



**Manual**

**TC3 C++**

**TwinCAT 3**

**Version:** 1.7  
**Date:** 2019-01-17  
**Order No.:** TC1300

**BECKHOFF**



# Table of contents

<b>1 Foreword</b> .....	<b>7</b>
1.1 Notes on the documentation.....	7
1.2 Safety instructions .....	8
<b>2 Overview</b> .....	<b>9</b>
<b>3 Introduction</b> .....	<b>10</b>
3.1 From conventional user mode programming to real-time programming in TwinCAT .....	12
<b>4 Requirements</b> .....	<b>18</b>
<b>5 Preparation - just once!</b> .....	<b>20</b>
5.1 Installation "Microsoft Windows Driver Kit 7 (WDK)" .....	20
5.2 Visual Studio - TwinCAT XAE Base toolbar .....	22
5.3 Prepare Visual Studio - Configuration and Platform toolbar .....	23
5.4 x64: driver signing .....	23
5.4.1 Signing drivers .....	24
5.4.2 Test signing .....	24
5.4.3 Delete test certificate .....	26
5.4.4 Customer Certificates .....	28
5.5 SecureBoot: Driver signing .....	29
<b>6 Modules</b> .....	<b>30</b>
6.1 The TwinCAT Component Object Model (TcCOM) concept .....	30
6.1.1 TwinCAT module properties .....	32
6.1.2 TwinCAT module state machine.....	39
6.2 Module-to-module communication .....	41
<b>7 Modules - Handling</b> .....	<b>44</b>
7.1 Export modules.....	44
7.2 Import modules.....	45
<b>8 TwinCAT C++ development</b> .....	<b>48</b>
<b>9 Quick start</b> .....	<b>50</b>
9.1 Create TwinCAT 3 project .....	50
9.2 Create TwinCAT 3 C++ project .....	51
9.3 Implement TwinCAT 3 C++ project .....	55
9.4 Compiling/building a TwinCAT 3 C++ project.....	56
9.5 Create TwinCAT 3 C++ Module instance .....	57
9.6 Create a TwinCAT task and apply it to the module instance .....	59
9.7 TwinCAT 3 enable C++ debugger .....	61
9.8 Activating a TwinCAT 3 project .....	62
9.9 Debug TwinCAT 3 C++ project.....	63
<b>10 Debugging</b> .....	<b>68</b>
10.1 Details of Conditional Breakpoints.....	71
10.2 Visual Studio tools .....	73
<b>11 Wizards</b> .....	<b>76</b>
11.1 TwinCAT C++ Project Wizard.....	76

11.2	TwinCAT Module Class Wizard .....	77
11.3	TwinCAT Module Class Editor (TMC).....	79
11.3.1	Overview .....	82
11.3.2	Basic Information .....	83
11.3.3	Data Types .....	83
11.3.4	Modules .....	101
11.4	TwinCAT Module Instance Configurator.....	124
11.4.1	Object .....	125
11.4.2	Context .....	126
11.4.3	Parameter (Init).....	126
11.4.4	Data Area.....	127
11.4.5	Interfaces .....	127
11.4.6	Interface Pointer .....	127
11.4.7	Data Pointer.....	128
11.5	Customer-specific project templates .....	128
11.5.1	Overview .....	128
11.5.2	Files involved .....	129
11.5.3	Transformations.....	130
11.5.4	Notes on handling.....	131
<b>12</b>	<b>Programming Reference .....</b>	<b>134</b>
12.1	File Description .....	135
12.1.1	Compilation procedure.....	137
12.2	Limitations .....	137
12.3	Memory Allocation .....	138
12.4	Interfaces.....	139
12.4.1	Interface ITcCyclic .....	140
12.4.2	Interface ITcCyclicCaller.....	141
12.4.3	Interface ITcFileAccess .....	143
12.4.4	Interface ITcFileAccessAsync.....	151
12.4.5	Interface ITcloCyclic .....	153
12.4.6	Interface ITcloCyclicCaller.....	154
12.4.7	Interface ITComObject.....	156
12.4.8	ITComObject interface (C++ convenience) .....	160
12.4.9	Interface ITcPostCyclic.....	161
12.4.10	Interface ITcPostCyclicCaller.....	162
12.4.11	Interface ITcRTTimeTask .....	164
12.4.12	Interface ITcTask.....	165
12.4.13	Interface ITcTaskNotification .....	168
12.4.14	Interface ITcUnknown .....	169
12.5	Runtime Library (RtlR0.h).....	171
12.6	ADS Communication .....	173
12.6.1	AdsReadDeviceInfo .....	173
12.6.2	AdsRead .....	175
12.6.3	AdsWrite .....	177
12.6.4	AdsReadWrite.....	179
12.6.5	AdsReadState.....	181



12.6.6	AdsWriteControl.....	183
12.6.7	AdsAddDeviceNotification .....	185
12.6.8	AdsDelDeviceNotification .....	187
12.6.9	AdsDeviceNotification.....	189
12.7	Mathematical Functions.....	190
12.8	Time Functions .....	192
12.9	STL / Containers.....	192
12.10	Error Messages - Comprehension .....	193
12.11	Module messages for the Engineering (logging / tracing) .....	193
<b>13</b>	<b>How to...?.....</b>	<b>197</b>
13.1	Using the Automation Interface .....	197
13.2	Windows 10 as target system up to TwinCAT 3.1 Build 4022.2 .....	197
13.3	Publishing of modules .....	197
13.4	Publishing modules on the command line .....	198
13.5	Clone .....	198
13.6	Renaming TwinCAT C++ projects .....	199
13.7	Access Variables via ADS .....	201
13.8	TcCallAfterOutputUpdate for C++ modules.....	201
13.9	Ordering Execution in one Task .....	201
13.10	Use Stack Size > 4kB .....	202
13.11	Setting version/vendor information .....	203
13.12	Delete Module .....	204
13.13	Initialization of TMC-member variables .....	205
13.14	Using PLC Strings as Method-Parameter .....	205
13.15	Third Party Libraries .....	206
13.16	Linking via TMC editor (TcLinkTo).....	207
<b>14</b>	<b>Troubleshooting .....</b>	<b>209</b>
14.1	Build - "Cannot open include file ntddk.h".....	209
14.2	Build - "The target ... does not exist in the project".....	209
14.3	Debug - "Unable to attach" .....	210
14.4	Activation – "invalid object id" (1821/0x71d).....	211
14.5	Error Message – VS2010 and LNK1123/COFF .....	211
14.6	Using C++ classes in TwinCAT C++ module .....	211
14.7	Using afxres.h.....	211
<b>15</b>	<b>C++-samples .....</b>	<b>213</b>
15.1	Overview.....	213
15.2	Sample01: Cyclic module with IO .....	215
15.3	Sample02: Cyclic C++ logic using IO from IO-task .....	216
15.4	Sample03: C++ as ADS server .....	216
15.4.1	Sample03: TC3 ADS Server written in C++.....	217
15.4.2	Sample03: ADS client UI in C#.....	221
15.5	Sample05: C++ CoE access via ADS .....	225
15.6	Sample06: UI-C#-ADS client uploading the symbolic from module .....	226
15.7	Sample07: Receiving ADS Notifications.....	231
15.8	Sample08: provision of ADS-RPC.....	232

15.9	Sample10: module communication: Using data pointer .....	235
15.10	Sample11: module communication: PLC module invokes method of C-module .....	236
15.10.1	TwinCAT 3 C++ module providing methods .....	237
15.10.2	Calling methods offered by another module via PLC .....	251
15.11	Sample11a: Module communication: C module calls a method of another C module .....	263
15.12	Sample12: module communication: Using IO mapping .....	264
15.13	Sample13: Module communication: C-module calls PLC methods .....	265
15.14	Sample19: Synchronous File Access .....	268
15.15	Sample20: FileIO-Write .....	269
15.16	Sample20a: FileIO-Cyclic Read / Write .....	269
15.17	Sample22: Automation Device Driver (ADD): Access DPRAM .....	270
15.18	Sample23: Structured Exception Handling (SEH) .....	272
15.19	Sample25: Static Library .....	274
15.20	Sample26: Execution order at one task.....	275
15.21	Sample30: Timing Measurement.....	277
15.22	Sample31: Functionblock TON in TwinCAT3 C++ .....	278
15.23	Sample35: Access Ethernet .....	279
15.24	Sample37: Archive data .....	280
15.25	TcCOM samples .....	281
15.25.1	TcCOM_Sample01_PlcToPlc .....	281
15.25.2	TcCOM_Sample02_PlcToCpp .....	291
15.25.3	TcCOM_Sample03_PlcCreatesCpp .....	295
<b>16</b>	<b>Appendix .....</b>	<b>300</b>
16.1	ADS Return Codes .....	300
16.2	Retain data .....	304
16.3	Creating and handling C++ projects and modules .....	307
16.4	Creating and handling TcCOM modules .....	311

# 1 Foreword

## 1.1 Notes on the documentation

This description is only intended for the use of trained specialists in control and automation engineering who are familiar with the applicable national standards.

It is essential that the documentation and the following notes and explanations are followed when installing and commissioning the components.

It is the duty of the technical personnel to use the documentation published at the respective time of each installation and commissioning.

The responsible staff must ensure that the application or use of the products described satisfy all the requirements for safety, including all the relevant laws, regulations, guidelines and standards.

### Disclaimer

The documentation has been prepared 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 prior announcement. No claims for the modification of products that have already been supplied may be made on the basis of the data, diagrams and descriptions in this documentation.

### Trademarks

Beckhoff®, TwinCAT®, EtherCAT®, Safety over EtherCAT®, TwinSAFE®, XFC® and XTS® are registered trademarks of and licensed by Beckhoff Automation GmbH.

Other designations used in this publication may be trademarks whose use by third parties for their own purposes could violate the rights of the owners.

### Patent Pending

The EtherCAT Technology is covered, including but not limited to the following patent applications and patents:

EP1590927, EP1789857, DE102004044764, DE102007017835

with corresponding applications or registrations in various other countries.

The TwinCAT Technology is covered, including but not limited to the following patent applications and patents:

EP0851348, US6167425 with corresponding applications or registrations in various other countries.

**EtherCAT** 

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

### Copyright

© Beckhoff Automation GmbH & Co. KG, Germany.

The reproduction, distribution and utilization of this document as well as the communication of its contents to others without express authorization are prohibited.

Offenders will be held liable for the payment of damages. All rights reserved in the event of the grant of a patent, utility model or design.

## 1.2 Safety instructions

### Safety regulations

Please note the following safety instructions and explanations!  
Product-specific safety instructions can be found on following pages or in the areas mounting, wiring, commissioning etc.

### Exclusion of liability

All the components are supplied in particular hardware and software configurations 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 engineering who are familiar with the applicable national standards.

### Description of symbols

In this documentation the following symbols are used with an accompanying safety instruction or note. The safety instructions must be read carefully and followed without fail!

#### **DANGER**

##### **Serious risk of injury!**

Failure to follow the safety instructions associated with this symbol directly endangers the life and health of persons.

#### **WARNING**

##### **Risk of injury!**

Failure to follow the safety instructions associated with this symbol endangers the life and health of persons.

#### **CAUTION**

##### **Personal injuries!**

Failure to follow the safety instructions associated with this symbol can lead to injuries to persons.

#### **NOTE**

##### **Damage to the environment or devices**

Failure to follow the instructions associated with this symbol can lead to damage to the environment or equipment.



#### **Tip or pointer**

This symbol indicates information that contributes to better understanding.

## 2 Overview

This chapter is all about TwinCAT 3 implementation in C/C++. The most important chapters are:

- [Start from scratch](#)  
Which platforms are supported? Additional installations to implement TwinCAT 3 C++ modules? Find all answers in [Requirements \[▶ 18\]](#) and [Preparation \[▶ 20\]](#). Limitations are documented [here \[▶ 137\]](#).
- [Quick start \[▶ 50\]](#)  
This is a “less than five minutes sample” to create a simple incrementing counter in C++ being executed cyclically. Counter value will be monitored and overwritten, debugging capabilities will be presented etc.
- [MODULES \[▶ 32\]](#)  
Modularization the basic philosophy of TwinCAT 3. Especially for C++ Modules it is required to understand the module concept of TwinCAT 3.  
Minimum is to read one article about the architecture of TwinCAT modules.
- [Wizards \[▶ 76\]](#)  
Documentation of visual components of the TwinCAT C++ environment.  
This includes on the one hand tools for creating projects and on the other hand tools for editing module and configuring instances of modules.
- [Programming Reference \[▶ 134\]](#)  
This chapter contains detailed information for programming in TwinCAT C++. For Example Interfaces as well as other TwinCAT provided functions for ADS communication and helper methods are located here.
- [The How to ...? \[▶ 197\]](#) Chapter contains useful hints while working with TwinCAT C++.
- [Samples \[▶ 213\]](#)  
Some Interfaces and their usage is best described by working code, which is provided as download including source code and solution.

## 3 Introduction

The approach of emulating classic automation devices such as programmable logic controllers (PLCs) and numeric controllers (NC) in software on a powerful standard hardware has been the state of the art for many years and is now pursued by many suppliers.

There are many advantages; however the most important advantage is certainly that the software is independent of the hardware to a very great extent. This means both that the performance capacity of the hardware can be adapted to the particular application and that it will be possible to benefit automatically from its general further development.

This applies in particular to PC hardware, where increases in performance are still continuing at a dramatic rate. The relative independence from a supplier which also results from this separation of software and hardware is also important to the user.

Since the PLC and the motion controller – and possibly other automation components – continue to exist as independent, logical units with this approach, there are very few changes in the application architecture in comparison with classic automation technology.

The PLC determines the logical operational sequence of the machine and assigns the motion controller to implement certain axis functions. Due to the increased performance of the controllers and the possibility to use higher-level programming languages (IEC 61131-3), complex machines can also 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.

### Best of both worlds

The ideal controller for modular machines borrows from both the local and the central control architecture. A central, powerful and as general as possible computer platform is “naturally” used as the control hardware.

The advantages of central control technology:

- lower overall costs
- available

- fast, and modular field bus systems (keyword EtherCAT)
- and the possibility of being able to access all the information of the entire system without loss of communication

are decisive arguments.

The advantages of the local approach already outlined above can also be put into practice in the central controller by means of appropriate 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 encapsulated and offer their functions to the outside via standardized interfaces or use appropriate functions of other modules or the runtime.

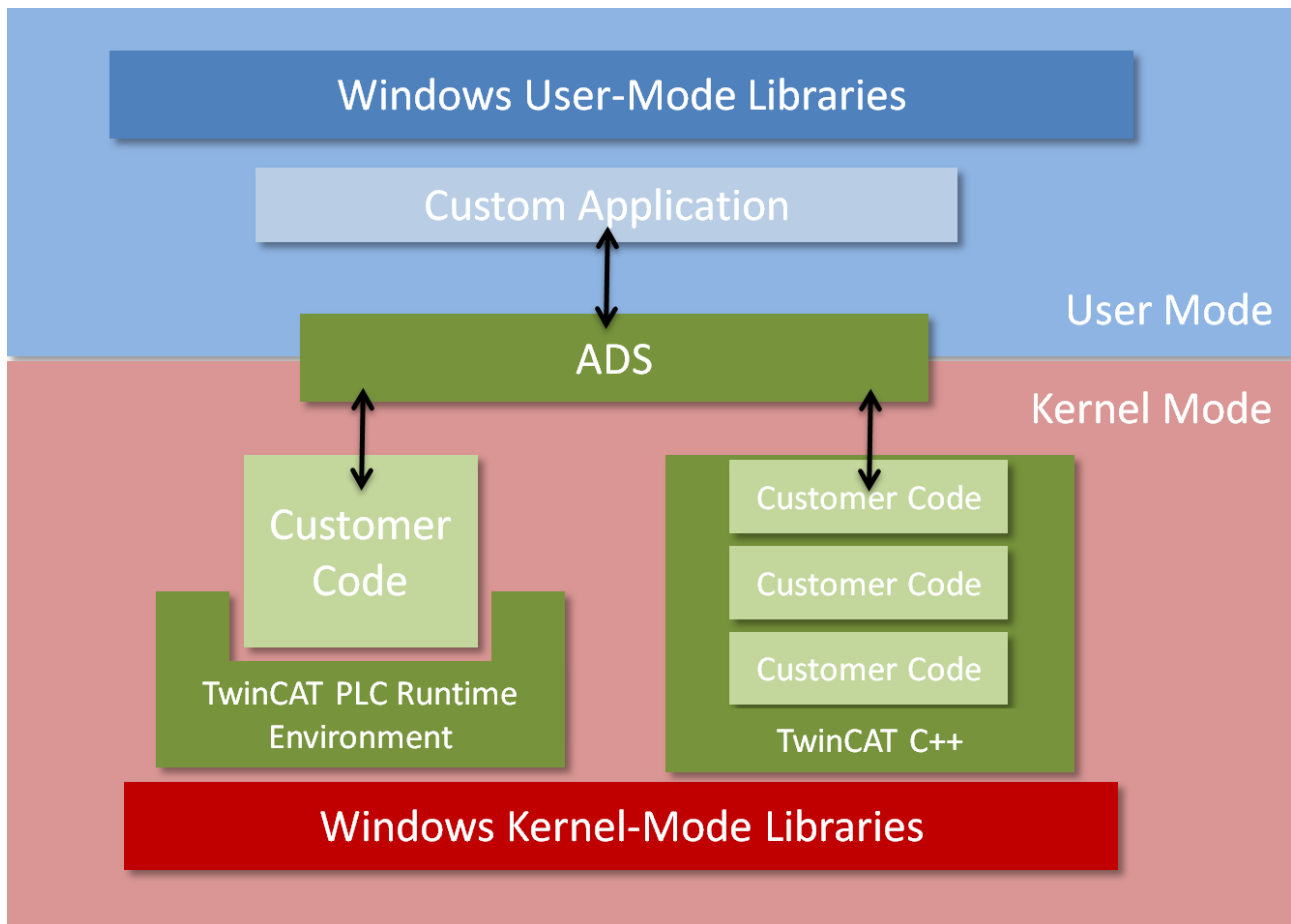
A meaningful profile formation takes place by the definition of these interfaces and the standardization of the appropriate parameters and process data. Since the individual modules are implemented within one runtime, direct calls to other modules are also possible – in turn via appropriate standardized interfaces. The modularization can therefore take place at meaningful limits, without having to give consideration to communication losses.

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 requirements of all instanced modules (memory, tasks and computing power) are fulfilled.

### **TwinCAT 3 Run-Time**

The TwinCAT runtime offers a software environment in which TwinCAT modules are loaded, implemented and managed. It offers additional basic functions so that the system resources can be used (memory, tasks, fieldbus and hardware access etc.). The individual modules do not have to be created using the same compiler and can therefore be independent of one another and can originate from different manufacturers.

A series of system modules is automatically loaded at the start of the runtime, so that their properties are available to other modules. However, access to the properties of the system modules takes place in the same way as access to the properties of normal modules, so that it is unimportant to the modules whether the respective property is made available by a system module or a normal module.



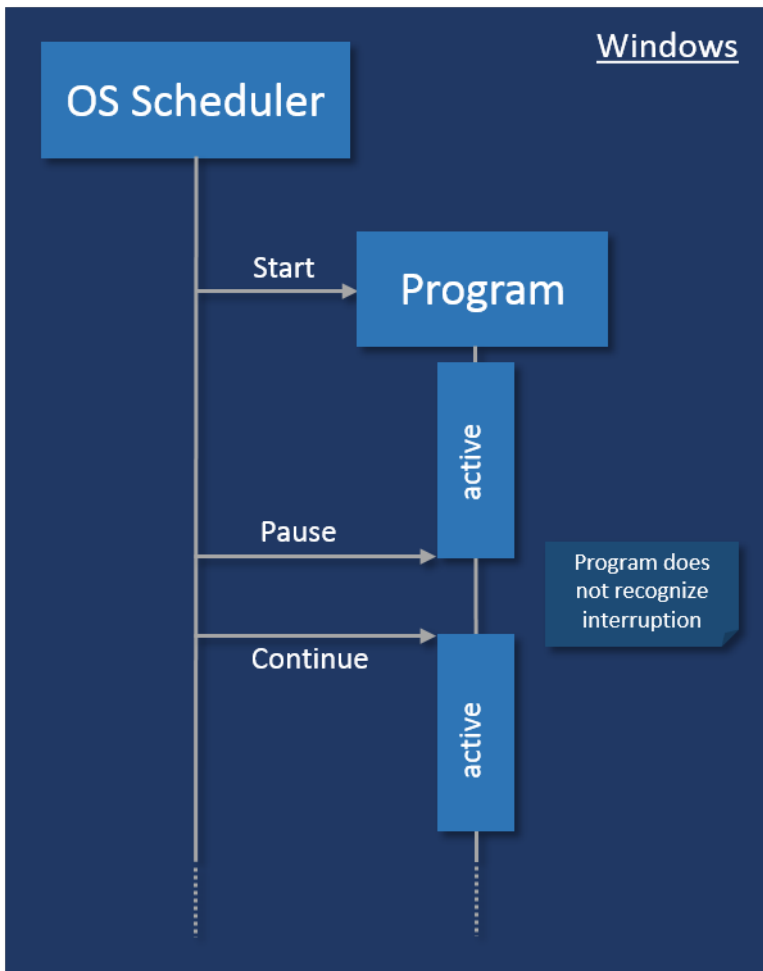
In contrast to the PLC, where customer code is executed within a runtime environment, TwinCAT C++ modules are not within such a hosted environment. As a consequence TwinCAT C++ modules are executed as Kernel Modules (.sys) – thus they are built with the kernel mode libraries.

### 3.1 From conventional user mode programming to real-time programming in TwinCAT

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.

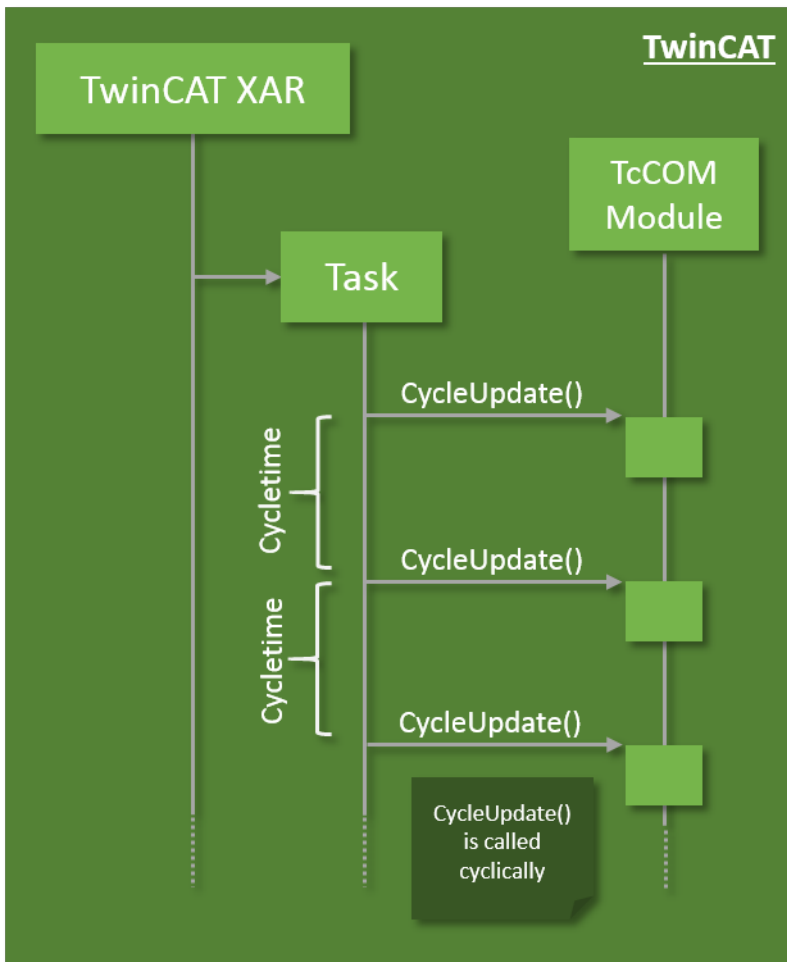




With conventional user mode programming, e.g. in C#, a program is created, which is then executed by an operating system.

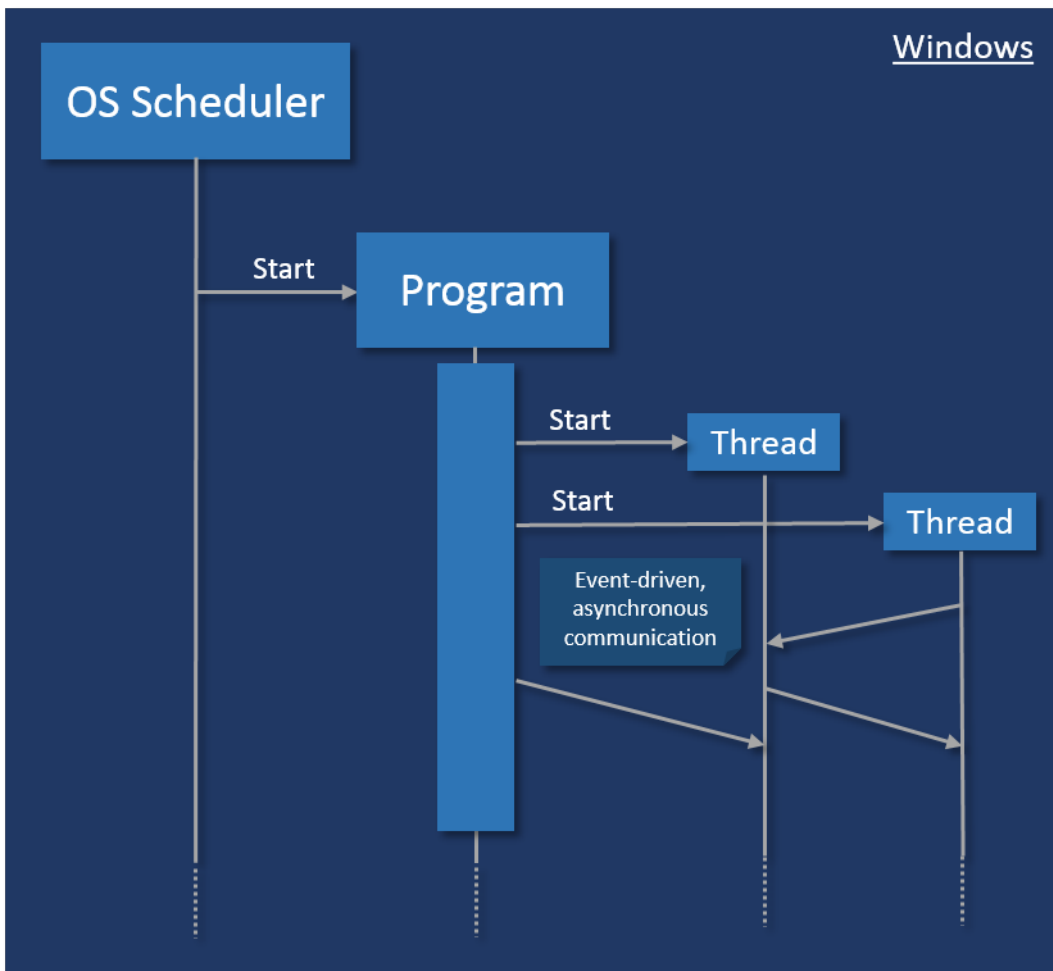
The program is started by the operating system and can run independently, i.e. it has full control over its own execution, including aspects such as threading and memory management. In order to enable multitasking, the operating system interrupts such a program at any time and for any period. The program 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).



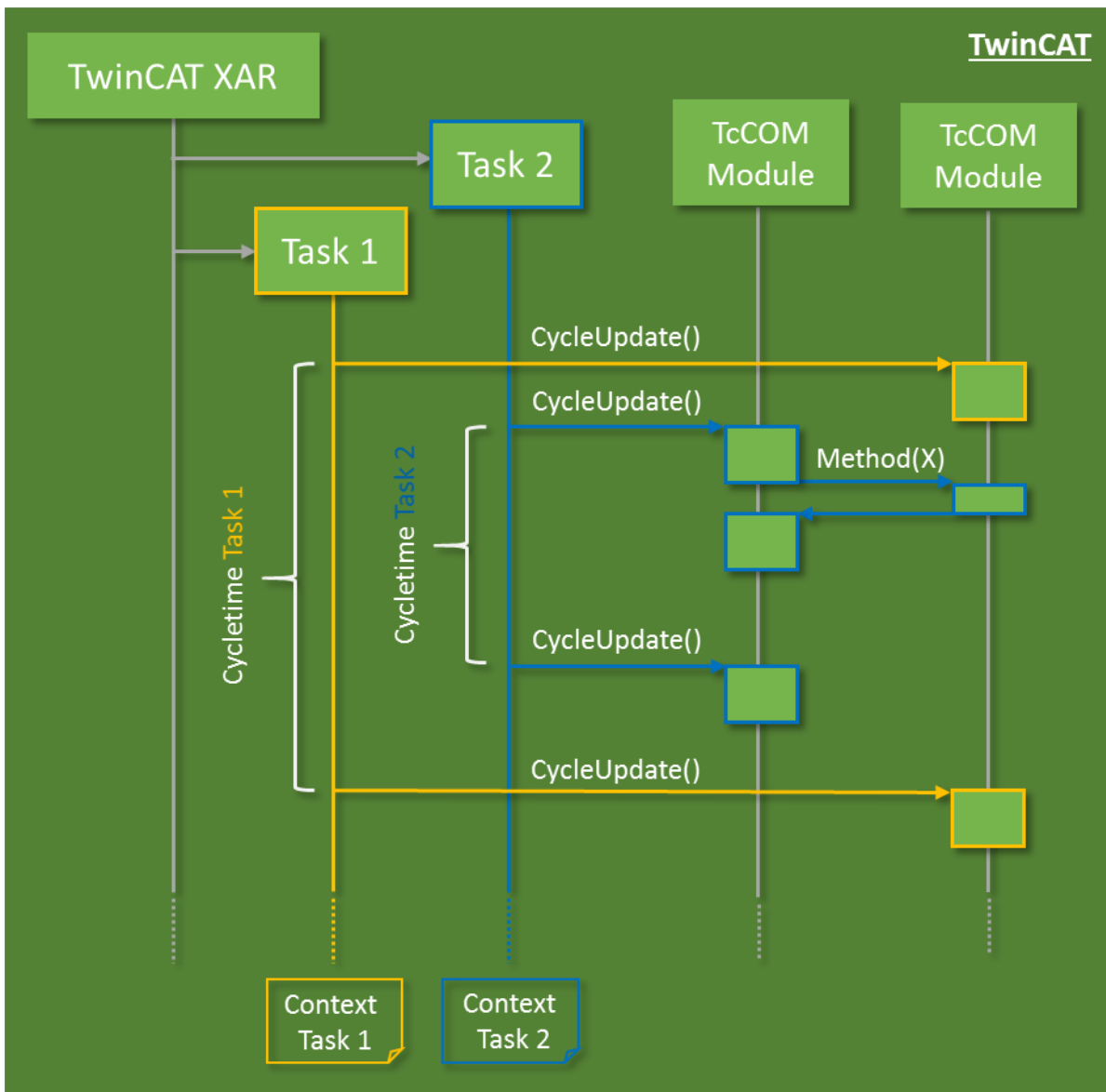
The basic idea of PLC is therefore adopted for TwinCAT C++: The TwinCAT real-time system manages the real-time tasks, handles the scheduling and cyclically calls an entry point in the program code. The program execution must be completed within the available cycle length and return the control. The TwinCAT 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 "Windows kernel context" and the "TwinCAT real-time context", not as a user mode application.

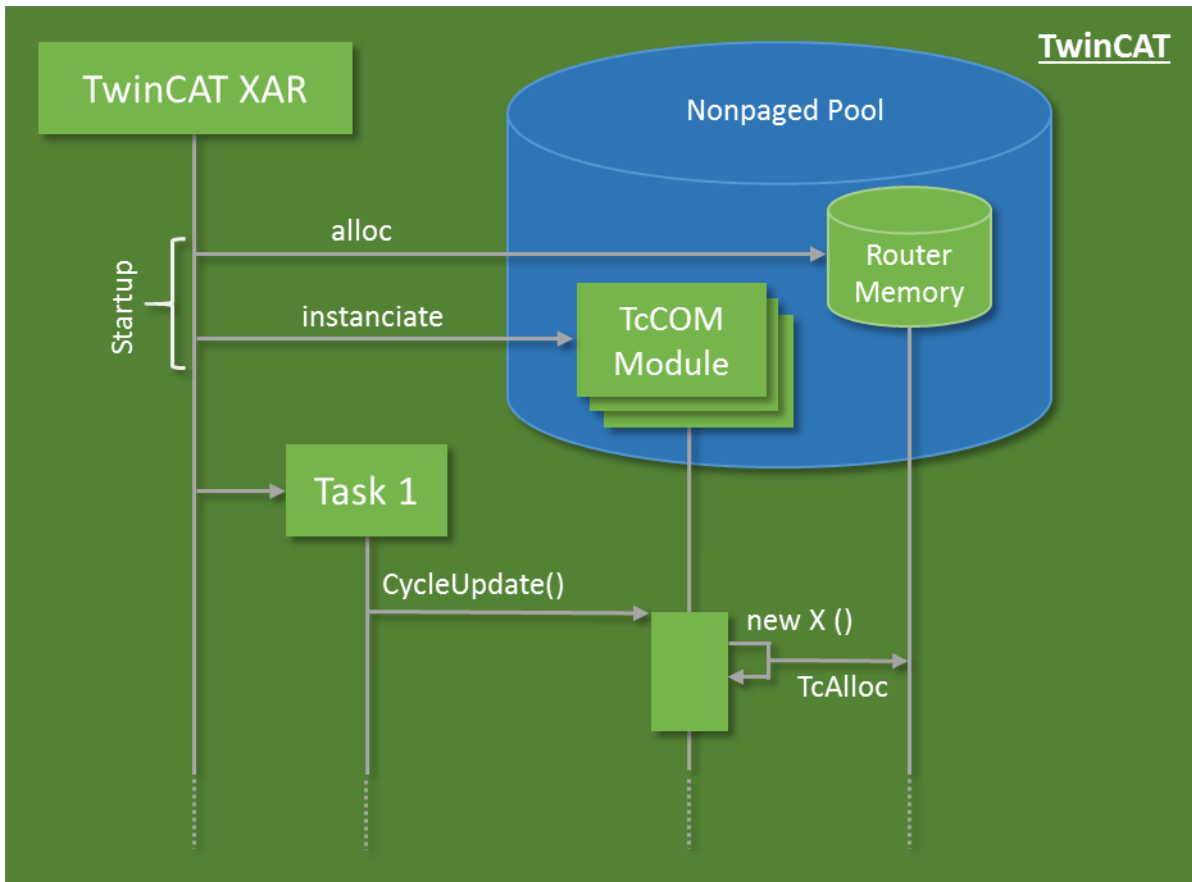
During startup/shutdown of the modules, code for (de)initialization is initially executed in the Windows kernel context; only the last phase and the cyclic calls are executed in the TwinCAT real-time context.

Details are described in the "[Module state machine \[▶ 39\]](#)" section.

### 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.

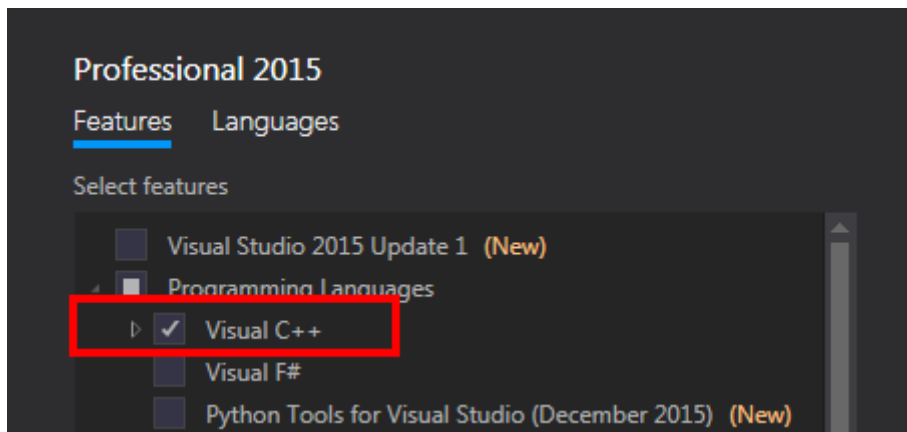
## 4 Requirements

### Overview of minimum requirements

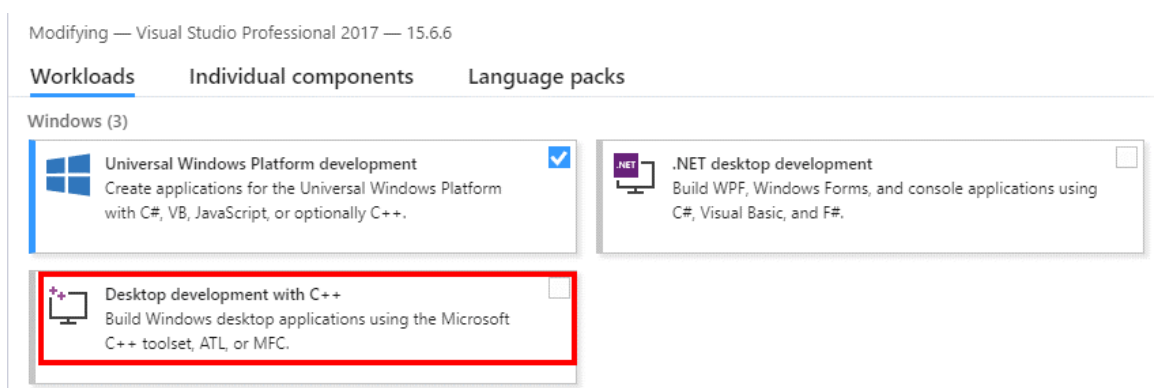
The implementation and debugging of TwinCAT 3 C++ modules requires:

#### The following must be installed on the engineering PC:

- Microsoft Visual Studio 2010 (with Service Pack 1), 2012, 2013 or 2015 Professional, Premium or Ultimate
  - When installing Visual Studio 2015, the Visual C++ development option must be manually selected, as this option is not selected with the automatic installation:



- When installing Visual Studio 2017, the "Desktop development with C++" option must be manually selected, as this option is not selected with the automatic installation:



- Microsoft "Windows Driver Kit" (WDK)  
To implement and debug C++ modules, Microsoft "Windows Driver Kit" (WDK) must be installed:  
[Installing "Windows Driver Kit" \(WDK\) \[▶ 20\]](#)
- TwinCAT 3 installation (XAE engineering)

#### On the runtime PC:

- IPC or Embedded CX PC with Microsoft operating system (Windows XP or Windows 7 or higher)
- Microsoft Visual Studio does **not** have to be installed
- Microsoft "Windows Driver Kit" (WDK) does **not** have to be installed.  
(No additional installation is required for the integration and application of existing binary C++ modules in a TwinCAT 3 PLC environment.)
- TwinCAT 3 installation (XAR runtime)

#### Limitations on the runtime PC

- TwinCAT 3.0 only supports 32-bit operating systems as target platform (runtime PC).  
TC3.0 can be used as engineering platform on x64 PCs. The program can be transferred to a 32bit (x86) remote PC over the network and executed there.

- TC3.1 also supports x64-bit operating systems as target platform (runtime PC) The drivers have to be signed, as documented [here \[▶ 23\]](#), which [requires a certificate \[▶ 28\]](#) for productive operation.
- The target runtime must be based on "Windows NT Kernel", such as Windows XP, Windows 7 or the embedded versions Windows XP Embedded, Windows Embedded Standard 2009, Windows Embedded Standard 7

## 5 Preparation - just once!

A PC for the engineering of TwinCAT C++ modules must be prepared. These steps only have to be performed once:

- [Microsoft Windows Driver Kit \(WDK\) \[► 20\]](#) must be installed and
- TwinCAT [Basis \[► 22\]](#) and the [configuration and platform \[► 23\]](#) toolbar have to be configured
- On x64 PCs modules have to be signed, in order to be able to be executed. See [Documentation for the setup of a test signing \[► 23\]](#).
- If the operating system of the target system requires enhanced validation for drivers, for example through a SecureBoot, a [corresponding signing \[► 29\]](#) has to be performed at Microsoft.

### 5.1 Installation "Microsoft Windows Driver Kit 7 (WDK)"

#### Overview

The implementation of TwinCAT 3 C++ modules requires parts of the "Windows Driver Kit" (WDK).

The installation is only necessary for the TwinCAT 3 engineering environment in order to be able to create and edit C++ modules. It is not required for the target platform of the TwinCAT 3 runtime.

1. Download "Windows Driver Kit 7.1" from the Microsoft Download Center <http://www.microsoft.com/downloads/en/details.aspx?displaylang=en&FamilyID=36a2630f-5d56-43b5-b996-7633f2ec14ff>



#### Windows Driver Kit Version 7.1.0

Language: **English**

**Download**

The Windows Driver Kit (WDK) Version 7.1.0 is an update to the WDK 7.0.0 release and contains the tools, code samples, documentation, compilers, headers and libraries with which software developers create drivers for Windows 7, Windows Vista, Windows XP, Windows Server 2008 R2, Windows Server 2008, and Windows Server 2003.

[+ Details](#)

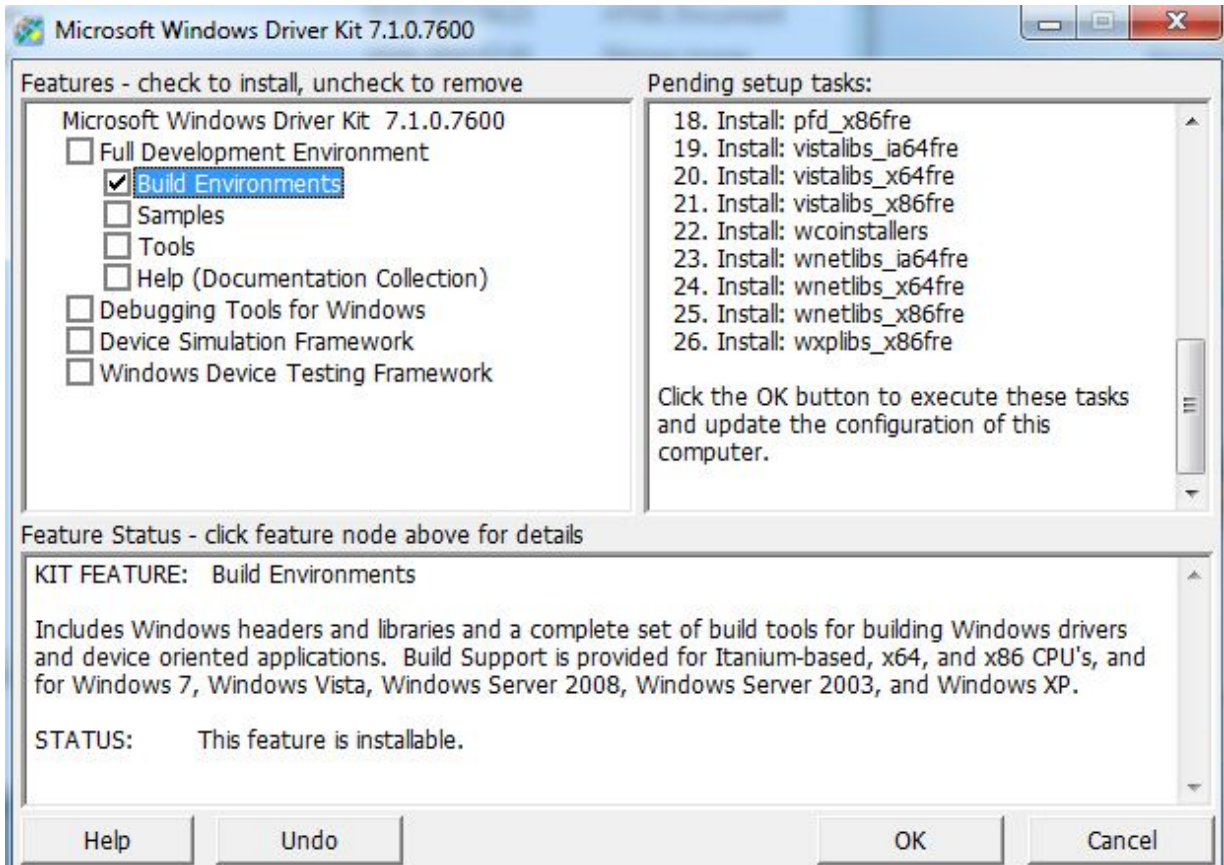
[+ System Requirements](#)

[+ Install Instructions](#)

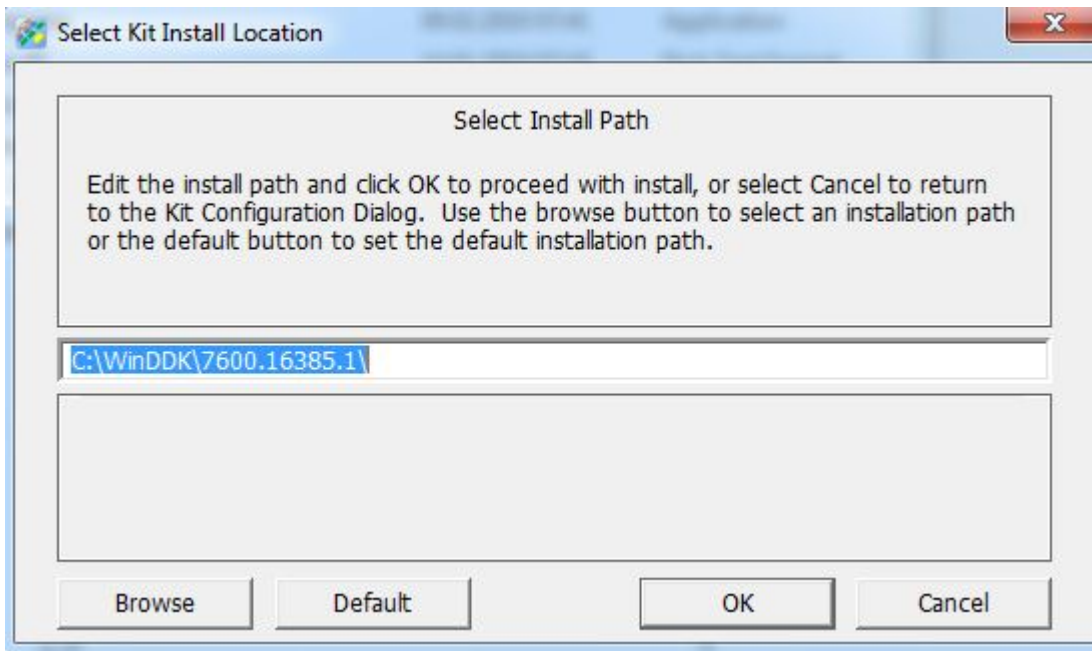
2. Following the download, either burn a CD of the downloaded ISO image or use a virtual (software-based) CD drive.
3. Start "KitSetup.exe" on CD/downloaded ISO image (start the installation with "Run As Administrator..." on Windows 7 PCs)



4. Select the option "Build Environment" - none of the other components are required by TwinCAT 3 - and click on "OK" to continue.

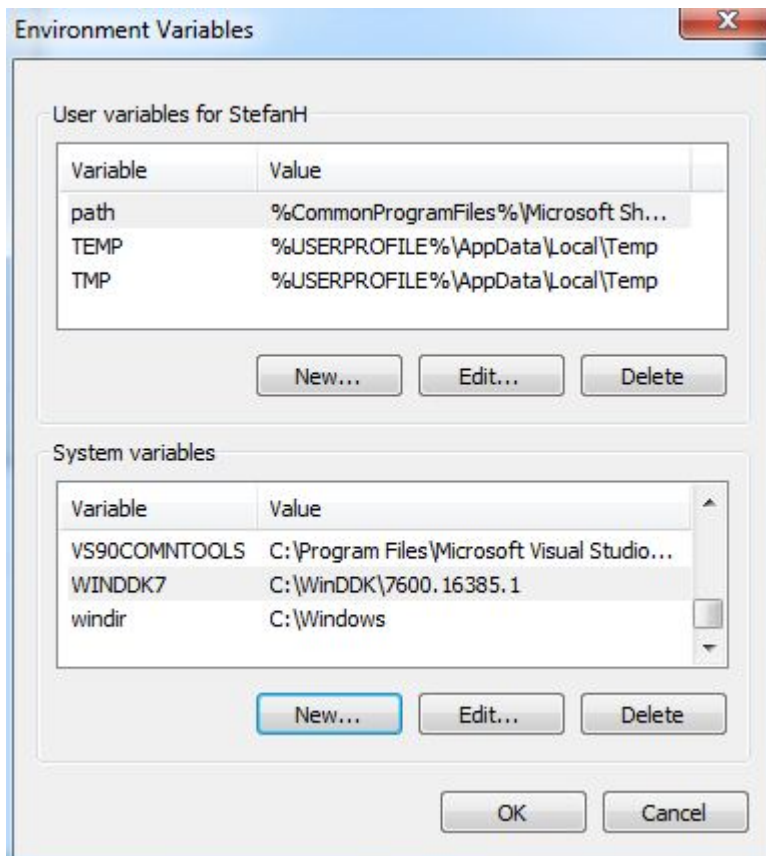


5. After accepting the Microsoft EULA license, select the destination folder for the installation. By default the root folder "C:\" will be selected - therefore "C:\WinDDK\7600.16385.1" will be suggested. The digits "7600..." may be different in the case of a newer version of the "Windows Driver Kit".
6. Start the installation with "OK"



7. In future TwinCAT 3 will take care of the following step. Now it needs to be done manually once.
8. Navigate to "Start" -> "Control Panel" -> "System" and select "Advanced system settings"
9. Select the "Advanced" tab and then click on "Environment Variables..."

10. In the lower area of "System variables", select "New..." and enter the following information:  
 Variable name WINDDK7  
 Variable value C:\WinDDK\7600.16385.1  
 The path may differ with a different version of the Windows Driver Kit or if a different installation path is specified.



11. Following the installation, log in again or restart the PC to confirm the new environment variable settings.

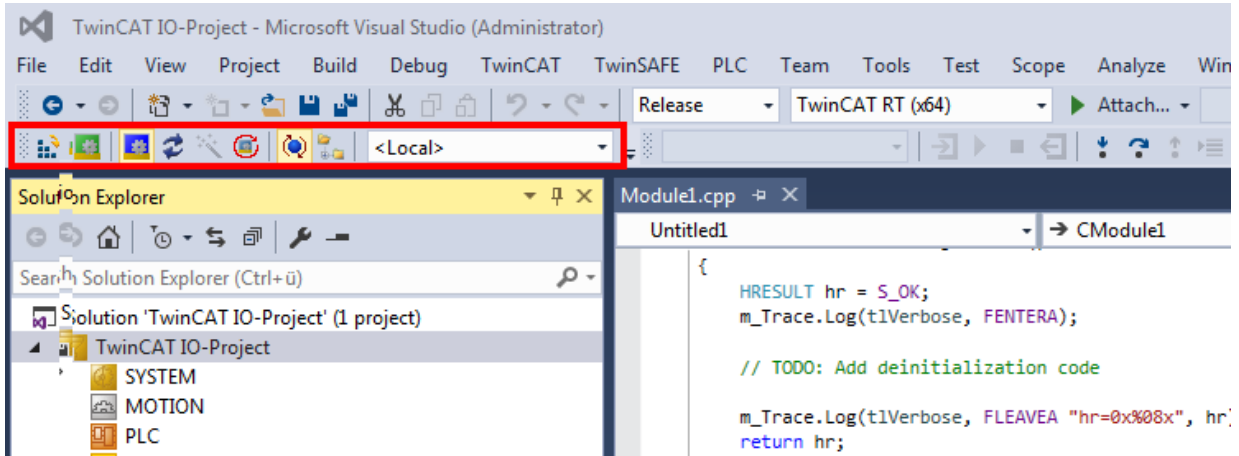
## 5.2 Visual Studio - TwinCAT XAE Base toolbar

### Add the "TwinCAT XAE Base" toolbar for efficient engineering

For better efficiency, TwinCAT 3 integrates own toolbar into the Visual Studio menu, which will support you in creating C++ projects. This toolbar should be added automatically to the Visual Studio menu by TwinCAT 3 Setup. However, if you would like to add it manually, you need to perform the following step:

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

⇒ The selected toolbar should now appear below the menu



### 5.3 Prepare Visual Studio - Configuration and Platform toolbar

#### Add the "Solution Configuration and Solution Platform" toolbar

The "Configuration and Platform" toolbar will enable you to select the target platform for building your project. This toolbar should be added automatically to the Visual Studio menu by TwinCAT 3 Setup. However, if you would like to add this toolbar manually, you need to perform the following steps:

1. Open the "View" menu and select "Toolbars\Customize"
2. Navigate to the tab "Commands"
3. Check the option box "Toolbar" and then select "Standard" in the list of toolbars
4. Click on "Add Command..."
5. Choose the "Build" category, select the command "Solution Configurations" and click on "Ok"
6. Repeat the last step for the command "Solution Platforms"
7. Finally, click on "Close"

⇒ Both commands should now appear below the menu bar



### 5.4 x64: driver signing

TwinCAT C++ modules must be signed so that they can be executed on x64 PCs.

**i** **Engineering requires no signing**

Only the execution requires certificates - engineering on an x64 machine with execution on an x86 machine requires no signing.

Since a published module should be executable on various PCs, signing is always necessary for publishing.

## 5.4.1 Signing drivers

### Overview

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

The signature, which is generated by the TwinCAT3 build process automatically, is used by 64bit Windows operating systems for authentication of the drivers to execute.

For signing a driver, a certificate is required. [This documentation](#) by Microsoft describes the process and background knowledge on how to retrieve a test and release certificate, which will be accepted by the 64bit Windows operating systems.

For using such a certificate in TwinCAT 3, configure the post-compile step of your x64 build target as documented in the [How to create a test certificate for test mode? \[▶ 24\]](#)

### Test Certificates

For testing purpose it is possible to create and use self-signed test certificates without any technical limitation.

The following tutorials describe the process on how to enable this capability.

Please make sure that you disable the capability and provide drivers signed by real certificates for production machines.

- [How to create a test certificate for test mode? \[▶ 24\]](#)
- [How to delete \(test-\) certificates? \[▶ 26\]](#)

### Further References:

MSDN, Test Certificates (Windows Drivers),

[http://msdn.microsoft.com/en-us/library/windows/hardware/ff553457\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/windows/hardware/ff553457(v=vs.85).aspx)

MSDN, MakeCert Test Certificate (Windows Drivers),

[http://msdn.microsoft.com/en-us/library/windows/hardware/ff548693\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/windows/hardware/ff548693(v=vs.85).aspx)

## 5.4.2 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**

**I** Developers may have a wide range of tools for creating certificates. Please follow this description exactly, in order to activate the test certificate mechanism.

---

One of the following commands must be executed

- **Visual Studio 2010 / 2012 prompt with administrator rights.** (via: All Programs -> Microsoft Visual Studio 2010/2012 -> Visual Studio Tools -> Visual Studio Command Prompt, then right-click on "Run as administrator")
- normal prompt (Start->Command Prompt) **with administrator rights**, then change to directory %WINDDK7%\bin\x86\, which contains the corresponding tools.

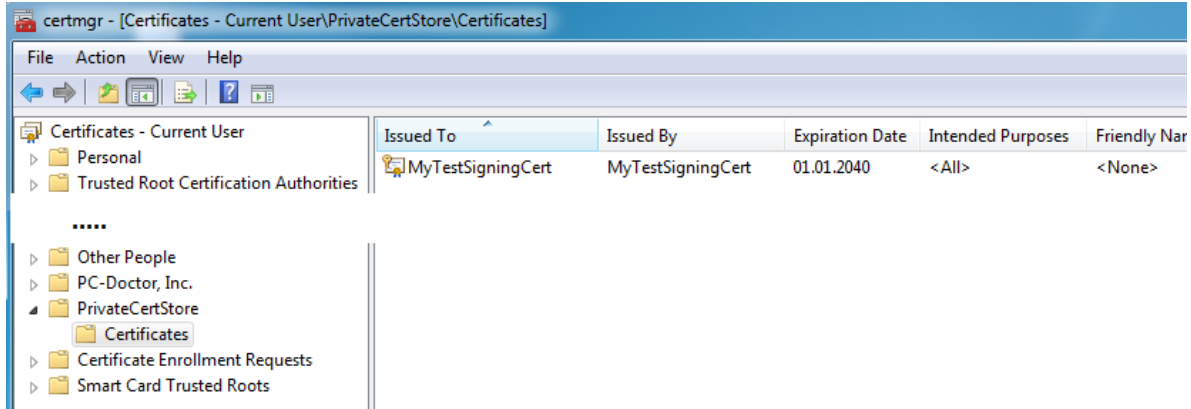
1. On XAE:

In the engineering system enter the following in the Visual Studio 2010 / 2012 prompt with administrator rights (see note above):

```
makecert -r -pe -ss PrivateCertStore -n CN=MyTestSigningCert
MyTestSigningCert.cer
```

⇒ This is followed by creation of a self-signed certificate, which is stored in the file "MyTestSigningCert.cer" and in the Windows Certificate Store

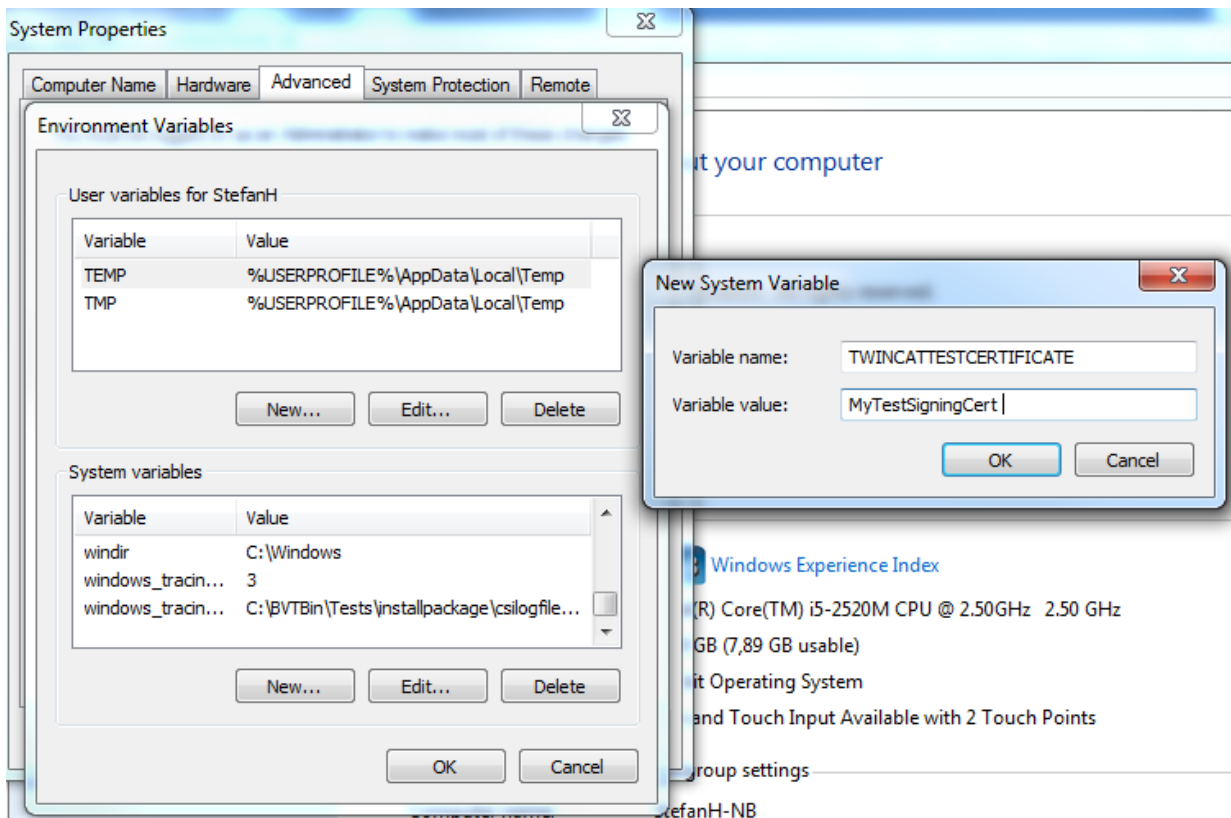
⇒ The result can be verified with mmc (Use File->Add/Remove Snap-in->Certificates):



2. On XAE:

Configuring the certificate such 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 step 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).**



3. On XAR (and XAE, if local test)

Activate signing mode, so that Windows can accept the self-signed certificates. This is possible on all systems, which can start the modules, i.e. engineering system or XAR (runtime) systems.



Use a command prompt to execute the following:

```
bcdedit /set testsigning yes
```

and restart the target system.

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



4. 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:

```
Output
Show output from: Build
1>----- Build started: Project: Untitled2, Configuration: Debug TwinCAT RT (x64) -----
1> header file << C:\TwinCAT\3.1\SDK\*_products\TwinCAT RT (x64)\Debug\Untitled2\Untitled2Version.h >> is up-to-date!
1> TcPch.cpp
1> Module1.cpp
1> Untitled2ClassFactory.cpp
1> Untitled2Driver.cpp
1> Untitled2.vcxproj -> C:\TwinCAT\3.1\SDK\*_products\TwinCAT RT (x64)\Debug\Untitled2.sys
1> The following certificate was selected:
1>   Issued to: MyTestSigningCert
1>
1>   Issued by: MyTestSigningCert
1>
1>   Expires:   Sun Jan 01 00:59:59 2040
1>
1>   SHA1 hash: E27A66E6A0C7BC0C86DFDD093DDF2486D1EE502E
1>
1>
1> Done Adding Additional Store
1> Successfully signed and timestamped: C:\TwinCAT\3.1\SDK\*_products\TwinCAT RT (x64)\Debug\Untitled2.sys
1>
1>
1> Number of files successfully Signed: 1
1>
1> Number of warnings: 0
1>
1> Number of errors: 0
1>
1>
===== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped =====
```

References:

[MSDN, test certificates \(Windows driver\)](#)

[MSDN, MakeCert test certificates \(Windows driver\),](#)

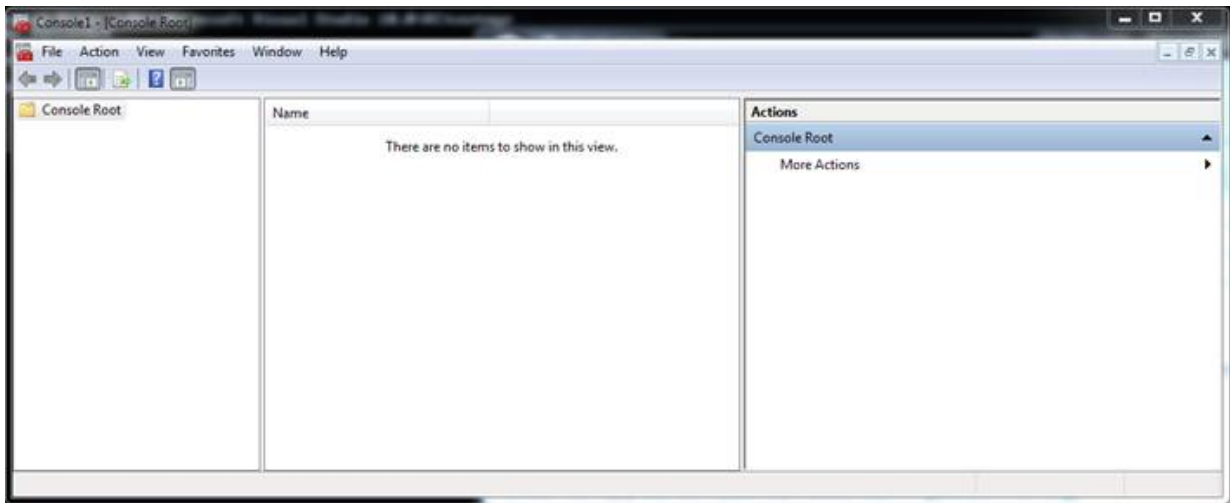
### 5.4.3 Delete test certificate

This article is about how to delete a test certificate.

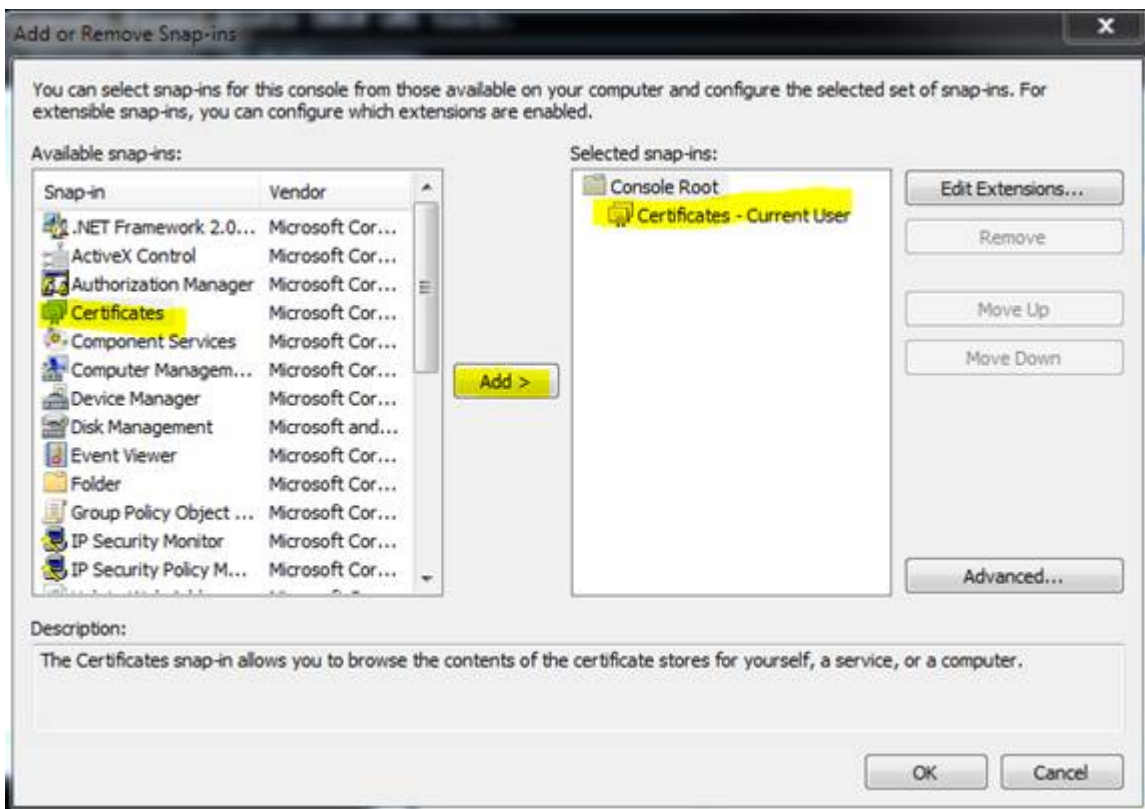
**Overview**

Deleting a certificate can be handled with the Microsoft Management Console:

1. Start the Management console MMC.exe via start menu or shell

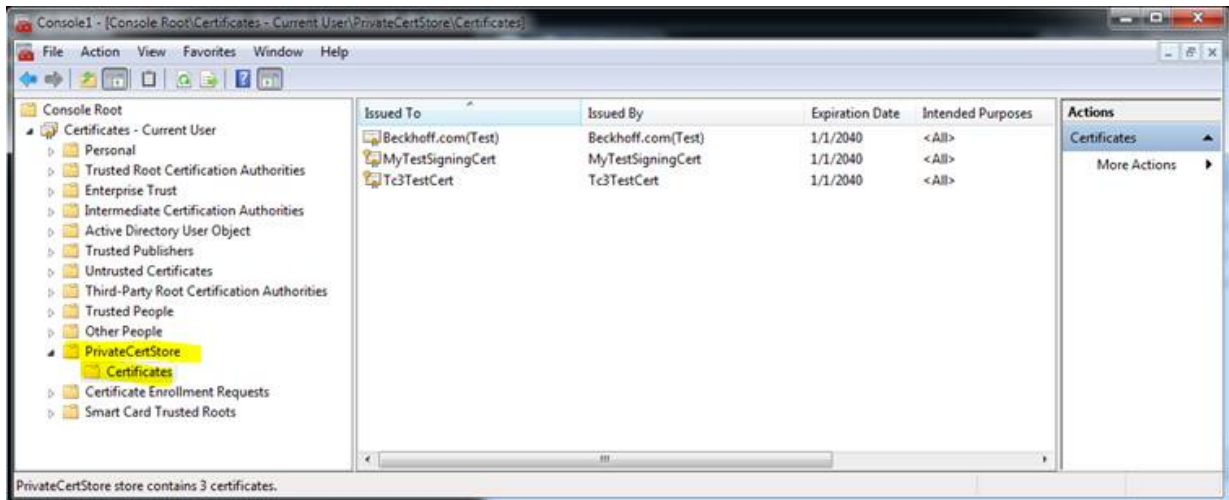


2. In the menu click "File" -> "Add/Remove Snap-in.." and select the certificate snap-in for the current user - finish with "OK"



⇒ The certificates should be listed in the node below "PrivateCertStore/Certificates"

## 3. Select the certificate to be deleted.

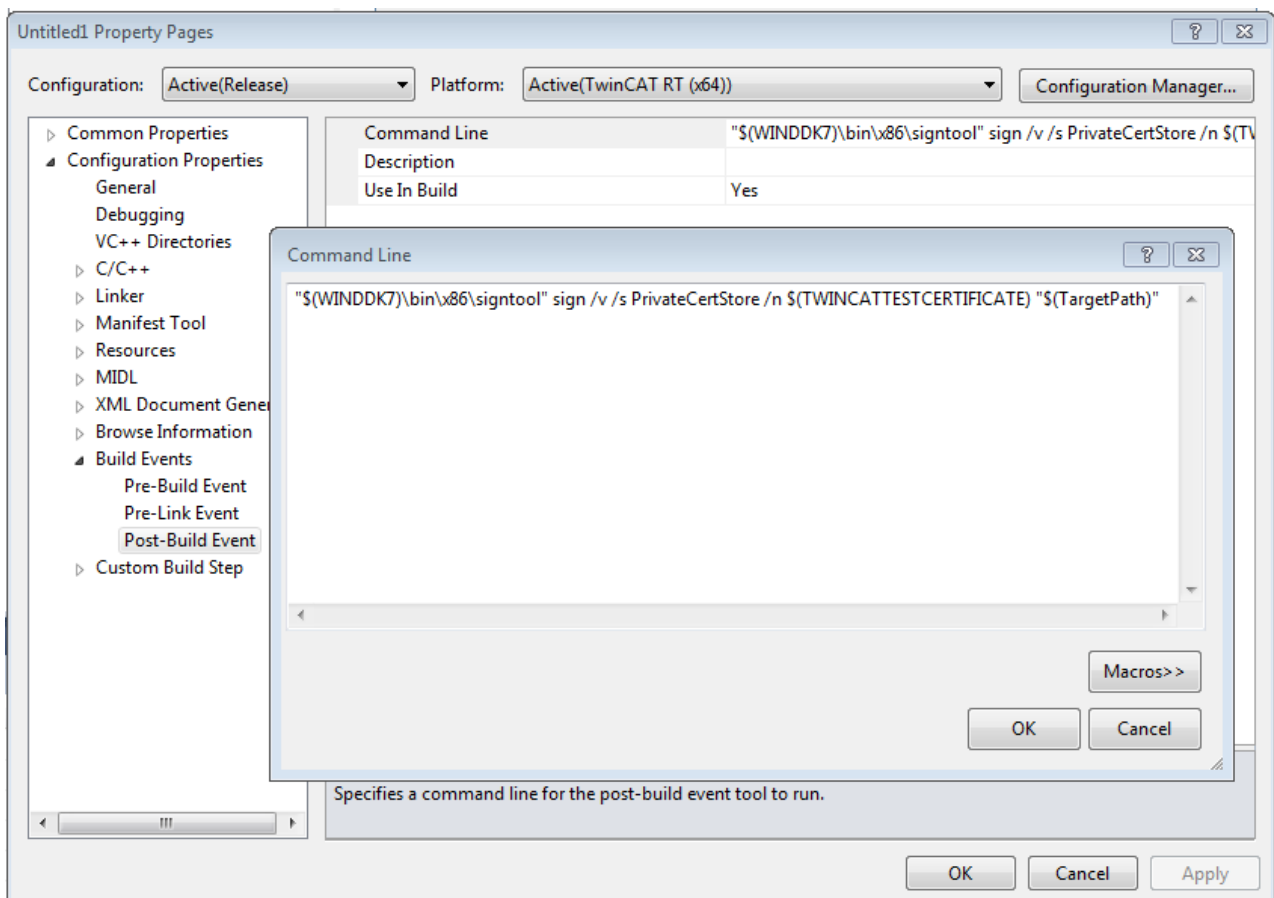


## 5.4.4 Customer Certificates

By using the TwinCAT C++ Class Wizard, the projects will be prepared using the prior described test certificate procedure on x64 targets.

This test signing system could be used for the whole engineering and testing process.

If one would like to establish an infrastructure and sign the kernel drivers with official Microsoft trusted certificates, the project properties' postbuild procedure provides the entry point:





The customer could either simply replace the value of the environment variable **TWINCATTESTCERTIFICATE** or identify another certificate to use. The customer could also modify the whole signing procedure using the sign tool.

## 5.5 SecureBoot: Driver signing

Systems may require enhanced validation of the Windows drivers. This is usually the case with systems with enabled SecureBoot.

In this case, the TwinCAT C++ drivers must also be signed by the "Attestation Signing" established by Microsoft in the same way as all other drivers that the operating system is to load. The procedure for this is documented in MSDN.

For development purposes, a shutdown of SecureBoot can simplify the development process on corresponding test systems.

## 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

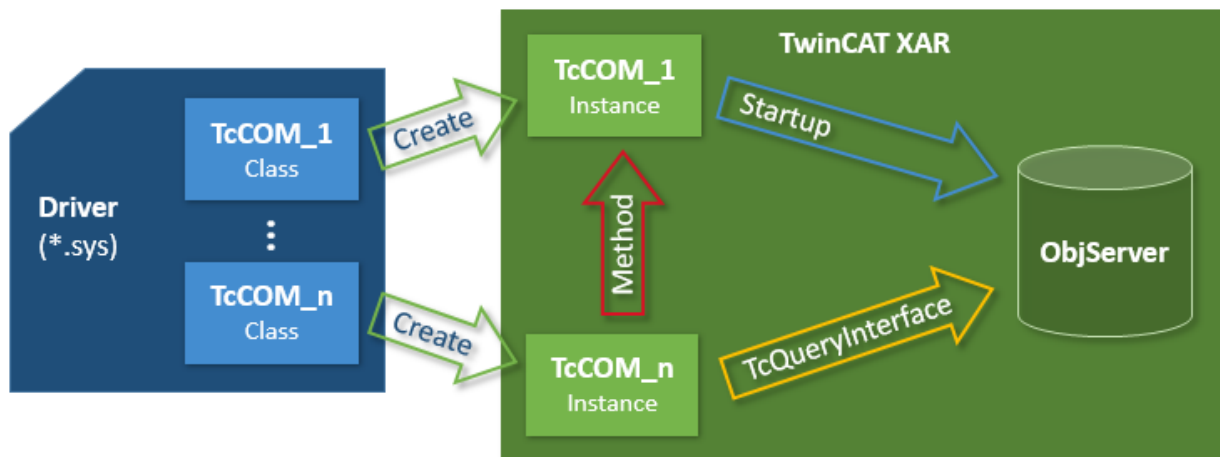
The TwinCAT Component Object Model defines the characteristics and the behavior of the modules. The model derived from the "Component Object Model" COM from Microsoft Windows describes the way in which various independently developed and compiled software components can co-operate with one another. To 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 IO. 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, is interfaces.

Interfaces define a set of methods that are combined in order 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 lifetime 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:

- ITCyclic is implemented by modules, which are called cyclically ("CycleUpdate"). The module can register via the ITCyclicCaller 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 device notifications 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. User-defined interfaces should only be defined if no suitable general interfaces are available.

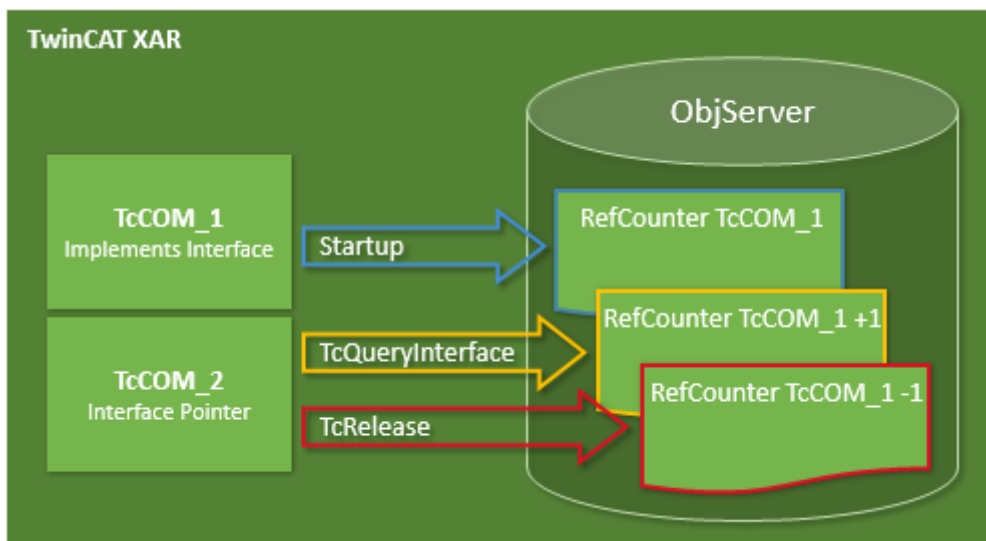
## Class Factories

"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 development 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.



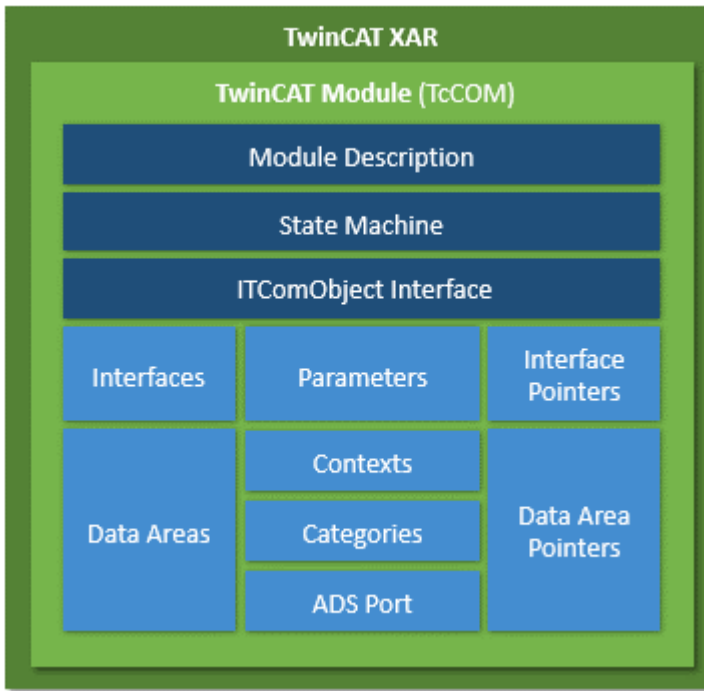
### 6.1.1 TwinCAT module properties

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.

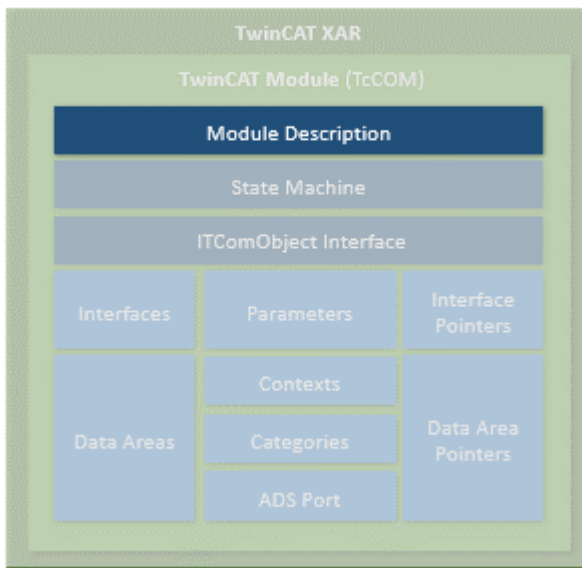
If a module is instantiated in the TwinCAT runtime, it registers itself with a central system instance, the ObjectServer. This makes it reachable and parameterizable for other modules and also for general tools. Modules can be compiled independently and can therefore also be developed, tested and updated 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.

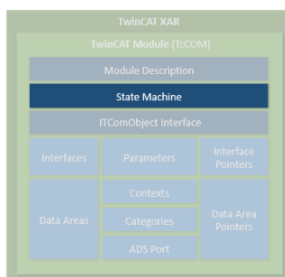
More complex configurations involving several modules can also be described in the class description files, which are preconfigured and linked for a specific application. Accordingly, a module for a complex machine unit, which internally consists of a number of submodules, can be defined and preconfigured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

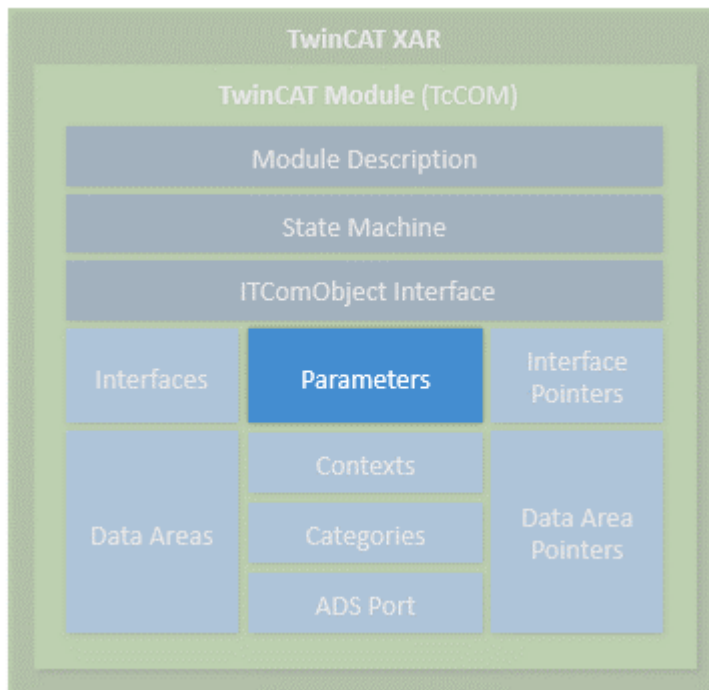


Each module contains a state machine, which describes the initialization state of the module and the means with which this state can be modified from outside. The state machine describes the states, which occur during starting and stopping of the module. This relates to module creation, parameterization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

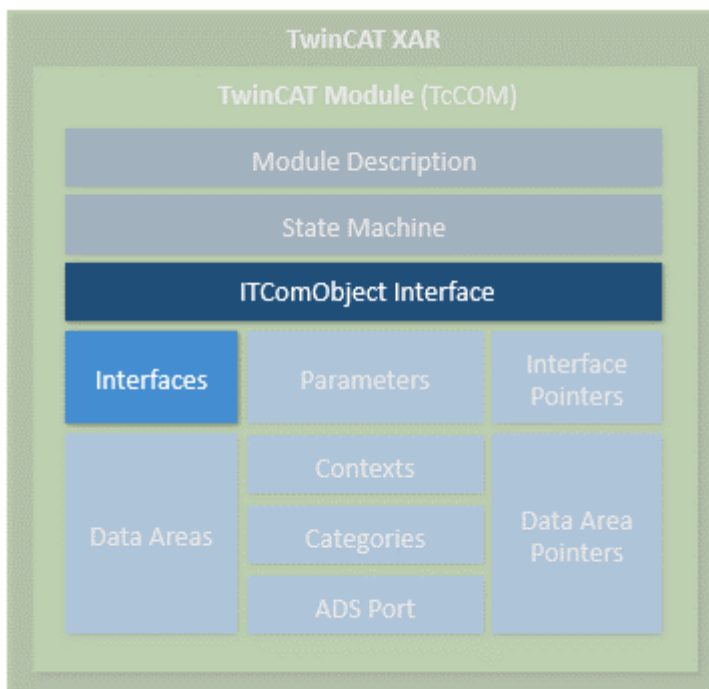
The state machine is [described \[► 39\]](#) in detail separately.

Parameter



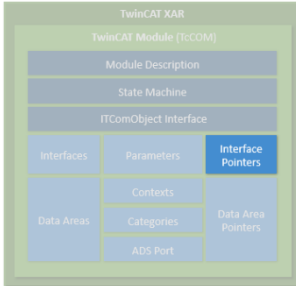
Modules can have parameters, which can be read or written during initialization or later at runtime (OP state). Each parameter is designated by a parameter ID. The uniqueness of the parameter ID can be global, limited global or module-specific. Further details can be found in the "ID Management" section. 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.

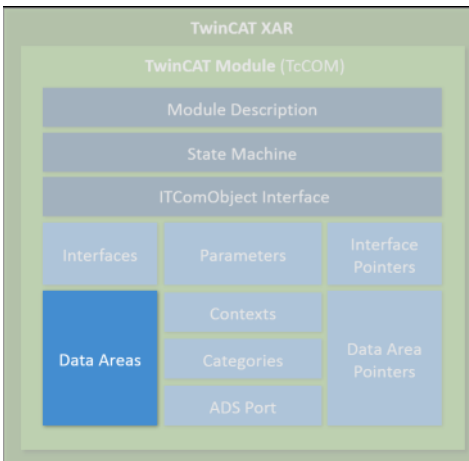
**Interface pointers**



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.

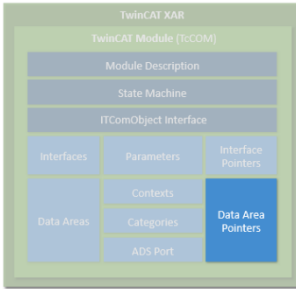
**Data areas**



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.

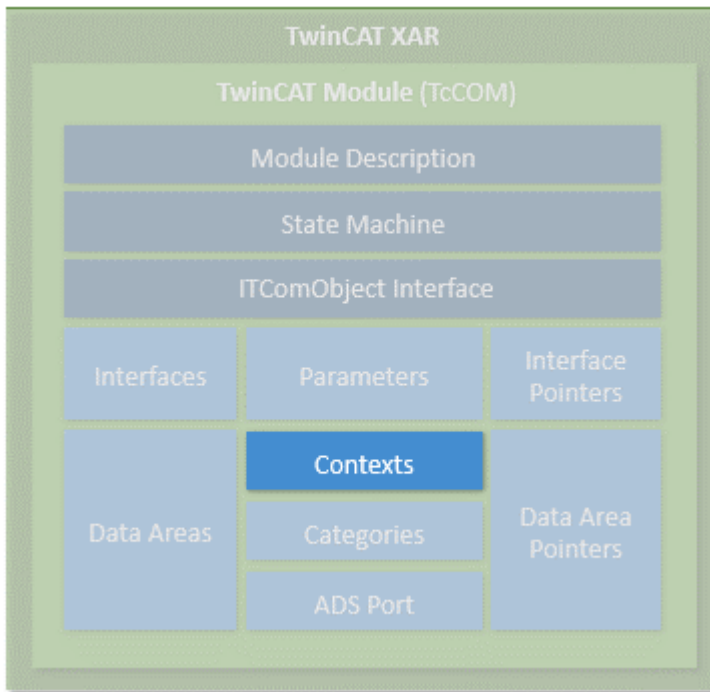


Data area pointer



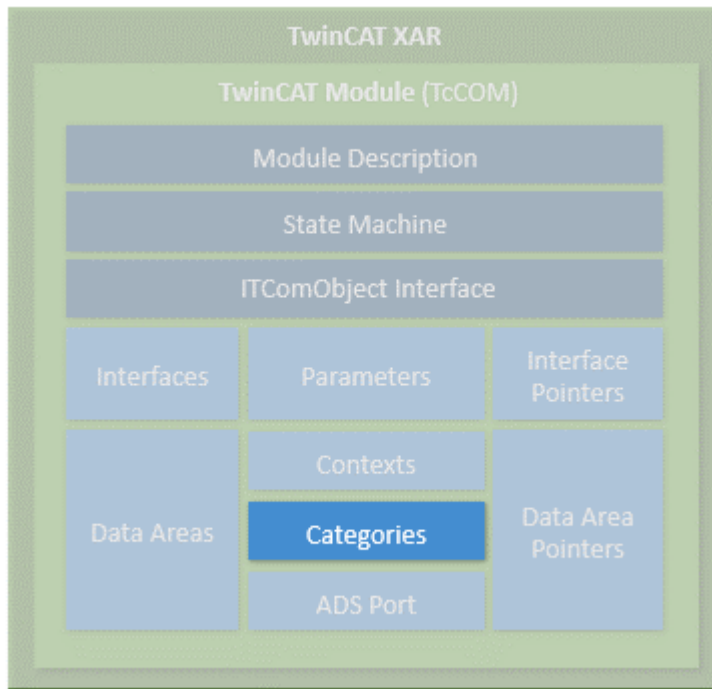
If a module wants to access the data area of other modules, it can contain data area pointers. These are normally set during initialization of the state machine to data areas or data area sections of other modules. The access is directly to the memory area, so that corresponding protection mechanisms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

Context



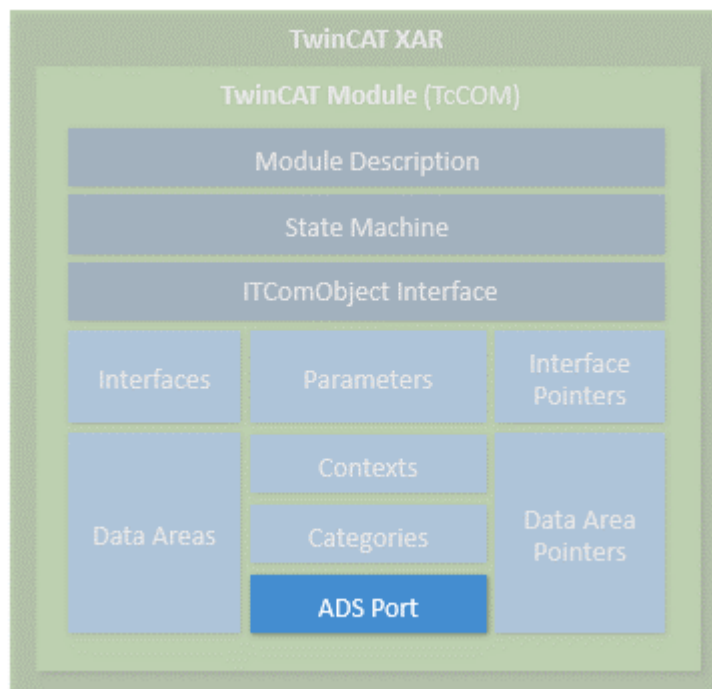
The context should be regarded as real-time task context. Context is required for the configuration of the modules, for example. Simple modules usually operate in a single time context, which therefore requires no detailed specification. Other modules may partly be active in several contexts (e.g. an 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 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++.

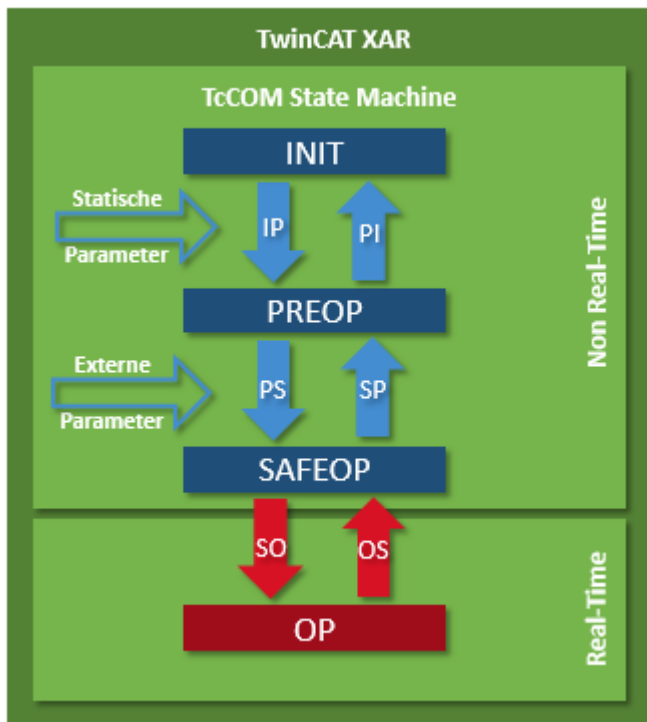
The PLC modules also register via the ObjectServer and can therefore be reached through it. PLC modules vary in terms of complexity. It makes no difference whether only a small filter module is generated or a complete PLC program is packed into a module. Due to the automation, each PLC program is a module 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,

resulting in the following state transitions: INIT to PREOP (IP), PREOP to SAFEOP (PS) and SAFEOP to OP (SO). In the opposite direction there are the following state transitions: 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 activated, 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.

**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

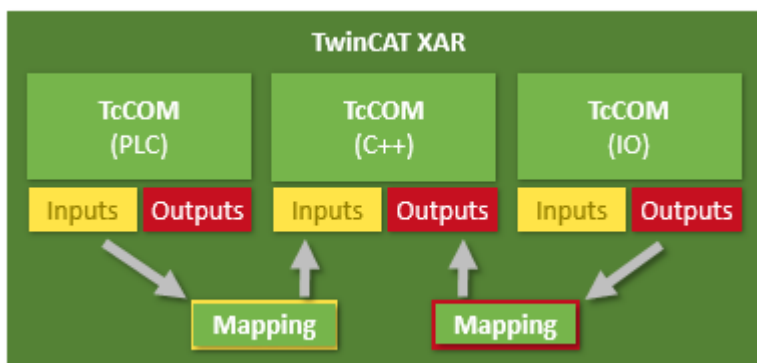
These four methods will now be described.

**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 \[► 112\]](#) that describe the corresponding inputs/outputs. These are then linked in the TwinCAT solution.

Through mapping, the data are provided or accepted at the task beginning (inputs) or task end (outputs) respectively. The data consistency is ensured by synchronous or asynchronous mapping.

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



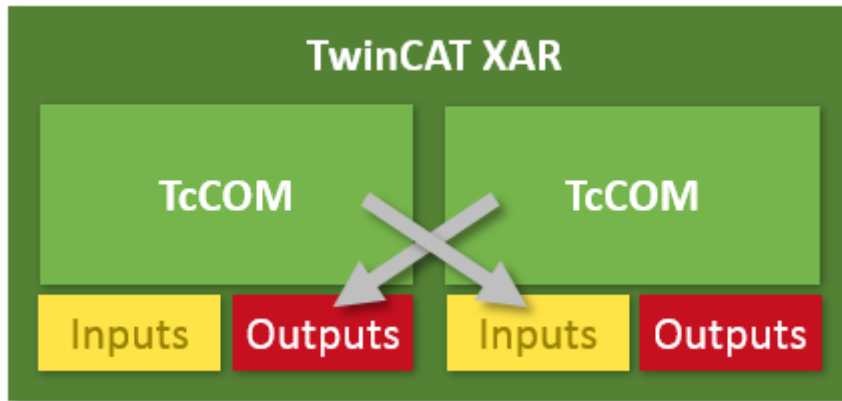
The following sample shows the realization:

[Sample12: Module communication: IO mapping used \[► 264\]](#)

### 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 realization:

[Sample10: Module communication: Use of data pointers \[► 235\]](#)

### 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 [► 103]), 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 realization:

[Sample11: Module communication: PLC module calls a method of a C-module \[► 236\]](#)

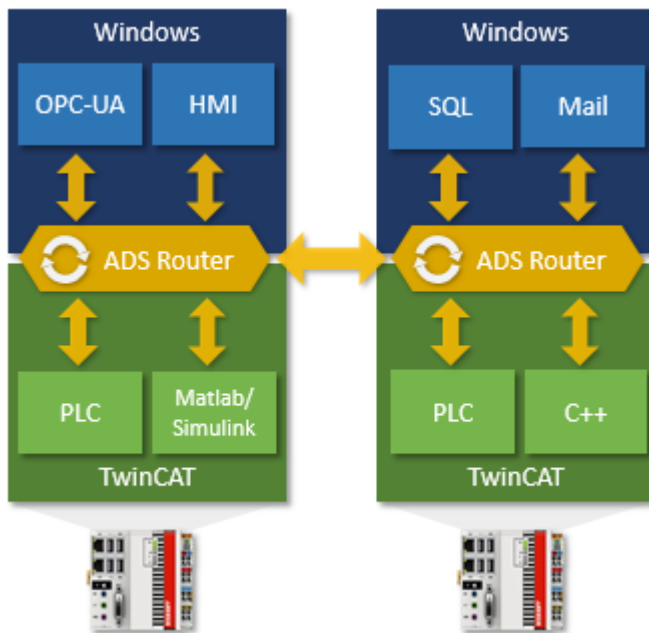
[Sample11a: Module communication: C-module cites a method in the C-module \[► 263\]](#)

[Further samples exist for the communication with the PLC. \[► 281\]](#)

### 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/CPU's. In this case TcCOM modules can be both clients (requesters) and servers (providers). The implementing language (PLC, C++, Matlab) is unimportant.



The following samples show the realization:

[Sample03: C++ as ADS server \[▶ 216\]](#)

[Sample06: UI-C#-ADS client uploads the symbols from the module \[▶ 226\]](#)

[Sample07: reception of ADS notifications \[▶ 231\]](#)

[Sample08: provision of ADS-RPC \[▶ 232\]](#)

## 7 Modules - Handling

TcCOM Modules are defined and implemented. Afterwards they could be

- Exchanged: [Export modules \[▶ 44\]](#), [Import modules \[▶ 45\]](#)
- Started

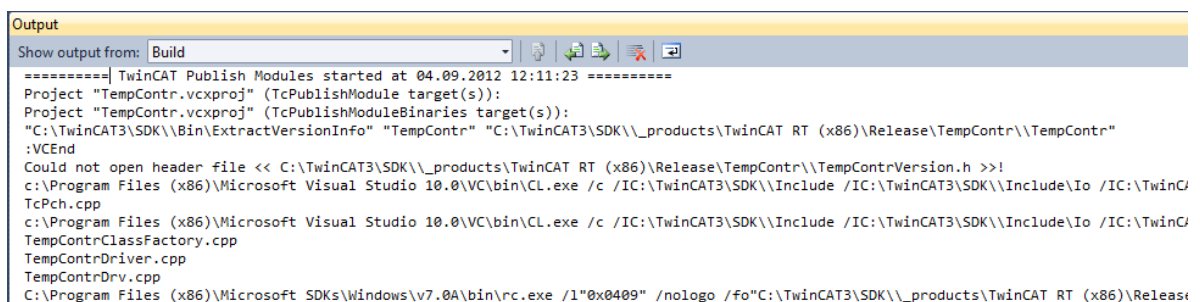
So this section describes handling of the Modules.

### 7.1 Export modules

This article covers how to export a TwinCAT 3 module, which could be used on any other TwinCAT machine.

The steps to be done are

1. Implementation of a TwinCAT 3 C++ project on one engineering PC equipped with Visual Studio version. Please refer to the [quick start sample \[▶ 50\]](#) Create a TwinCAT 3 project, implement the TwinCAT modules as described, compile and test the module before exporting.
2. Since the resulting module should be capable of being used on any machine, TwinCAT will generate 32bit as well as 64bit versions of the module. Since x64bit modules need to be signed, the machine exporting the module needs to have a certificate installed. Please see [x64: driver signing \[▶ 23\]](#) about how to generate and install a certificate. (Step 3 could be left out on an engineering or 32bit system)
3. To export a TC 3 C++ module just right click the module-project in the solution tree and select "TwinCAT Publish Modules"
  - ⇒ As a result the module will be compiled (Rebuild) - the successfully export is shown in the output view "Build"



```

Output
Show output from: Build
===== TwinCAT Publish Modules started at 04.09.2012 12:11:23 =====
Project "TempContr.vcxproj" (TcPublishModule target(s)):
Project "TempContr.vcxproj" (TcPublishModuleBinaries target(s)):
"C:\TwinCAT3\SDK\Bin\ExtractVersionInfo" "TempContr" "C:\TwinCAT3\SDK\products\TwinCAT RT (x86)\Release\TempContr\TempContr"
:VCEnd
Could not open header file << C:\TwinCAT3\SDK\products\TwinCAT RT (x86)\Release\TempContr\TempContrVersion.h >>!
c:\Program Files (x86)\Microsoft Visual Studio 10.0\VC\bin\CL.exe /c /IC:\TwinCAT3\SDK\Include /IC:\TwinCAT3\SDK\Include\Io /IC:\TwinC
TcPch.cpp
c:\Program Files (x86)\Microsoft Visual Studio 10.0\VC\bin\CL.exe /c /IC:\TwinCAT3\SDK\Include /IC:\TwinCAT3\SDK\Include\Io /IC:\TwinC
TempContrClassFactory.cpp
TempContrDriver.cpp
TempContrDrv.cpp
C:\Program Files (x86)\Microsoft SDKs\Windows\v7.0A\bin\rc.exe /I"0x0409" /nologo /fo"C:\TwinCAT3\SDK\products\TwinCAT RT (x86)\Release
  
```

Most important is the success report at the end:

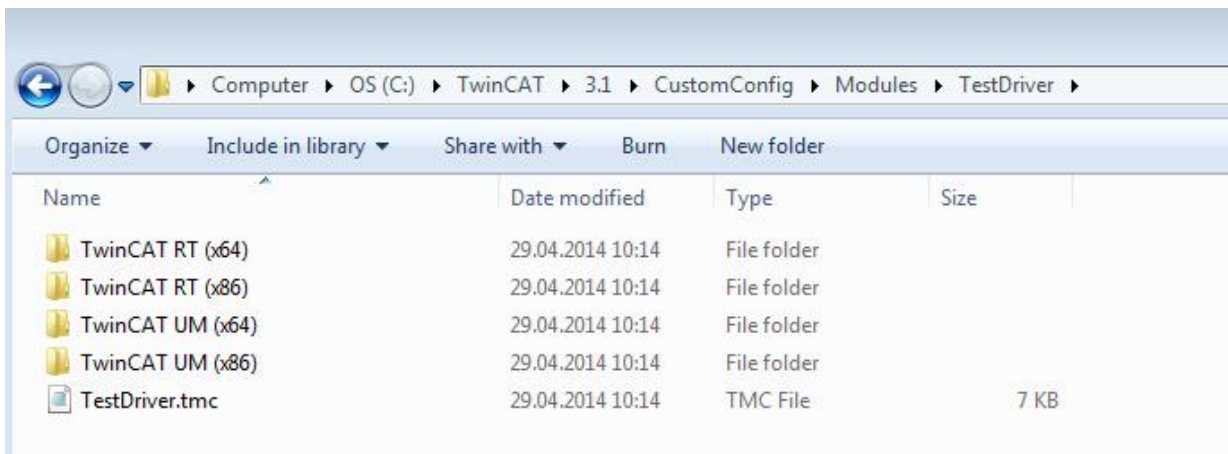


```

Output
Show output from: Build
TcPch.cpp
c:\Program Files (x86)\Microsoft Visual Studio 10.0\VC\bin\CL.exe /c /IC:\TwinCAT3\SDK\Includ
TempContrClassFactory.cpp
TempContrCtrl.cpp
TempContrDrv.cpp
TempContrW32.cpp
C:\Program Files (x86)\Microsoft SDKs\Windows\v7.0A\bin\rc.exe /D _UNICODE /D UNICODE /I"0x040
c:\Program Files (x86)\Microsoft Visual Studio 10.0\VC\bin\link.exe /ERRORREPORT:QUEUE /OUT:"C
"C:\TwinCAT3\SDK\_products\TwinCAT UM (x86)\Release\TempContr\TcPch.obj"
"C:\TwinCAT3\SDK\_products\TwinCAT UM (x86)\Release\TempContr\TempContrClassFactory.obj"
"C:\TwinCAT3\SDK\_products\TwinCAT UM (x86)\Release\TempContr\TempContrCtrl.obj"
"C:\TwinCAT3\SDK\_products\TwinCAT UM (x86)\Release\TempContr\TempContrDrv.obj"
"C:\TwinCAT3\SDK\_products\TwinCAT UM (x86)\Release\TempContr\TempContrW32.obj"
  Creating library C:\TwinCAT3\SDK\_products\TwinCAT UM (x86)\Release\TempContrW32.lib and ol
TempContr.vcxproj -> C:\TwinCAT3\SDK\_products\TwinCAT UM (x86)\Release\TempContrW32.dll
C:\Program Files (x86)\Microsoft SDKs\Windows\v7.0A\bin\mt.exe /nologo /verbose /out:"C:\TwinC
Done building project "TempContr.vcxproj".
Project "TempContr.vcxproj" (TcPublishAdditionalFiles target(s)):
Done building project "TempContr.vcxproj".
Done building project "TempContr.vcxproj".
===== TwinCAT Publish Modules finished at 04.09.2012 12:11:29 =====
    
```

The binary files and the TMC module description are exported into folder "TempContr" under "C:\TwinCAT3.x\CustomConfig\Modules"

- Just copy folder "TempContr" to any other TwinCAT 3 machine for import.



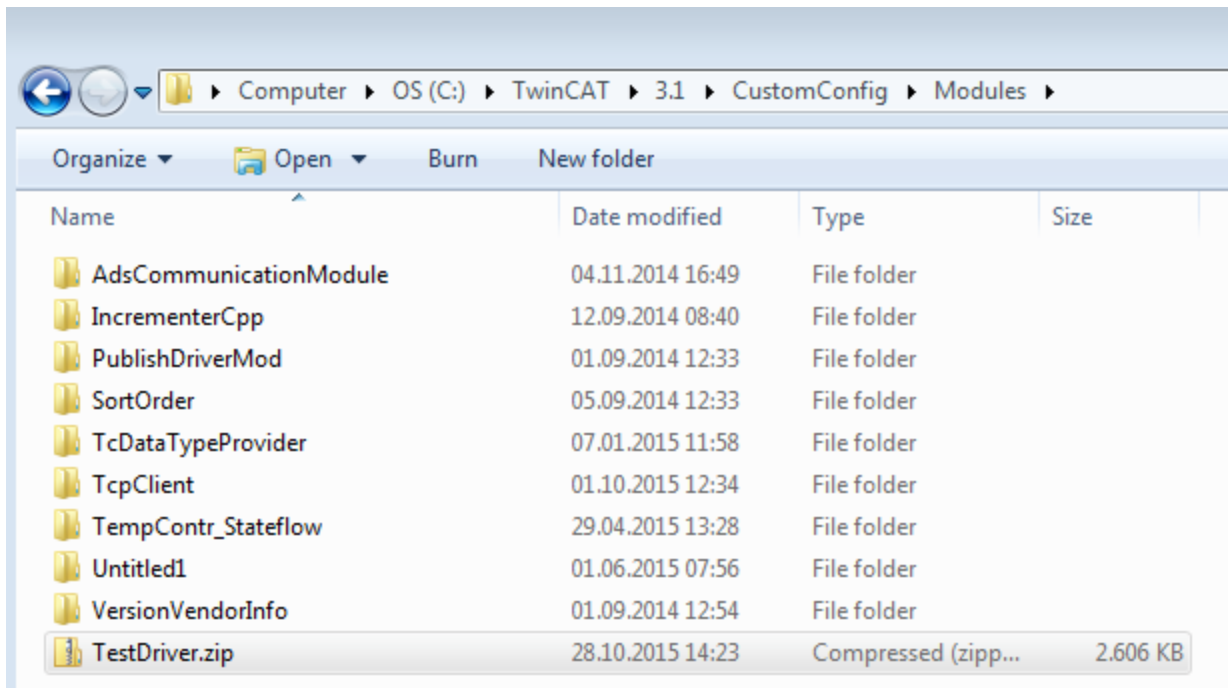
## 7.2 Import modules

This article describes how a binary TC3 module can be imported and integrated into a "PC/IPC" controller with TwinCAT 3 XAE (without the full version of Visual Studio).

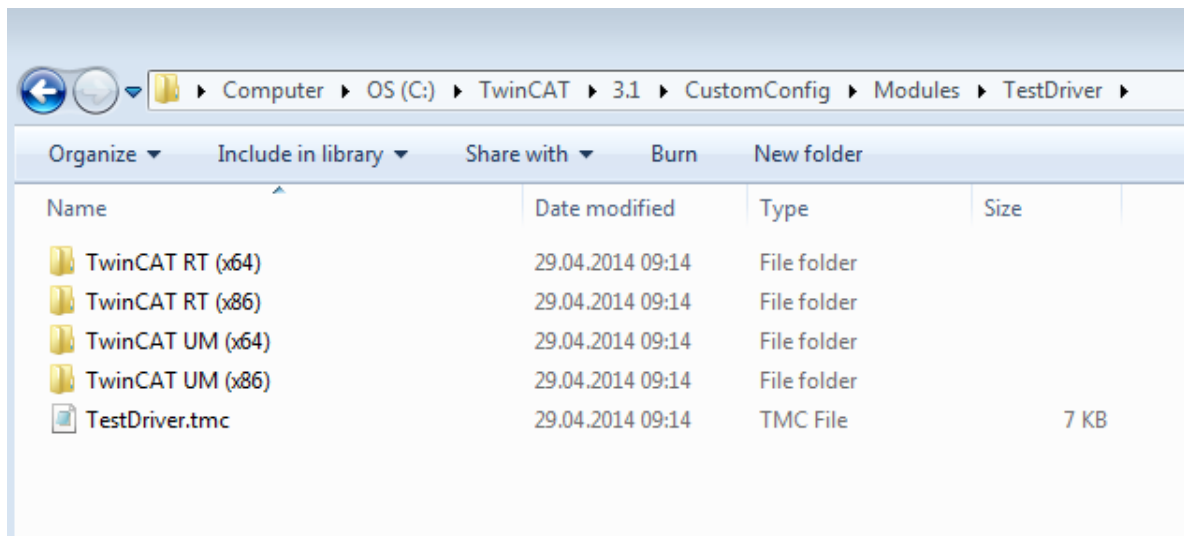
The binary TC3 module was implemented on another PC and exported beforehand.

The following steps have to be carried out

1. Copy the binary module to the destination folder ".\TwinCAT\3.x\CustomConfig\Modules" on the second IPC with TwinCAT XAE without the full version of Visual Studio. The "TestDriver.zip" archive is unpacked in this example.

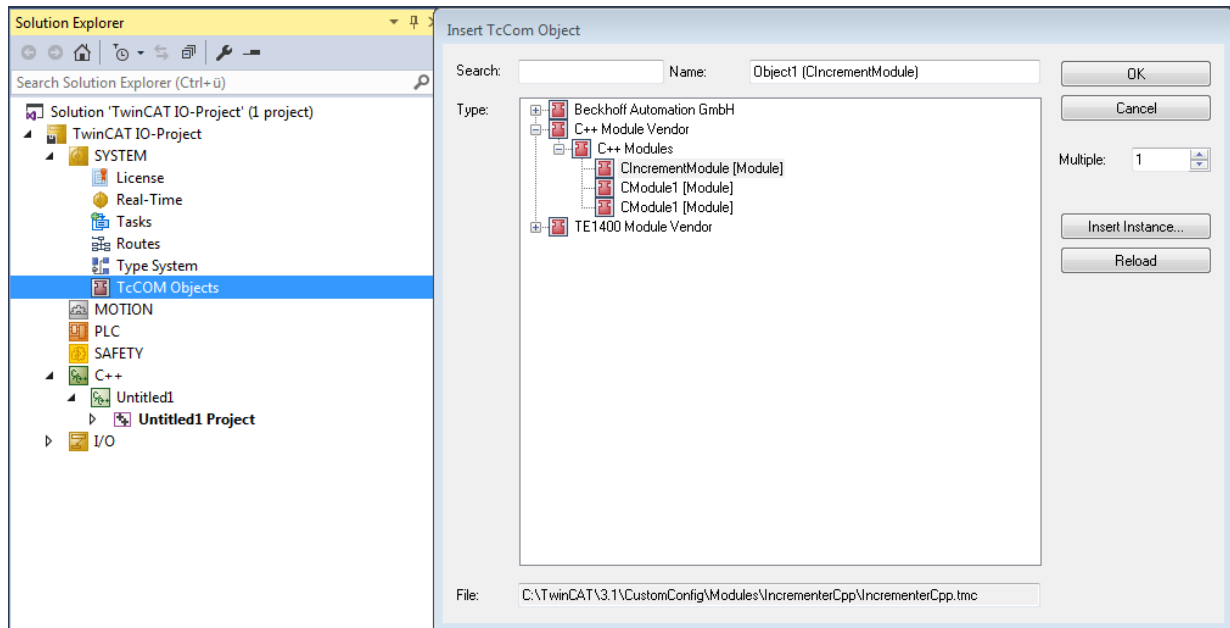


- ⇒ After that the "TestDriver" module provides the binary modules (in the RT and UM subfolders) and the corresponding TwinCAT Module Class \*.tmc file "TestDriver.tmc".



2. Start the TwinCAT XAE environment and create a TwinCAT 3 project.

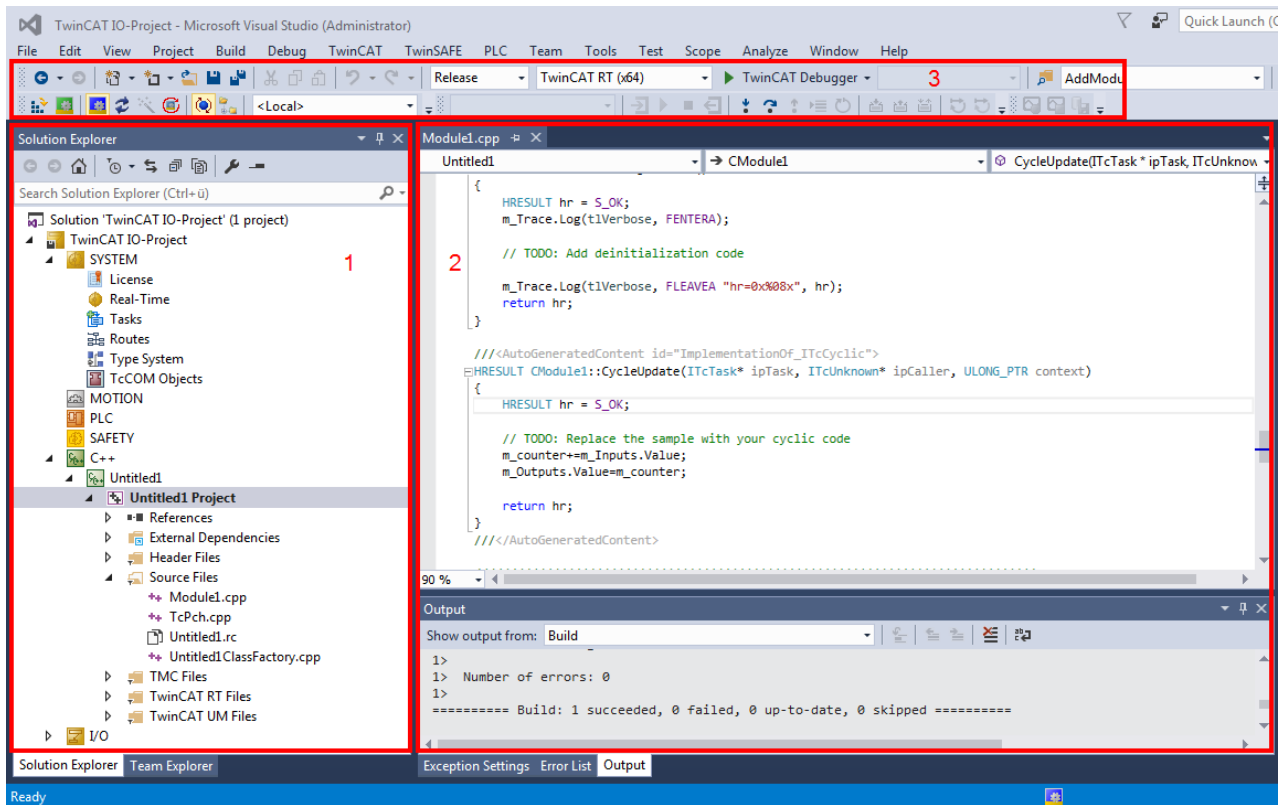
3. Right-click on "**System->TcCOM Objects**" and select "**Add New Item...**".



4. The new CTestModule module is listed in the dialog box that appears. Create a module instance by selecting the module name and continue with "**OK**".
5. The instance of the **TestModule** module now appears under "**TcCom Objects**".
6. The next steps as before: generate a new task and
7. go to "Context" of the module instance and link the C++ module instance with the previously added "**Task 1**".
8. Activate configuration

## 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 perhaps several modules [▶ 30], 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 [▶ 41] for module interaction.
2. The Visual Studio editor for Visual C++ is used for programming. Note in particular the drop-down boxes for fast navigation within a file. In the lower section the result of the compile process is output. The user can switch to TwinCAT messages (cf. [Module messages for the Engineering \(logging / tracing\)](#) [▶ 193]). The usual features such as breakpoints (cf. [Debugging](#) [▶ 68]) 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 (here "<Local>") and some other buttons offer quick access to frequently used functions. "TwinCAT Debugger" is the button for establishing the connection with the target system with regard to C++ modules (the PLC uses a separate 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](#) [▶ 50] guide describes the individual common steps.

1. Type declaration and module type:

The [TwinCAT Module Class Editor \(TMC\)](#) [▶ 79] and TMC code generator is 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).

The code generator can be started over and over again: The code generation takes note of and preserves the programmed user code.

## 2. Programming

The familiar Visual Studio C++ programming environment is used for the development and [debugging \[► 68\]](#) of the user-defined code within the code template.

## 3. Instantiating [modules \[► 30\]](#)

The program describes a class, which is instantiated as objects. The [TwinCAT Module Instance Configurator \[► 124\]](#) is used for configuring the instance. General configuration elements are: allocate 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 (see [Export modules \[► 44\]](#) / [Import modules \[► 45\]](#))

During publishing of a module, the drivers for all platforms are created, and the module is prepared for distribution. The created directory can be distributed without the need to transfer the source code. Only binary code with the interface description is transferred.

## 7. Signature (see [x64: driver signing \[► 23\]](#))

The TwinCAT drivers must be signed for x64 run times, since 64-bit Windows versions require that kernel modules are signed. This therefore applies to the x64 development and to publishing of modules, since these modules contain the x64 binary code (if not disabled, as described [here \[► 197\]](#)).

The signature process can be user-defined, as described [here \[► 28\]](#).

## 8. Activation

The TwinCAT C++ driver can be activated like any other TwinCAT project via "Activate Configuration". The dialog then requests to switch TwinCAT to RUN mode.

[Debugging \[► 68\]](#) in real-time (which is familiar from IEC61131-based systems) and the setting of (conditional) breakpoints is possible for TwinCAT C++ modules.

⇒ 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.

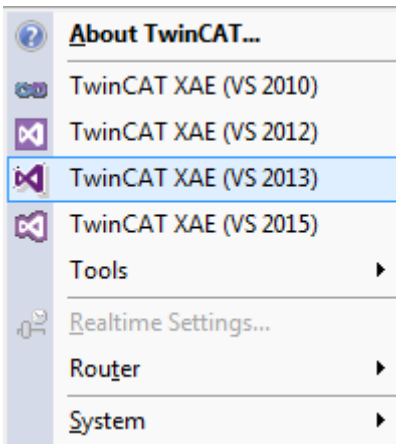
Each step in the creation of a module that runs in a real-time context is described in detail.

Before the quick start, please pay attention to the [Preparation - just once!](#) [▶ 20]

### 9.1 Create TwinCAT 3 project

#### Start the TwinCAT Engineering Environment (XAE)

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

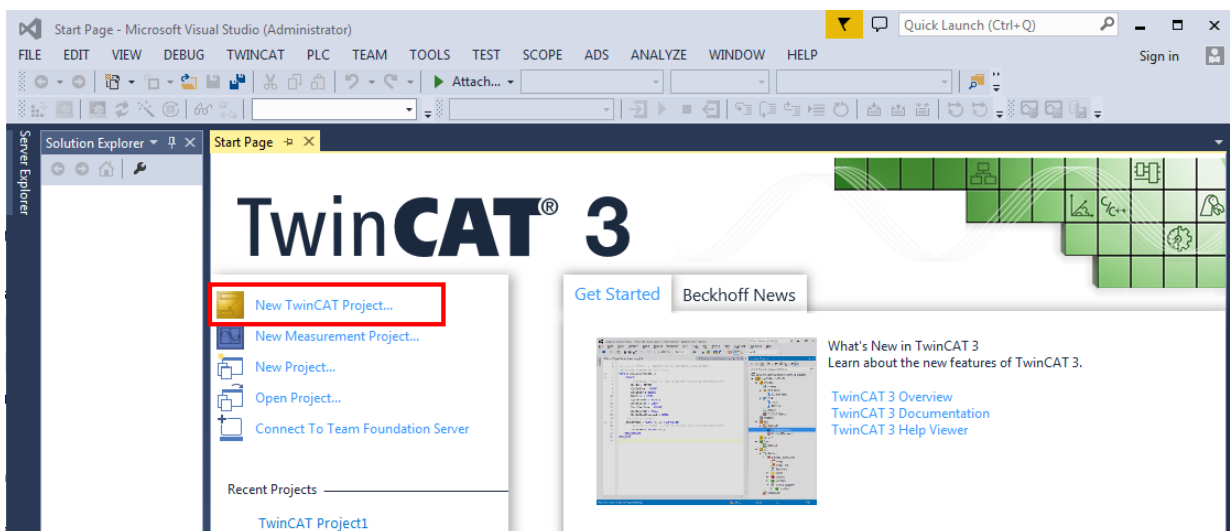


The Visual Studio versions recognized during the installation and supported by TwinCAT are thereby offered. Alternatively, Visual Studio can also be started via the Start menu.

#### TwinCAT 3 C++ - Create project

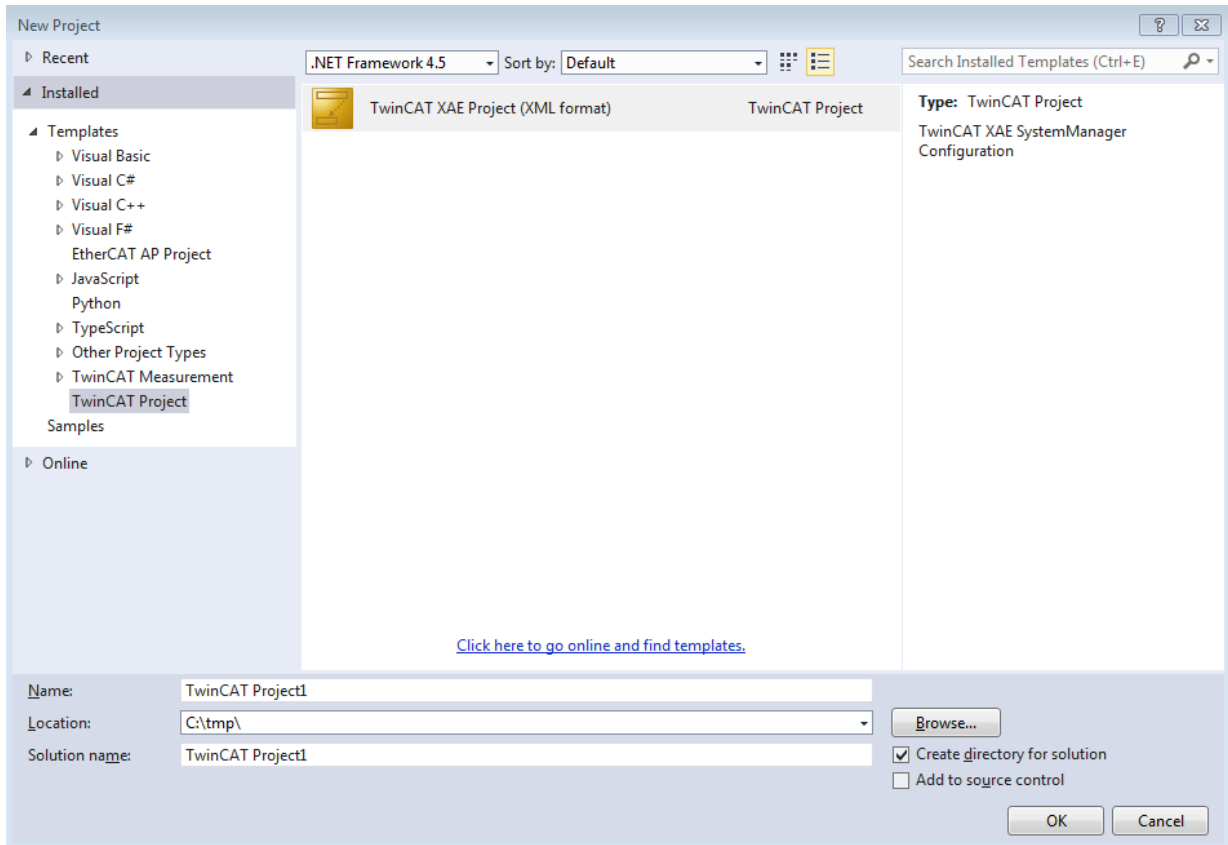
Carry out the following steps to create a TwinCAT C++ project:

1. Select "New TwinCAT Project ..." via the Start page.

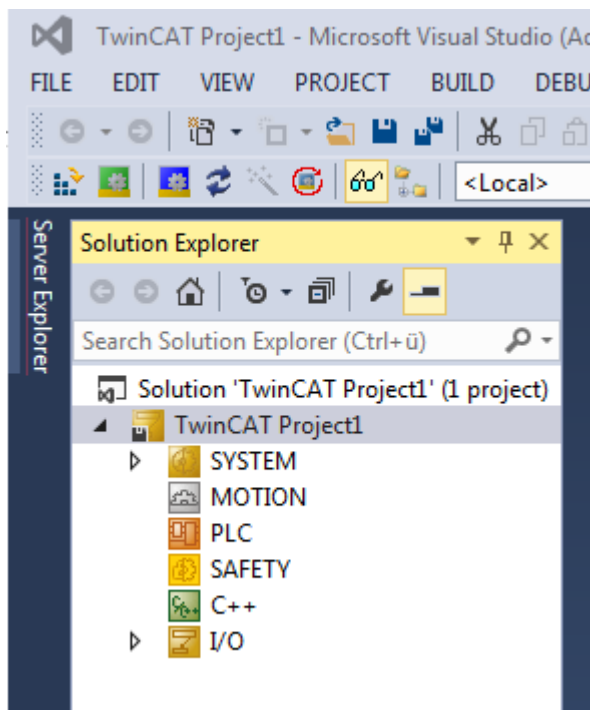


2. Alternatively, a new project can be created by clicking on File -> New -> Project.
  - ⇒ All existing project templates are displayed.
3. Select "TwinCAT XAE Project"; a suitable project name can optionally be entered.

- Click on "OK". From now on you cannot select or change the name of the directory. Retain the default settings (selected option "Create directory for solution").



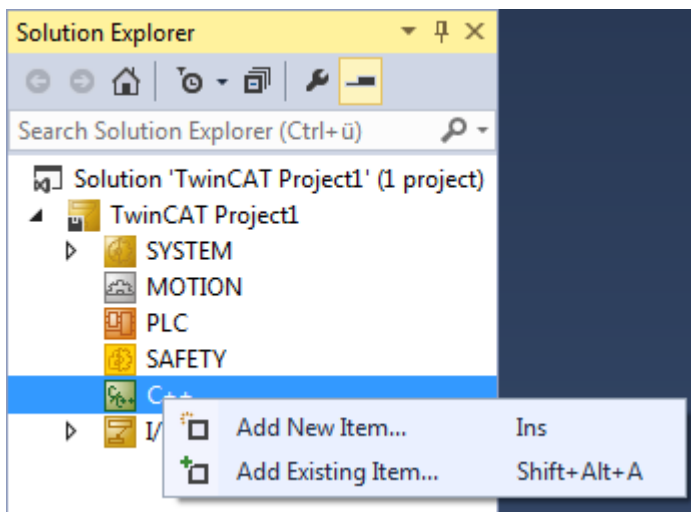
⇒ The Visual Studio Solution Explorer then displays the TwinCAT 3 project.



## 9.2 Create TwinCAT 3 C++ project

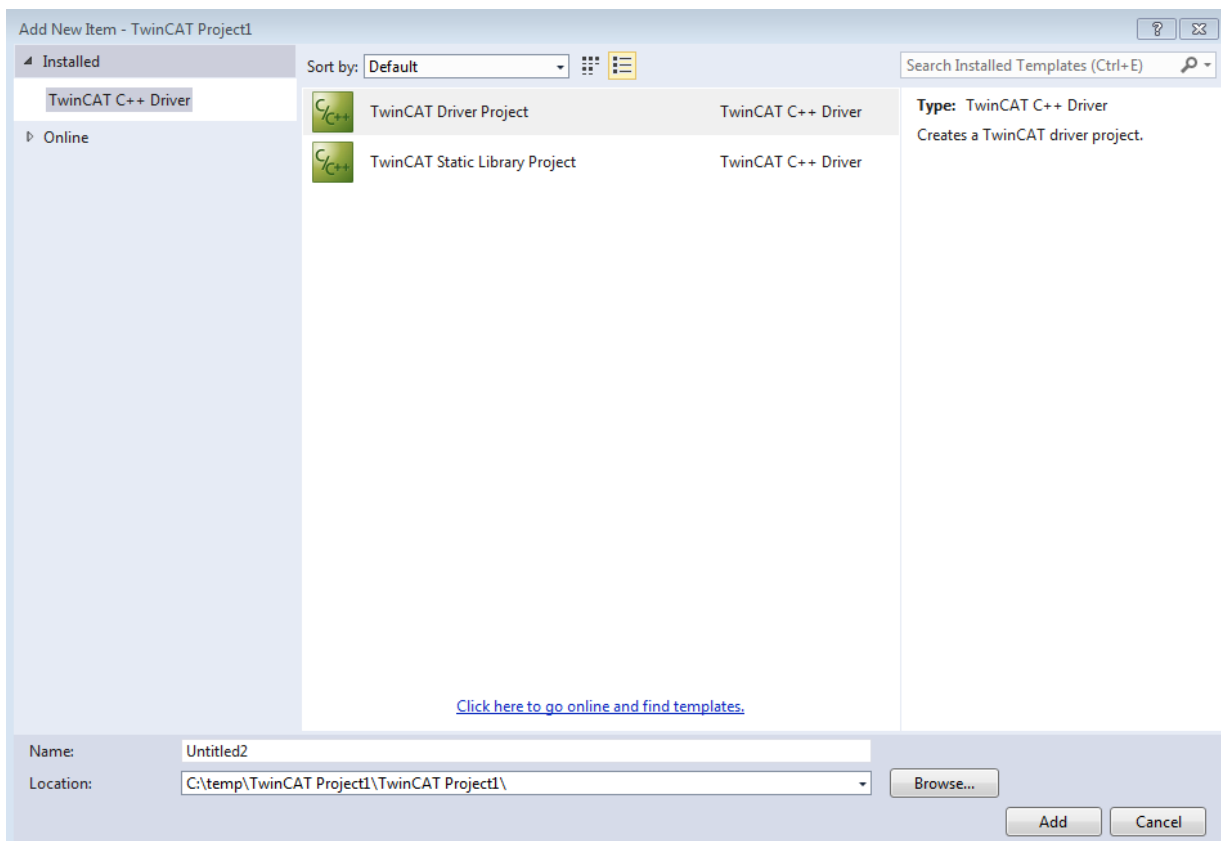
After creating a TwinCAT 3 project, open the "C++" node and carry out the following steps:

1. Right-click on "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. [Requirements](#) [▶ 18]).



⇒ The "[TwinCAT C++ Project Assistant](#) [▶ 76]" is shown and all existing project templates are listed.

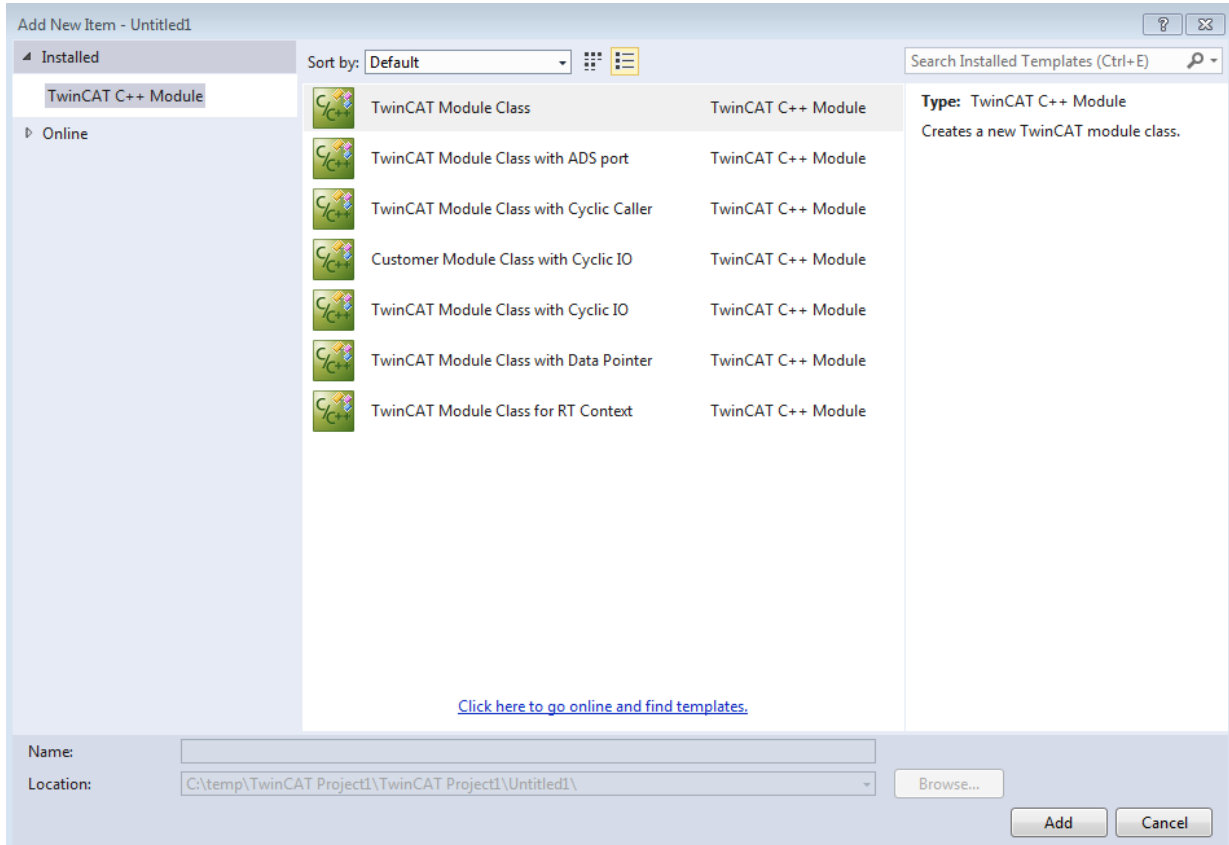
2. A) Select "TwinCAT Driver Project", optionally enter a related project name and click on "OK".  
 B) Alternatively, use the "TwinCAT Static Library Project", which provides an environment for the programming of static TC-C++ libraries (see [Example 25](#) [▶ 274])



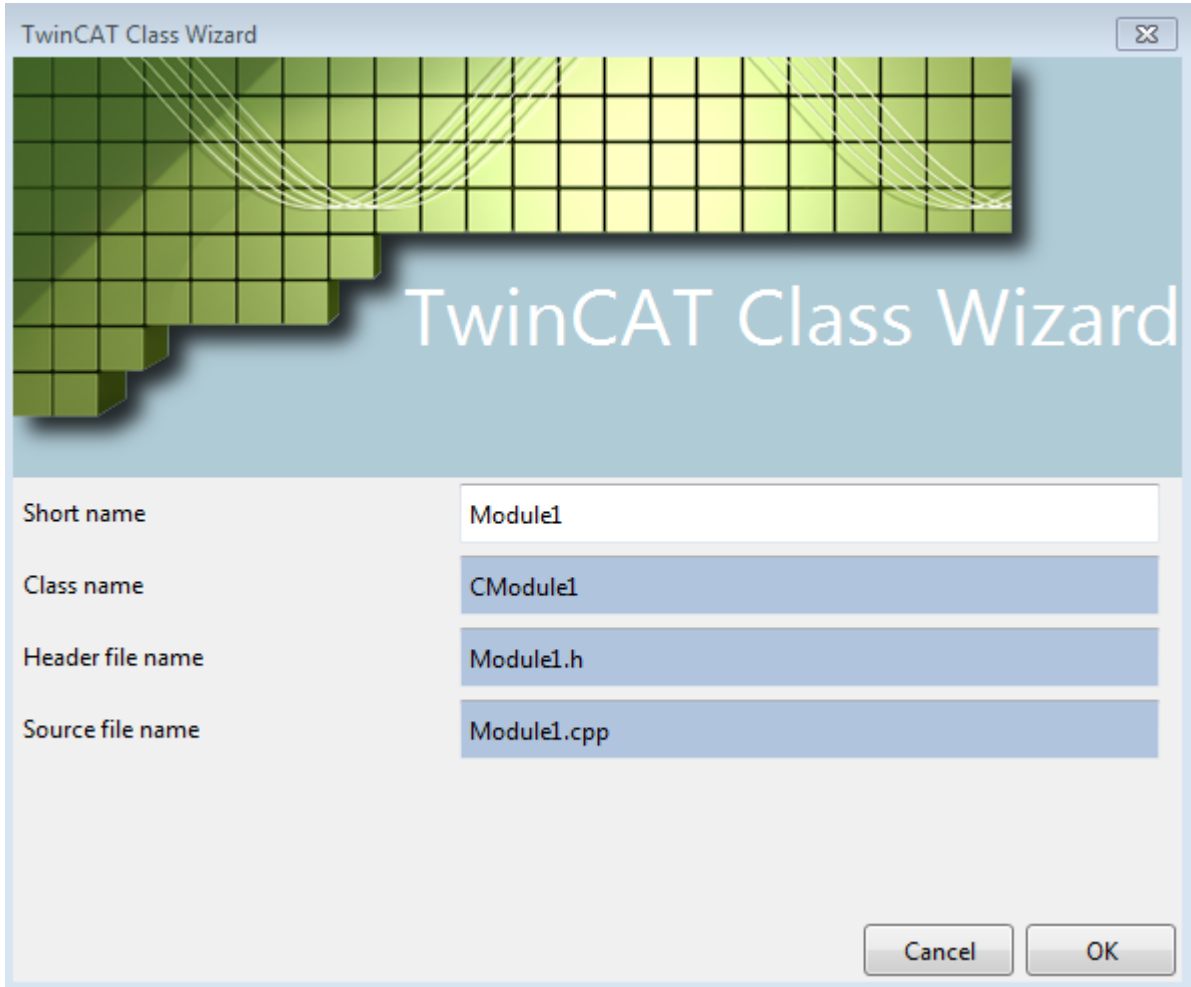
⇒ The "[TwinCAT Module Assistant](#) [▶ 77]" is displayed.



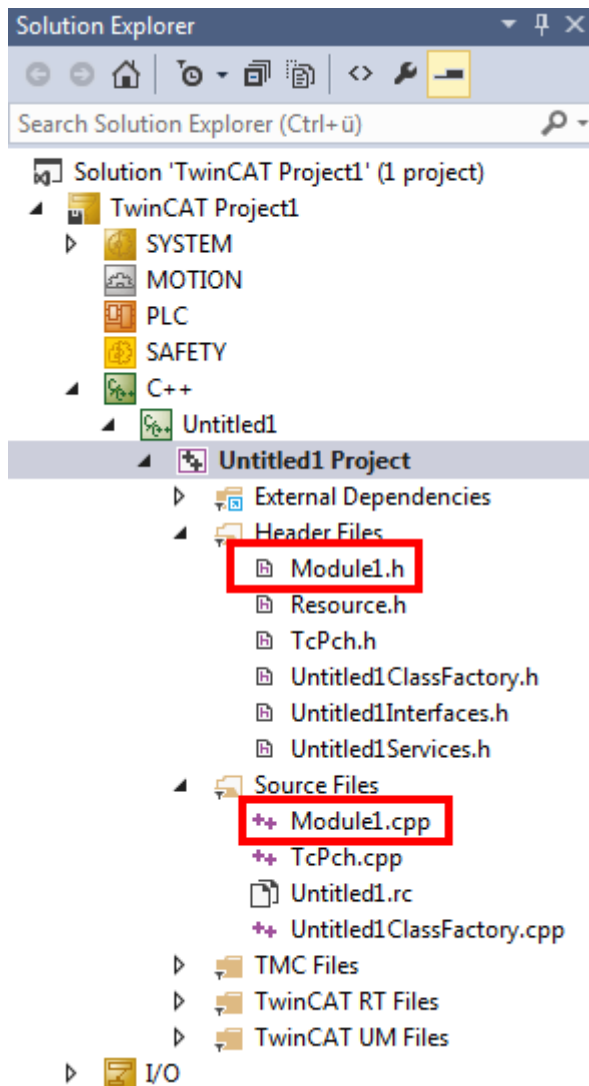
- In this case select "TwinCAT Module Class with Cyclic I/O" and click on "OK". A name is not necessary and also cannot be entered here.



4. Enter a unique name in the dialog window "TwinCAT Class Wizard" or continue with the suggestion "Object 1".



⇒ A TwinCAT 3 C++ project with a driver will then be created on the basis of the selected template:

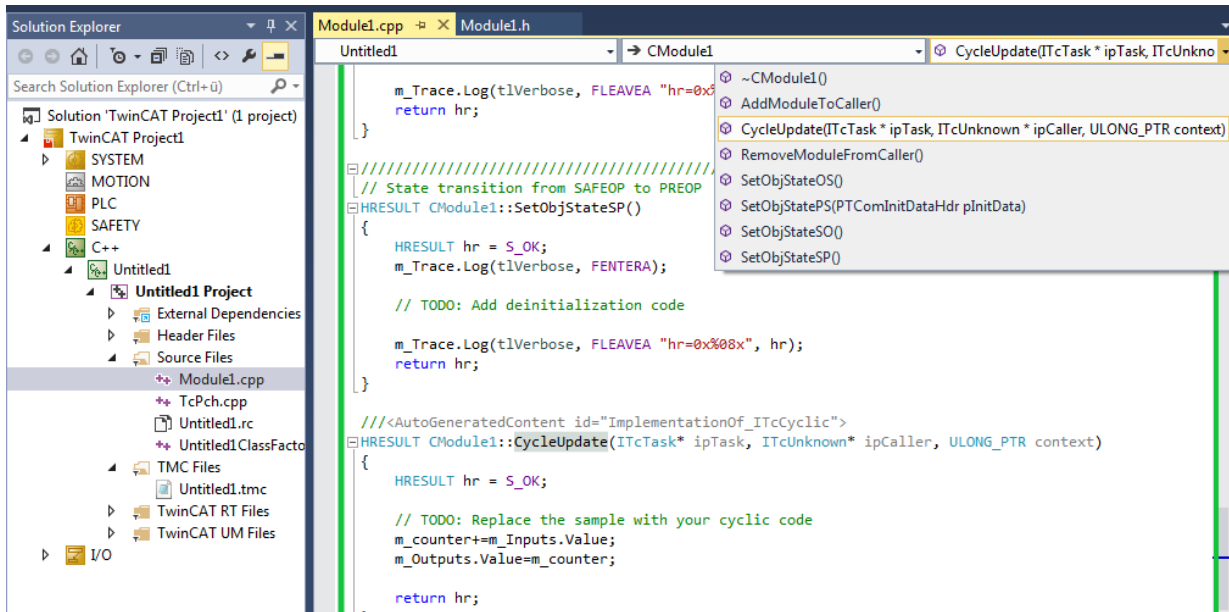


## 9.3 Implement TwinCAT 3 C++ project

This article describes how the example project can be changed.

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

1. The `<MyClass>::CycleUpdate()` method is cyclically called – this is the point where the cyclic logic is to be positioned. The entire cyclic code is inserted at this point. The drop-down menus at the top edge of the editor can be used for navigation as shown in the screenshot

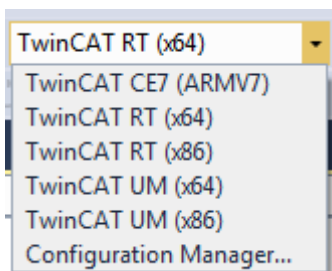


2. In this case a counter is incremented by the value of the "Value" variable in the input image (`m_Inputs`). Replace a line in order to increment the counter without dependence on the value of the input image. Replace this line:  
`m_counter+=m_Inputs.Value;`  
 by this one:  
`m_counter++;`
3. Save the modifications.

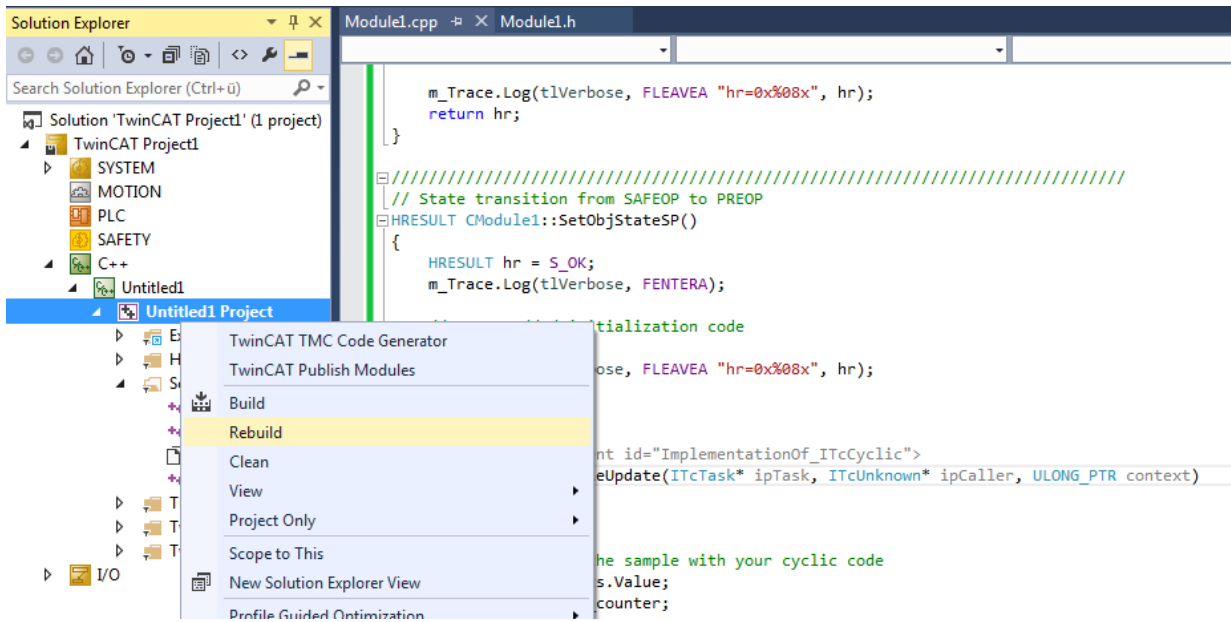
## 9.4 Compiling/building a TwinCAT 3 C++ project

This article describes how an already implemented C++ module class is created (compiled).

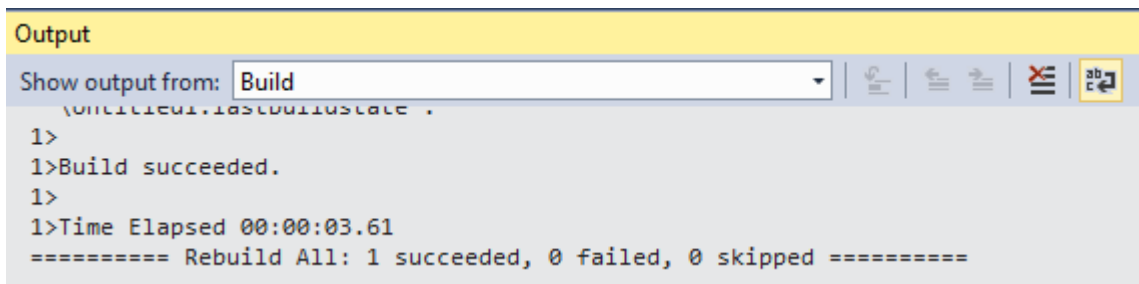
1. Select the target platform according to which the compilation should be carried out. TwinCAT checks this setting when selecting a target system and changes it if necessary after a prompt. The project is also deactivated if an unsupported target platform is selected.



2. Right-click on the TwinCAT 3 C++ project and select "Build" or "Rebuild"



⇒ The compiler output window must look like this if the code has been correctly written (i.e. no syntax errors):



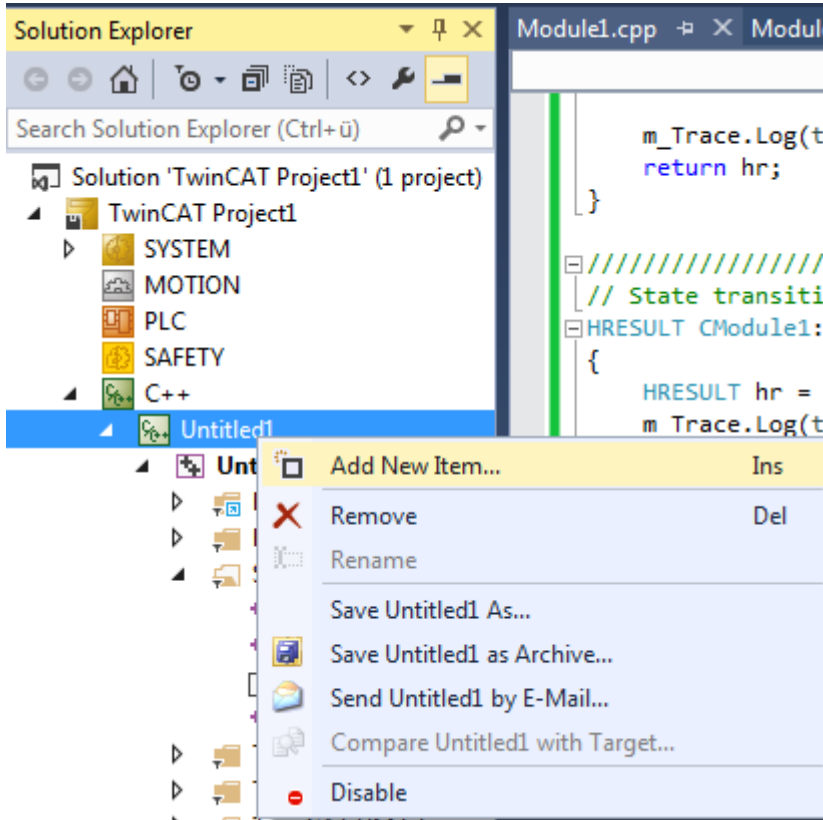
⇒ Following successful compilation/creation, the new TwinCAT C++ module is provided for the specific target platform in the "\_Deployment" subfolder of the project directory.

## 9.5 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.

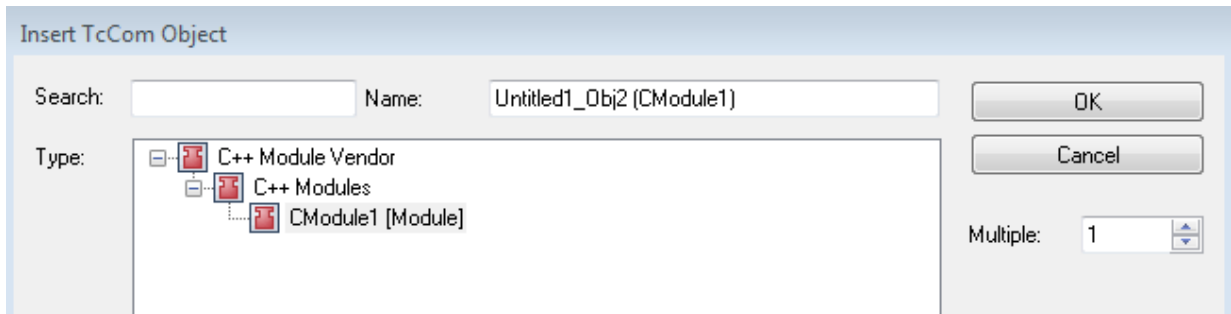
**After creating a TwinCAT C++ module, open the "C++ - Configuration" node and follow these steps to create an instance.**

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. The default name can be used or alternatively a new instance name can be entered and confirmed with "OK" (in this example the default name was selected).



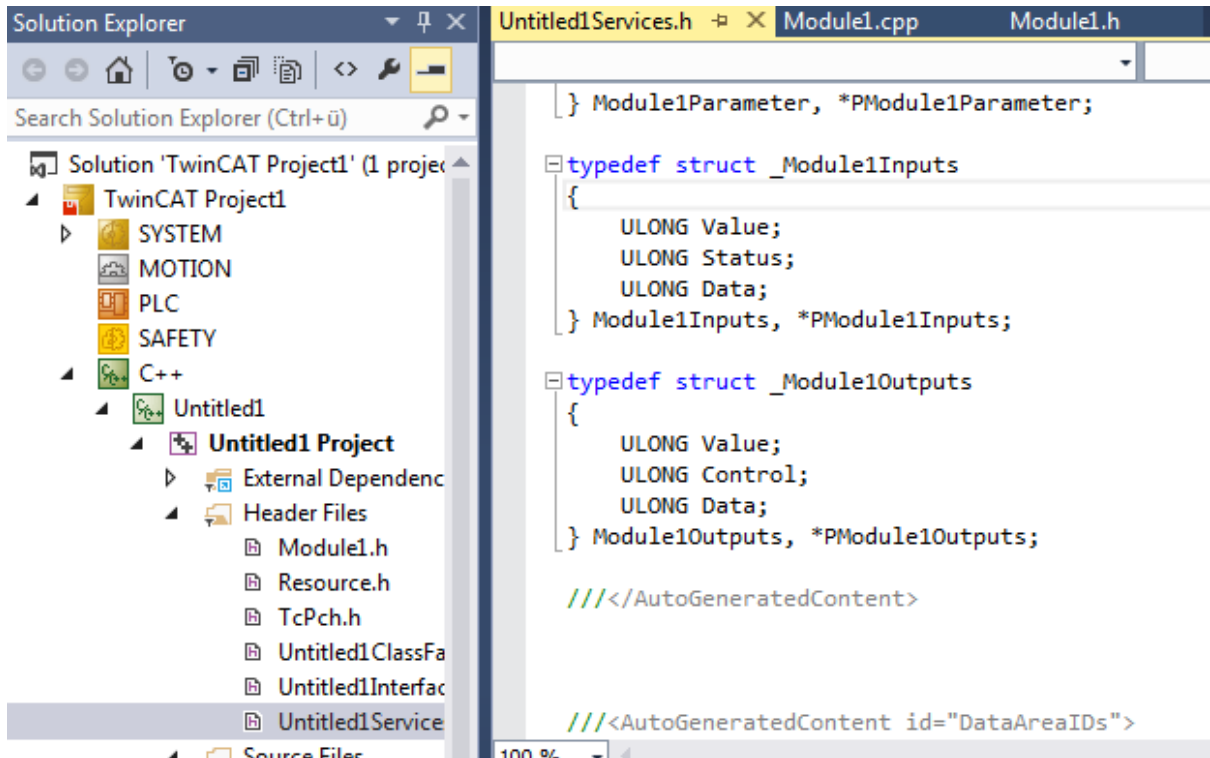
⇒ The new instance "Untitled1\_Obj2 (CModule1)" becomes part of the TwinCAT 3 solution: the new node is located precisely under the TwinCAT 3 C++ source "Untitled1 Project".

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

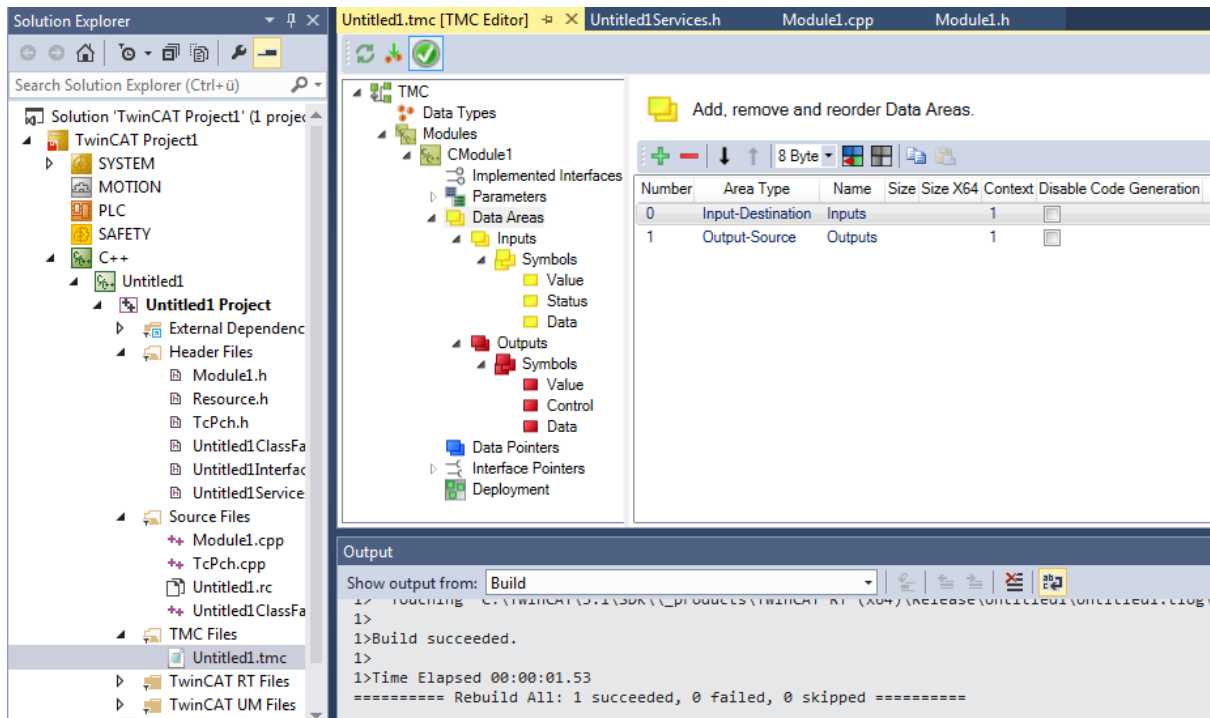
- Input area: Value, Status, Data
- Output area: Value, Control, Data

The description of these interfaces corresponds in two places:

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



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



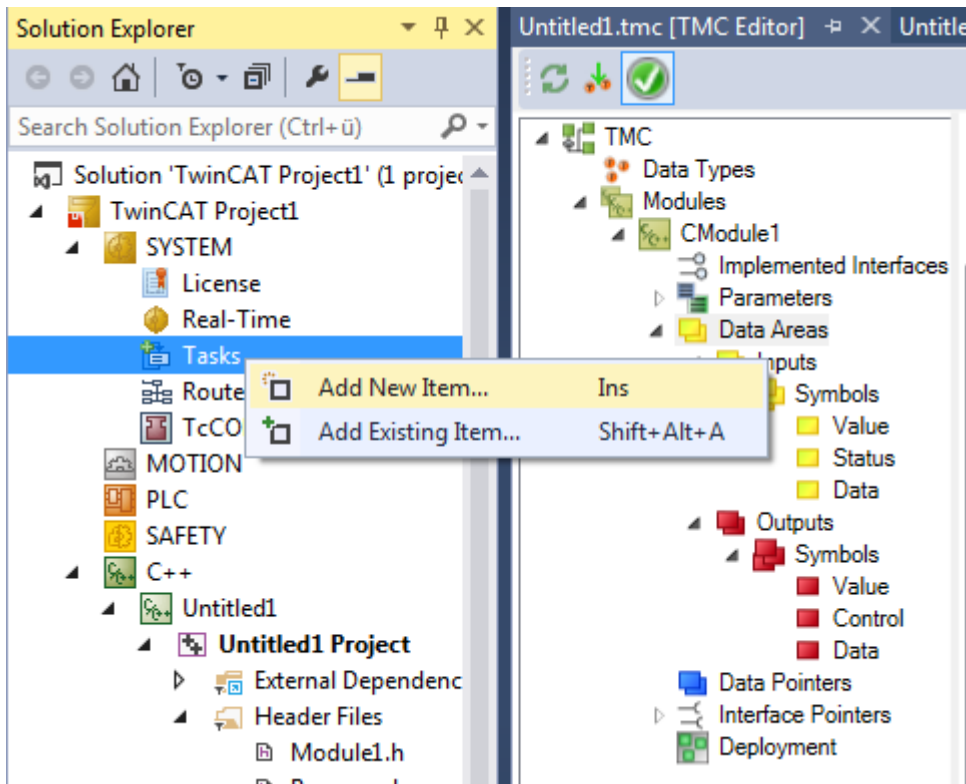
## 9.6 Create a TwinCAT task and apply it to the module instance

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.

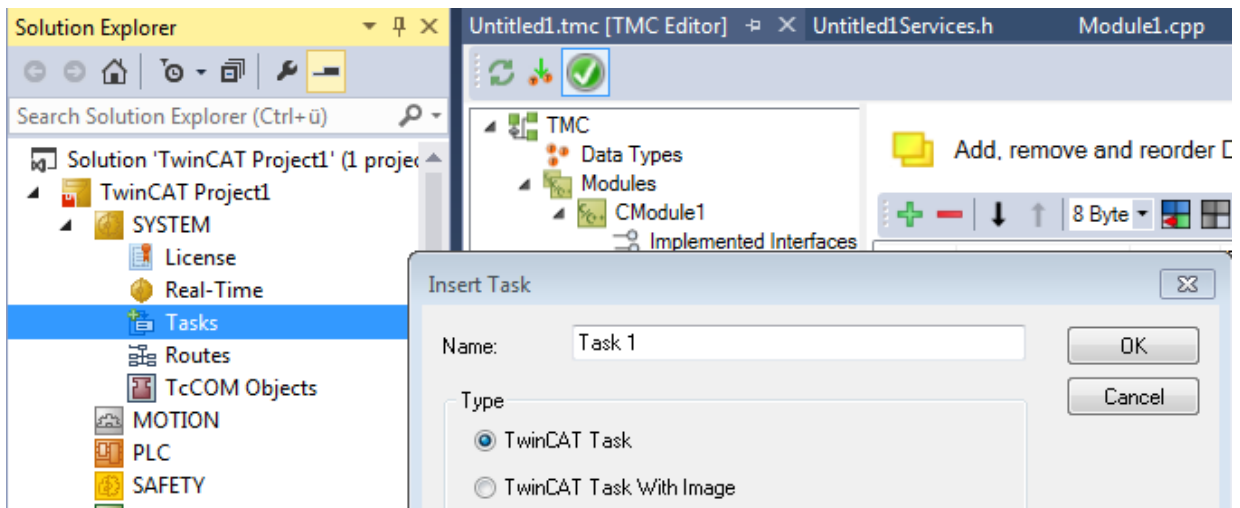
This configuration step only has to be carried out once. No new task needs to be configured for subsequent creations/new compilations of the C++ module in the same project.

## Creating a TwinCAT 3 task

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



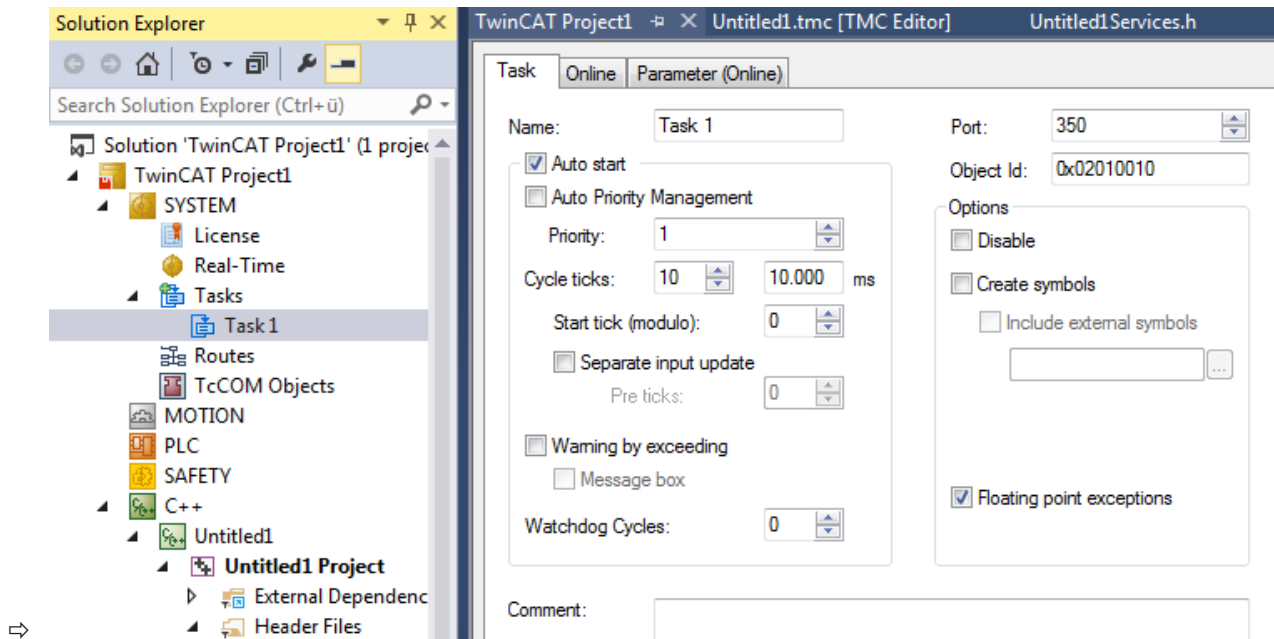
2. Enter a unique name for the task (or retain the default name).  
In this example the I/O image interface is provided by a C++ module instance, so that no image is necessary at the task for triggering the execution of the C++ module instance.



⇒ The new task with the name "Task 1" is created.

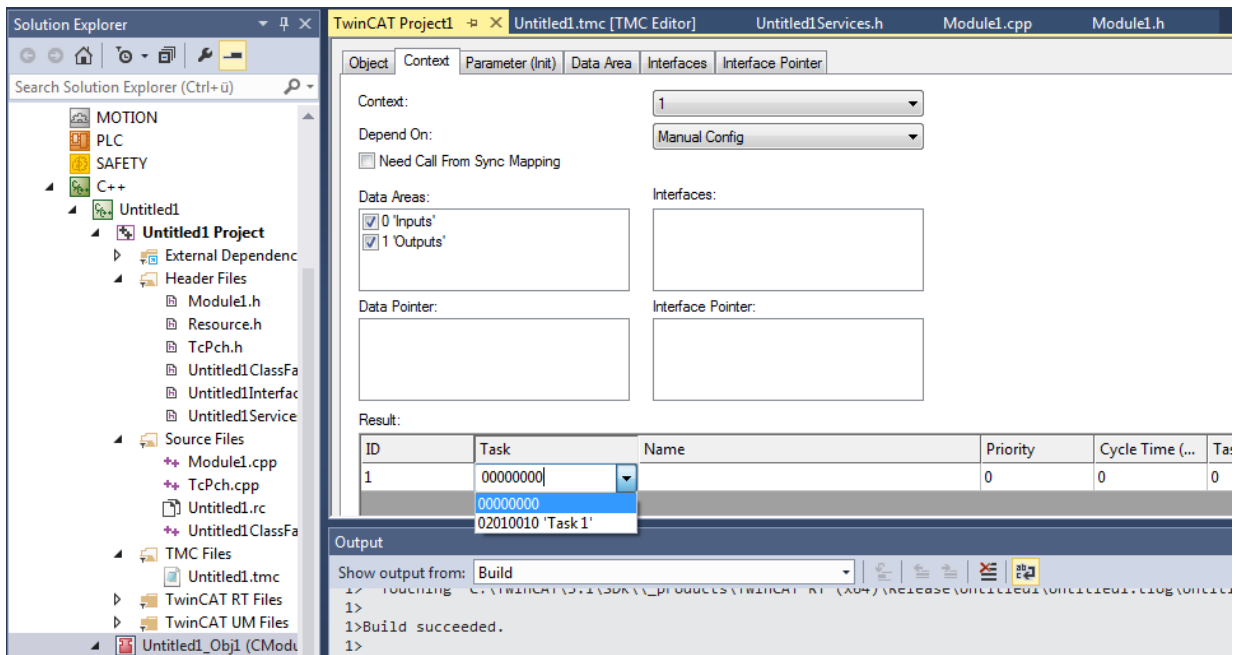
3. The task can now be configured; double-click on the task to do this.  
The most important parameters are "Auto start" and "Priority":  
"Auto start" must be activated in order to automatically start a task that is to be cyclically executed. The "Cycle ticks" define the timing of the clock in relation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

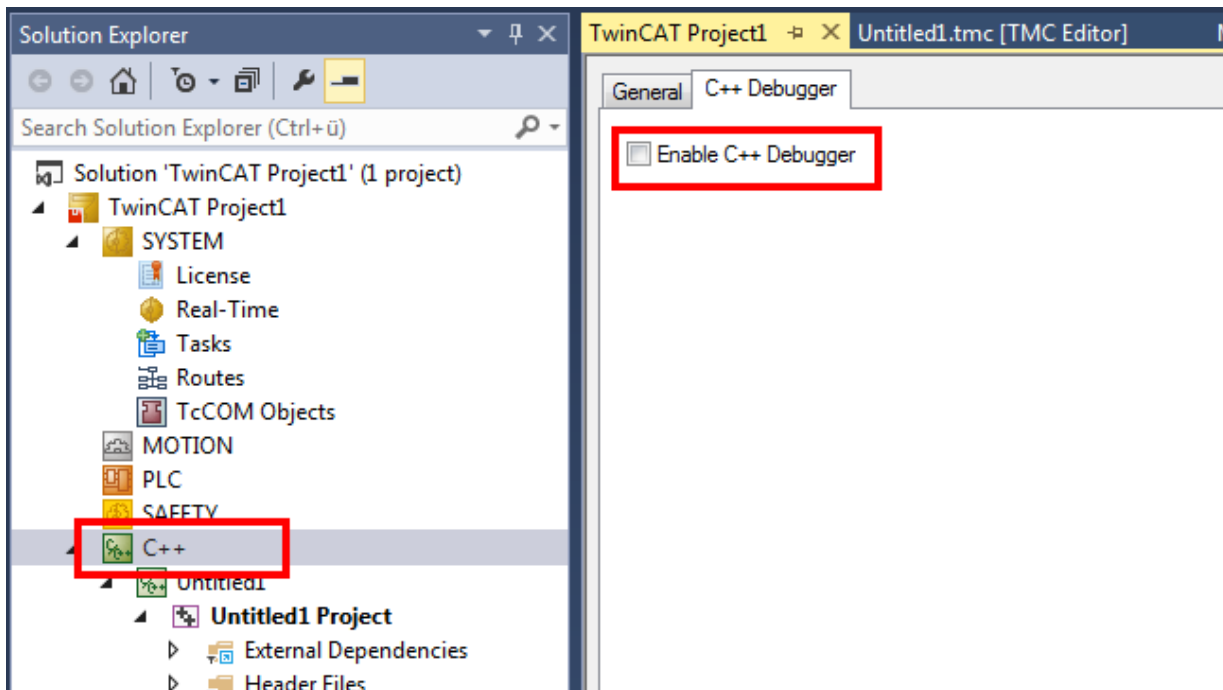


⇒ On completion of this step the "Interface Pointer" is configured as a "CyclicCaller". The configuration is now complete!

**9.7 TwinCAT 3 enable C++ debugger**

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

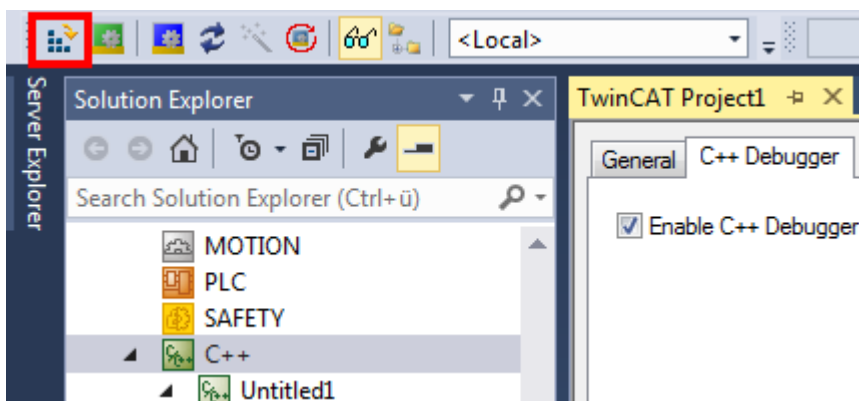
1. Select the "C++ Debugger" tab on the C++ node of the solution, select "Enable C++ Debugger" and switch on "Enable C++ Debugger"



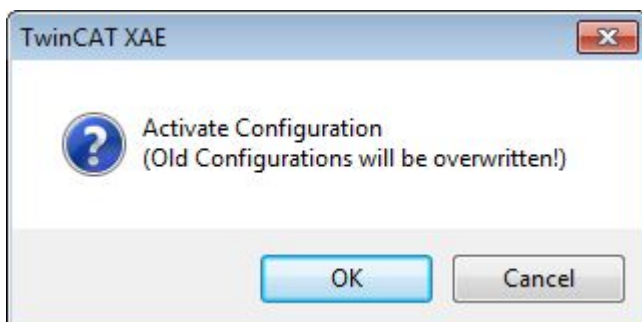
## 9.8 Activating a TwinCAT 3 project

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

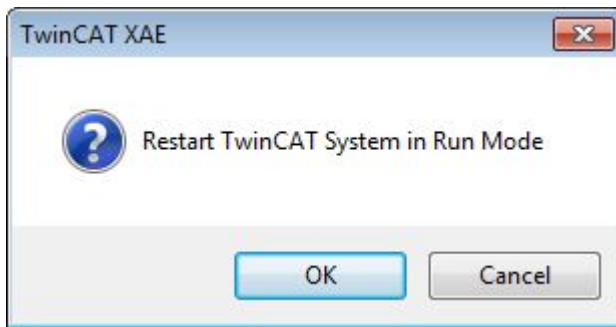
1. Click on the symbol "Activate Configuration" – all required files for the TwinCAT project are transferred to the target system:



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



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



- ⇒ In the case of **"OK"**, the TwinCAT 3 project switches to Run mode. In the case of **"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.



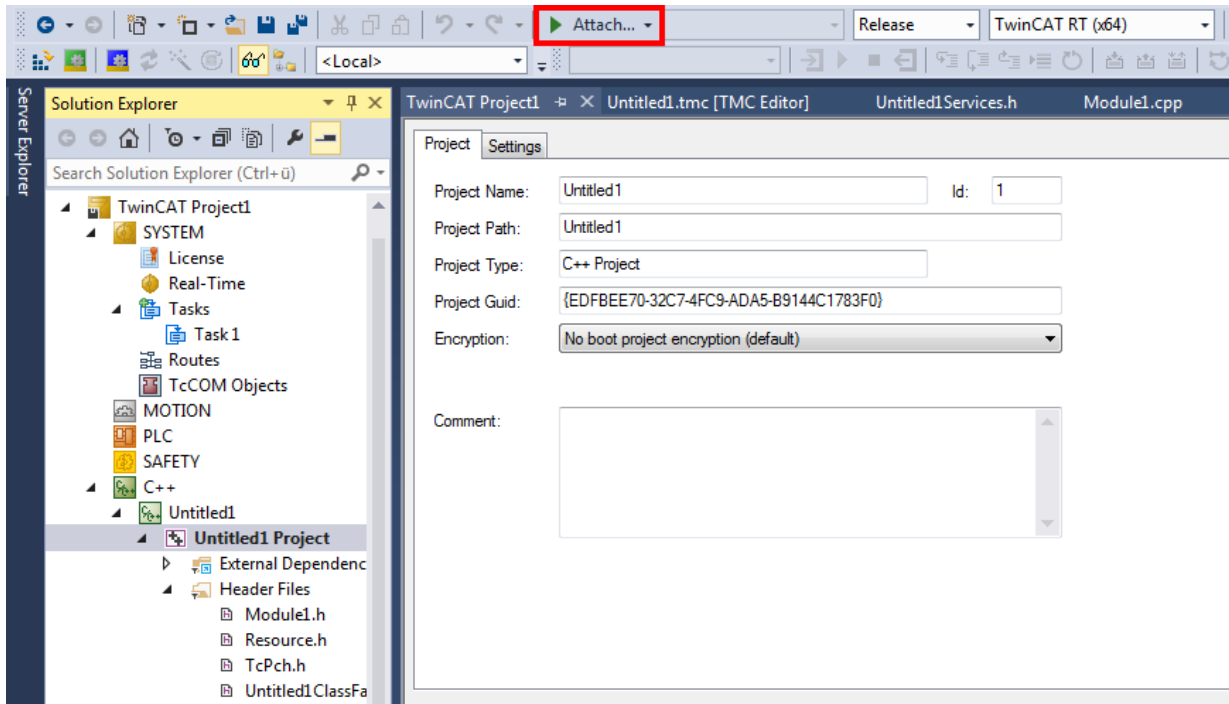
## 9.9 Debug TwinCAT 3 C++ project

This article describes the debugging of the TwinCAT 3 C++ example project.

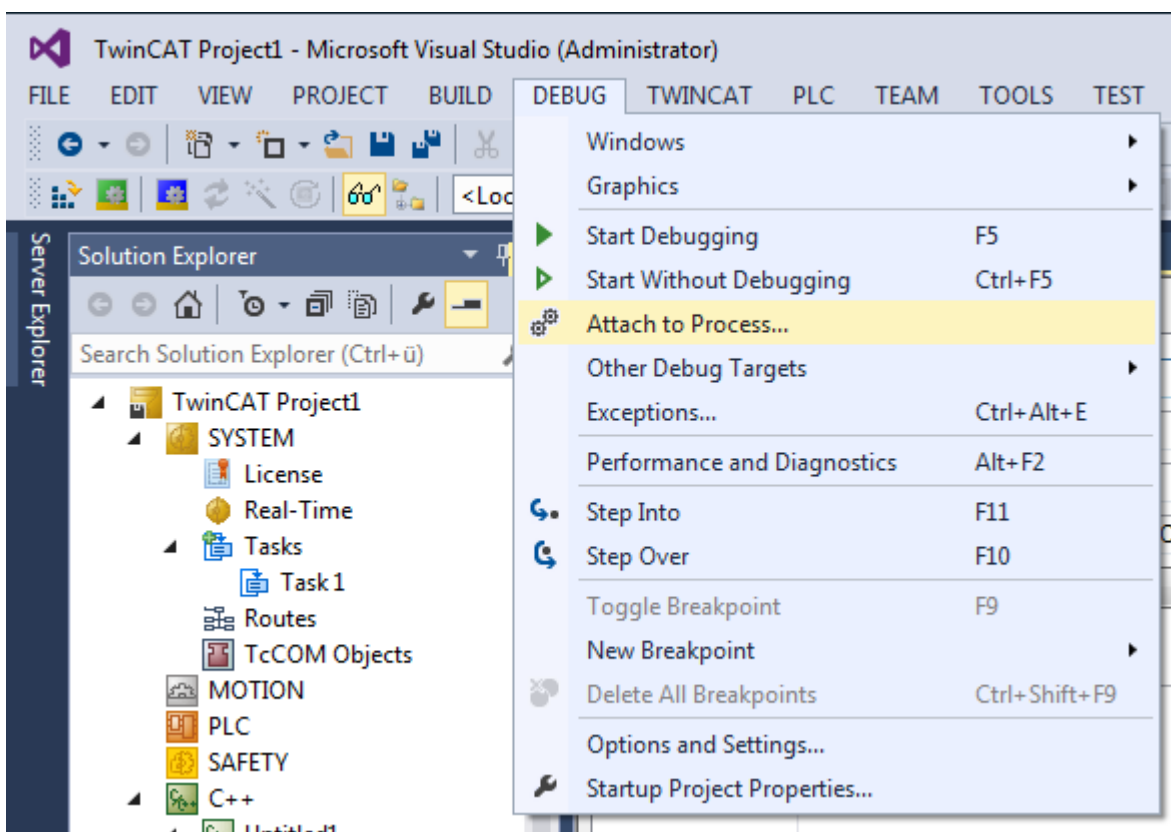
### Attachment to the C++ runtime

After switching on the C++ debugging in the TwinCAT project and activating the complete project, the TwinCAT Engineering (XAE) can now be used to connect to the target system for debugging.

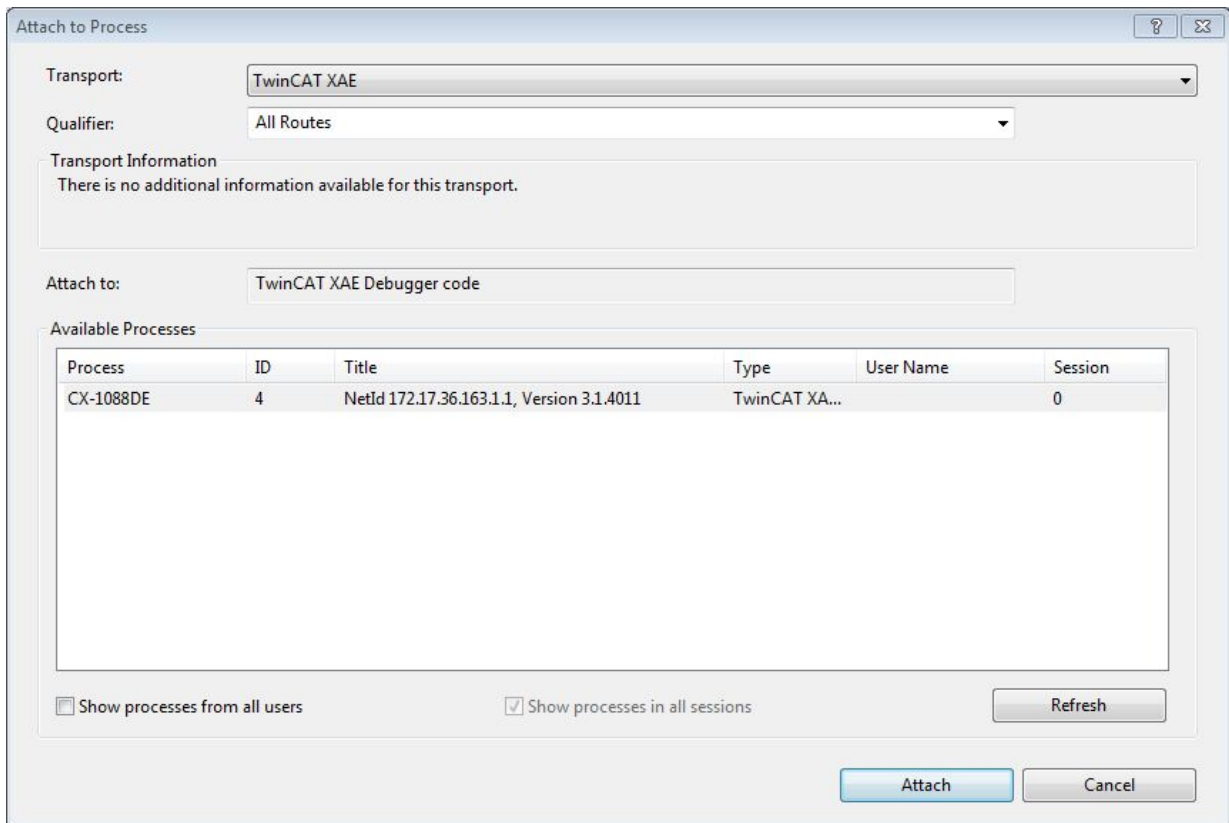
1. A) Click on the "Attach..." button familiar from Visual Studio in order to connect to the TwinCAT debugger on the target system:



- B) Alternatively, select "Debug" -> "Attach to process..." in the Visual Studio environment:



2. Do not select the "Default" setting of Visual Studio for the transport, use "TwinCAT XAE" instead. Target system (or "All Routes") as qualifier and connect by "Attach".



⇒

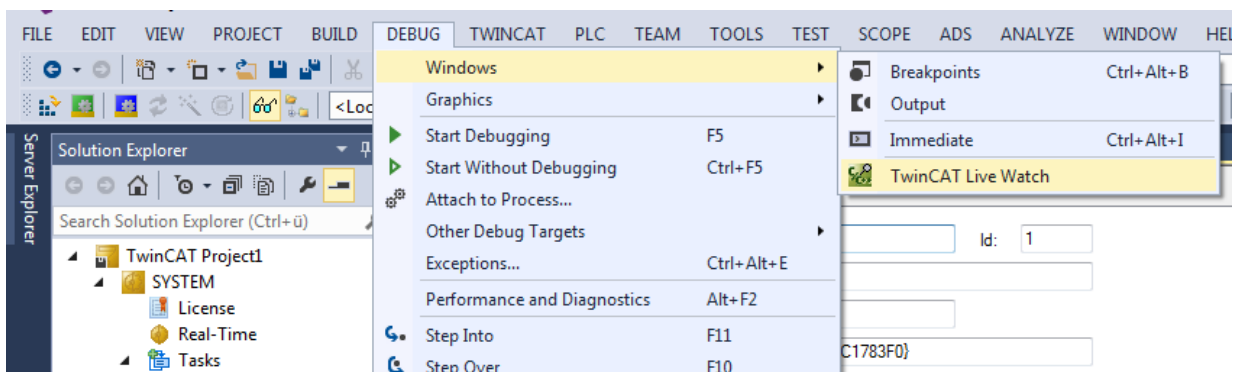
**Monitoring C++ member variables (without breakpoints)**

The "normal" Visual Studio debugging mechanism is available – setting of breakpoints, step execution, etc. Their usage depends on the process to be monitored:

If TwinCAT runs on a real machine with axis movements, the user will probably not wish to set any breakpoints just for monitoring variables. On reaching a breakpoint the execution of a task would be stopped and, depending on the configuration, the axis would immediately come to a halt or, perhaps even worse, would continue to move in an uncontrolled fashion – a very unfortunate situation.

**TwinCAT 3 therefore offers the option to monitor process variables without setting breakpoints:**

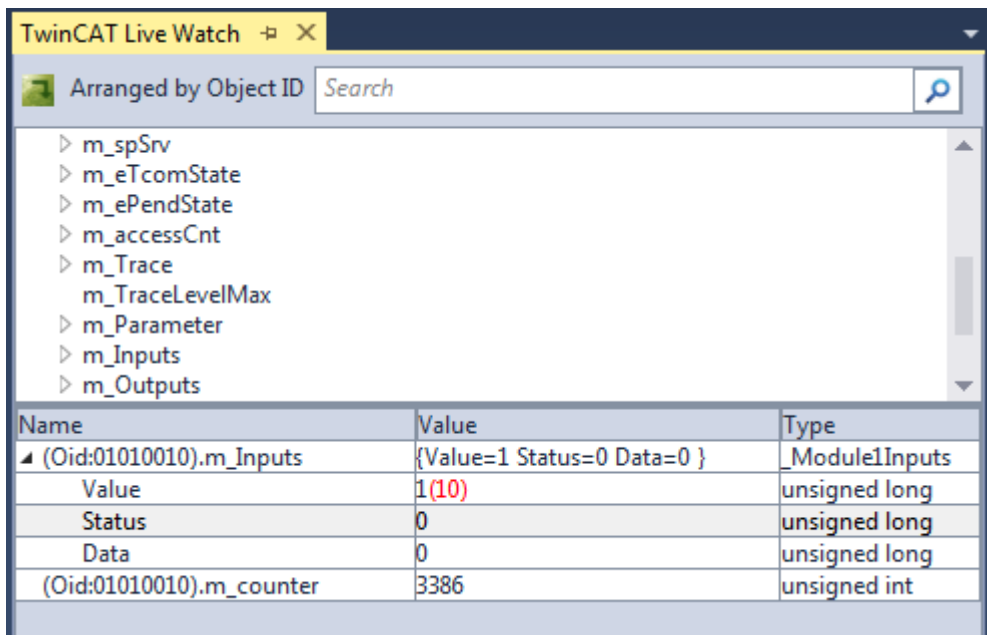
1. Select "Debug" -> "Windows" -> "TwinCAT Live Watch"



⇒ The "TwinCAT Live Watch" windows show a list of all the variables in the module. Variables placed in the watch list by drag & drop are monitored without setting breakpoints.

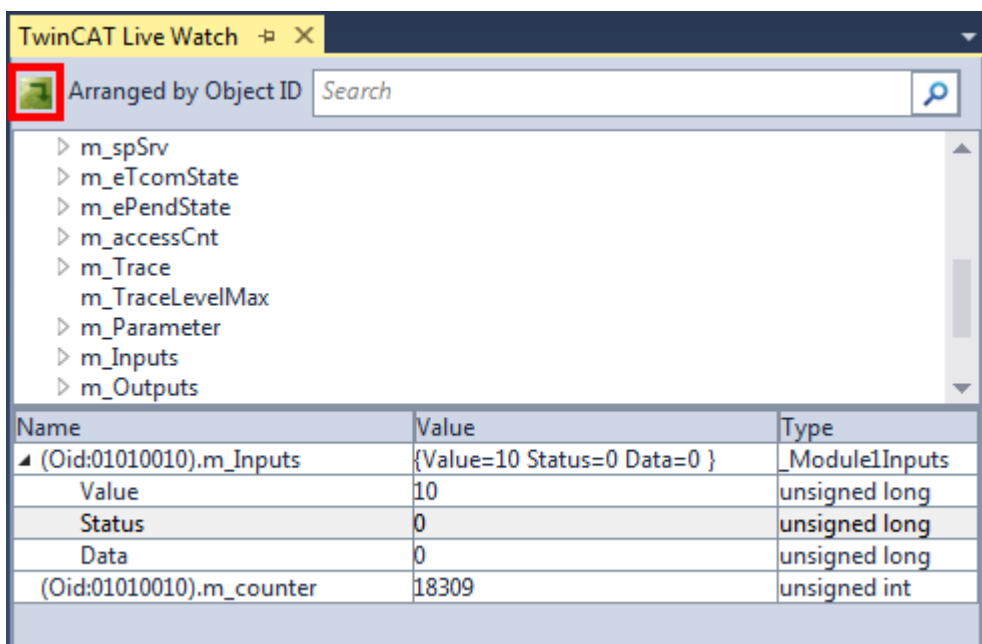
2. In order to change the value of a monitoring variable, simply enter a new value.

⇒ The new value is displayed in red and in brackets.



3. Click on the green symbol

⇒ the new value is written into the process.

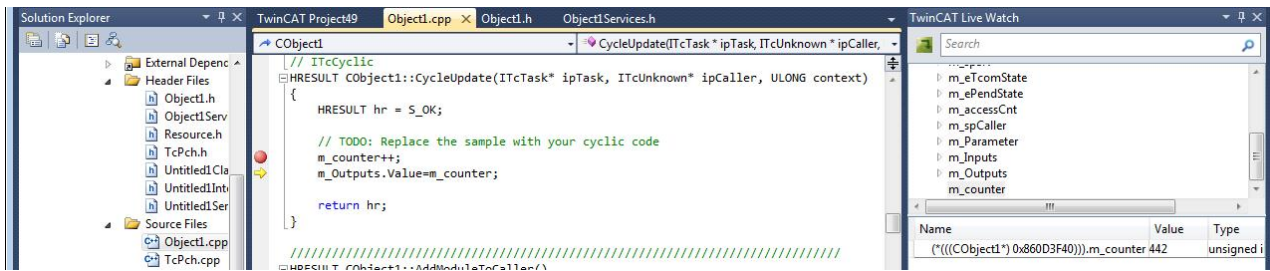


### Setting breakpoints

The setting of breakpoints in the conventional way is also possible.

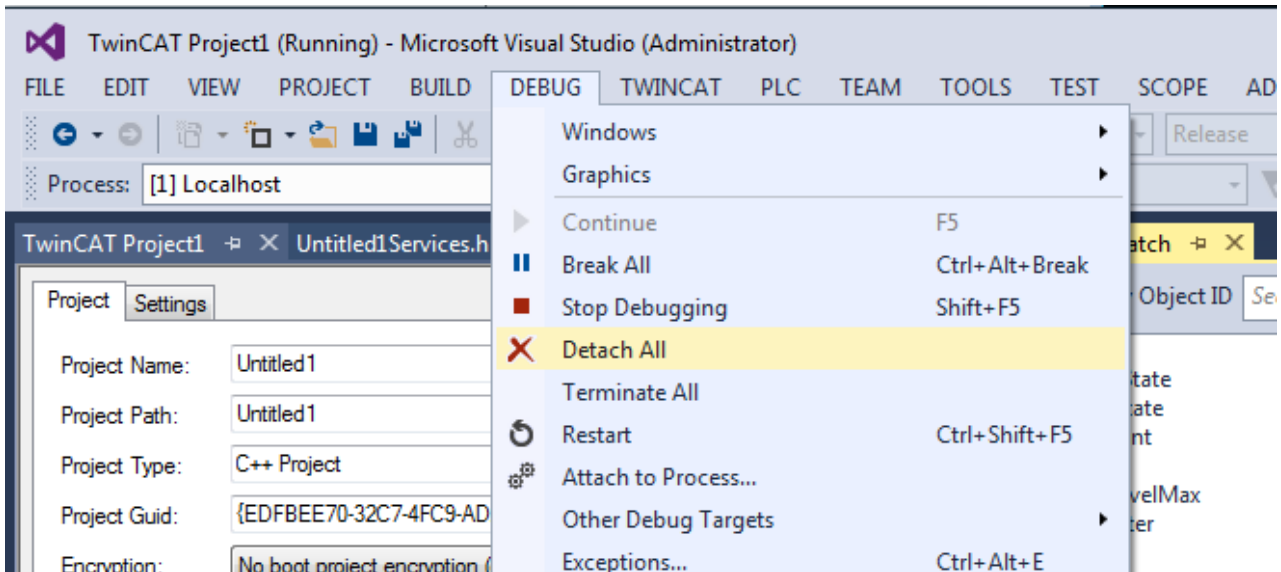
**⚠ WARNING**

**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.



### Detaching the debugger from the process

Click on "Debug" -> "Detach All".



## 10 Debugging

TwinCAT C++ offers various mechanisms for debugging TwinCAT C++ modules running under real-time conditions.

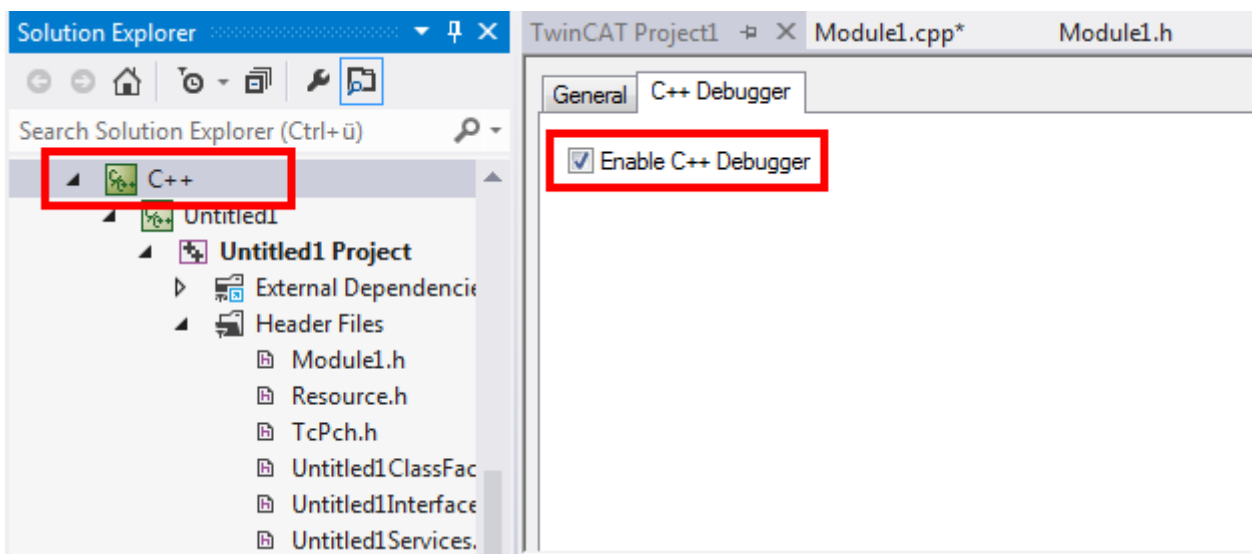
Most of them correspond to the mechanisms that are familiar from the normal C++ development environment. The world of automation requires additional, slightly different debugging mechanisms, which are documented here.

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

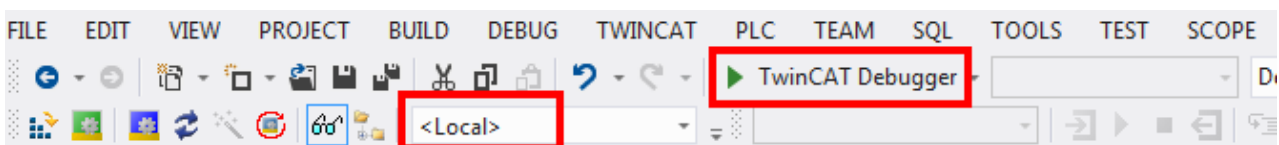
### Debugging needs to be enabled.

This could be configured by the C++ node of the solution:

Double click on the C++ node and switch to the “C++ Debugger” tab for hitting the checkbox.



For using any TwinCAT C++ debugging, the TwinCAT Engineering needs to be connected to the execution (XAR) system by the “TwinCAT Debugger” Button:



### 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.



```

    ///<AutoGeneratedContent id="ImplementationOf_ITcCyclic">
    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_Inputs.Value;
    }
    
```

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

```

    ///<AutoGeneratedContent id="ImplementationOf_ITcCyclic">
    HRESULT CModule1::CycleUpdate(ITcTask* ipTask, ITcUnknown* ipCaller, ULONG_PTR context)
    {
        HRESULT hr = S_OK;

        // TODO: Replace the sample with your cyclic code
    }
    
```

The code can be 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 the code at a breakpoint is only stopped 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.

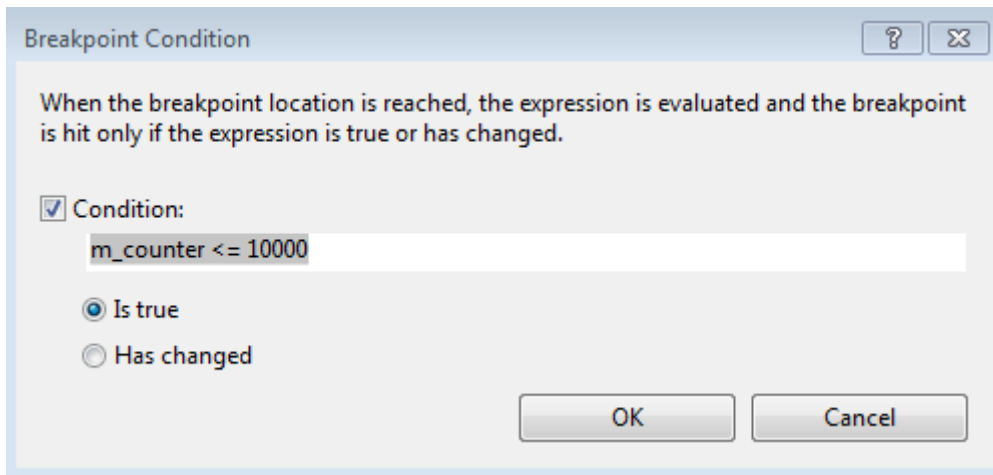
Make sure that the changed behavior of the controlled system does not cause any damage and be sure to note the plant documentation.

```

    ///<AutoGeneratedContent id="ImplementationOf_ITcCyclic">
    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_Inputs.Value;
    }
    
```

Select "Condition..." to open the condition window:



Details of the conditions and how they are to be formulated can be found [here](#) [▶ 71].

### Live Watch

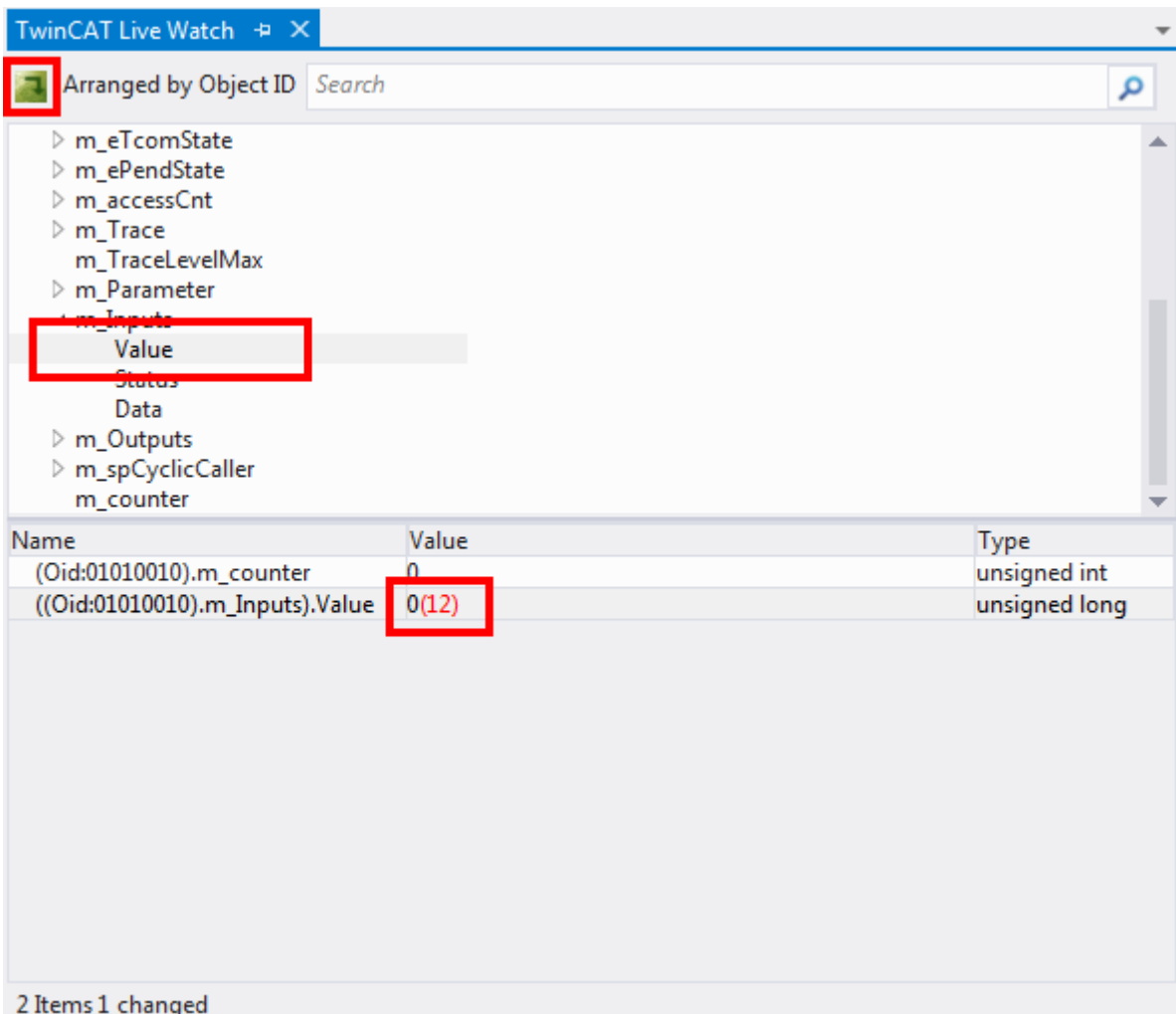
During engineering and development of machines, it is not always reasonable to halt the system by a breakpoint since this influences the behavior.

TwinCAT PLC projects provide online view and manipulation of variables during RUN state without interrupting the real time.

TwinCAT C++ projects provide similar behavior for C++ code via the “Live Watch” window.

The “Live Watch” window could be opened by Debug->Windows->TwinCAT Live Watch.

To open the window first connect to the real time system (hit the “TwinCAT Debugger” button) thus Visual Studio switches to the Debug perspective, otherwise no data could be provided.



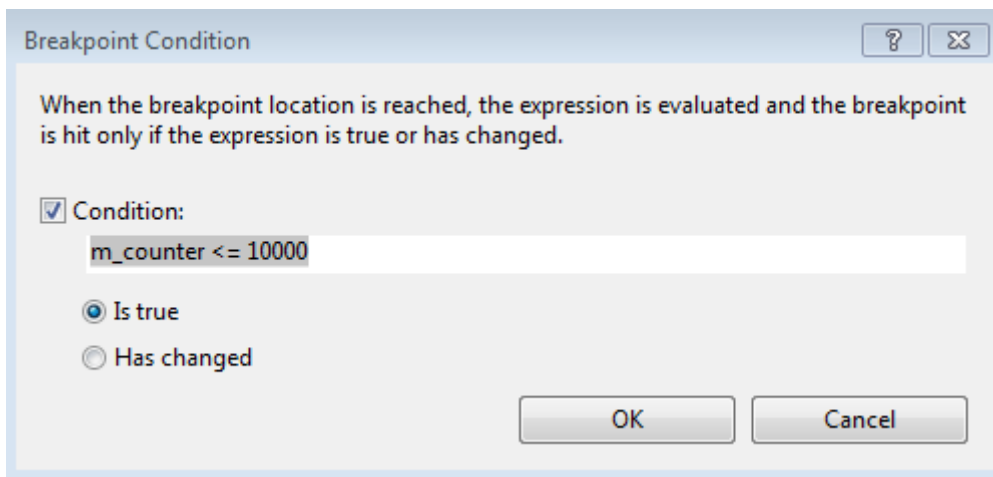
The TwinCAT Live Watch Window is divided into two parts.

In the upper part, all member variables could be explored. By double-clicking them they will be added to the lower part, where current value is displayed.

You can edit these values in the "Value"-Field By clicking on the value. The new value is enclosed by brackets and marked in red color. For actually writing the value, hit the icon at the upper left corner.

## 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 by logical terms, comparable to a CNF (conjunctive normal form). They are combined of && connected “Maxterms”:

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

where each Maxterm is a combination of || connected conditions:

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

Provided relational operator: ==, !=, <=, >=, <, >

For determining the available variables see the Live Watch Window. All listed variables could be used for formulating conditions. These are 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
```

Additional notes:

- **Monitoring of module instances:**  
The OID of the object is stored in `m_objId`, thus monitoring the OID could be i.e. `m_objId == 0x01010010`
- **Monitoring of tasks:**  
A special variable `#taskId` is provided to access OID of the calling task. I.e. `#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 Engineering extends these Visual Studio tools, so that debugging of C++ code that runs on a target system is also possible 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 option 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.

### Call stack

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

Name
0xffff8800a4a1574()
TcRtsObjects.sys!CTask::TaskEntryPoint() Line 570
TcRtsObjects.sys!CTask::CycleTask() Line 1127
TcRtsObjects.sys!CADT::ExecTask() Line 602
Untitled1.sys!CModule1::CycleUpdate(ITcTask* ipTask, ITcUnknown* ipCaller, unsigned __int64 context) Line 179
Untitled1.sys!CModule1::Add(unsigned long a, unsigned long b, unsigned long* res) Line 227

### 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.

Locals		
Name	Value	Type
[-] this	0xfffffa801a1d57b0	CModule1*
[+] IComObject	{}	IComObject
[+] ITcADI	{}	ITcADI
[+] ITcWatchSource	{}	ITcWatchSource
[+] ITcCyclic	{}	ITcCyclic
[+] Calc	{}	Calc
[+] m_refCnt	{value=2 }	AUTO_ULONG
[+] m_objId	{value=16842768 }	AUTO_ULONG
[+] m_parentObjId	{value=0 }	AUTO_ULONG
[+] m_objName	{str={...} }	AUTO_NAMESTR
[+] m_spSrv	{m_pInterface={...} m_oid=0 }	_tc_com_ptr_t<_tc_com_IIIID<ITCo
[+] m_eTcomState	{value={...} }	AUTO_TCOM_STATE
[+] m_ePendState	{value={...} }	AUTO_TCOM_STATE_INVALID
[+] m_accessCnt	{value=1 }	AUTO_ULONG
[+] m_Trace	{m_TraceLevelMax={...} m_spSrv={...} }	CTcTrace
m_TraceLevelMax	tlAlways (0)	TcTraceLevel
[+] m_Parameter	{data1=0 data2=0 data3=0.0 }	_Module1Parameter
[+] m_Inputs	{Value=123 Status=0 Data=0 }	_Module1Inputs
[+] m_Outputs	{Value=108117 Control=0 Data=0 }	_Module1Outputs
[+] m_spCyclicCaller	{m_info={...} }	_tc_com_ptr_t_listinfo<_tc_com_III
m_counter	0	unsigned int
hr	21	HRESULT
a	123	unsigned long
b	108117	unsigned long
[-] res	0xfffffa801a1d5824	unsigned long*
	108117	unsigned long

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

Watch 1	
Name	Value
a	123
b	108117
*(res)	108117

**Memory view**

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

The screenshot shows a debugger interface with two main windows: 'Autos' and 'Memory1'.

**Autos Window:**

Name	Value	Type
Status	0	unsigned
Data	0	unsigned
m_Inputs.Value	1	unsigned
m_counter	11	unsigned
▲ this	0xfffffa8022ceca...	CModule
▶ ITCComObject	{}	ITComO
▶ ITCADI	{}	ITcADI
▶ ITCWatchSourc	{}	ITcWatc
▶ ITCcyclic	{}	ITcCyclic
▶ m_refCnt	{value=2}	AUTO_U
▶ m_objId	{value=16842768}	AUTO_U
▶ m_parentObjId	{value=0}	AUTO_U
▶ m_objName	{str={...}}	AUTO_N

**Memory1 Window:**

Address: 0xfffffa8022ceca8

Address	Hex	ASCII
0xFFFFFA8022CECAC8	28 1f 4f 22 80 fa ff ff 28 1f 4f 22	(.O"€úÿÿ(.O"
0xFFFFFA8022CECAD4	80 fa ff ff 28 1f 4f 22 80 fa ff ff	€úÿÿ(.O"€úÿÿ
0xFFFFFA8022CECAE0	38 ca ce 22 80 fa ff ff 00 00 00 00	8ÉÏ"€úÿÿ....
0xFFFFFA8022CECAEC	00 00 00 00 64 00 00 00 ab ab ab ab	....d...««««
0xFFFFFA8022CECAF8	0b 00 00 00 ab ab ab ab ab ab ab ab	....««««««««
0xFFFFFA8022CECB04	00 00 00 00 30 6c f2 22 80 fa ff ff	....01d"€úÿÿ
0xFFFFFA8022CECB10	80 6e 8d 08 80 f8 ff ff 00 00 00 00	€n..€øÿÿ....
0xFFFFFA8022CECB1C	00 00 00 00 12 00 08 02 54 4d 73 63	.....TMsc
0xFFFFFA8022CECB28	40 17 06 04 00 f8 ff ff 00 00 00 00	@...øÿÿ....
0xFFFFFA8022CECB34	00 00 00 00 20 86 43 13 80 fa ff ff	.... .C.€úÿÿ
0xFFFFFA8022CECB40	50 c4 c2 22 80 fa ff ff 48 cb ce 22	PÄÄ"€úÿÿHÉÏ"
0xFFFFFA8022CECB4C	80 fa ff ff 00 00 00 00 00 00 00 00	€úÿÿ.....
0xFFFFFA8022CECB58	00 00 00 00 00 00 00 00 a0 c4 c2 22	..... ÄÄ"
0xFFFFFA8022CECB64	80 fa ff ff b0 06 45 13 80 fa ff ff	€úÿÿ°.E.€úÿÿ

# 11 Wizards

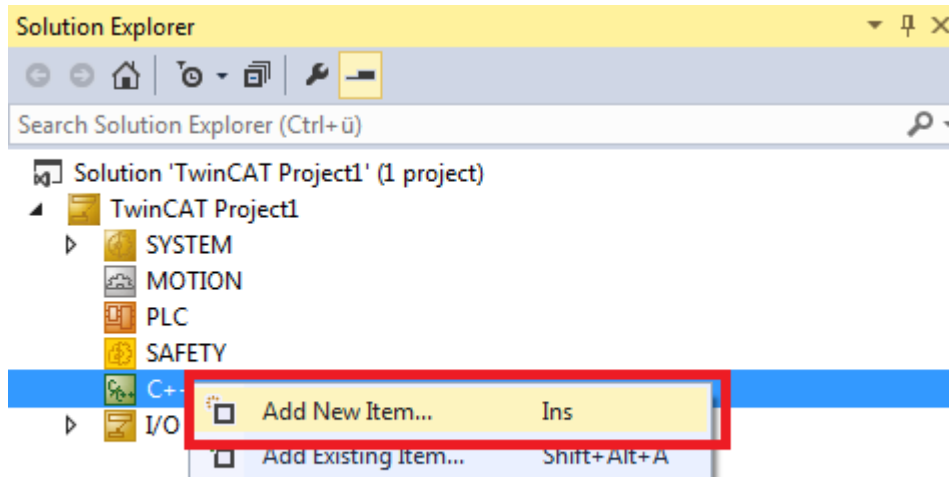
For ease of entrance in engineering the TwinCAT C++ system provides Wizards.

- The [TwinCAT Project Wizard](#) [▶ 76] creates a TwinCAT C++ project. For Driver projects, the TwinCAT Class Wizard will be started afterwards.
- The [TwinCAT Module Class Wizard](#) [▶ 77] is automatically started during creation of a C++ module. This wizard provides different “ready to use” projects as entry points for own development.
- The [TwinCAT Module Class Editor](#) [▶ 79] (TMC) is a graphical editor for defining the data structures, parameters, data areas, interfaces and pointers. It generates a TMC file, which will be used by the TMC Code generator.
- From the defined Classes instances will be generated and could be configured via the [TwinCAT Module Instance Configurator](#) [▶ 124]

## 11.1 TwinCAT C++ Project Wizard

After creating a TwinCAT project, one could add a C++ project by using the TwinCAT C++ project wizard:

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

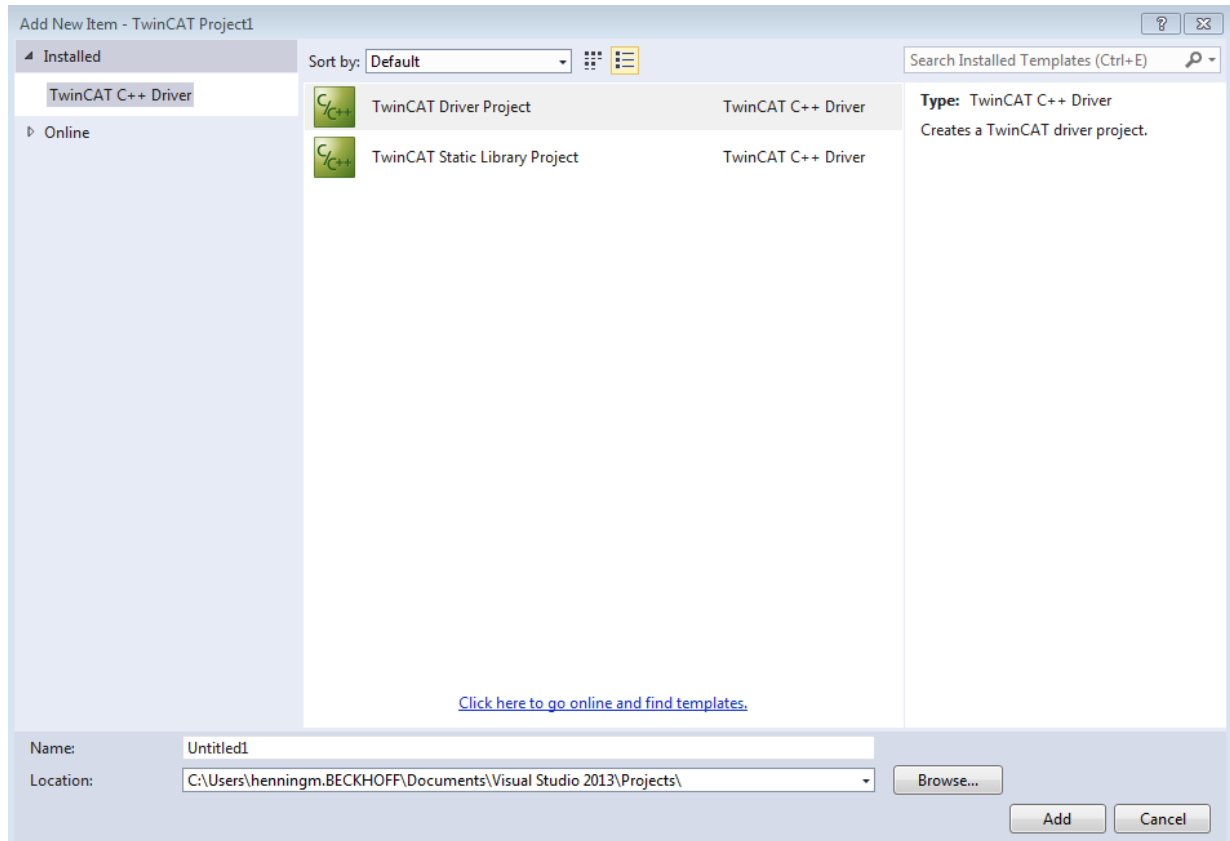


TwinCAT offers two C++ projects:



Driver project: Projects, which contains one or more modules to be executed

Static Library: Projects providing C++ functions used by (different) TwinCAT C++ drivers.



2. Select one of the project templates, provide a name and location.

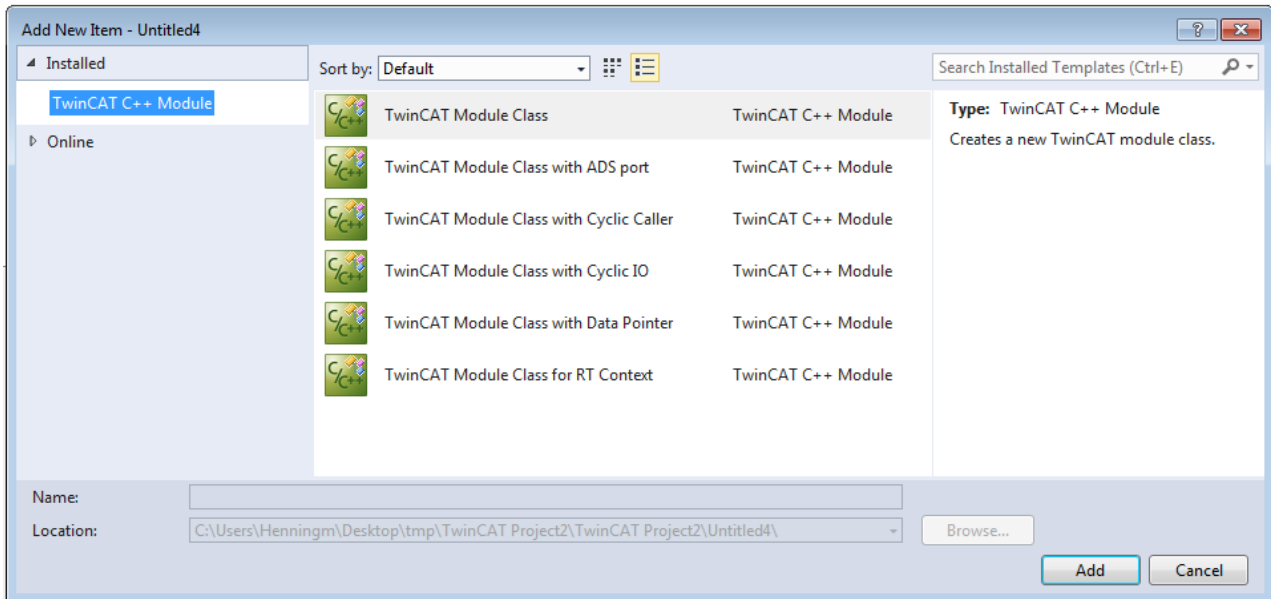
⇒ The TwinCAT C++ project will be created

⇒ For a driver, the TwinCAT C++ class wizard [▶ 77] will be started

## 11.2 TwinCAT Module Class Wizard

TwinCAT 3 offers different class templates

- TwinCAT Modules Class
- TwinCAT Modules Class with ADS port
- TwinCAT Modules Class with Cyclic Caller
- TwinCAT Modules Class with Cyclic IO
- TwinCAT Modules Class with Data Pointer
- TwinCAT Modules Class for RT Context



### 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](#) [▶ 236]

### TwinCAT Modules Class with ADS port

This template provides both the C++ module and acts as an ADS-Server and an ADS client.

- ADS Server:  
Can be run as a single instance of this template the C++ module and can be pre-configured with a specific ADS-port number (e.g. 25023).  
Allows multiple instances of this template the C++ modules will each get its own unique ADS-port number from TwinCAT 3 assigned (e.g. 25023, 25024, 25025...).  
The ADS messages being analyzed and managed can be specified by implementing the C++ module.  
ADS handling to access the input / output data areas don't have to be implemented with own ADS message handling.
- ADS Client:  
This template provides sample code how to initiate an ADS call by sending out an ADS message to an ADS partner.

As the modules is acting as ADS-client or ADS-server communicating with each other via ADS messages the two modules (caller=client and the cally=server) can run in same or different real-time contexts on same or different CPU-cores.

As ADS is capable to cross network the two modules can also run on different machines in the network..

See [sample 03](#) [▶ 216], [ADS Communication](#) [▶ 173]

### TwinCAT Modules Class with Cyclic Caller

It allows a C++ program to be called cyclically but with no access to the outside world.

This is not commonly 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 \[► 50\]](#), [sample 01 \[► 215\]](#)

### TwinCAT Modules Class with Data Pointer

As the "TwinCAT Module Class with Cyclic IO" this template creates a new TwinCAT module class which implements the cyclic caller interface and which has an input and output data area for linking to other logical input/output images or to physical IO terminals.

Additionally, this template provides "Data Pointers" which allow to access data areas from other modules by pointer.

#### Important to understand:

Unlike the Cyclic I/O data area where the data is copied between modules cyclically, when using the C++ "Data Pointers" there is only one data area and it is owned by the target module. When writing from another C++ module via the "Data Pointer mechanism" to the target module will effect on the target module's data area will be immediate. (Not necessarily at the end of a cycle)

When the module is executed at runtime the call happens immediately blocking the original process (it's a pointer...). Due to this both modules (the caller and the callee) must be in **same real-time context** and must be executed on same CPU core.

The configuration of "Data Pointer" is done with the [TwinCAT Module Instance Configurator \[► 124\]](#).

See [sample 10 \[► 235\]](#)

### TwinCAT Module Class for real-time context

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

As described [here \[► 39\]](#), 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 TC. In this case the transitions are executed like for a normal module.

The [TcCOM 03 sample \[► 295\]](#) illustrates the application of such a module.

## 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.

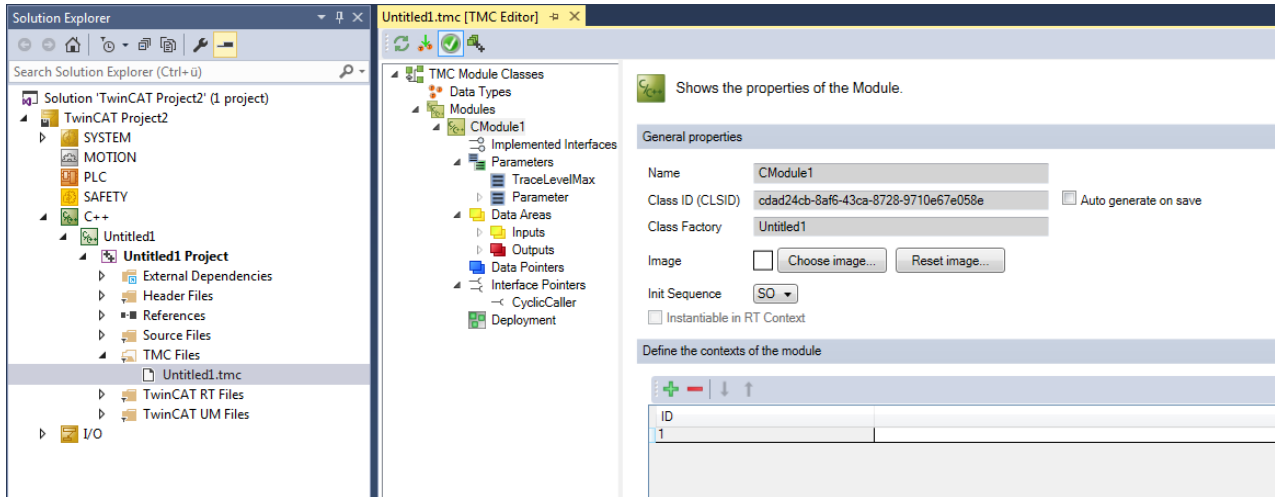
To put it briefly: Everything that is visible from outside must be defined with this editor.

The basic idea is:

1. The TMC Editor can be used to modify the module description file (TMC file). This file contains all the information that can be accessed in the TwinCAT system itself, including 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 the TMC file of a module. The graphical editor opens:



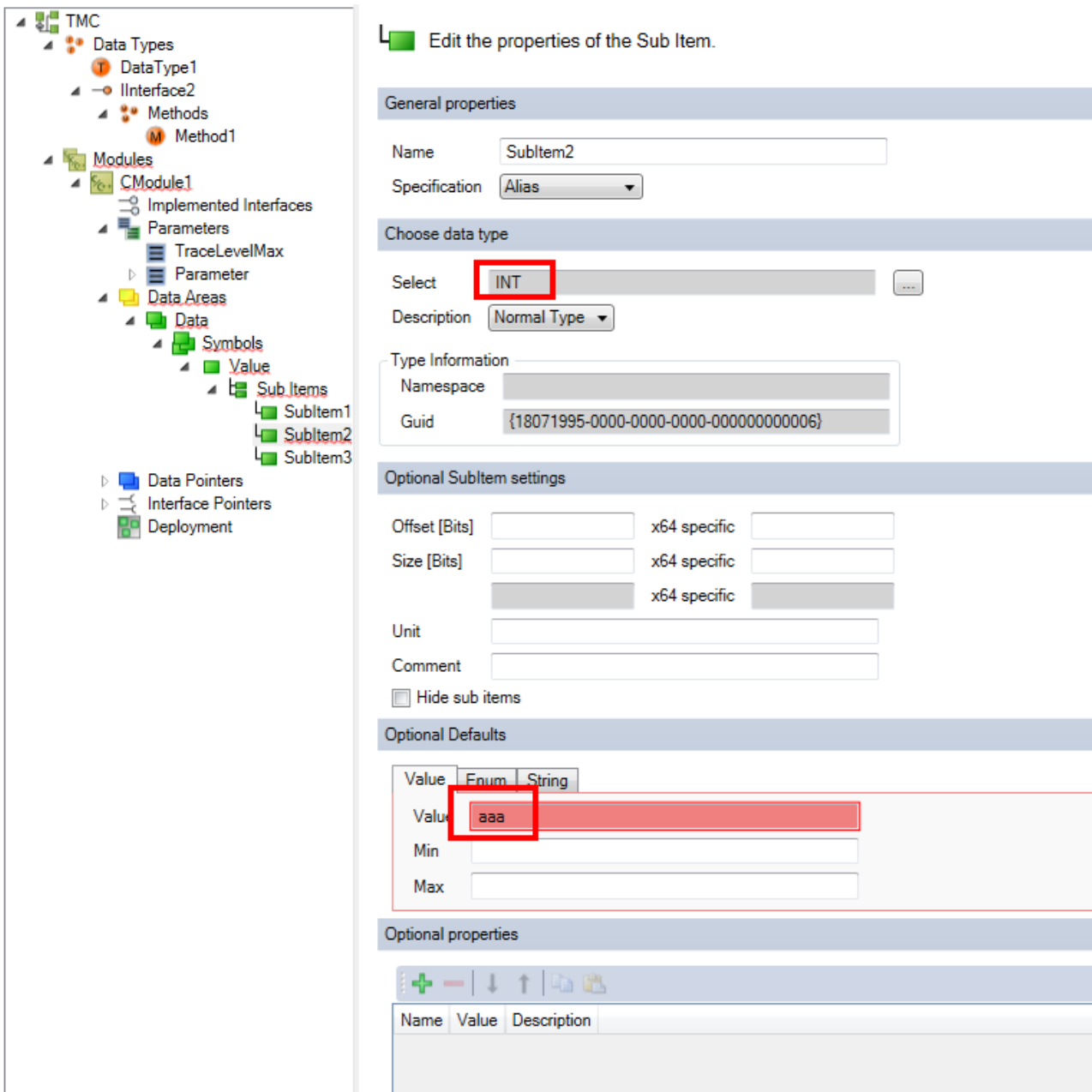
Functionalities of the TMC editor:

- 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.

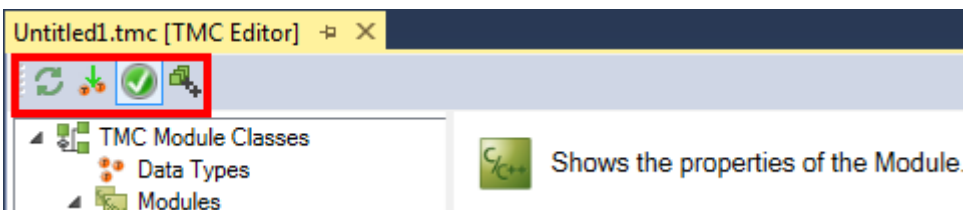
For example, in the event of problems (alignment, invalid standard definitions, ...) within the TMC, the user is guided to the relevant location via red flags within the TMC tree:



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
- Enabling/disabling the user help (see above)
- Starting 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

Shows the properties of the Module.

**General properties**

Name	CModule1
Class ID (CLSID)	6a84bf6d-ddd8-463b-b910-a3dd1cc48121
Class Factory	Untitled1
Image	<input type="checkbox"/> Choose image... Reset image...
Init Sequence	SO
<input type="checkbox"/> Instantiable in RT Context	

**Define the contexts of the module**

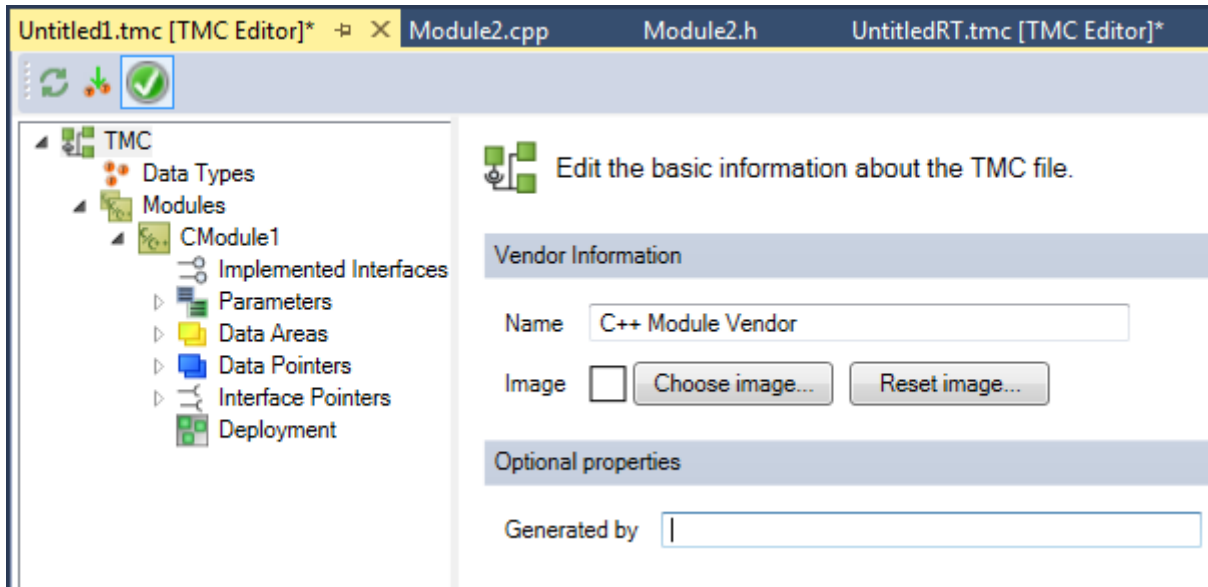
ID	
1	

#### User interface

- [TMC \[▶ 83\]](#): Edit the basic vendor information of the C++ module and add a picture
- [Data Types \[▶ 84\]](#): Add, remove and reorder data types
- [Modules \[▶ 101\]](#): Shows the modules of the driver
- [Implemented Interfaces \[▶ 103\]](#): Shows the implemented interfaces of the module.
- [Parameters \[▶ 104\]](#): Add, remove and reorder your parameters
  - [TraceLevelMax \[▶ 111\]](#): Parameter which controls the amount of log messages – predefined for (nearly) every Module
- [Data Areas \[▶ 112\]](#): Add, remove and reorder Data Areas.
- [Data Pointers \[▶ 119\]](#): Add, remove and reorder Data Pointers.
- [Interface Pointers \[▶ 121\]](#): Add, remove and reorder Data Pointers.
- [Deployment \[▶ 122\]](#): Define the files which should be deployed.

### 11.3.2 Basic Information

Basic information regarding the TMC file



#### Vendor information

**Name:** Edit the module name

**Choose Image:** Insert a 16x16 pixel bitmap icon

**Reset image:** Reset module image to default

#### Optional properties

**Generated by:** This field is used to indicate by whom the file was generated and will be managed. Please note that filling this field disables changes (disables all edit operations) in the TMC editor.

### 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.

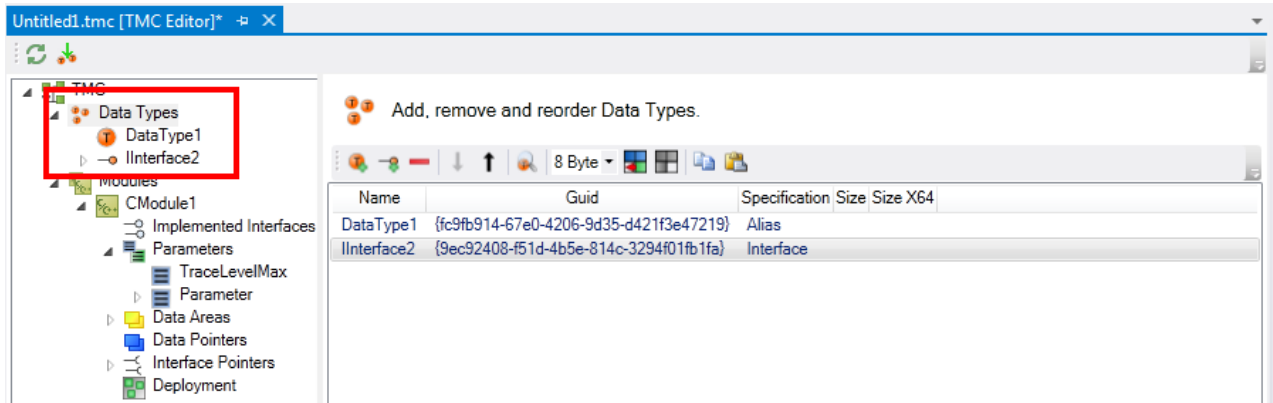
The TwinCAT Engineering system (XAE) publishes these data types in relation to all other nested projects of the TwinCAT project, so that they can also be used in PLC projects, for example (as described [here](#) [▶ 236]).

NOTE
<p><b>Name conflict</b></p> <p>A name collision can occur if the driver is used in combination with a PLC module.</p> <ul style="list-style-type: none"> <li>Do not use any of the keywords that are reserved for the PLC as names.</li> </ul>

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



Deletes the selected type



Moves the selected element down one position



Moves the selected element up one position



Finds unused types

8 Byte ▾

Select byte alignment



Align selected data type (alignment)  
This function loops through all used data types (recursion). If 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 for the data type

**GUID:** Unique ID of the data type

**Specification:** Specification of the data type

**Size:** Size of data type if explicitly specified

**Size X64:** Different size of data type for x64 platform



### 11.3.3.2 Add / modify / delete data types

The TwinCAT Module Class (TMC) Editor allows adding, modifying, and deleting data types used by TwinCAT C++ modules.

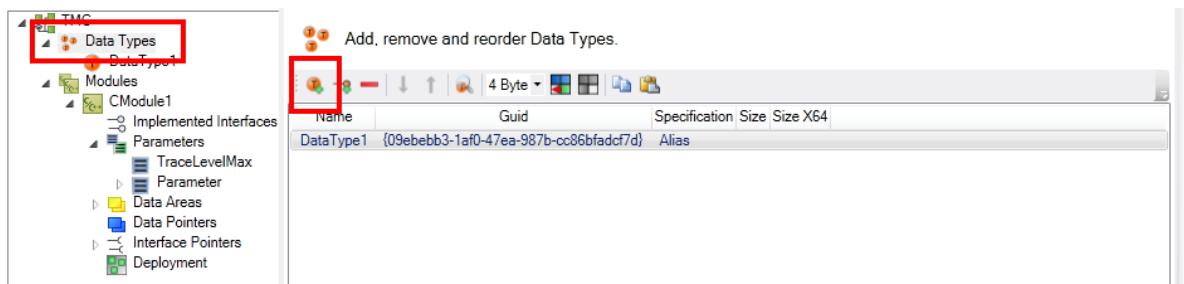
This article describes how to

- [Step 1: Create a new data type \[▶ 85\]](#) in the TMC file
- [Step 2: Start TwinCAT TMC Code Generator \[▶ 88\]](#) to generate C++ code based on module description in the TMC file
- [Make use \[▶ 101\]](#) of the data types

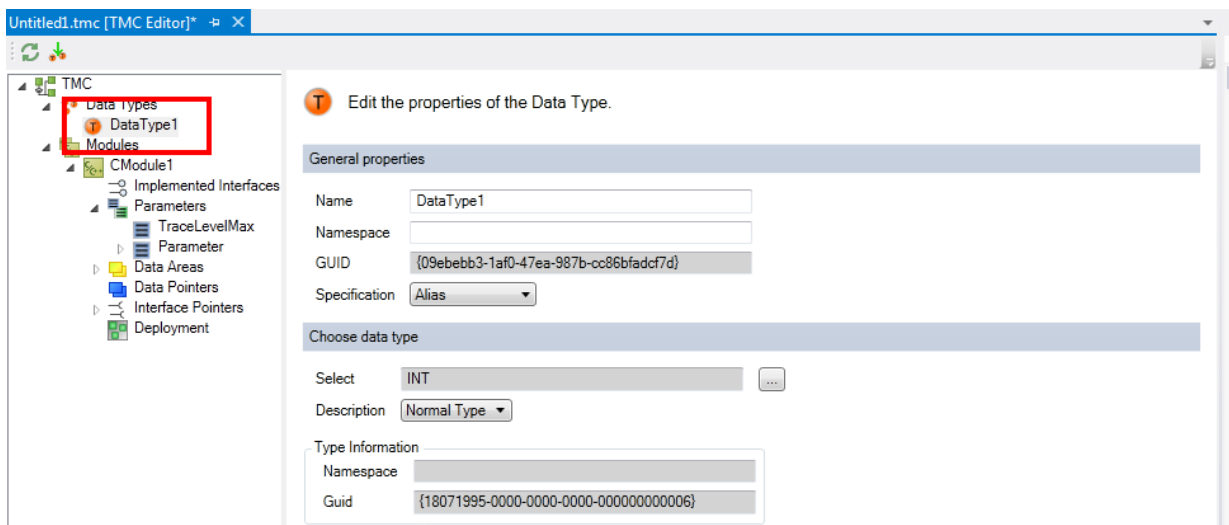
#### Step 1: Create new data type

1. After starting the TMC editor, select the node **"Data Types"**.
2. List of data types and interfaces will be extended with a new data type by clicking the "+" button "Add a new data area".

⇒ As a result a new **"Data type"** is listed as new entry:



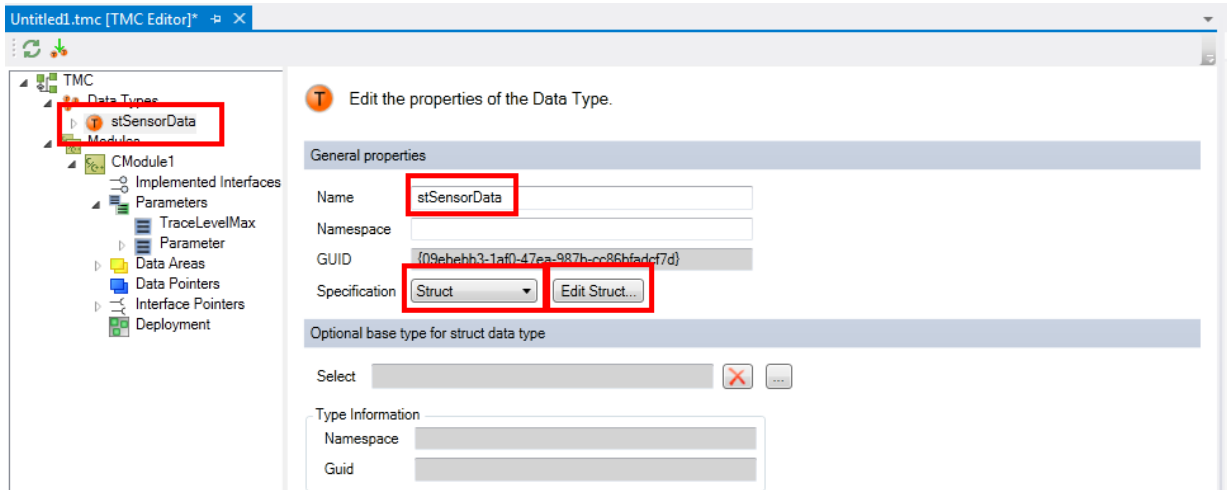
3. Select the generated "Data Type1" to get details of the new data type.



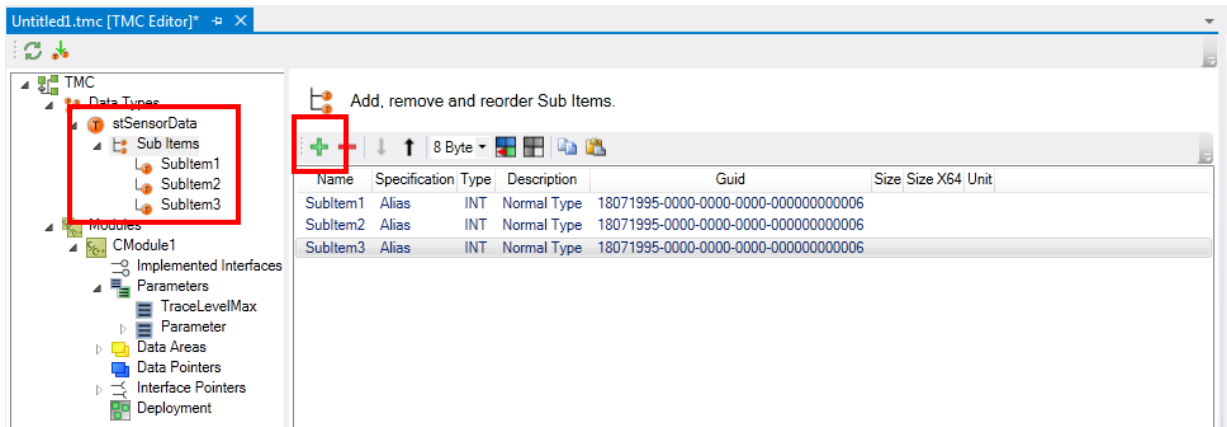
4. Specify the data type.  
See [here \[▶ 95\]](#) for details.

## 5. Rename the data type.

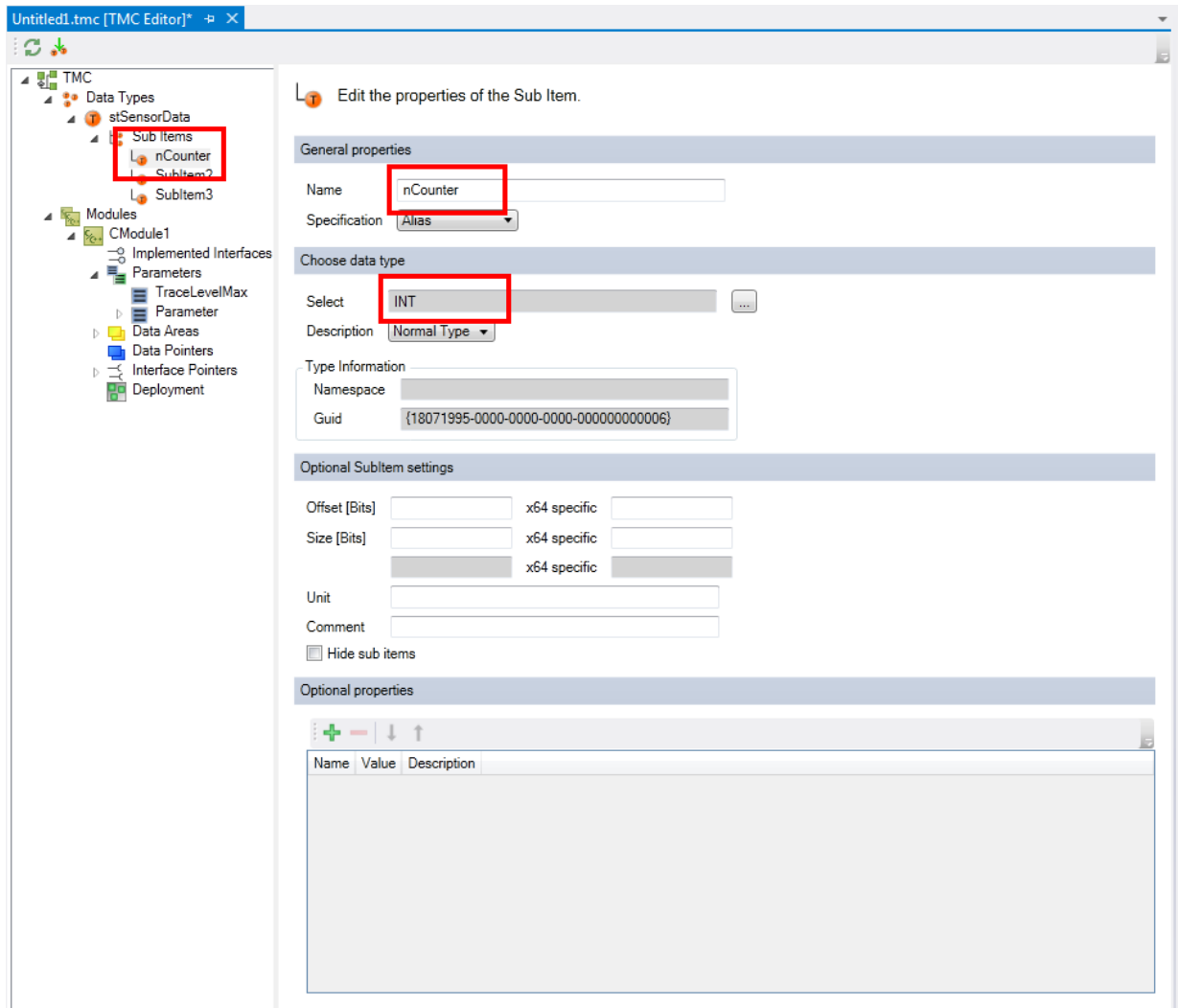
In this sample "**stSensorData**", select the specification "STRUCT" and click on "Edit Struct".



## 6. Insert new sub items to the structure by clicking on the "Add a new sub item button".



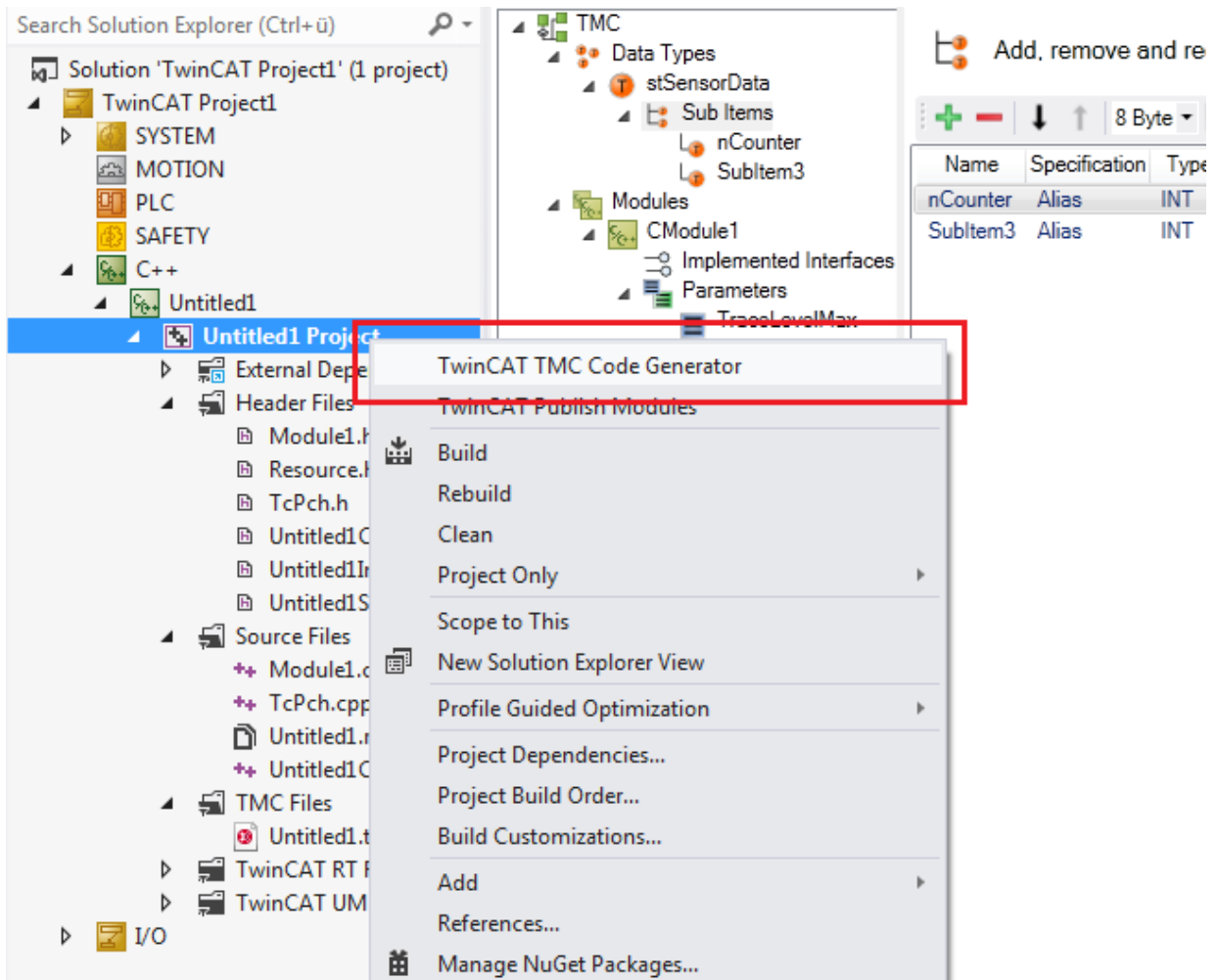
- By double-clicking on the sub item you can edit the properties. Rename the sub item and select an appropriate data type



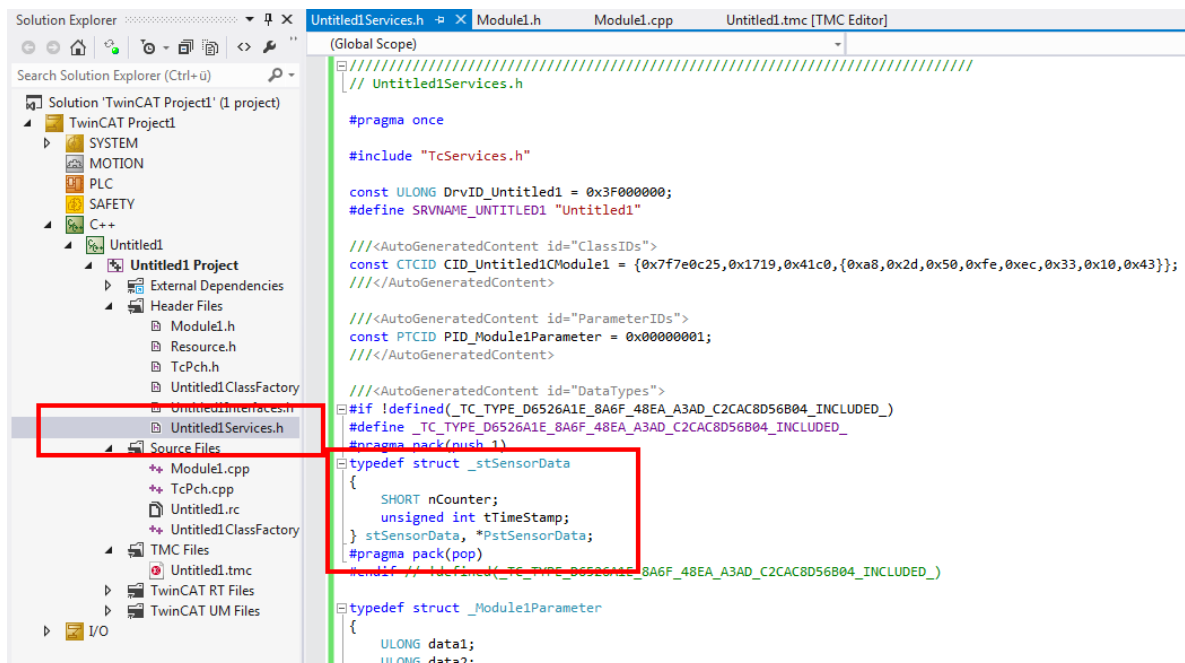
- Rename the other sub items and select a data type:
- Save your changes of the TMC file.

## Step 2: Start TwinCAT TMC Code Generator to create code of module description

10. Right Click on your project file and select "TwinCAT TMC Code Generator" to create the source code of your data type:



⇒ You will see the data type declaration in the module header file "Untitled1Services.h"



⇒ If you add an additional data type or sub item, execute the TwinCAT TMC Code Generator again.

### 11.3.3.3 Add / modify / delete Interfaces

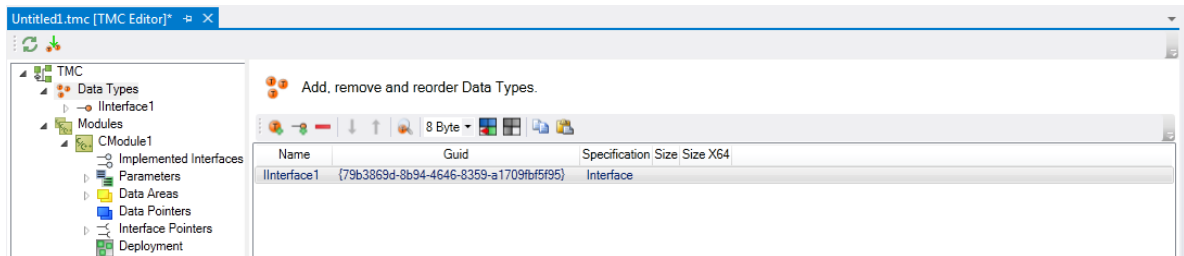
Interfaces of a TwinCAT module can be added, edited and deleted with the help of the TwinCAT Module Class (TMC) editor.

This article describes:

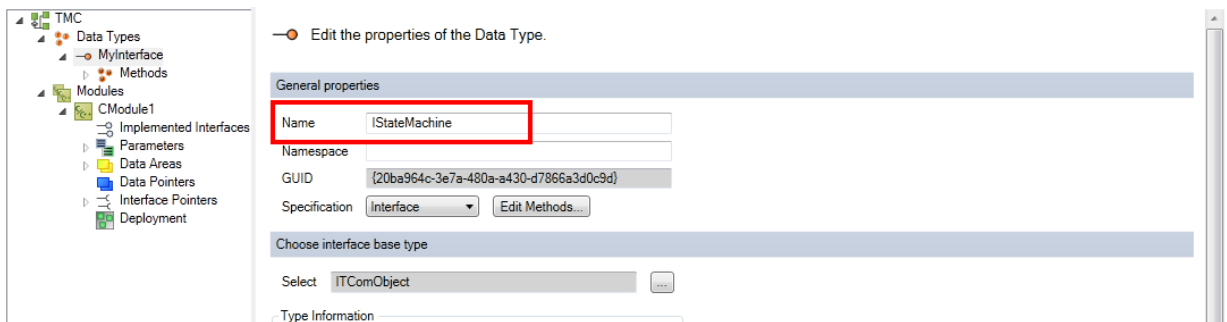
- [Step 1: Create a new interface \[▶ 89\]](#) in the TMC file
- [Step 2: \[▶ 89\]Add methods \[▶ 89\]](#) to the interface in the TMC file
- [Step 3: Use the interface \[▶ 91\]](#) by adding it to the "Implemented Interfaces" of the module.
- [Step 4: Start the TwinCAT TMC \[▶ 93\]](#) Code Generator to generate code for the module description.
- [Optional change of the interface \[▶ 93\]](#)

#### Step 1: Create new interface

1. After starting the TMC editor, select the node **"Data Types"**.
2. List of interfaces will be extended with a new interface by clicking the "Add a new interface".  
⇒ As a result **"Interface1"** is listed as new entry:



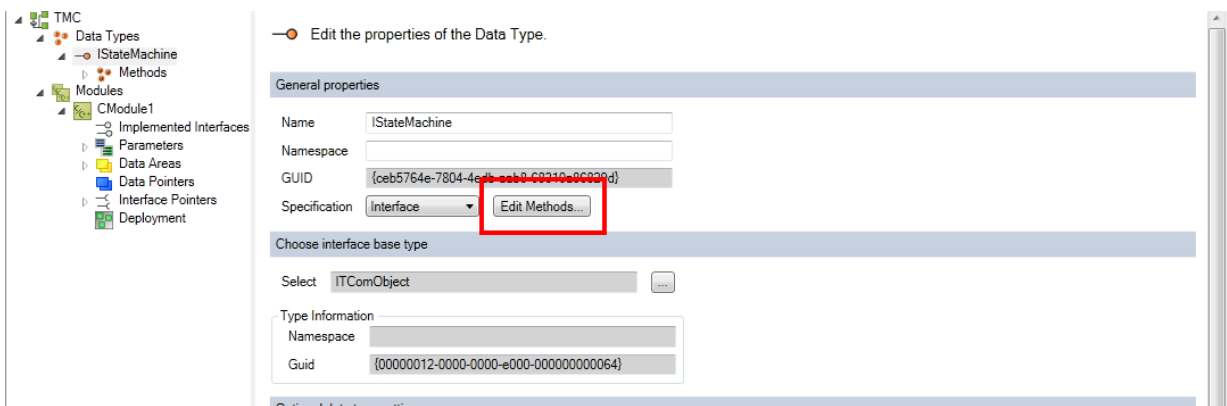
3. Either select the corresponding node in the tree or double click on the line in the table to open the details



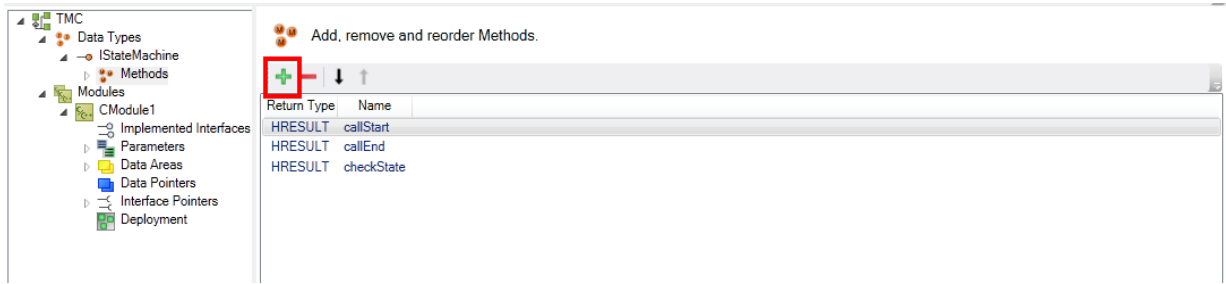
4. Renaming this to a more useful name - in this sample **"IStateMachine"**.

#### Step 2: Add methods to the interface

5. Click **"Edit Methods..."** to get the list of methods of this interface:



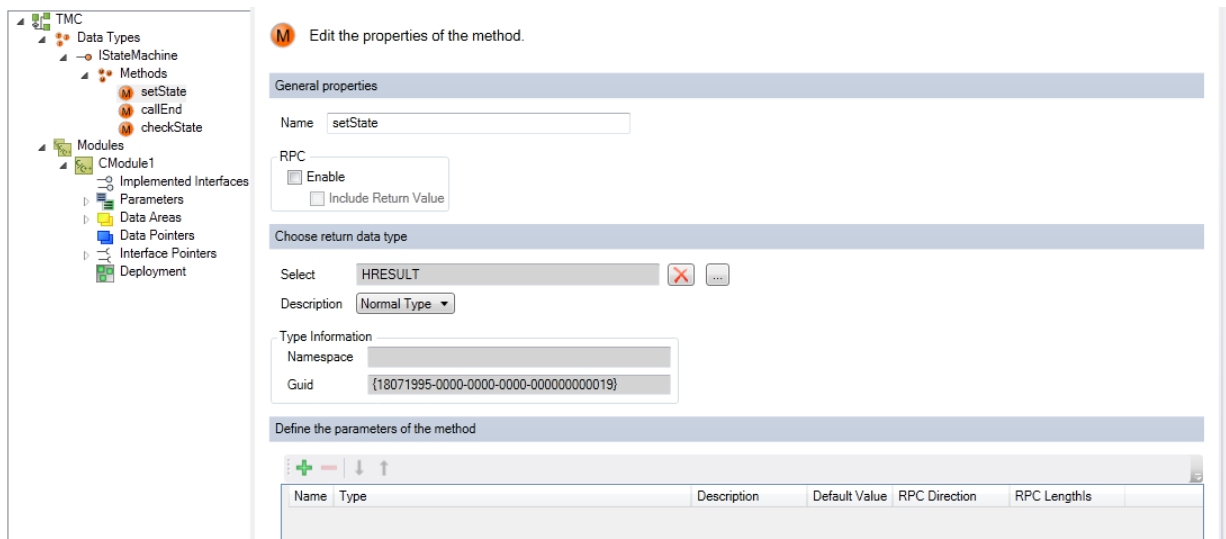
6. Clicking the "+" button will generate a new default method "Method1":



7. Double click on the method or select the node in the tree to open details

8. Renaming the default "Method1" to a more useful name".

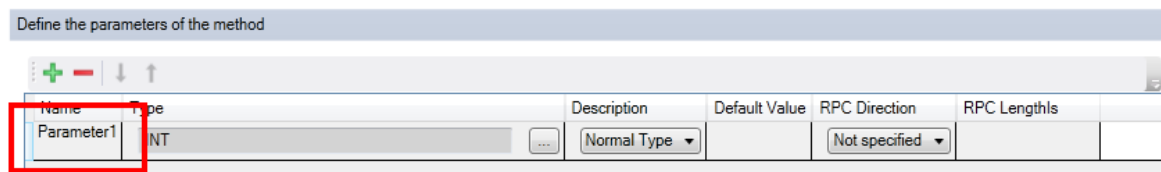
9. After that clicking on "Add a new parameter" allows to add / modify parameters of method "SetState"



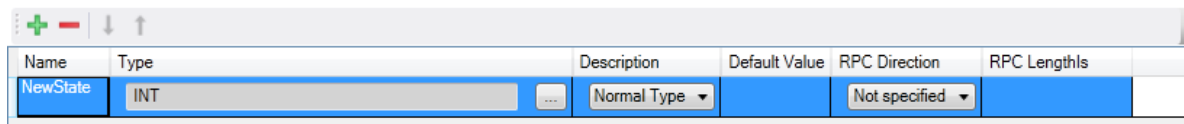
⇒ By default the new parameter "Parameter1" as "Normal Type" of "INTEGER" is created.

10. Click on the name "Parameter1" allows modifying the name.

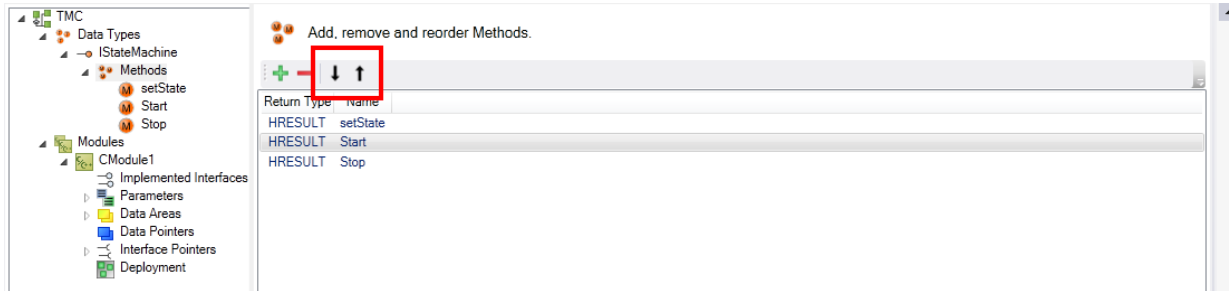
⇒ The "Normal Type" could also be changed into Pointers etc. - also the data type itself can be selected



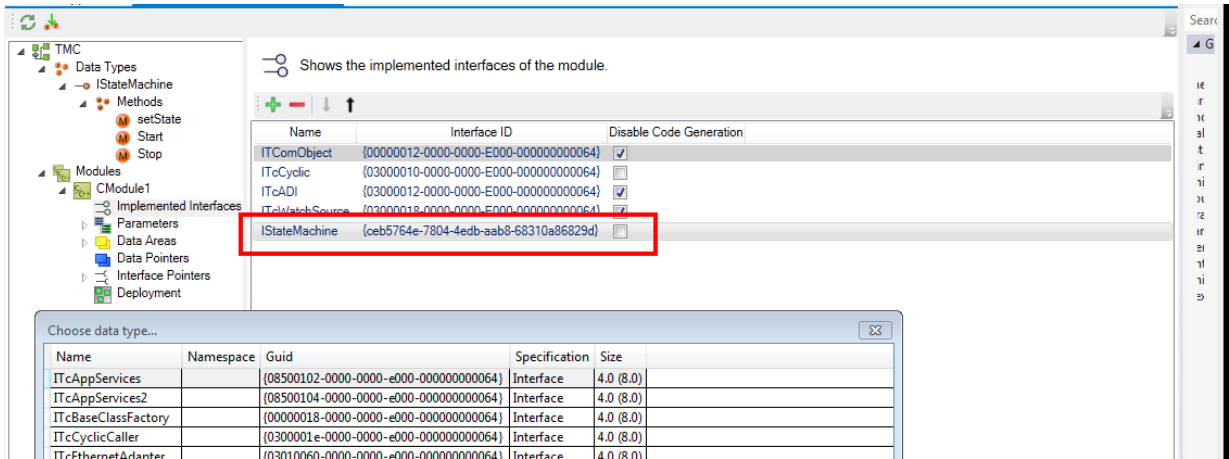
⇒ In this case "NewState" is the new name - the other settings are not changed



11. Repeating the step 2 "Add methods to interface" will list all methods - with the "up" / "down" buttons the methods can be reordered.



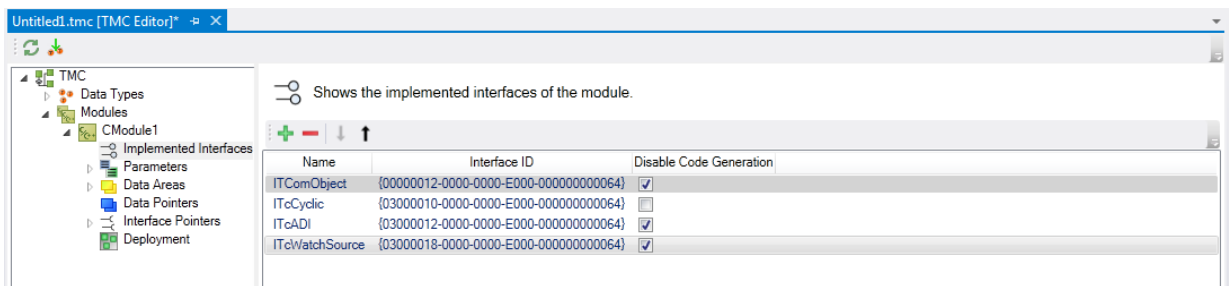
12. The interface is ready to be implemented by your module.



**Step 3: Add the new interface to "Implemented Interfaces"**

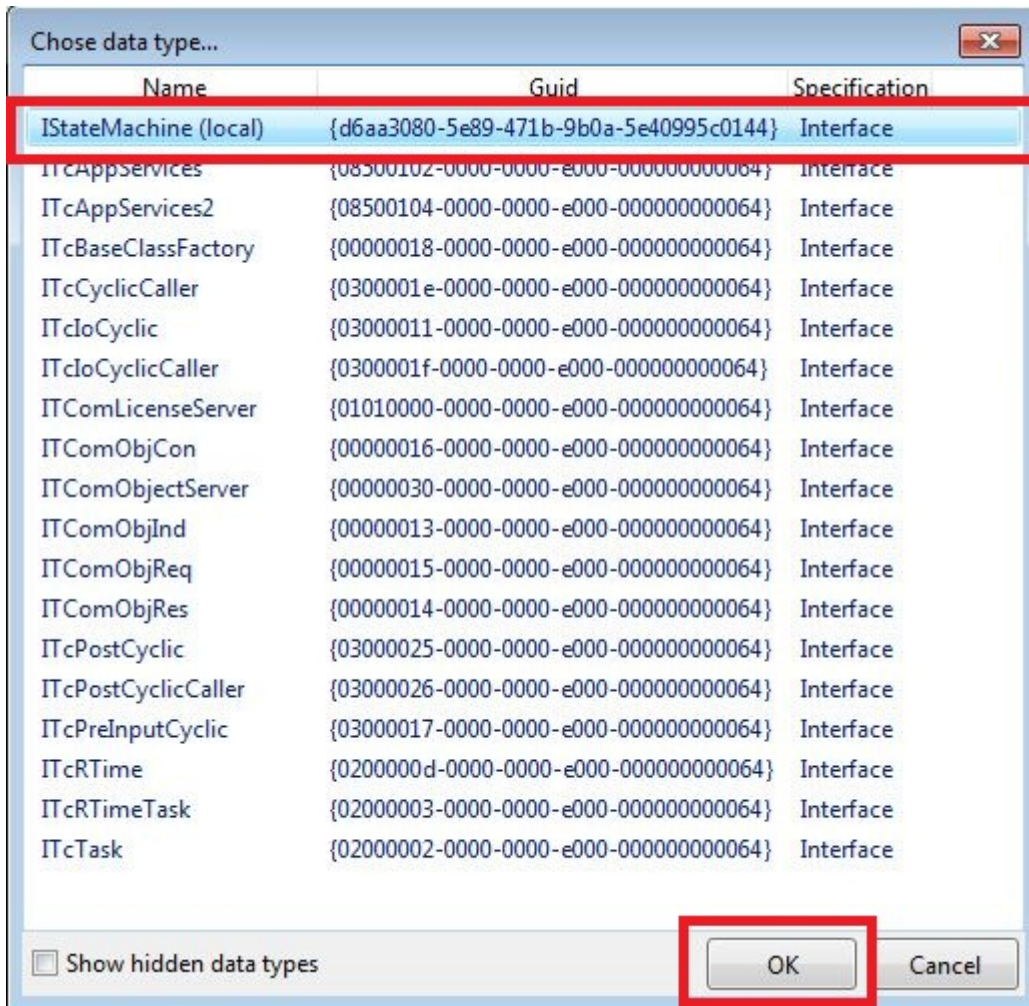
13. Select the module that is to be extended by the new interface - in this case select the destination "Modules->CModule1".

14. The list of implemented interfaces is extended by a new interface with "Add a new interface to the module" by clicking on the "+" button.

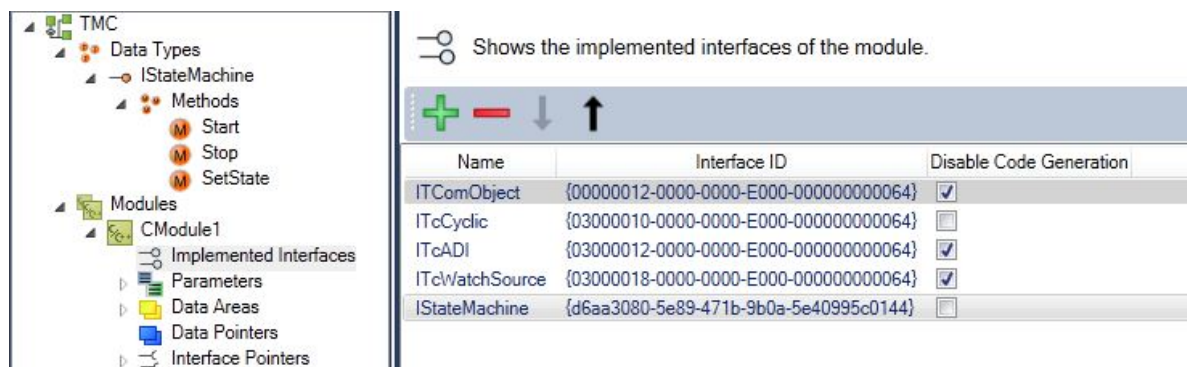




15. All available interfaces are listed - select the new template "IStateMachine" and end with "OK"



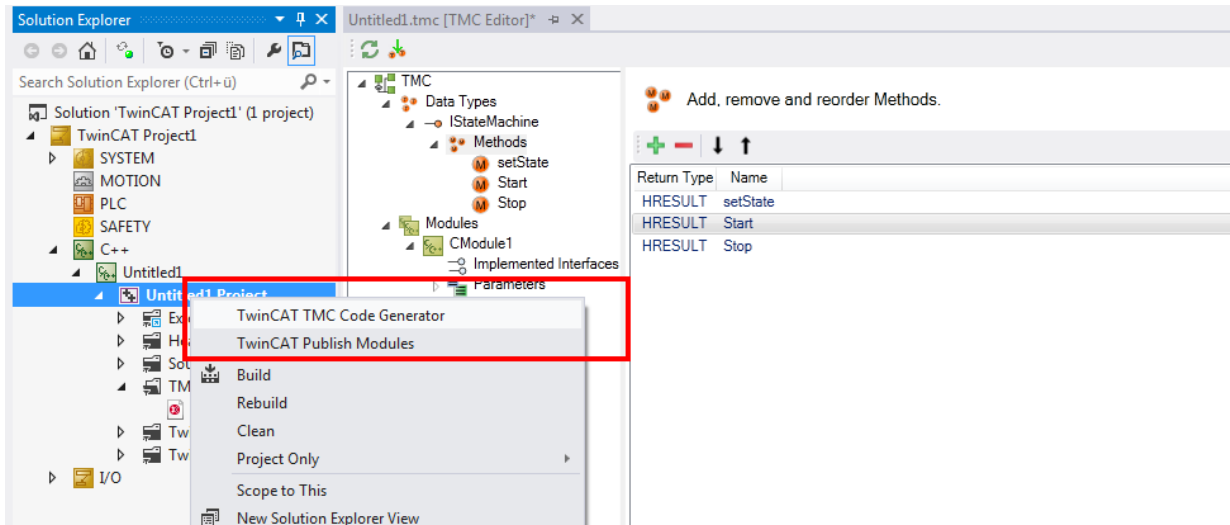
⇒ As a consequence, the new interface "IStateMachine" is now part of the module description.



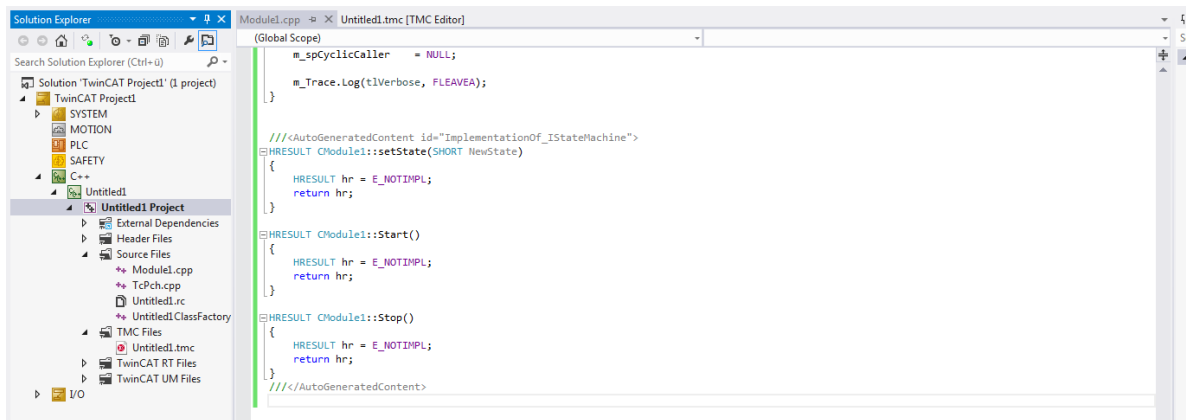


**Step 4: Start TwinCAT TMC Code Generator to create code of module description**

16. To generate the C/C++ code out of this module descriptions right click in the C/C++ project and select the "TwinCAT TMC Code Generator"



⇒ As a result the module "Module1" contains the new interfaces  
 CModule1: Start()  
 CModule1: Stop()  
 CModule1: SetState(SHORT NewState)



17. Finished - the custom code can now be added to this area.

**Optional change of the interface**

**i 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

**Edit the properties of the Data Type.**

**General properties**

Name:

Namespace:

Guid:

Specification:

**Choose data type**

Select:  ...

Description:

**Type Information**

Namespace:

Guid:

**Optional data type settings**

Size [Bits]:  x64 specific

C/C++ Name:

default:

x64 specific:

Unit:

Comment:

Hide sub items

Persistent (even if unused)

**Optional Defaults**

Value	Enum	String
Value	<input type="text" value="4"/>	
Min	<input type="text" value="1"/>	
Max	<input type="text" value="5"/>	

**Optional properties**

Name	Value	Description

**Datatype Hides**

Guid:

## General properties

**Name:** user-defined name of the data type

### NOTE

#### 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.

Example: an enumeration with a namespace "A":

 Edit the properties of the Data Type.

General properties	
Name	<input type="text" value="ASampleEnum"/>
Namespace	<input type="text" value="A"/>
GUID	<input type="text" value="{41d4a207-3a09-4316-9d89-0dd1881ab8c4}"/>
Specification	<input type="text" value="Enumeration"/>

The following code is generated:

```

//<AutoGeneratedContent id="DataTypes">
#if !defined(_TC_TYPE_41D4A207_3A09_4316_9D89_ODD1881AB8C4_INCLUDED_)
#define _TC_TYPE_41D4A207_3A09_4316_9D89_ODD1881AB8C4_INCLUDED_
enum A_ASampleEnum : SHORT {
One,
Two,
Three
};
#endif // !defined(_TC_TYPE_41D4A207_3A09_4316_9D89_ODD1881AB8C4_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 [▶ 98]:** create a user-defined array
- **Enumeration [▶ 99]:** create a user-defined enumeration
- **Struct [▶ 99]:** generate a user-defined structure
- **Interface [▶ 100]:** generate a new interface

### Choose data type

**Select:** Select data type – these could be base data types of TwinCAT or user-defined data types. There are data types defined, which are equivalent to the PLC data types (like TIME, LTIME etc.). Please see Data Types of the PLC for a detailed description.

**Description:** Define if the type is a pointer, a reference or a value by selecting one of

- Normal type
- Is pointer
- Is pointer to pointer
- Is pointer to pointer to pointer
- a reference

### Type Information

- **Namespace:** Defined of the 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

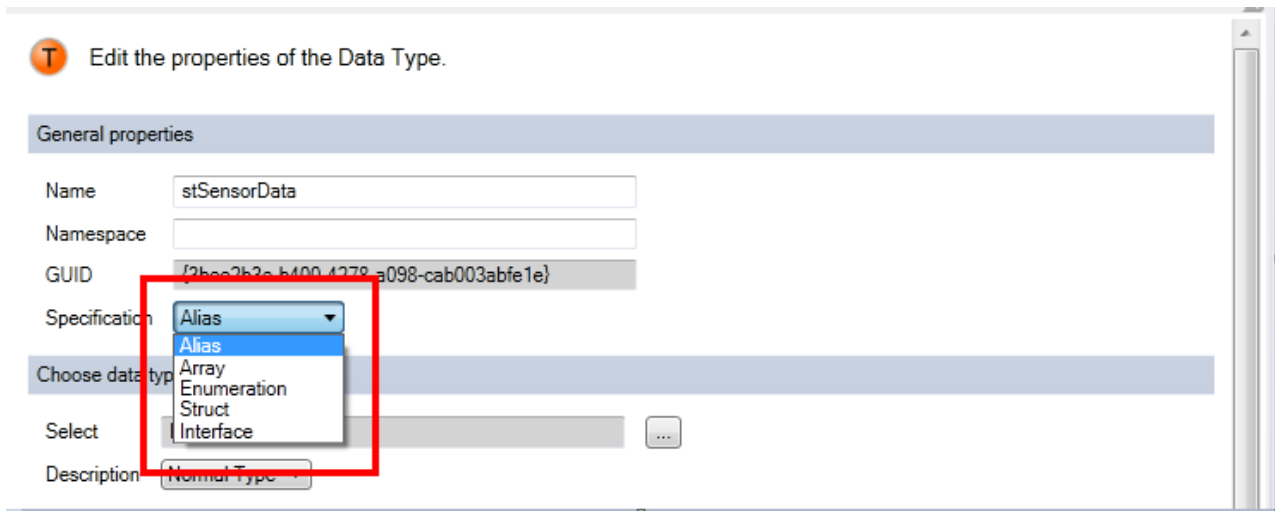
A table of name, value and description for annotating the data type.  
This information is provided within the TMC and also TMI files.  
TwinCAT functions as well as customer programs can use these properties.

### 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.

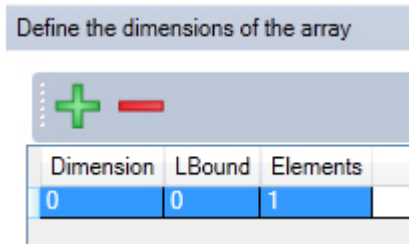
## 11.3.3.5 Specification

This section describes the Specification of data types.



### 11.3.3.5.1 Array

**Array:** Create a user specific array



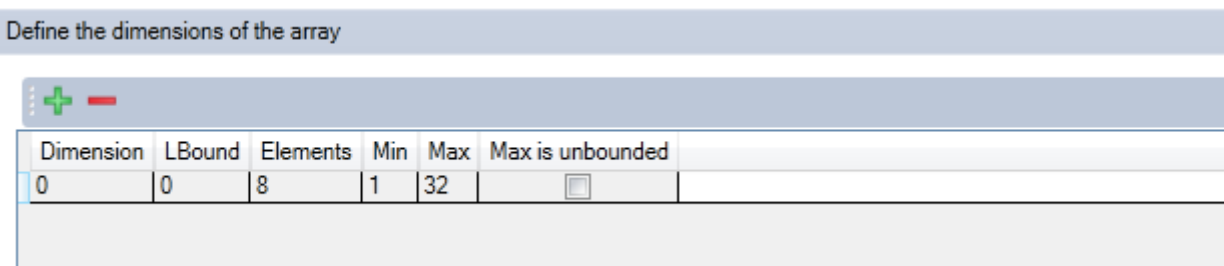
You get a new dialog to add (+) or remove (-) array elements.

**Dimension:** Dimension of the array

**LBound:** Left bound of the array (default = 0)

**Elements:** Amount of elements

### Dynamic Arrays for Parameters and Data Pointers



For [Parameters \[► 104\]](#) and [Data Pointers \[► 119\]](#) TwinCAT 3 can handle arrays, which are of dynamic length.

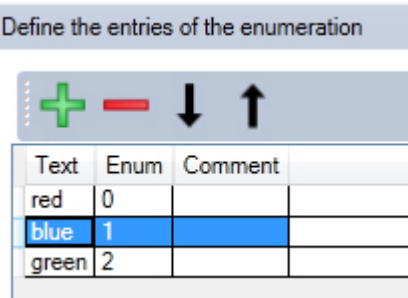
**Min:** Minimum size of the array

**Max:** Maximum size of the array

**Max is unbounded:** Indicates that there is no maximum limit of 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.

**NOTE**

**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

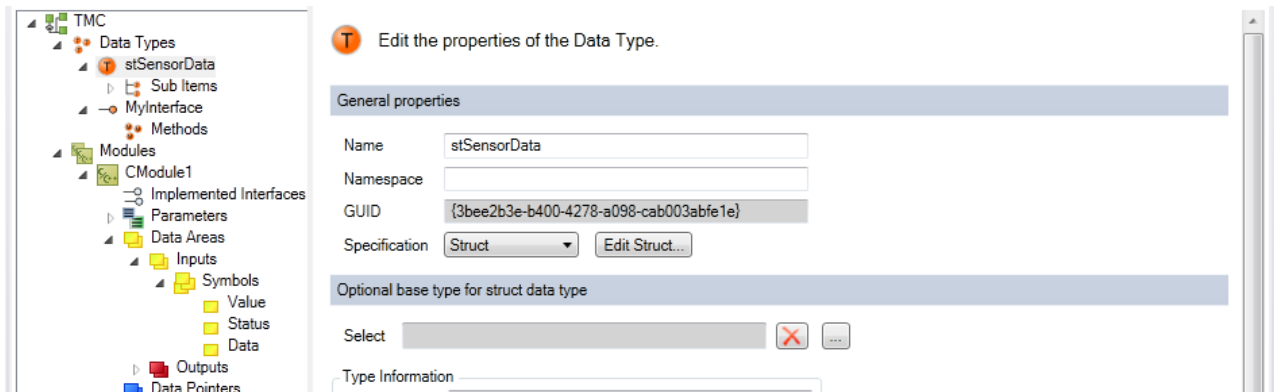
**Enum:** Suitable integer value

**Comment:** Optional comment

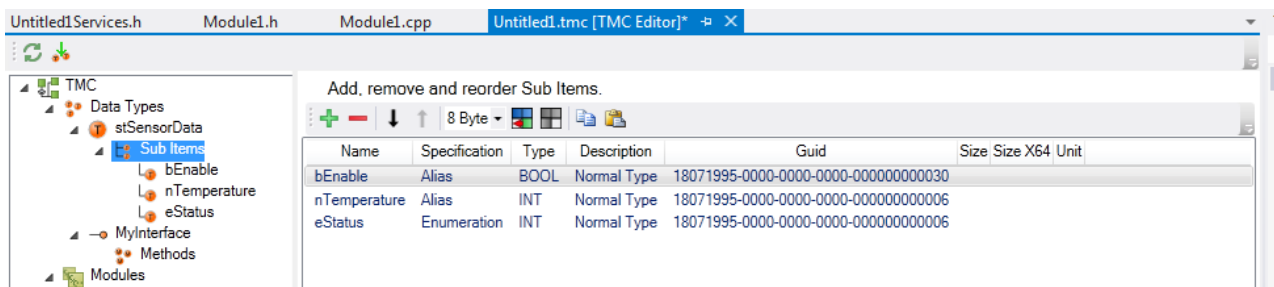
### 11.3.3.5.3 Struct

**Struct:** Create a user specific structure

Please select the “Sub Items” node or click the “Edit Struct” button to switch to this table:



You get a new dialog to add (+) or remove (-) an element. Use the arrows to adjust the ordering.



**Name:** Name of the element

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

**Type:** Type of variable

**Size:** Size and offset of the subitem.

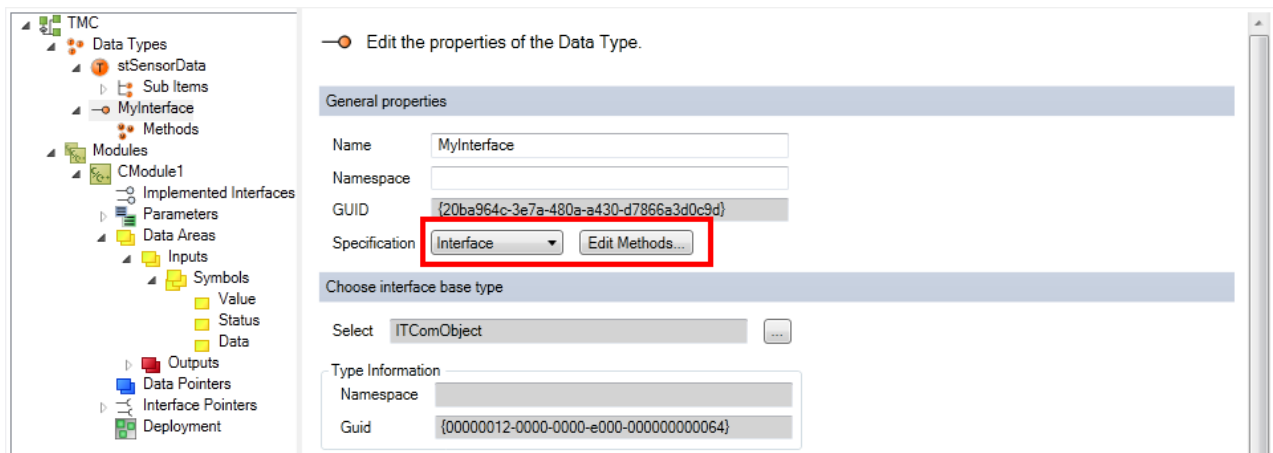
**Size X64:** Different size for x64 platform is additionally provided.

**Unit:** Optional unit

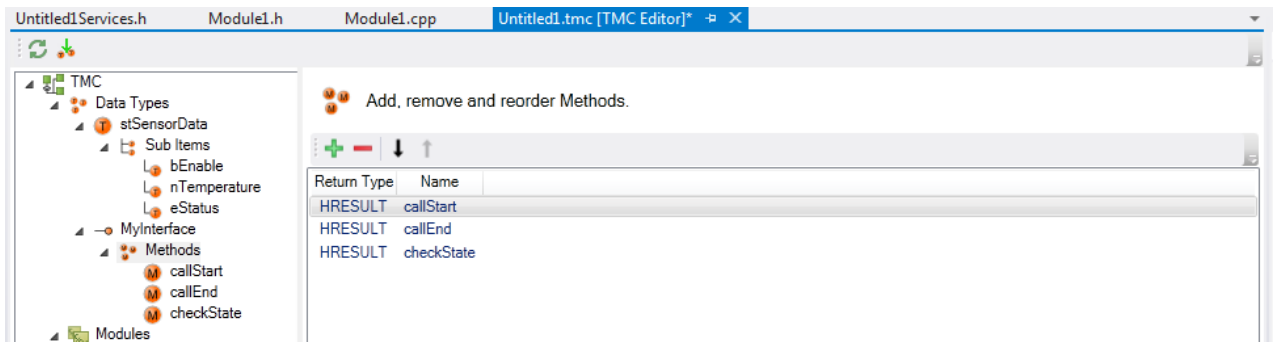
By selecting node of the data type or double-clicking on the table entry one would get to the details of the subitems configuration page. This is similar to the [Data type properties](#) [► 95].

### 11.3.3.5.4 Interfaces

**Interfaces:** Create a user specific interface.



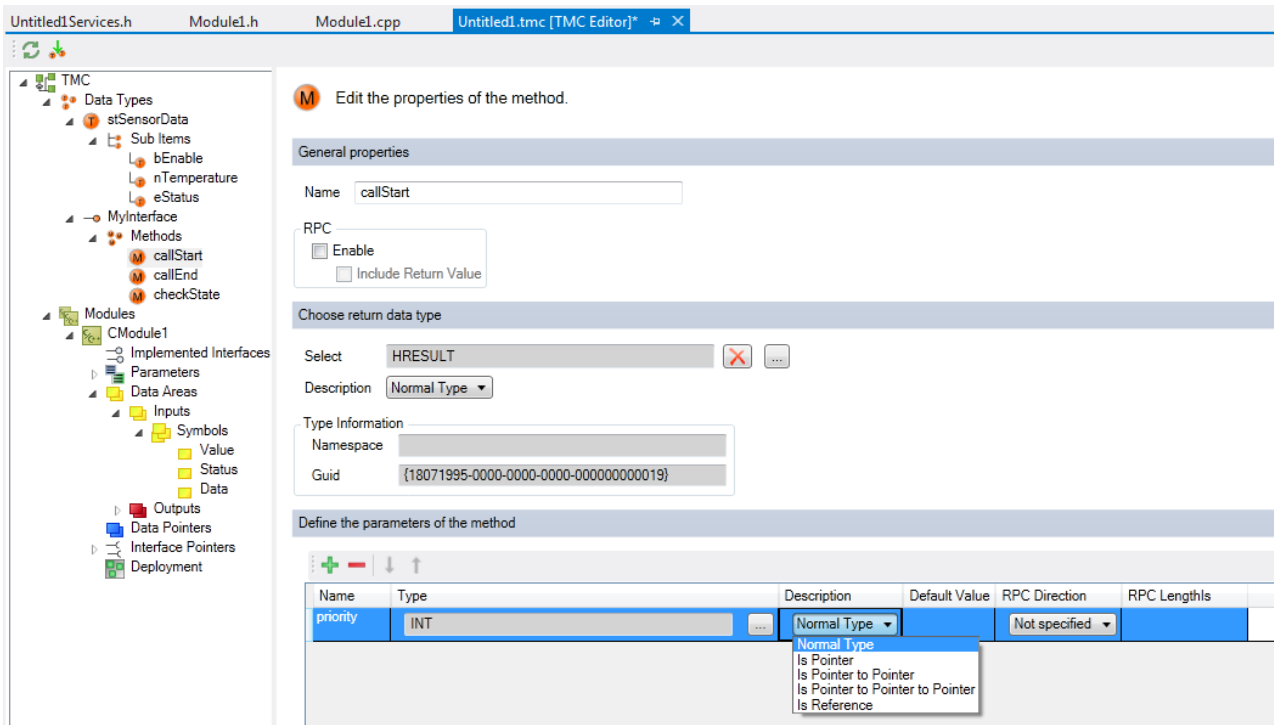
Please select the “Methods” node or click the “Edit Methods” button to switch to this table:



### Method Parameters

Select the node of the method or double click to see the details of a method:





**Name:** The name of the method

**RPC enable: Enable** “Remote Procedure Calls” from outside for this method.

- **Include Return Value:** Enable propagation of method’s return value

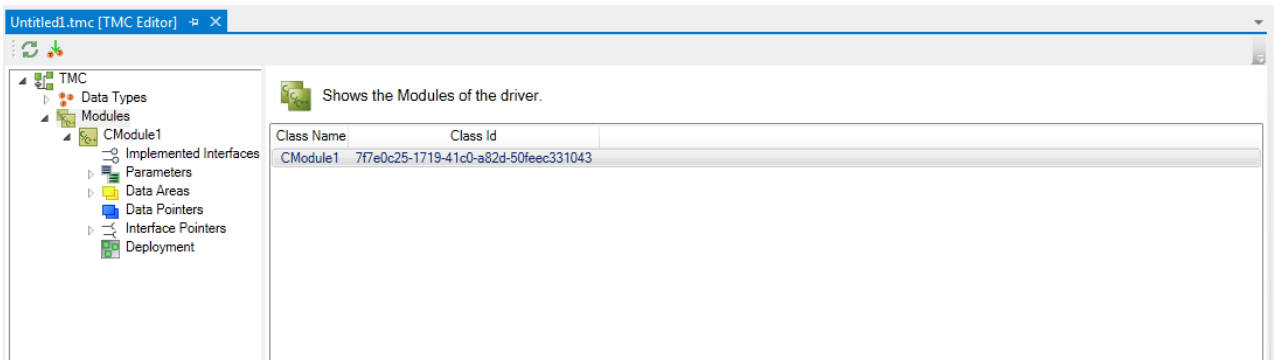
Fields are equivalent to the [Data type properties \[▶ 95\]](#).

**Define the parameter of the method**

- Name
- Type: Known from the [Data type properties \[▶ 95\]](#)
- Description: Known from the [Data type properties \[▶ 95\]](#)
- Default Value: Default value of this parameter; only numbers are allowed
- RPC-Direction: Equivalent to PLC function blocks each parameter could either be IN, OUT or INOUT. Additionally, NONE could be defined to ignore this parameter on Remote Procedure Calls.

### 11.3.4 Modules

**Modules:** Shows modules of the driver

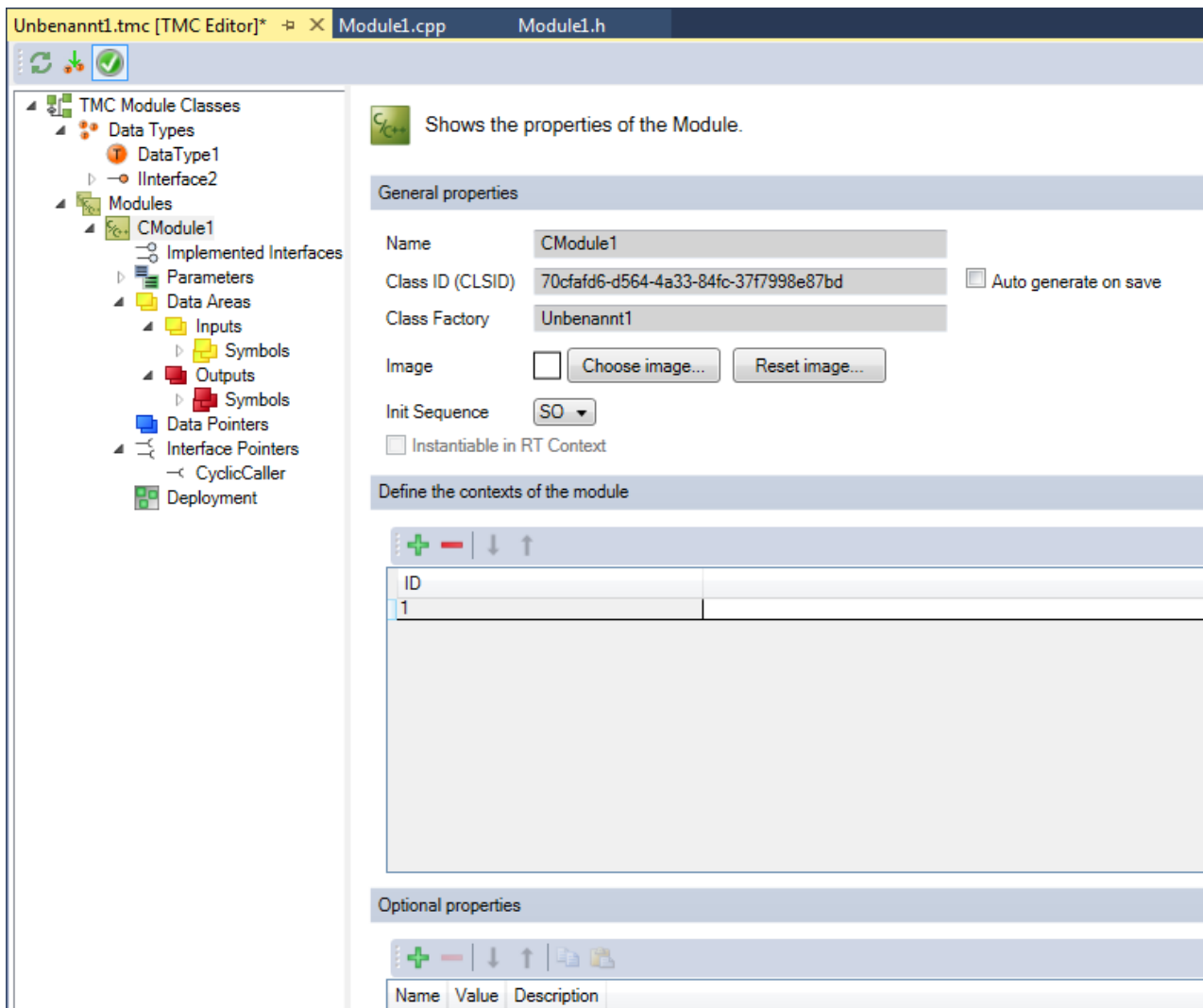


**Class Name:** Name of the module

**Class ID:** Unique ID of the module

### Modules 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

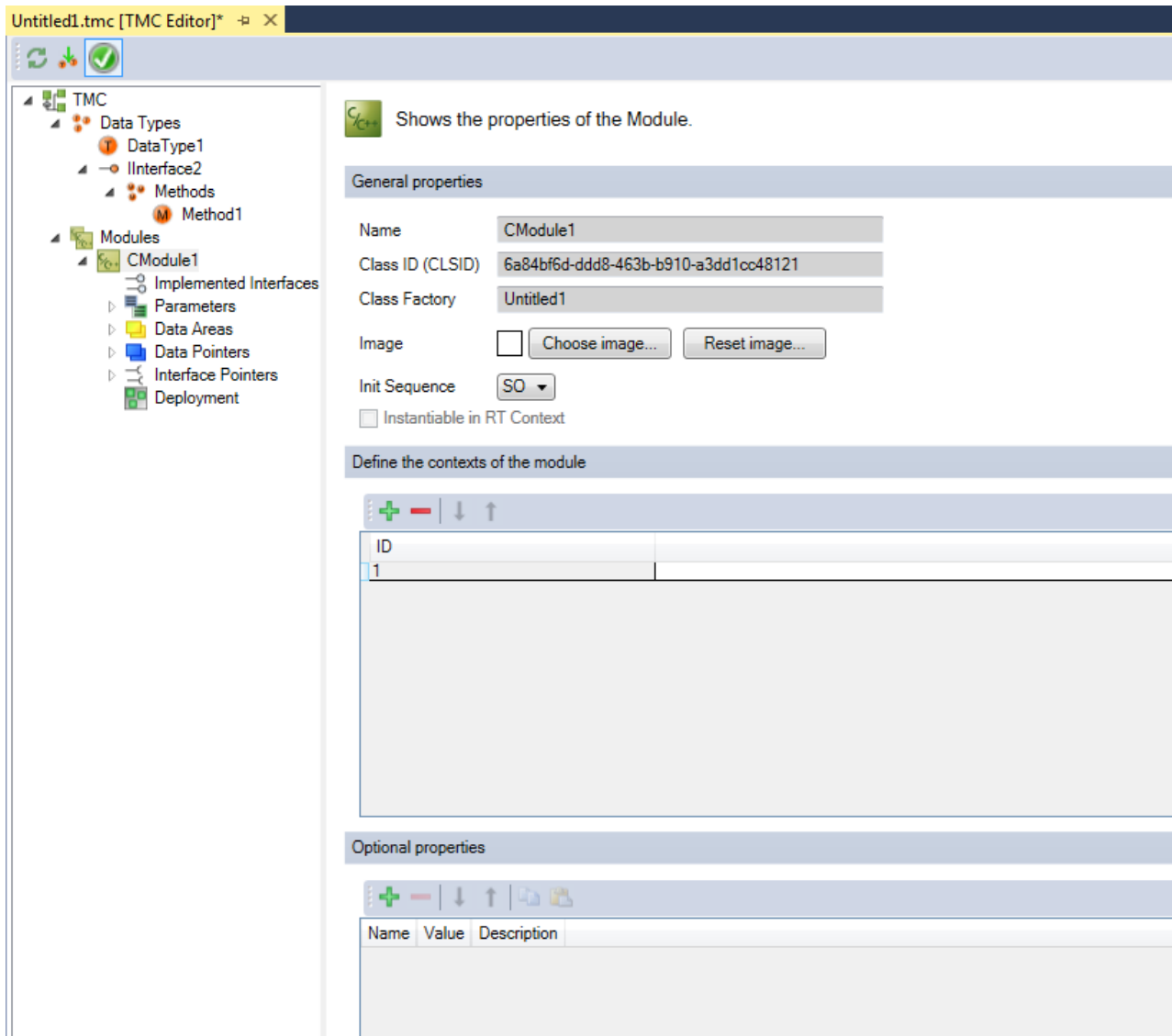
**Auto generate on save:** Enables TwinCAT to generate the ClassID via the module parameters during saving. If the ClassID changes during import of the binary modules, the corresponding ClassIDs have to be adjusted. Thus, TwinCAT can detect the interface change.

**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 \[▶ 125\]](#) 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](#) [▶ 77]



**Define the contexts of the module**

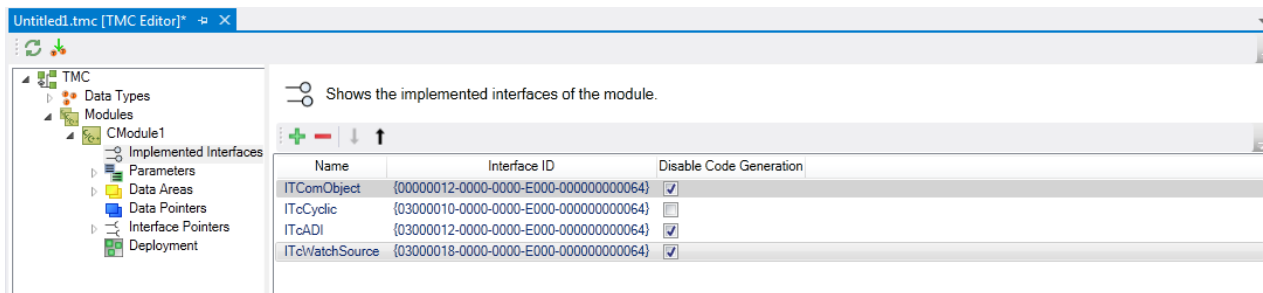
You can add (+) or remove (-) contexts for the module. Use the arrows to adjust the ordering. Context Id should be a non-zero integer.

**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:** Manipulate and view the implemented interfaces of the module

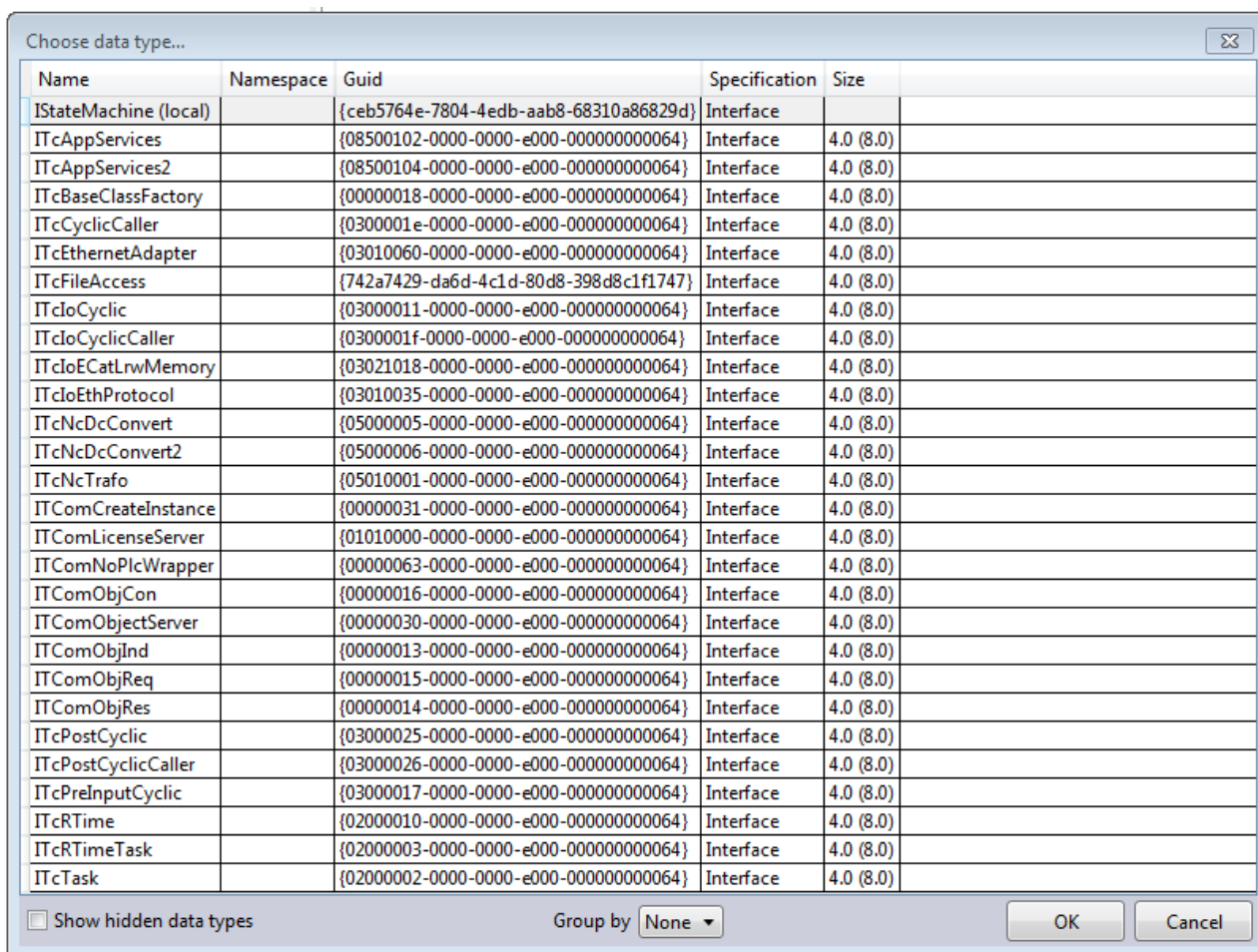


**Name:** Name for the interface

**Interface ID:** Unique ID of the interface

**Disable Code Generation:** Switch to enable/disable the code generation

You can add (+) or remove (-) contexts for the module. Use the arrows to adjust the ordering.



### 11.3.4.2 Parameters

A TcCOM module instance is defined through various parameters.

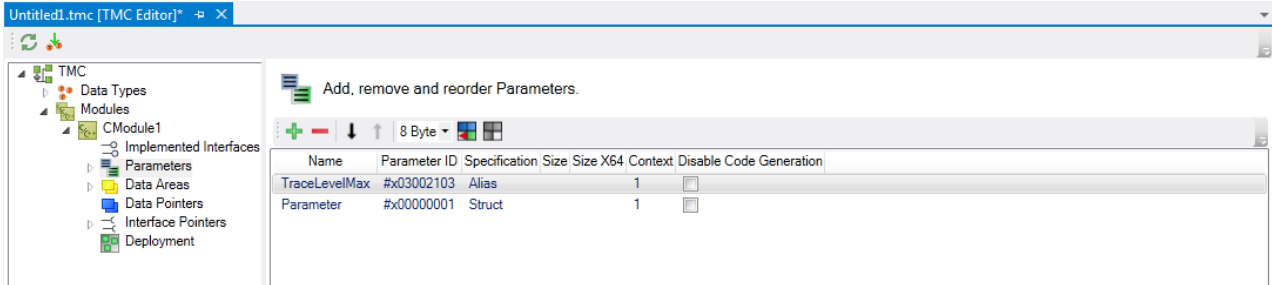
TwinCAT supports three types of parameter IDs (PTCID) in the section "Configure the parameter ID".

- "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 \[▶ 126\]](#) 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



**Icon**



**Function**

Add a new parameter



Deletes the selected type



Move the selected item one position down



Move the selected item one position up



Select byte alignment



Align selected data type



Reset the data layout of the selected data type

**Name:** Name for the interface

**Parameter ID:** Unique ID of the parameter

**Specification:** Data type of the parameter

**Size:** Size of the parameter. For x64 different sizes are possible.

**Context:** Context ID of the parameter

**Disable Code Generation:** Switch to enable/disable the code generation

### 11.3.4.2.1 Add / modify / delete parameters

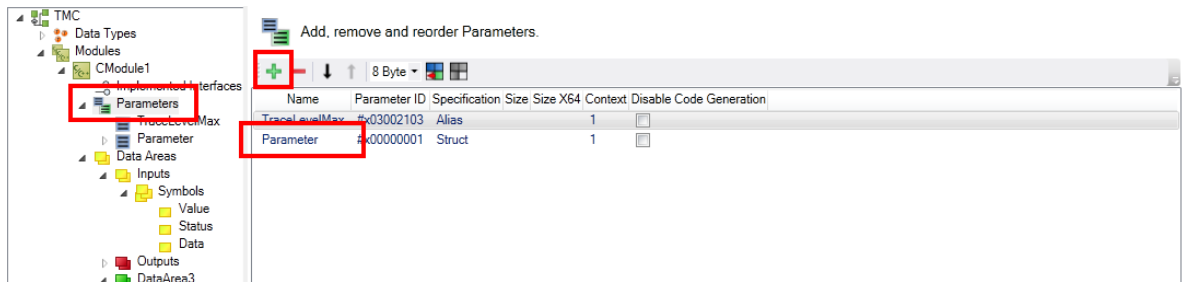
The TwinCAT Module Class (TMC) Editor allows adding / modifying and deleting features and functionalities of a TwinCAT class.

This article describes how to

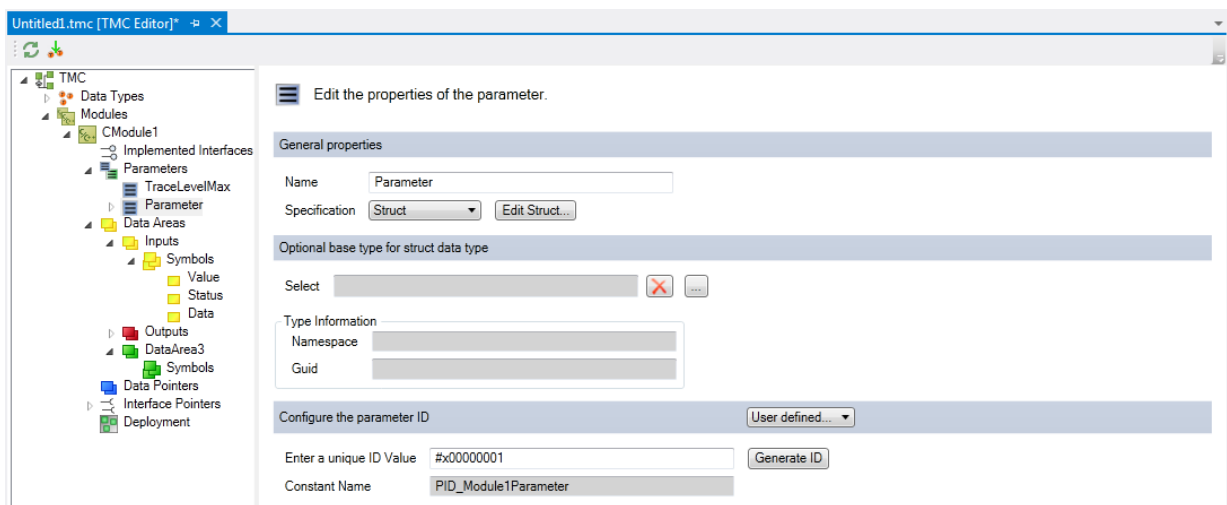
- [Step 1: Create new parameter \[▶ 106\]](#) in the TMC file
- [Step 2: Start TwinCAT TMC Code Generator \[▶ 107\]](#) to create code of module description in the TMC file
- [Step 3: Statemachine transitions \[▶ 108\]](#)

### Step 1: Create new parameter

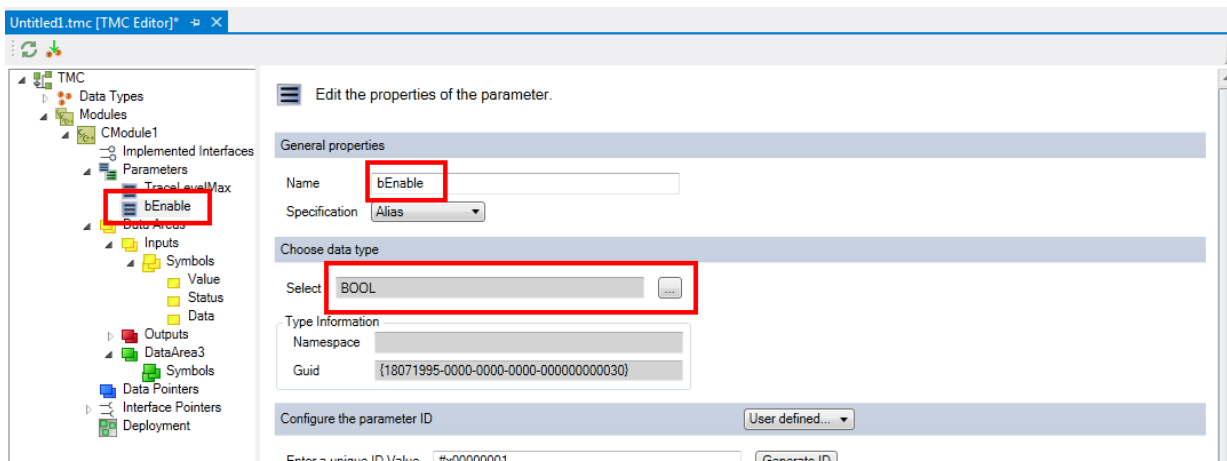
1. After starting the TMC editor, select the target **"Parameters"**.
2. List of parameters will be extended with a new parameter by clicking the "+" button "Add a new parameter".
  - ⇒ As a result a new **"Parameter"** is listed as new entry:



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



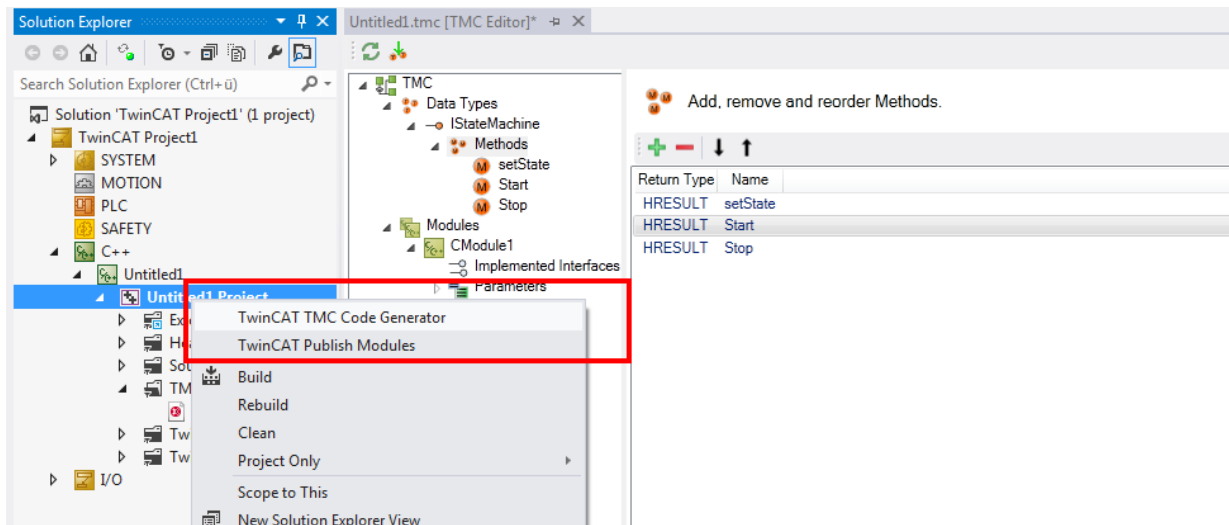
4. Configure parameter comparable to the [Data Types \[▶ 83\]](#)
5. Renaming this to a more usefully name - in this sample **"bEnable"** and select the data type "BOOL".



6. Save your changes of the TMC file.

**Step 2: Start TwinCAT TMC Code Generator to create code of module description**

7. Right Click on your project file and select "TwinCAT TMC Code Generator" to get the parameter in your source code:



⇒ You will see the parameter declaration in the module header file "Module1.h"

```

///<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_ITWATCHSOURCE(CModule1)

// Set parameters of CModule1
BEGIN_SETOBJPARA_MAP(CModule1)
    SETOBJPARA_DATAAREA_MAP()
    ///<AutoGeneratedContent id="SetObjectParameterMap">
    SETOBJPARA_VALUE(PID_TraceLevel, m_TraceLevelMax)
    SETOBJPARA_VALUE(PID_Module1bEnable, m_bEnable)
    SETOBJPARA_ITPTR(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_TraceLevel, m_TraceLevelMax)
    GETOBJPARA_VALUE(PID_Module1bEnable, m_bEnable)
    GETOBJPARA_ITPTR(PID_Ctx_TaskOid, m_spCyclicCaller)
    ///</AutoGeneratedContent>
END_GETOBJPARA_MAP()

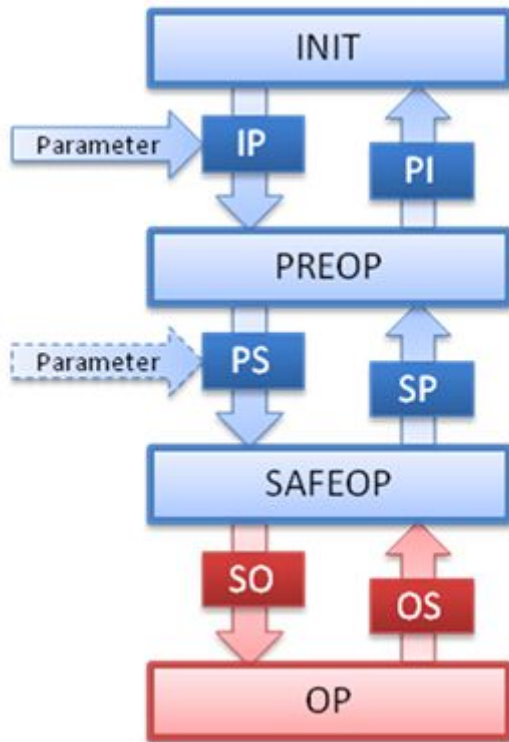
```

If you add an additional parameter, use the TwinCAT TMC Code Generator again.

### Step 3: State machine transitions

Note the different state transitions of your [state machine](#) [► 39]:

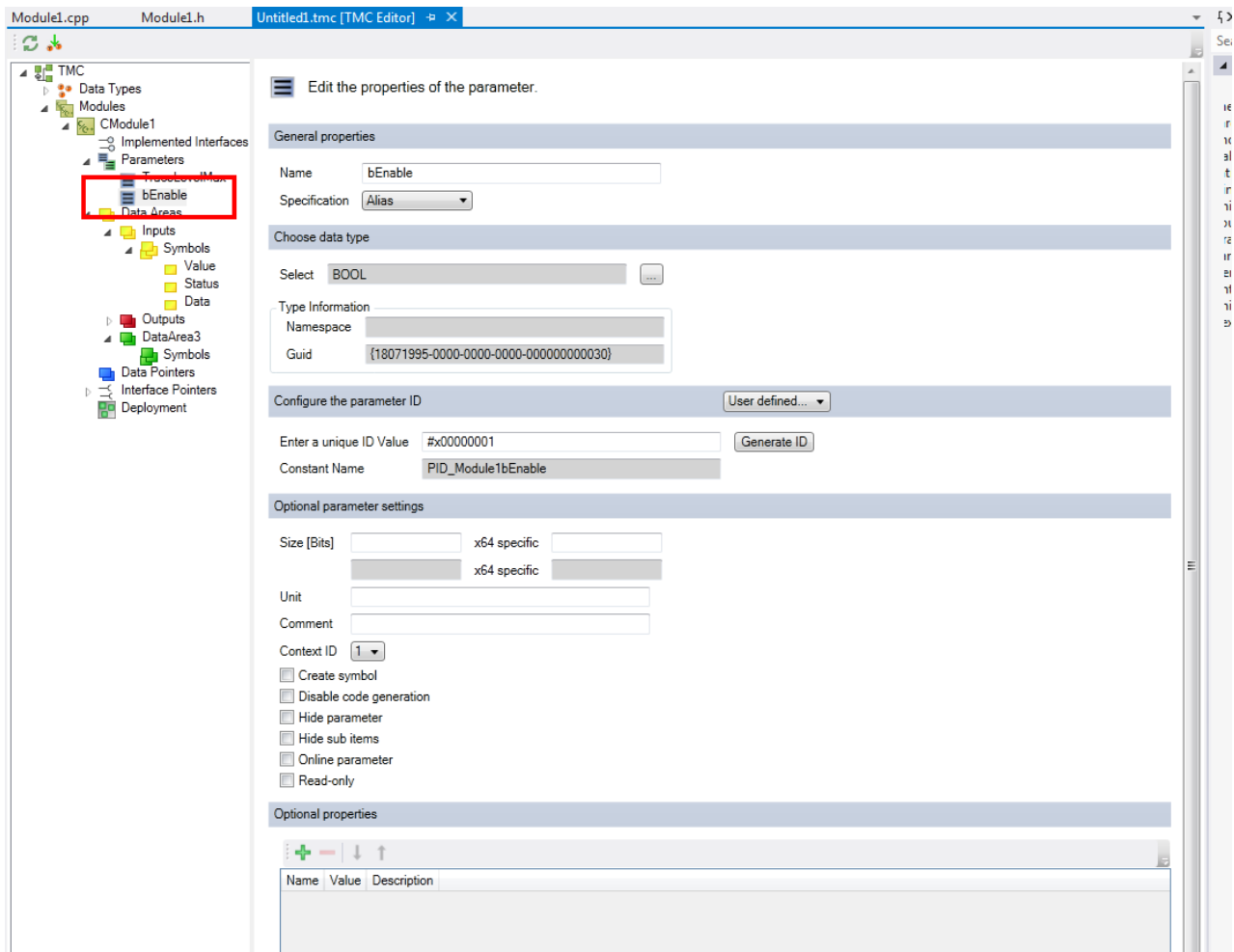




The parameters are specified during the transition Init->Preop and perhaps Preop->Safeop.

### 11.3.4.2.2 Parameter properties

**Parameter properties:** Edit the properties of the parameter



### General properties

**Name:** Name for the interface

**Specification:** Data type of the parameter. Please see Specification.

### Choose data type

**Select:** Select data type

### Type Information

- **Namespace:** User-defined namespace for the data type
- **GUID:** Unique ID of the data type

**Enter a unique ID Value:** Enter a unique ID Value. See [Parameters](#) [► 104].

**Constant Name:** Source code name of the parameter ID

### Optional parameter settings

**Size [Bits]:** Calculated size in bits (white boxes) and in “byte.bit” notation (grey boxes). For x64 special size configuration is provided.

**Unit:** A unit of the variable

**Comment:** Comment, which will be visible e.g. in the instance configurator

**Context ID:** Context, which is used when the parameter is accessed by ADS

**Create Symbol:** Default setting for ADS symbol creation.

**Disable Code Generation:** Switch to enable/disable the code generation

**Hide parameter:** Switch to hide/unhide parameter in system manager view

**Hide sub items:** If data type has subitems, system manager will not provide access to the subitems. This should be used for example on large arrays.

**Online parameter:** Set as online parameter

**Read-only:** Switch to read-only for system manager access

**Optional Properties**

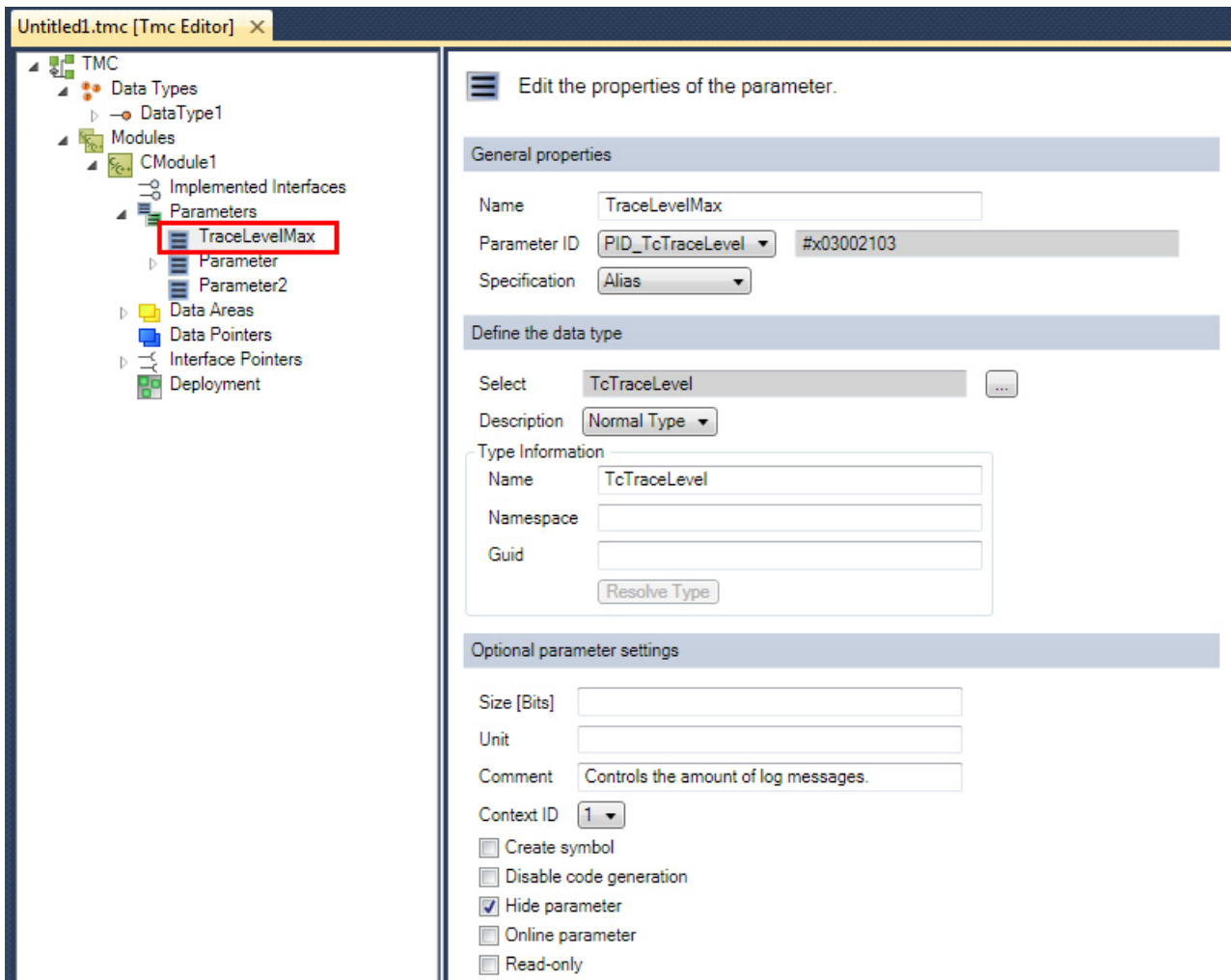
A table of name, value and description for annotating the parameter.  
 This information is provided within the TMC and also TMI files.  
 TwinCAT functions as well as customer programs can use these properties.

**11.3.4.2.3 TraceLevelMax**

**TraceLevelMax:** Parameter which defines the trace level.  
 This is a predefined parameter provided by most TwinCAT module templates (except for the empty TwinCAT module template).

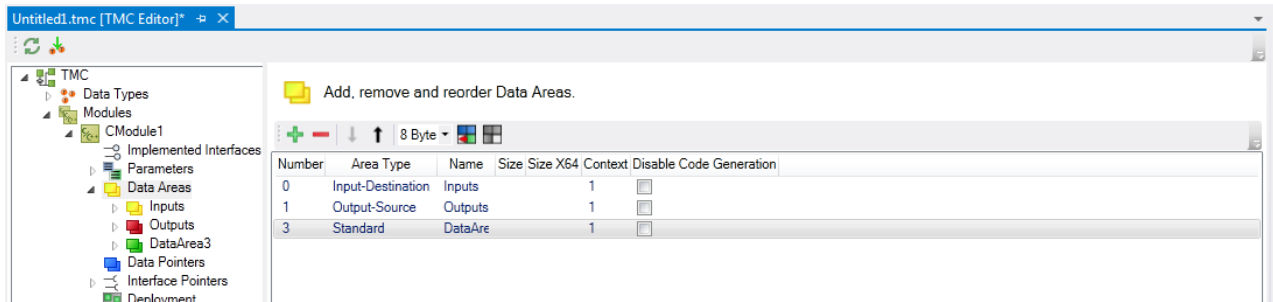
Settings for this parameter should not be changed.

See [Module messages for the Engineering \(logging / tracing\) \[► 193\]](#)



### 11.3.4.3 Data Areas

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



#### Symbol



8 Byte ▾



#### Function

Add a new data area

Delete 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

#### NOTE

##### 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

**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 the code generation

#### 11.3.4.3.1 Add / modify / delete data areas and variables

The TwinCAT Module Class (TMC) Editor allows adding / modifying and deleting features and functionalities of a TwinCAT class.

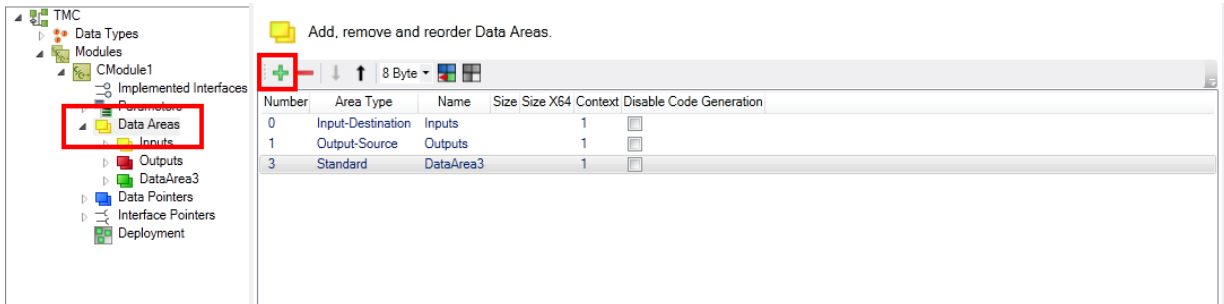
This article describes how to

- create a new data area in the TMC file
- [create new variables \[► 118\]](#) in a data area

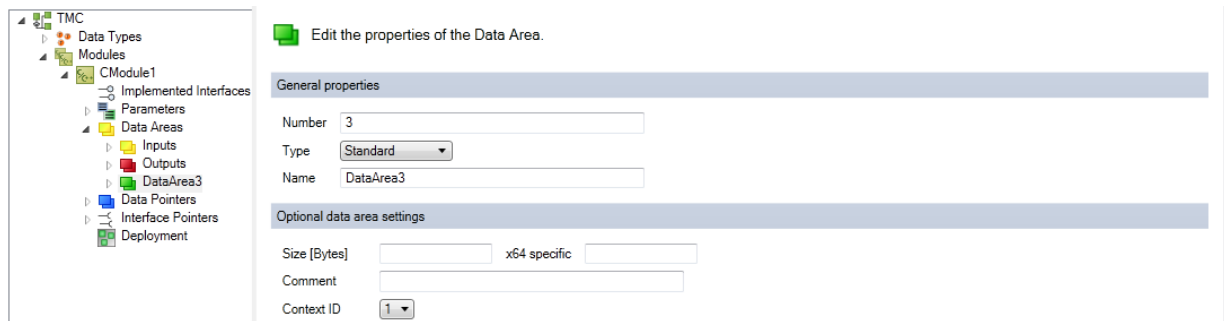
- [modify e.g. the name or data type \[▶ 118\]](#) of existing variables in the TMC file
- [delete existing variables \[▶ 119\]](#) in the TMC file

**Create a new data area**

1. After starting the TMC editor, select the node “Data Areas” of the module
2. By clicking on the + button a new data area is created



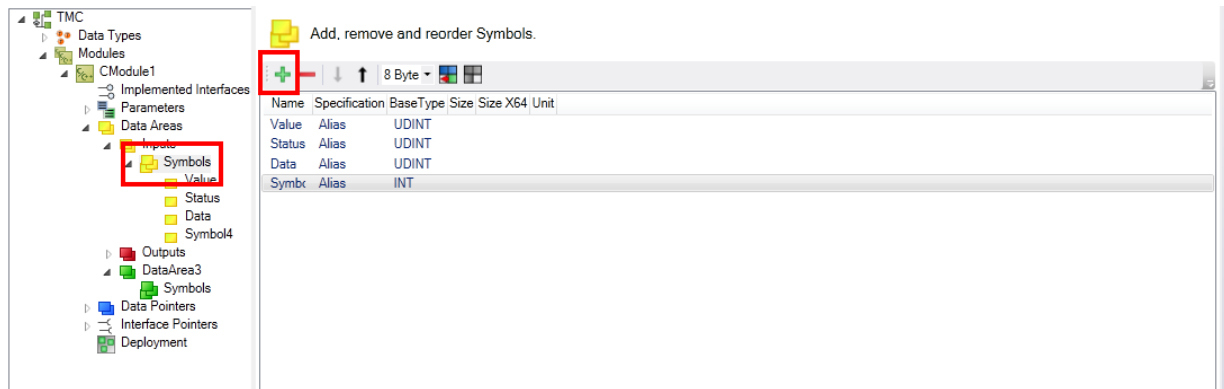
3. Double click on the table or click on the node to get to the properties of the data area



4. Rename the data area

**Create a new variable**

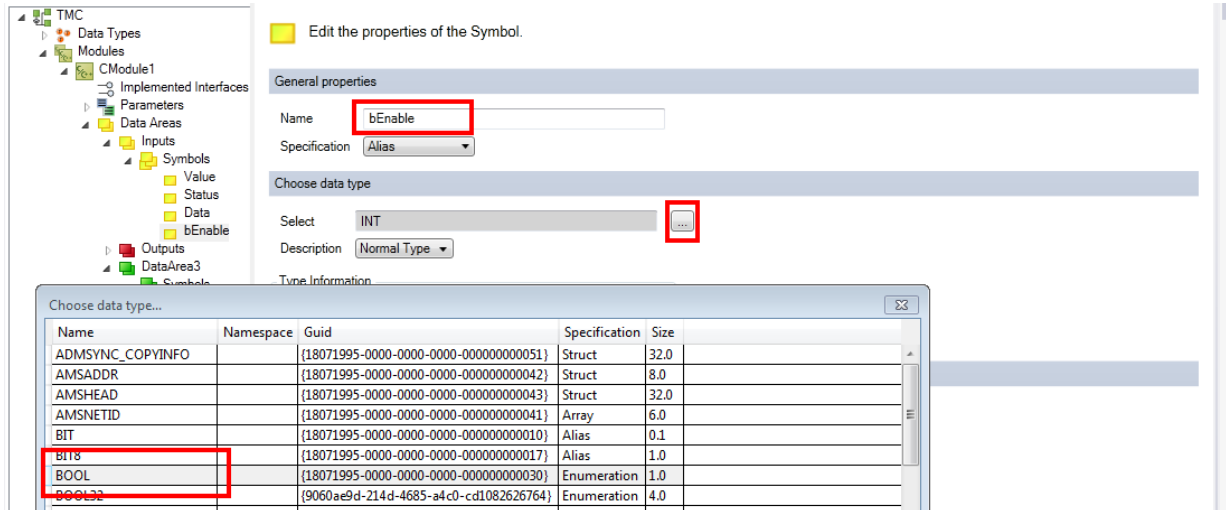
5. Select the subnode “Symbols” of the data area.
6. This data area could be extended with a new variable by clicking the "+" button. As a result "Symbol4" is listed as new entry.



**Modify name or data type of existing variables**

7. Select subnode "Symbol4" or double click on the line. The variable properties will open

8. Enter new name e.g. "bEnableJob" and change type to BOOL

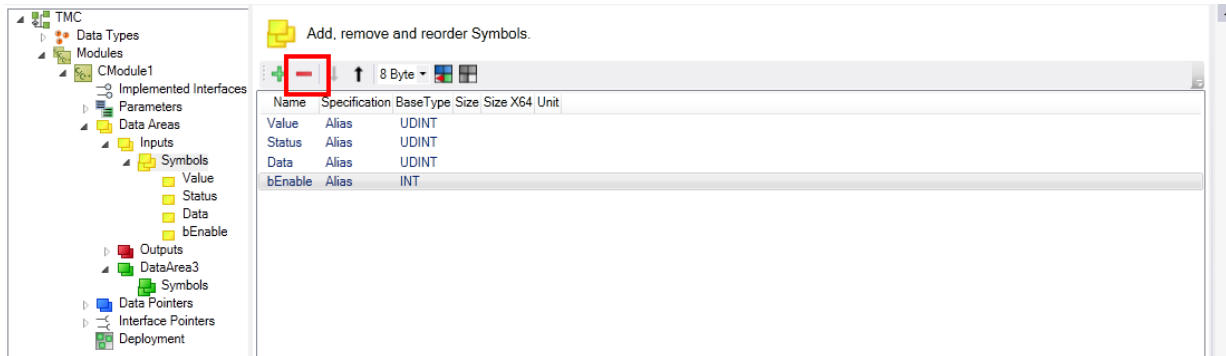


⇒ Finally, the new variable "bEnableJob" was created in the data area "Input".

**Note** Don't forget to rerun the TMC Code generator

### Delete existing variables

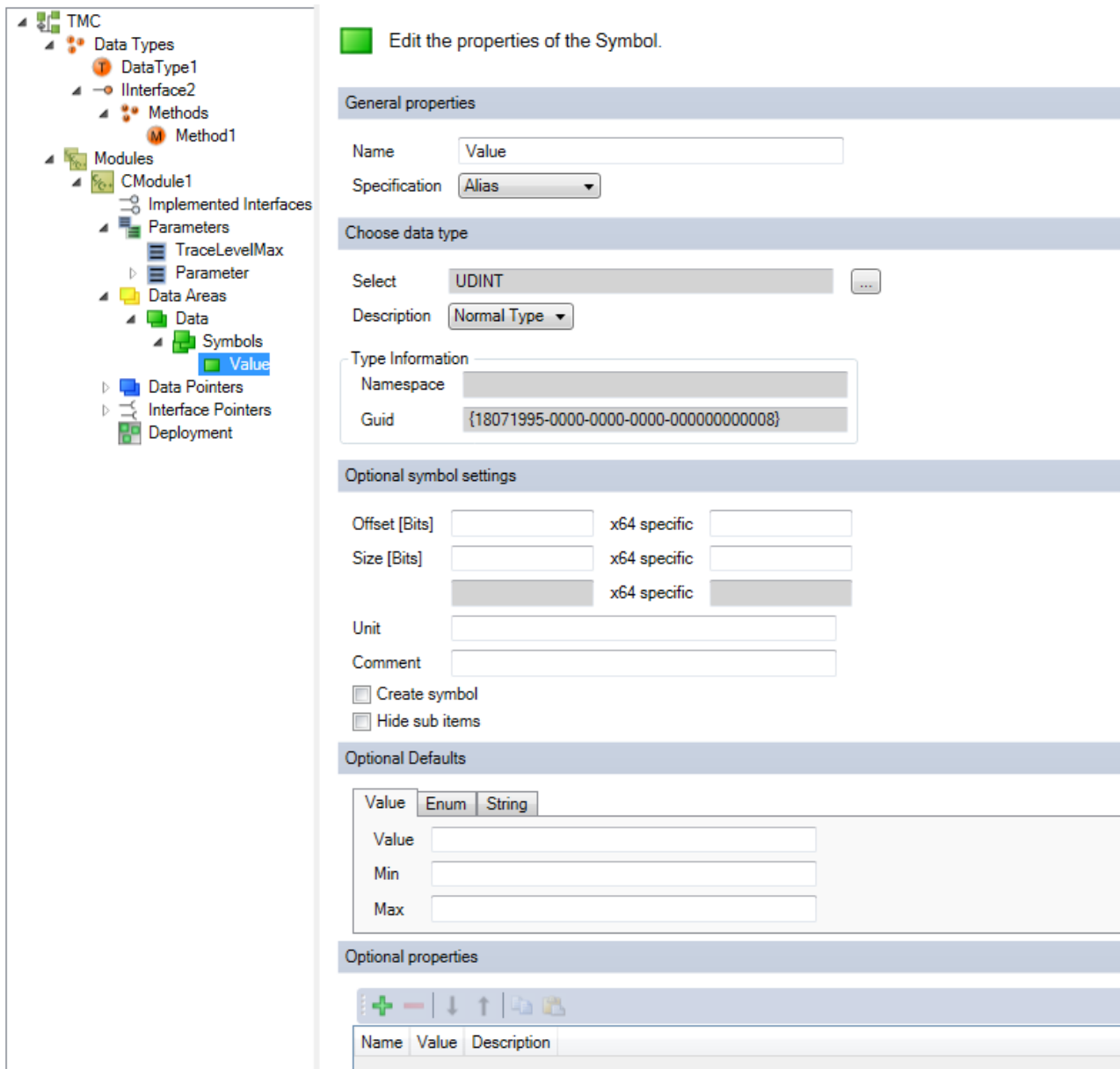
1. To delete existing variables in the data area select the variable and click on delete icon:  
In this demo select "MachineStatus1" and click "Delete symbol"



2. Rerun the TMC Code generator

### 11.3.4.3.2 Data Areas Properties

**Data Areas Properties:** Dialog to edit the data area properties



**General properties**

**Number:** Number of the data area

**Type:** Defines the purpose and the location of data area:

Linkable data areas in the System Manager:

- Input-Source
- Input-Destination
- Output-Source
- Output-Destination
- Retain-Source (for use with NOV-RAM memory, see [annex \[▶ 304\]](#))
- 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. For x64 special size configuration is provided

**Comment:** Optional a comment, which will be visible e.g. in the instance configurator

**Context ID:** Context ID of all symbols of this data area. Used to determine the mapping.

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

**Create Symbol:** Default setting for ADS symbol creation

**Disable Code Generation:** Switch to enable/disable the code generation

#### **Optional Defaults**

Depending on data type the default could be defined.

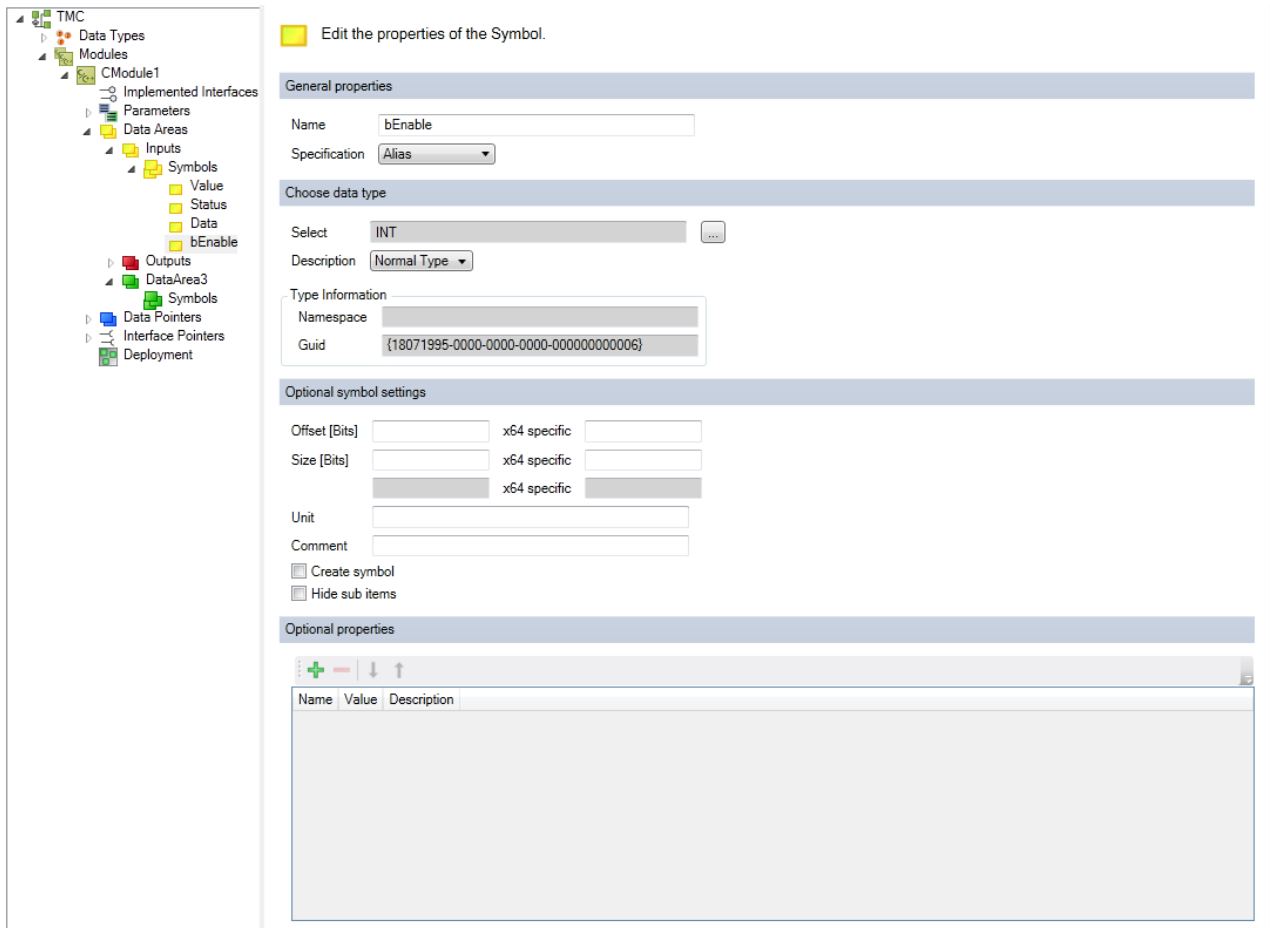
#### **Optional Properties**

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 to edit the symbols of a data area





**General properties**

**Name:** Name for the symbol

**Specification:** Data type of the symbol, see [Data type properties \[► 95\]](#)

**Choose data type**

**Select:** Select data type – these could be base data types of TwinCAT or user-defined data types

**Description:** Define if the type is

- Normal type
- Is pointer
- Is pointer to pointer
- Is pointer to pointer to pointer
- a reference

**Type Information**

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

**Optional data type settings**

**Offset [Bits]:** Offset of the symbol within the data area. Different offset for x64 platform can be set.

**Size [Bits]:** Size in bits, if specified. Different size for x64 platform can be set.

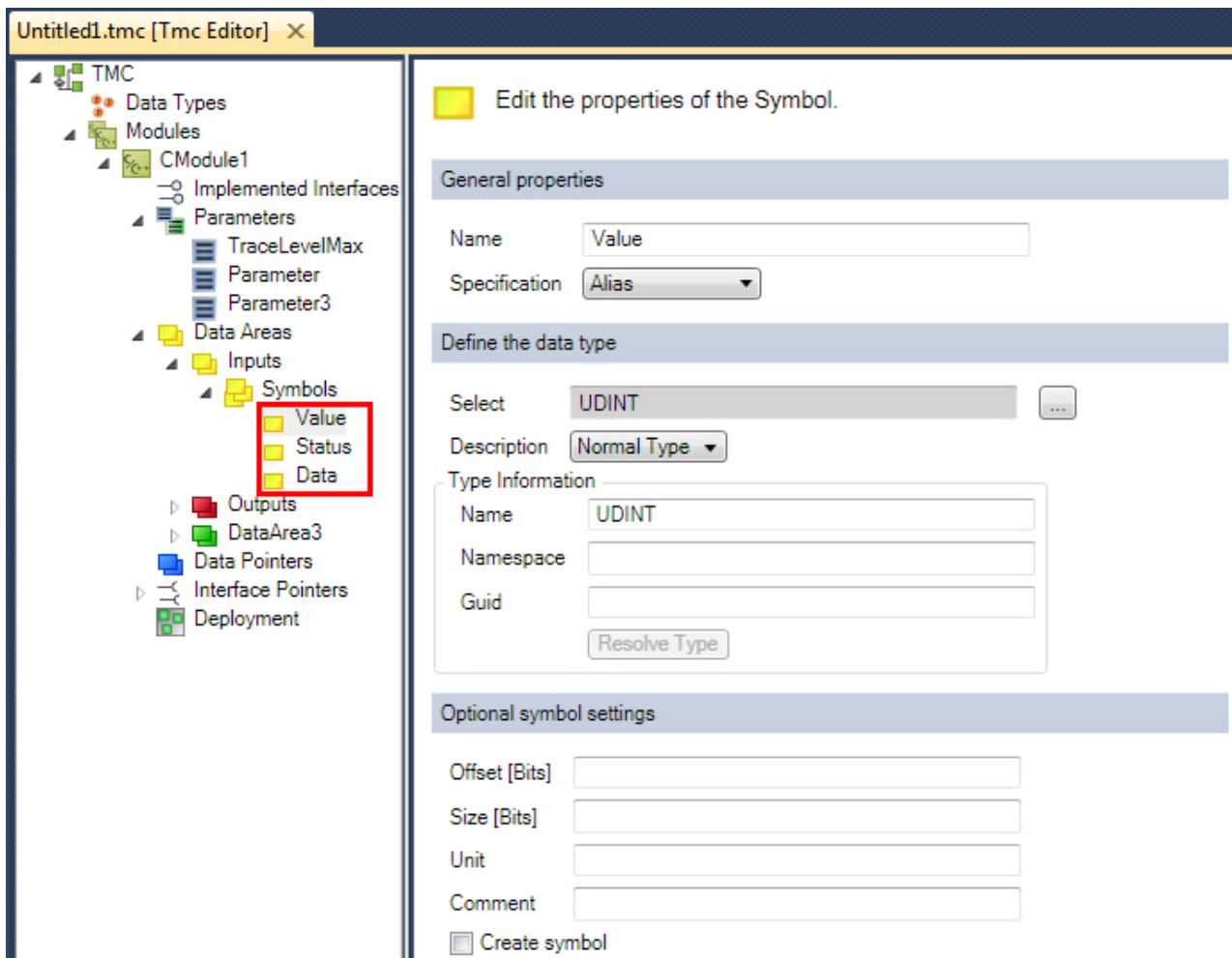
**Comment:** Optional a comment, which will be visible e.g. in the instance configurator

**Create Symbol:** Default setting for ADS symbol creation

**Hide sub items:** If variable has subitems, system manager will not provide access to the subitems. This should be used for example on large arrays.

## TwinCAT Module Class Editor - Data Areas Symbols Properties

**Data Areas Symbols Properties:** Dialog to edit the data area symbols properties



### General Properties

**Name:** Name for the interface

**Specification:** Data type of the parameter

Available specifications are:

- **Alias:** Create an alias of a default data type (e.g. INT)
- **Array:** Create a user specific array
- **Enumeration:** Create a user specific enum
- **Struct:** Create a user specific structure
- **Interface:** Create a new interface

### Define the data type

**Select:** Select data type

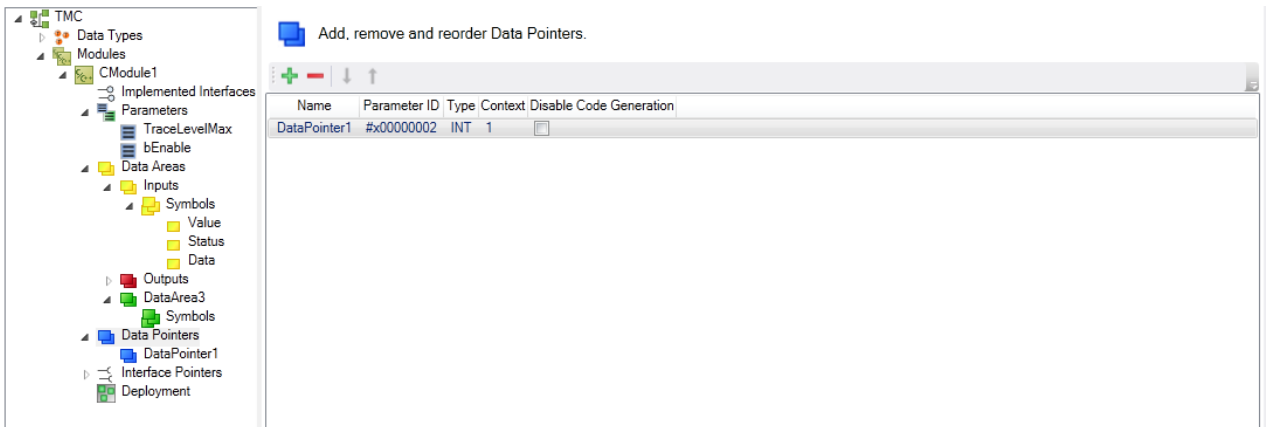
**Description:** Define description

**Type Information**

- Name:** Name of the selected default type
- Namespace:** User-defined namespace for the data type
- GUID:** Unique ID of the data type
- Optional data type settings**
- Offset [Bits]:** Memory offset
- Size [Bits]:** Calculated size in bits
- Unit:** Optional
- Comment:** Optional
- Create symbol:** Default setting for ADS symbol creation

**11.3.4.4 Data Pointers**

**Data Pointer:** Dialog to adjust the data pointers of your module



**Icon**



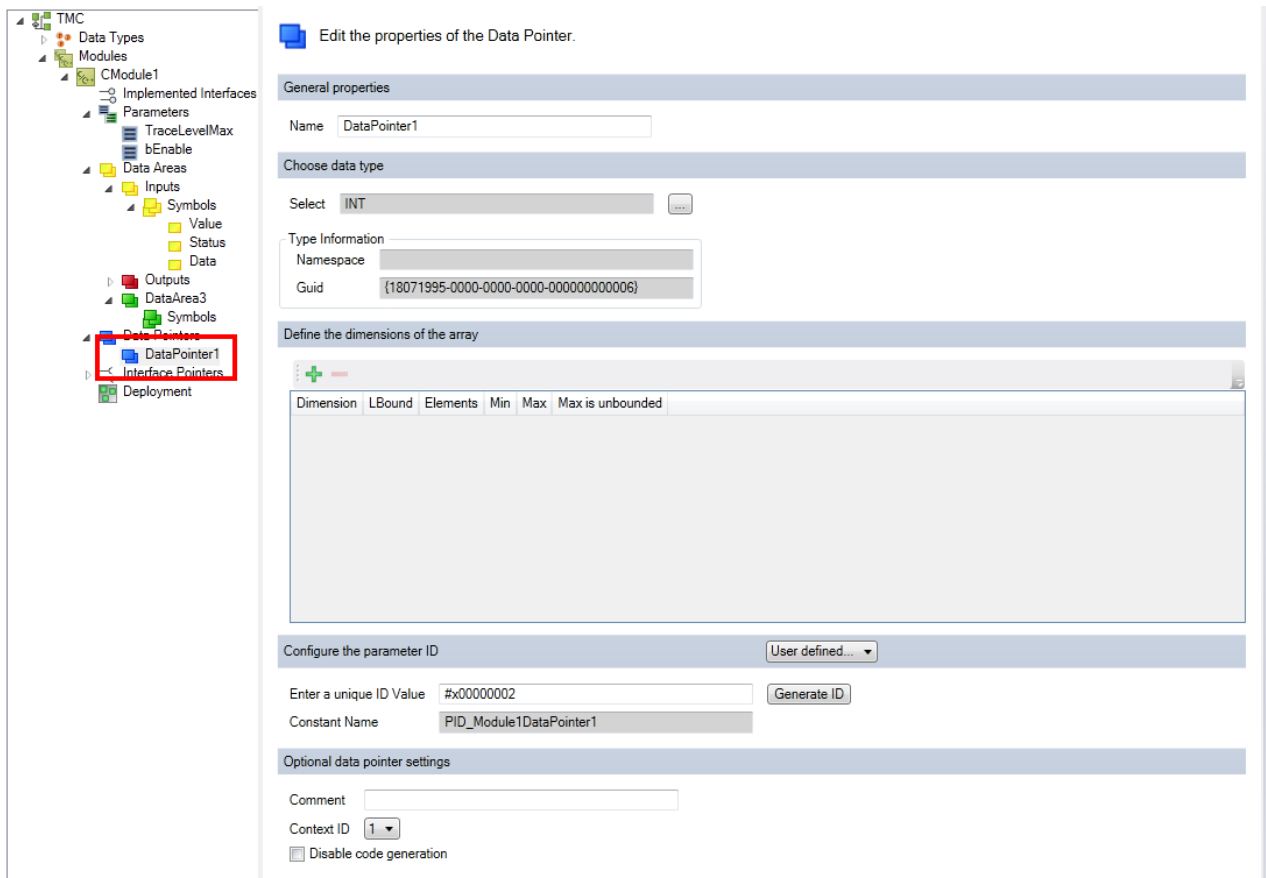
**Function**

- Add a new data pointer
- Deletes the selected data pointer
- Move the selected item one position down
- Move the selected item one position up

- 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:** Switch to enable/disable the code generation

### 11.3.4.4.1 Data Pointer Properties

**Data Pointer Properties:** Edit the Data Pointer properties



#### 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

Please see [here](#) [▶ 98].

#### Configure the parameter ID

**Enter a unique ID Value:** Enter a unique ID Value. See [Parameters](#) [▶ 104].

**Constant Name:** Source code name of the parameter ID

#### Optional data pointer settings

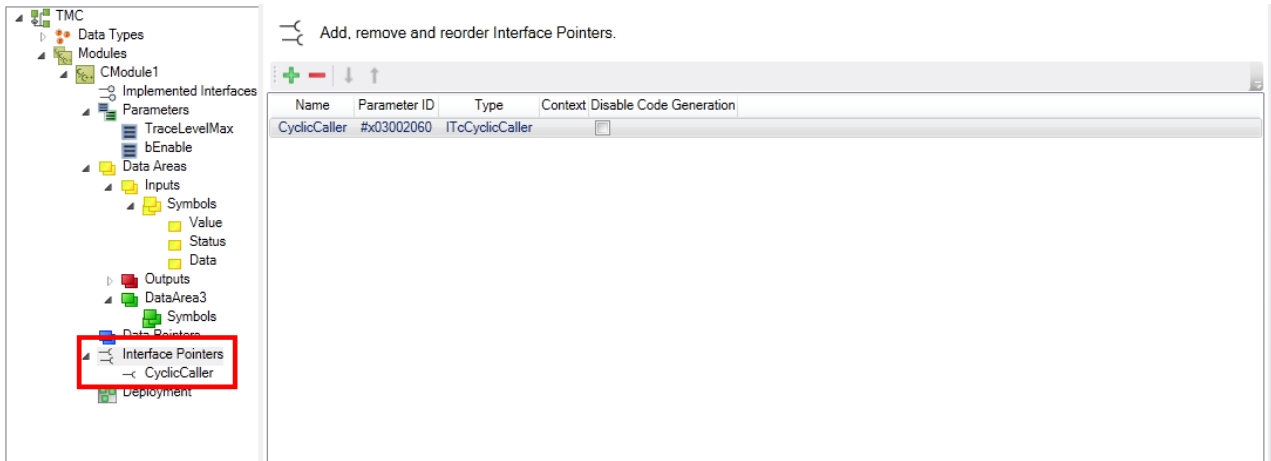
**Comment:** Comment, which will be visible e.g. in the instance configurator

**Context ID:** Context ID of the data pointer.

**Disable Code Generation:** Switch to enable/disable the code generation

### 11.3.4.5 Interface Pointers

**Interface Pointers :** Add, remove and reorder Interface Pointers



**Icon**



**Function**

Add interface pointer

Deletes the selected pointer

Move the selected item one position down

Move the selected item one position up

**Name:** Name of the interface

**Parameter ID:** Unique ID of the interface pointer

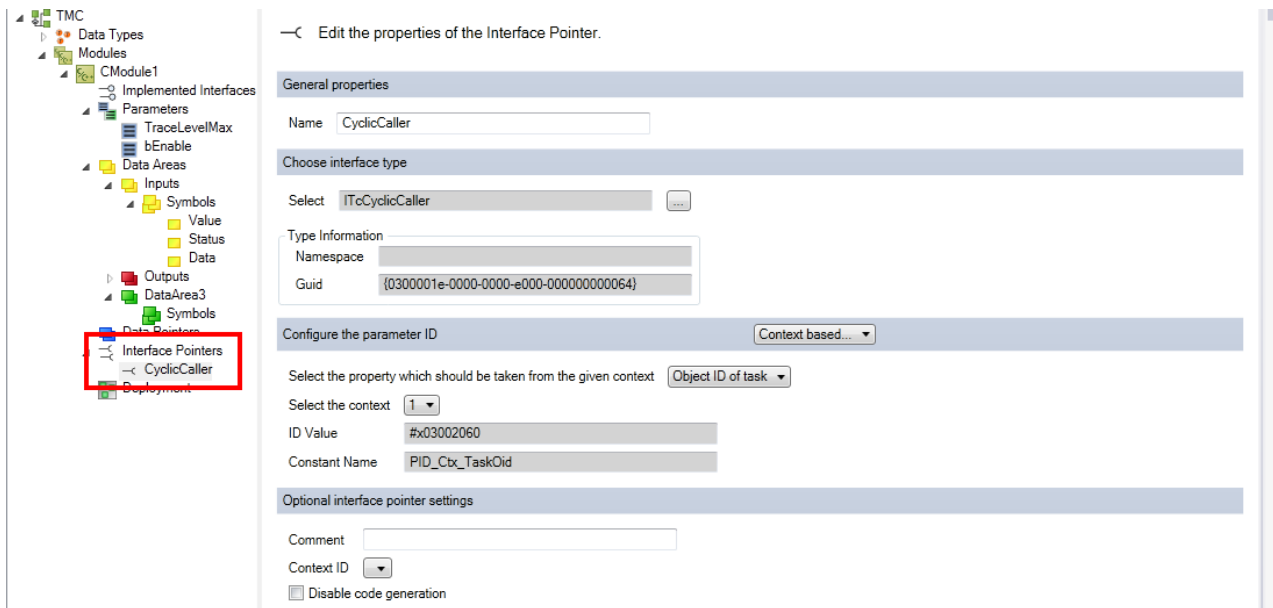
**Type:** Type of interface pointer

**Context:** Context of the interface

**Disable Code Generation:** Switch to enable/disable the code generation

#### 11.3.4.5.1 Interface Pointer Properties

**Interface Pointer Properties:** Edit the Interface Pointer properties



## General Properties

**Name:** Name of the interface pointer

## Choose the base interface

**Select:** Selection of the interface

## Type Information

- **Namespace:** namespace for the interface
- **GUID:** Unique ID of the interface

## Configure the parameter ID

See [Parameters \[► 104\]](#).

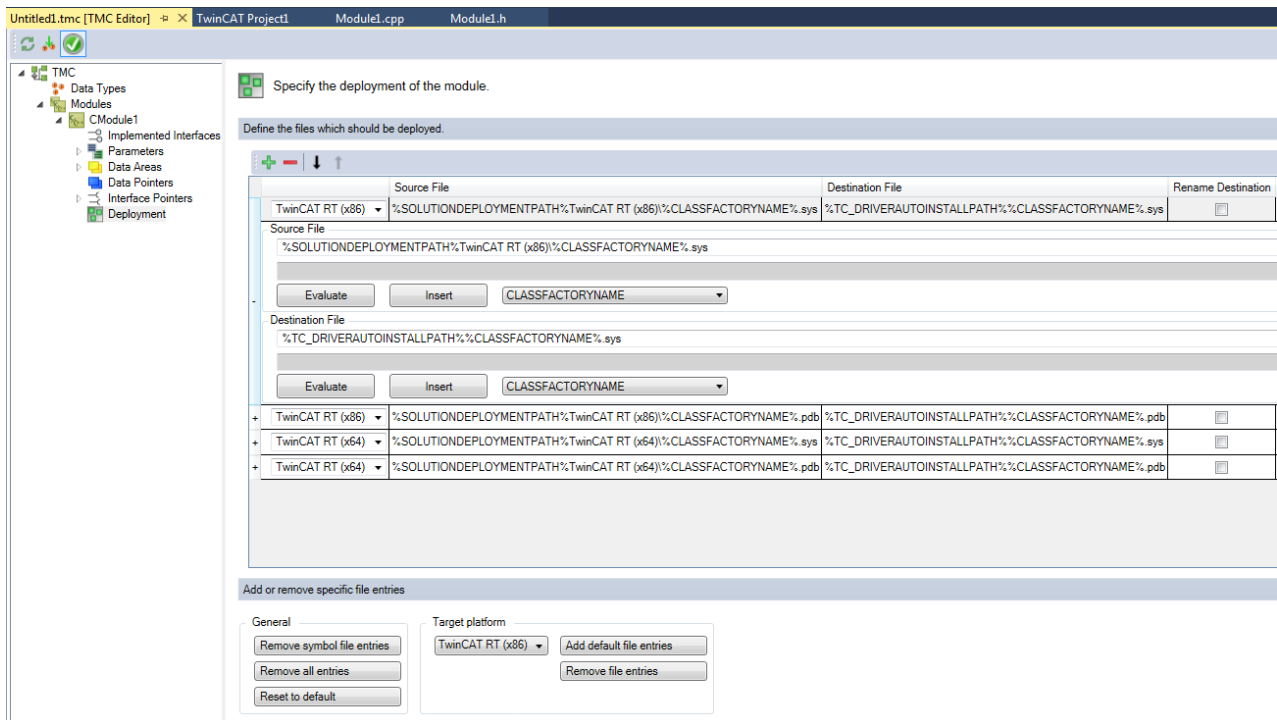
**Comment:** Optional

**Context ID:** Context ID of the interface pointer.

**Disable Code Generation:** Switch to enable/disable the code generation

## 11.3.4.6 Deployment

**Deployment:** Specify storage locations for the provided modules on the target system



**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:** target file is renamed before the new file is transferred. Since this is required for Windows 10, it is done implicitly.

The individual entries can be expanded or collapsed by clicking on "+" or "-" on the left.

**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	Default value
	\HKLM\Software\Beckhoff\Twin-CAT3	
%TC_INSTALLPATH%	InstallDir	C:\TwinCAT\3.x \
%TC_CONFIGPATH%	ConfigDir	C:\TwinCAT\3.x \Config\
%TC_TARGETPATH%	TargetDir	C:\TwinCAT\3.x \Target\
%TC_SYSEMPATH%	SystemDir	C:\TwinCAT\3.x \System\
%TC_BOOTPRJPATH%	BootDir	C:\TwinCAT\3.x \Boot\
%TC_RESOURCEPATH%	ResourceDir	C:\TwinCAT\3.x \Target\Resource\
%TC_REPOSITORYPATH%	RepositoryDir	C:\TwinCAT\3.x \Repository\
%TC_DRIVERPATH%	DriverDir	C:\TwinCAT\3.x \Driver\
%TC_DRIVERAUTOINSTALLPATH%	DriverAutoInstallDir	C:\TwinCAT\3.x \Driver\AutoInstall\
%TC_SYSSRVEXEPATH%		C:\TwinCAT\3.x \SDK\Bin \TwinCAT UM (x86)\
%CLASSFACTORYNAME%		<Name of the Class Factory>

("x" is replaced by the installed TwinCAT version)

## 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.

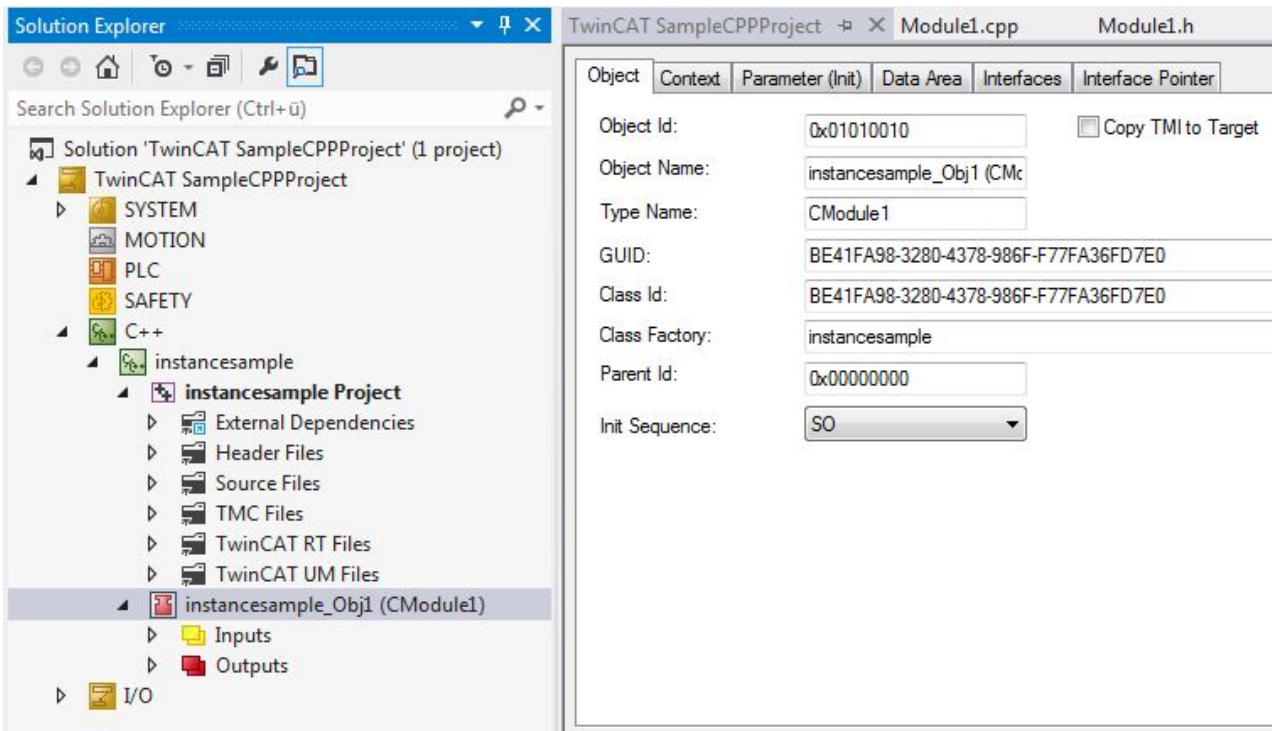
The configuration applies to the context (including the task calling the module), parameters and pointers.

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

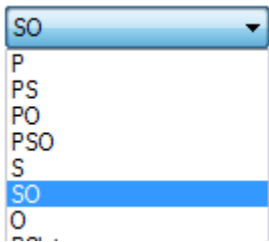
Double-click on the generated instance to open the configuration dialog with several windows.



### 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 class GUID
- **Class Id:** Class ID of the implementation class (GUID and ClassId are usually identical)
- **Class Factory:** Refers to the driver, which provides the Class Factory that was used for the development of the module instance.
- **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 [here](#) [▶ 39] 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

ID	Task	Name	Priority	Cycle Time (µs)	Task Port	Symbol Port	Sort Order
1	02010010	Task 1	1	10000	350	350	0

- **Context:** Selects the context to configure (see TMC Editor for adding different contexts).  
**Note** A dataarea is associated with a context
- **Data Areas / Interfaces / Data Pointer and Interface Pointer:** Each instance could be configured to have items defined in TMC or not.
- **Result Table:** List of IDs which need to be configured. At least the context („Task“column) needs to be configured to the task.

## 11.4.3 Parameter (Init)

PTCID	Name	Value	CS	Unit	Type	Comment
0x00000001	Parameter	...	<input type="checkbox"/>			

List of all parameters (as defined in TMC) could be initialized by values for each instance.

Special ParameterIDs (PTCID) are used to set values automatically. These are configured via the TMC Editor's parameter dialogue as described [here \[► 104\]](#).

The CS (CreateSymbol) checkbox creates the ADS Symbols for each parameter, thus it is accessible from outside

### 11.4.4 Data Area

Area No	Name	Type	Size	CS	Elements	Owner	Comment
- 0	Inputs	InputDst	12	<input type="checkbox"/>	3 Symbols		
	Value	UDINT	4.0 (Offs: 0.0)	<input type="checkbox"/>			
	Status	UDINT	4.0 (Offs: 4.0)	<input type="checkbox"/>			
	Data	UDINT	4.0 (Offs: 8.0)	<input type="checkbox"/>			
- 1	Outputs	OutputSrc	12	<input type="checkbox"/>	3 Symbols		
	Value	UDINT	4.0 (Offs: 0.0)	<input type="checkbox"/>			
	Control	UDINT	4.0 (Offs: 4.0)	<input type="checkbox"/>			
	Data	UDINT	4.0 (Offs: 8.0)	<input type="checkbox"/>			

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

The CS (CreateSymbol) checkbox creates the ADS Symbols for each parameter, thus the variable is accessible from outside

### 11.4.5 Interfaces

IID	Name
00000012-0000-0000-E000-000000000064	ITComObject
03000010-0000-0000-E000-000000000064	ITcCyclic
03000012-0000-0000-E000-000000000064	ITcADI
03000018-0000-0000-E000-000000000064	ITcWatchSource

List of all implemented interfaces (as defined in TMC)

### 11.4.6 Interface Pointer

PTCID	Name	OTCID	Object Name	IID	Type
0x03002060	CyclicCaller	02010010	Task 1	0300001E-0000-0000...	ITcCyclicCaller

List of all Interface Pointers (as defined in TMC).

Special ParameterIDs (PTCID) are used to set values automatically. These are configured via the TMCEditor's parameter dialogue as described [here](#) [▶ 104].

The OTCID column defines the pointer to the instance, which should be used.

## 11.4.7 Data Pointer

PTCID	Name	OTCID	Object Name	Area No	Offset	Size
0x00000003	DataPointer1	0x00000000		0	0	0

List of all Data Pointers (as defined in TMC).

Special ParameterIDs (PTCID) are used to set values automatically. These are configured via the TMC Editor's parameter dialogue as described [here](#) [▶ 104].

The OTCID column defines the pointer to the instance, which should be used.

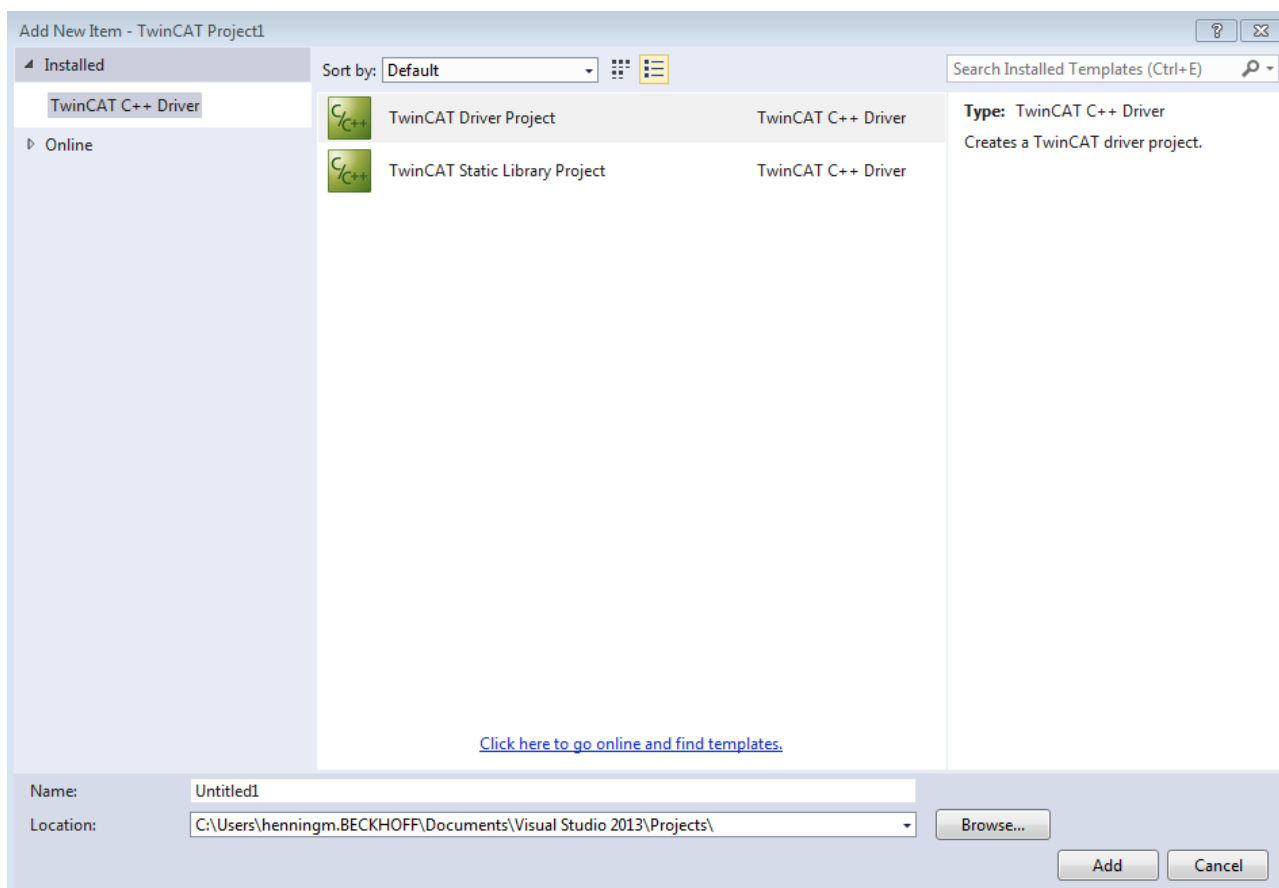
## 11.5 Customer-specific project templates

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).

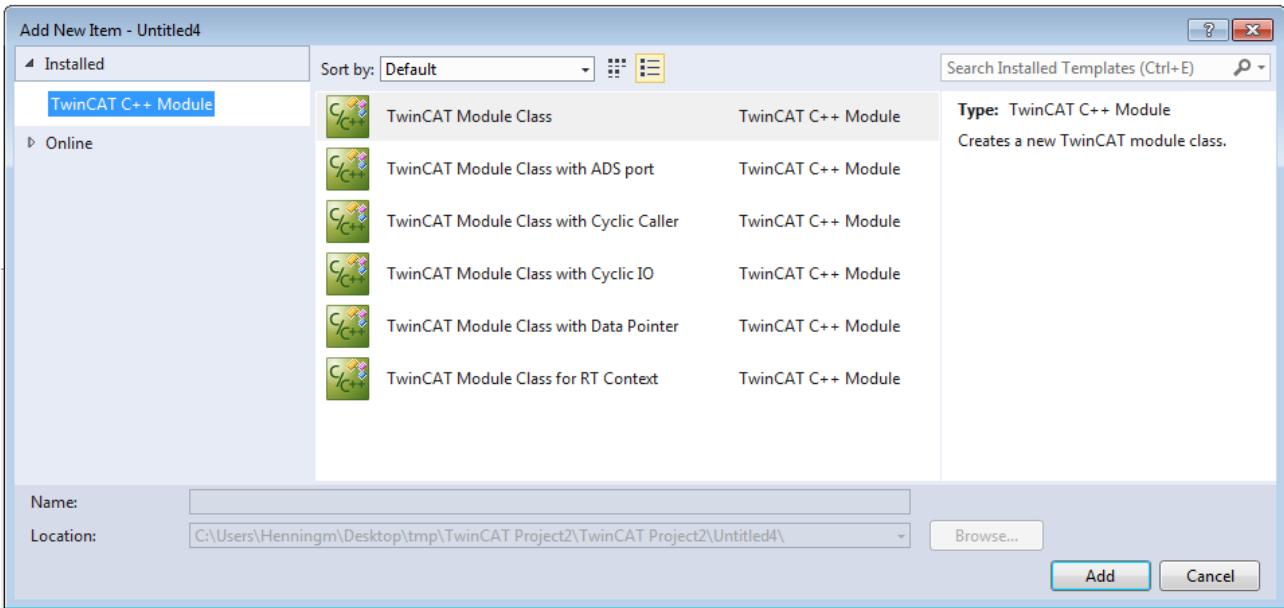
This section of the documentation describes how customers can realize their own project templates.

### 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 driver. The purpose of this framework is to register a TwinCAT driver in the system. The actual function of the driver is implemented in TwinCAT modules.

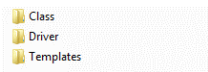


The TwinCAT Class Wizard is automatically started on creating a new driver project in order to add the first TwinCAT driver module. The different modules are generated by the same TwinCAT Class Wizard, but the specific design of the module is realized using templates.



### 11.5.2 Files involved







Virtually all relevant information is contained in the directory **C:\TwinCAT\3.x\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** (for "TwinCAT C++ Project Wizard") and **Class directory** (for the "TwinCAT Module Class Wizard"), where each project type encompasses 3 files:

 TcDriverWizard.ico	10.06.2015 11:14	Icon	267 KB
 TcDriverWizard.vmdir	10.06.2015 11:14	VSDIR File	1 KB
 TcDriverWizard.vsz	10.06.2015 11:14	Visual Studio Wiza...	1 KB
 TcStaticLibraryWizard.ico	10.06.2015 11:14	Icon	267 KB
 TcStaticLibraryWizard.vmdir	10.06.2015 11:14	VSDIR File	1 KB
 TcStaticLibraryWizard.vsz	10.06.2015 11:14	Visual Studio Wiza...	1 KB

The **.vmdir** 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 the MSDN can be found here: <https://msdn.microsoft.com/de-de/library/Aa291929%28v=VS.71%29.aspx>.

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

Both assistants have a .xml file as a parameter that describes the transformations of, for example, source data 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 templates 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 possibly 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 number 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</Target-File>
```
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. <code>&lt;PlatformToolset&gt;v100&lt;/PlatformToolset&gt;</code>
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.

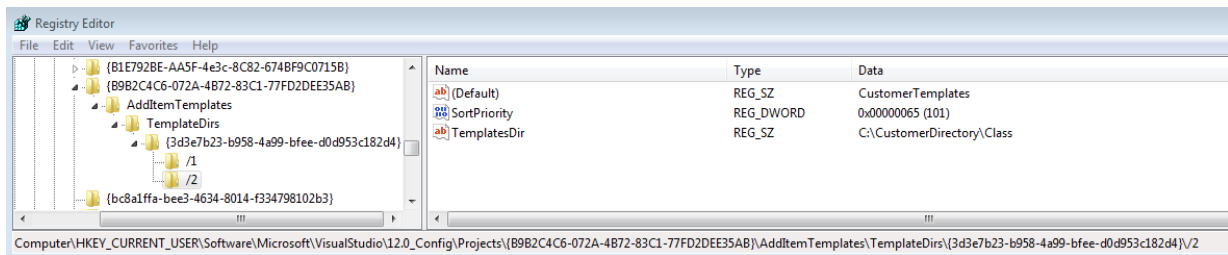
1. In the registry, expand the search path (in this case V12.0, i.e. for VS 2013) in which the node /2 is created:  
**Registry Key:** `HKEY_CURRENT_USER\Software\Microsoft\VisualStudio\12.0_Config`



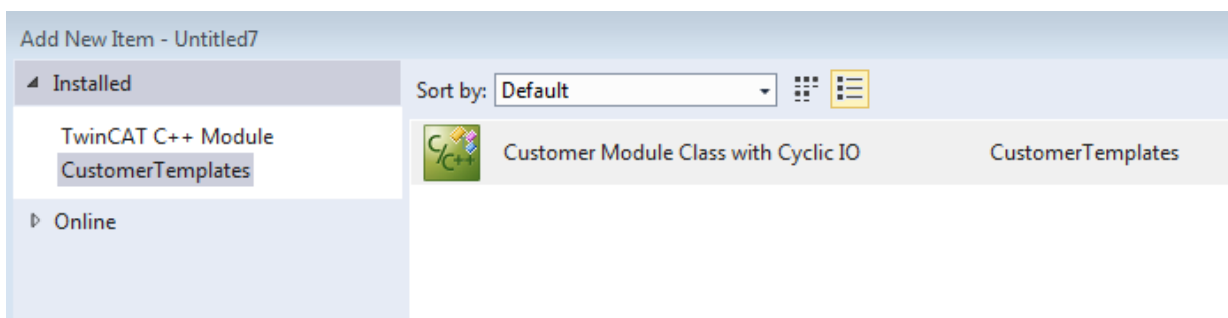
```

\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 given a version number in the version management system and is also not affected by TwinCAT installations/updates.

### Quick start

A general entry to the assistant environment in the 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 a copy template  
In **C:\TwinCAT\3.x\Components\Base\CppTemplate\Templates**








 CustomerModuleCyclicIO	20.08.2015 10:29	File folder
 TcDriverWizard	21.07.2015 13:02	File folder
 TcModuleAdsPort	21.07.2015 13:02	File folder
 TcModuleCyclicCaller	21.07.2015 13:02	File folder
 TcModuleCyclicIO	21.07.2015 13:02	File folder
 TcModuleDataPointer	21.07.2015 13:02	File folder
 TcModuleEmpty	21.07.2015 13:02	File folder
 TcModuleRT	21.07.2015 13:02	File folder
 TcStaticLibrary	21.07.2015 13:02	File folder



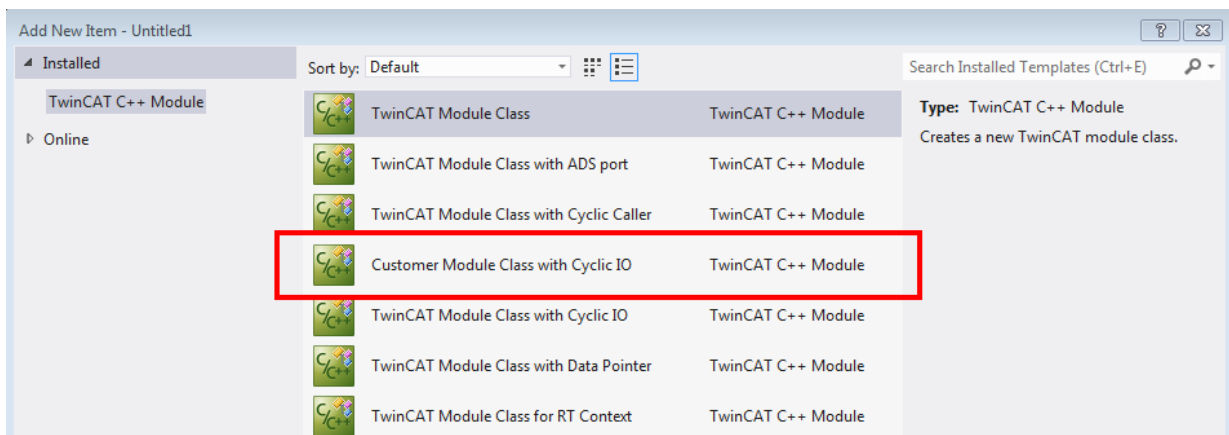
2. Rename the .xml file within the folder

 CustomerModuleCyclicIOConfig.xml	10.06.2015 11:14	XML Document	1 KB
 TcModuleCyclicIO.cpp	10.06.2015 11:14	C++ Source	7 KB
 TcModuleCyclicIO.h	10.06.2015 11:14	C/C++ Header	2 KB
 TcModuleCyclicIO.tmc	10.06.2015 11:14	TMC File	5 KB

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

 CustomerModuleCyclicIOWizard.ico	10.06.2015 11:14	Icon	265 KB
 CustomerModuleCyclicIOWizard.vmdir	20.08.2015 10:35	VSDIR File	1 KB
 CustomerModuleCyclicIOWizard.vsz	10.06.2015 11:14	Visual Studio Wiza...	1 KB
 TcModuleAdsPortWizard.ico	10.06.2015 11:14	Icon	265 KB
 TcModuleAdsPortWizard.vmdir	10.06.2015 11:14	VSDIR File	1 KB
 TcModuleAdsPortWizard.vsz	10.06.2015 11:14	Visual Studio Wiza...	1 KB
 TcModuleCyclicCallerWizard.ico	10.06.2015 11:14	Icon	265 KB

- Now reference the copied .vsz file in the .vsdir file and adapt the description.
  - Enter the .xml file created in step 2 in the .vsz file.
  - You can now make changes to the source files in the Template/CustomModuleCyclicIO/ directory. The .xml takes care of replacements when generating a project from this template.
- ⇒ The "TwinCAT Module Class Wizard" now displays the new project for selection:



If the vsxproj, for example, is 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 generated 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.

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

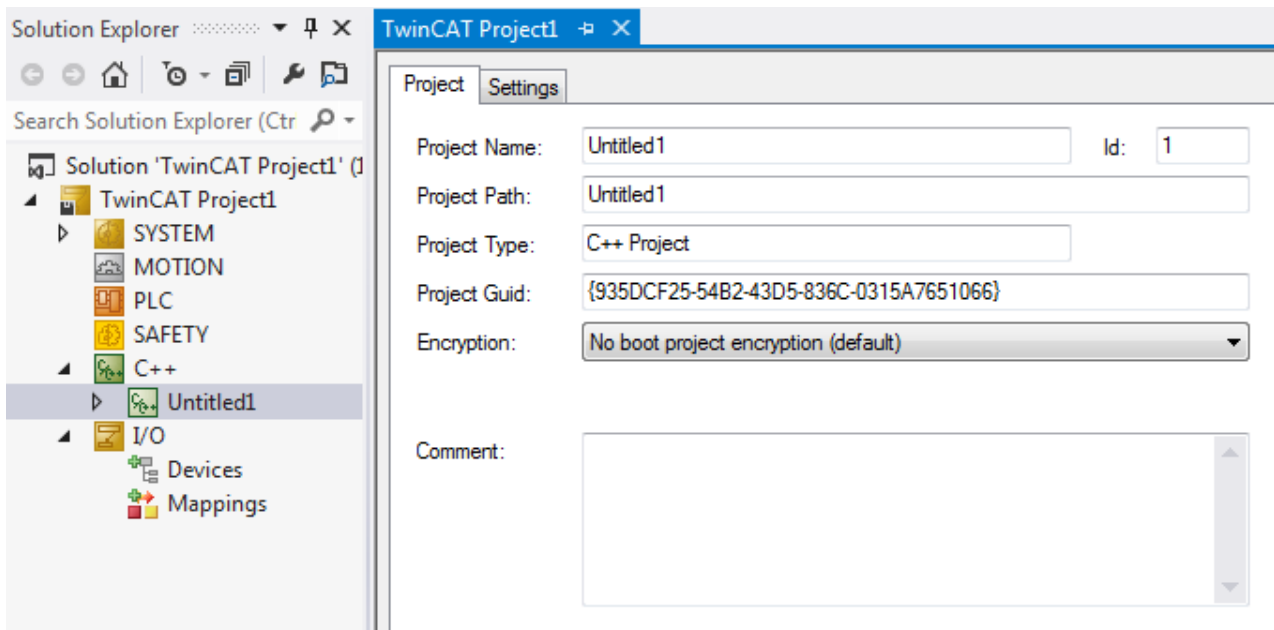
## 12 Programming Reference

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 \[▶ 213\]](#), which contain the valuable information on the handling of the modules and interfaces.

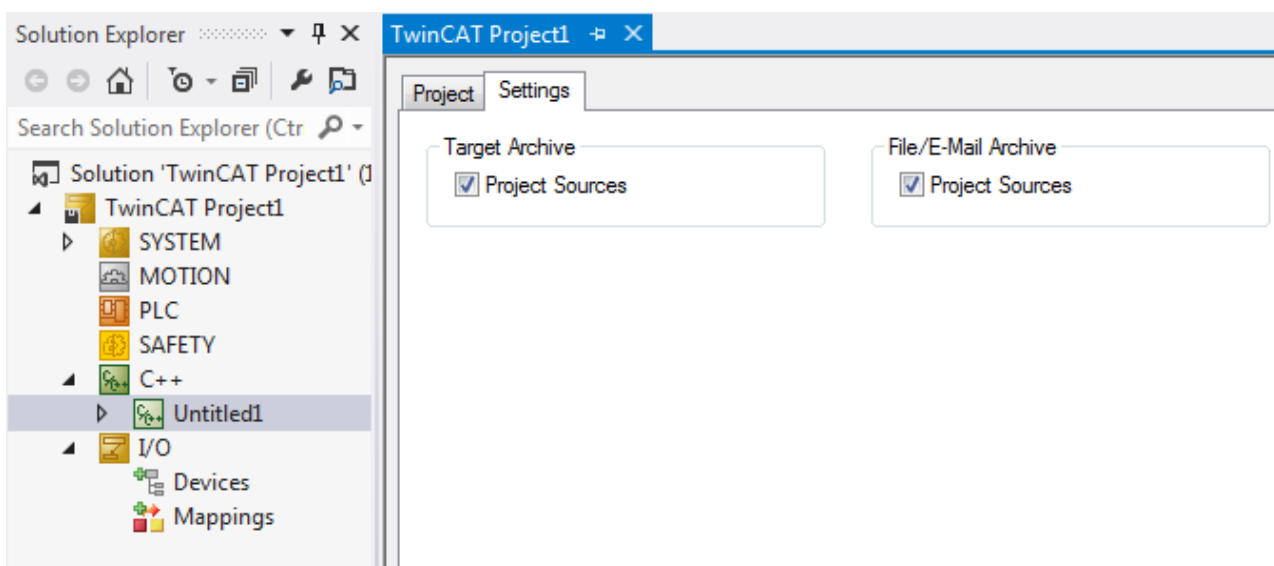
### C++ project properties

A TwinCAT C++ project has some properties, which can be accessed by double-clicking on the TwinCAT C++ project (project name here "Untitled1").



Renaming is not possible at this stage (see [Renaming TwinCAT C++ projects \[▶ 199\]](#))

Encryption is not implemented for C++ projects.



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 File Description

During development of a TwinCAT C++ module, one could come into contact with files on the file system directly. This could be of interest either for understanding how the system works or for special use cases like manual file transfer etc.

Here is a list of files, which are C++ module related.

File	Description	Additional Information
<b>Engineering / XAE</b>		
*.sln	Visual Studio Solution File, hosts TwinCAT and non-TwinCAT projects	
*.tsproj	TwinCAT Project, collecting all nested TwinCAT projects like TwinCAT C++ or TwinCAT PLC projects	
_Config/	Folder contains additional configuration files (*.xti) which 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 <a href="#">TwinCAT Module Class Editor (TMC) [► 79]</a>
*.rc	Resource File	See <a href="#">Setting version/vendor information [► 203]</a>
*.vcxproj.*	Visual Studio C++ Project files	
*ClassFactory.cpp/.h	Class factory for this TwinCAT Driver	
*Ctrl.cpp/.h	Driver loading and unloading for TwinCAT UM platform	
*Driver.cpp/.h	Driver loading and unloading for TwinCAT RT platform	
*Interfaces.cpp/.h	Declaration of TwinCAT COM interface classes	
*W32.cpp./def/.idl		
*.cpp/.h	One C++/Header file per TwinCAT module in driver. Custom code goes here.	
Resource.h	Needed by *.rc file	
TcPch.cpp/.h	Used for precompiled header creation	
%TC_INSTALLPATH%\CustomConfig\Modules\*	Published TwinCAT driver package usually C:\TwinCAT\3.x\CustomConfig\Modules\*	See <a href="#">Export modules [► 44]</a>
<b>Runtime / XAR</b>		
%TC_BOOTPRJPATH%\CurrentConfig\*	Current Configuration setup usually C:\TwinCAT\3.x\Boot	
%TC_DRIVERAUTOINSTALLPATH%\*.sys/pdb	Compiled, platform specific driver <ul style="list-style-type: none"> <li>C:\TwinCAT\3.x\Driver\AutoInstall (System load)</li> <li>C:\TwinCAT\3.x\Driver\AutoLoad (TcLoader load)</li> </ul>	
%TC_BOOTPRJPATH%\TMI\OBJECTID.tmi	TwinCAT Module Instance file Describes variables of driver Filename is "ObjectID.tmi" Usually C:\TwinCAT\3.x\Boot\TMI\OTCID.tmi	
<b>Temporary Files</b>		
*.sdf	IntelliSense database	
*.suo / *.v12.suo	User and Visual Studio specific files	
*.tsproj.bak	Automatically generated backup file of tsproj	
ipch/	Intermediate directory created for precompiled header	

## 12.1.1 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 selecting the target system.

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

When a "Build" or a "Rebuild" is selected (and thus also in the case of "Activate Configuration"), the following steps take place:

1. The sources are located in the respective project directory.
2. The compilations are generated according to the specific architecture in C:\TwinCAT\3.1\sdk\\_products\  
e.g. in C:\TwinCAT\3.1\sdk\\_products\TwinCAT RT (x64)\Debug\<ProjectName>
3. After that the link procedure places the .sys/.pdb file, similarly according to the specific architecture, in C:\TwinCAT\3.1\sdk\\_products\  
e.g. in C:\TwinCAT\3.1\sdk\\_products\TwinCAT RT (x64)\Debug\
4. A copy of the .sys/.pdb is placed in the \_Deployment/ subdirectory of the project directory, e.g. in Project Directory/\_Deployment/TwinCAT RT (x64)\
5. Pressing the "Activate Configuration" button leads to .sys/.pdb being transferred from \_Deployment/ of the project directory to the target system (if applicable it is a local copy)

## 12.2 Limitations

TwinCAT 3 C++ modules [▶ 30] are executed in Windows kernel mode. Developers must therefore be aware of some limitations:

- Win32 API is not available in kernel mode (see [below \[▶ 137\]](#)).
- 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 \[▶ 206\]](#))
- The memory capacity for dynamic allocation in a real-time context is limited by the router memory (this can be configured during engineering) (see [Memory Allocation \[▶ 138\]](#))
- A subset of the C++ runtime library functions (CRT) is supported
- C++ exceptions are not supported.
- Runtime Type Information (RTTI) is not supported (see [below \[▶ 138\]](#))
- Subset of STL is supported (see [STL / Containers \[▶ 192\]](#))
- Support for functions from math.h through TwinCAT implementation (see [Mathematical Functions \[▶ 190\]](#))

### 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	<a href="#">Tracing</a> [ <a href="#">▶ 193</a> ]
File I/O	See <a href="#">Interface ITcFileAccess</a> [ <a href="#">▶ 143</a> ], <a href="#">Interface ITcFileAccessAsync</a> [ <a href="#">▶ 151</a> ] and <a href="#">Sample19: Synchronous File Access</a> [ <a href="#">▶ 268</a> ], <a href="#">Sample20: FileIO-Write</a> [ <a href="#">▶ 269</a> ], <a href="#">Sample20a: FileIO-Cyclic Read / Write</a> [ <a href="#">▶ 269</a> ]
Synchronization	See <a href="#">Sample11a: Module communication: C module calls a method of another C module</a> [ <a href="#">▶ 263</a> ]
Visual C CRT	See <a href="#">RtlR0.h</a>

### 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](#) [[▶ 169](#)]

## 12.3 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.

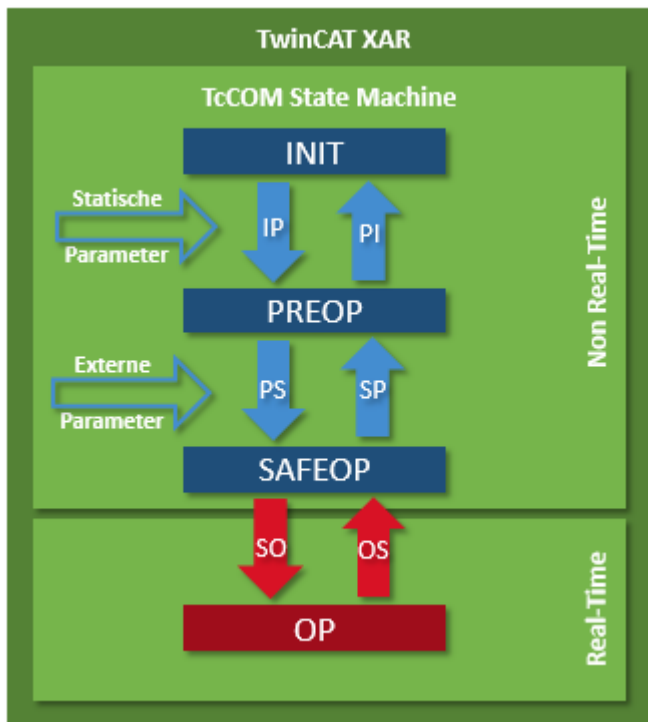
It is also possible to allocate and release memory dynamically.

- Operator `new` / `delete`
- `TcMemAllocate` / `TcMemFree`

This memory allocation can be used in the [transitions](#) [[▶ 39](#)] 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.



**Global class instances**

Global instances must release memory allocated in the real-time context before the destructor.

TwinCAT supports up to 32 global class instances.

Global class instances include the following constructs:

- Definition in the global scope
- Definition as a static class variable
- Local, static variables in methods

**12.4 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
<a href="#">ITcUnknown [▶ 169]</a>	ITcUnknown defines the reference count as well as the querying of a reference to a more specific interface.
<a href="#">ITComObject [▶ 156]</a>	The ITComObject interface is implemented by every TwinCAT module.
<a href="#">ITcCyclic [▶ 140]</a>	The interface is implemented by TwinCAT modules that are called once per task cycle.
<a href="#">ITcCyclicCaller [▶ 141]</a>	Interface for logging the ItcCyclic interface of a module onto and off from a TwinCAT task.
<a href="#">ITcFileAccess [▶ 143]</a>	Interface for accessing the file system
<a href="#">ITcFileAccessAsync [▶ 151]</a>	Asynchronous access to file operations.
<a href="#">ITcPostCyclic [▶ 161]</a>	The interface is implemented by TwinCAT modules that are called once per task cycle following the output update.
<a href="#">ITcPostCyclicCaller [▶ 162]</a>	Interface for logging the ITcPostCyclic interface of a module onto and off from a TwinCAT task.
<a href="#">ITcIoCyclic [▶ 153]</a>	This interface is implemented by TwinCAT modules that are called during the input update and output update within a task cycle.
<a href="#">ITcIoCyclicCaller [▶ 154]</a>	Interface for logging the ITcloCyclic interface of a module onto and off from a TwinCAT task.
<a href="#">ITcRTimeTask [▶ 164]</a>	Query of extended TwinCAT task information.
<a href="#">ITcTask [▶ 165]</a>	Query of the time stamp and task-specific information of a TwinCAT task.
<a href="#">ITcTaskNotification [▶ 168]</a>	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.x\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 \[▶ 190\]](#).
- Subset of STL [\[▶ 192\]](#).
- TwinCAT runtime [RtIR0.h \[▶ 171\]](#)
- Methods for [ADS communication \[▶ 173\]](#)
- Classes / functions with names beginning with "Os" must not be used in a real-time context.

## 12.4.1 Interface ITcCyclic


Interface ITcCyclic Interface is implemented by TwinCAT modules which should be 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

Icon	Name	Description
	<a href="#">CycleUpdate [▶ 141]</a>	Called once per task cycle if interface has been registered with a cyclic caller.



**Remarks**

The ITcCyclic interface is implemented by TwinCAT modules. This interface is passed to method ITcCyclicCaller::AddModule() when a module registers itself with a task, typically as the last initialization step in the SafeOP to OP transition. After registration the method CycleUpdate() of the module instance is called.

**12.4.1.1 Method ITcCyclic:CyclicUpdate**

The method CyclicUpdate usually called by a TwinCAT task, after the interface has been registered.

**Syntax**

```
HRESULT TCOMAPI CycleUpdate(ITcTask* ipTask, ITcUnknown* ipCaller, ULONG_PTR context)
```

**Parameters**

**ipTask:** (type: ITcTask) refers to the current task context.

**ipCaller:** (type: ITcUnknown) refers to the calling instance.

**Context:** (type: ULONG\_PTR) context contains the value which has been passed to method ITcCyclicCaller::AddModule()

**Return Value**

It is recommended to always return S\_OK. Currently, the return value is ignored by TwinCAT Tasks.

**Description**

Within a task cycle the method CycleUpdate() is called after InputUpdate() has been for all registered module instances. Therefore, this method should be used to implement cyclic processing.

**12.4.2 Interface ITcCyclicCaller**



Interface to register or unregister a module's ITcCyclic interface with a TwinCAT task.

**Syntax**

```
TCOM_DECL_INTERFACE("0300001E-0000-0000-e000-000000000064", ITcCyclicCaller)
struct__declspec(novtable) ITcCyclicCaller : public ITcUnknown
```

Required include: TcIoInterfaces.h

**Methods**

Icon	Name	Description
	AddModule [ <a href="#">▶ 142</a> ]	Register module which implements the ITcCyclic interface.
	RemoveModule [ <a href="#">▶ 142</a> ]	Unregister the previously registered ITcCyclic interface of a module.

**Remarks**

The ITcCyclicCaller interface is implemented by TwinCAT tasks. A module uses this interface to register its ITcCyclic interface with a task, typically as the last initialization step in the SafeOP to OP transition. After registration the method CycleUpdate() of the module instance is called. The interface is also used to unregister the module from being called by a task.

### 12.4.2.1 Method ITcCyclicCaller:AddModule

Register a module's ITcCyclic interface with cyclic caller, i.e. a TwinCAT task.

#### Syntax

```
virtual HRESULT TCOMAPI
AddModule(STcCyclicEntry* pEntry, ITcCyclic* ipMod, ULONG_PTR
context=0, ULONG sortOrder=0)=0;
```

#### Parameters

**pEntry:** (type: STcCyclicEntry) [in] Pointer to a list entry, which is inserted into the internal list of the cyclic caller. See also description.

**ipMod:** (type: ITcCyclic) [in] Interface pointer which will be used by cyclic caller

**context:** (type: ULONG\_PTR) [optional] a context value which is passed to the ITcCyclic::CyclicUpdate() method on each call.

**sortOrder:** (type: ULONG) [optional] the sort order can be used to control the order of execution if different module instances are executed by the same cyclic caller.

#### Return Value

Type: HRESULT

On success the method returns S\_OK. If cyclic caller, i.e. the TwinCAT task, is not in OP state, the error ADS\_E\_INVALIDSTATE is returned.

#### Description

A TwinCAT module class usually uses a smart pointer to refer to the cyclic caller of type ITcCyclicCallerPtr. The object id of the task is stored in this smart point and a reference to the task can be obtained using the TwinCAT object server. In addition the smart pointer class already contains a list entry. Therefore the smart pointer can be used as first parameter for the AddModule method.

The following sample code shows the registration 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;

    }

}

}
```

### 12.4.2.2 Method ITcCyclicCaller:RemoveModule

Unregister a module instance from being called by a cyclic caller.

#### Syntax

```
virtual HRESULT TCOMAPI
RemoveModule(STcCyclicEntry* pEntry)=0;
```

**Parameters**

**pEntry:** (type: STcCyclicEntry) refers to the list entry which should be removed from the internal list of the cyclic caller.

**Return Value**

If the entry is not in the internal list, the method returns E\_FAIL.

**Description**

Similar to the method AddModule() the smart pointer for the cyclic caller is used as list entry when the module instance should be removed from cyclic caller.

Declaration and usage of smart pointer:

ITcCyclicCallerInfoPtr m\_spCyclicCaller;

```
if (
m_spCyclicCaller ) {
m_spCyclicCaller->RemoveModule(m_spCyclicCaller);
}
m_spCyclicCaller = NULL;
```

### 12.4.3 Interface ITcFileAccess

Interface to access 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**

Icon	Name	Description
	<a href="#">FileOpen [▶ 144]</a>	Opens a file
	<a href="#">FileClose [▶ 145]</a>	Closes a file
	<a href="#">FileRead [▶ 145]</a>	Reads from a file
	<a href="#">FileWrite [▶ 146]</a>	Writes to a file
	<a href="#">FileSeek [▶ 146]</a>	Sets position in file
	<a href="#">FileTell [▶ 147]</a>	Retrieves position in file
	<a href="#">FileRename [▶ 147]</a>	Renames a file
	<a href="#">FileDelete [▶ 147]</a>	Deletes a file
	<a href="#">FileGetStatus [▶ 148]</a>	Gets status of a file
	<a href="#">FileFindFirst [▶ 149]</a>	Searches for a file, first iteration
	<a href="#">FileFindNext [▶ 149]</a>	Searches for a file, next iteration
	<a href="#">FileFindClose [▶ 150]</a>	Closes a file search
	<a href="#">MkDir [▶ 150]</a>	Creates a directory
	<a href="#">Rmdir [▶ 151]</a>	Deletes a directory

## Remarks

The ITcFileAccess interface used to access files from file systems. Since the provided methods are blocking this should not be used in CycleUpdate() / realtime context. The derived interface [ITcFileAccessAsync \[► 151\]](#) adds a Check() Method, which could be used instead.

Please have a look at [Sample20a: FileIO-Cyclic Read / Write \[► 269\]](#).

The interface is implemented by module class CID\_TcFileAccess.

### 12.4.3.1 Method ITcFileAccess:FileOpen

Opens a file

#### Syntax

```
virtual HRESULT TCOMAPI FileOpen(PCCH szFileName, TcFileAccessMode AccessMode, PTcFileHandle phFile);
```

#### Parameters

**szFileName:** (type: PCCH) [in] the filename to open

**AccessMode:** (type: TcFileAccessMode) [in] Access mode of the File, see *TcFileAccessServices.h*

**phFile:** (type: TcFileHandle) [out] returned file handle

#### Return Value

Type: HRESULT

On success the method returns S\_OK.

Error codes of special interest:

- ADS\_E\_TIMEOUT when timeout (5 seconds) has elapsed.

Further [ADS statuscodes \[► 300\]](#) could occur.

#### Description

The method returns a handle to access the file, which name is defined in szFileName.

AccessModes could be used as following:

```
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;
```

### 12.4.3.2 Method ITcFileAccess:FileClose

Closes a file

#### Syntax

```
virtual HRESULT TCOMAPI FileClose(PTcFileHandle phFile);
```

#### Parameters

**phFile:** (type: TcFileHandle) [out] returned file handle

#### Return Value

Type: HRESULT

On success the method returns S\_OK.

Error codes of special interest:

- ADS\_E\_TIMEOUT when timeout (5 seconds) has elapsed.

Further [ADS status codes](#) [▶ 300] could occur.

#### Description

The method closes a file defined by the phFile.

### 12.4.3.3 Method ITcFileAccess:FileRead

Read data from a file.

#### Syntax

```
virtual HRESULT TCOMAPI  
FileRead(TcFileHandle hFile, PVOID pData, UINT cbData, PUINT pcbRead);
```

#### Parameters

**hFile:** (type: TcFileHandle) [in] refers to the prior opened file

**pData:** (type: PVOID) [out] location of the data to be read

**cbData:** (type: PVOID) [in] maximum size of data to be read (size of memory behind pData)

**pcbRead:** (type: PUINT) [out] size of read data

#### Return Value

Type: HRESULT

If any data could be read, S\_OK is returned.

Error codes of special interest:

- ADS\_E\_TIMEOUT when timeout (5 seconds) has elapsed.

Further [ADS status codes](#) [▶ 300] could occur.

#### Description

This method retrieves data from a file defined by the file handle. Data will be stored in pData while pcbRead provides length of given data.

### 12.4.3.4 Method ITcFileAccess:FileWrite

Write data to a file.

#### Syntax

```
virtual HRESULT TCOMAPI  
FileWrite(TcFileHandle hFile, PCVOID pData, UINT cbData, PUINT pcbWrite);
```

#### Parameters

**hFile:** (type: TcFileHandle) [in] refers to the prior opened file

**pData:** (type: PVOID) [in] location of the data to be written

**cbData:** (type: PVOID) [in] size of data to be written (size of memory behind pData)

**pcbRead:** (type: PUINT) [out] size of written data

#### Return Value

Type: HRESULT

If any data could be written, S\_OK is returned.

Error codes of special interest:

- ADS\_E\_TIMEOUT when timeout (5 seconds) has elapsed.

Further [ADS status codes](#) [▶ 300] could occur.

#### Description

This method writes data to a file defined by the file handle. Data will be read from pData while pcbRead provides length of data.

### 12.4.3.5 Method ITcFileAccess:FileSeek

Sets position in file.

#### Syntax

```
virtual HRESULT TCOMAPI FileSeek(TcFileHandle hFile, UINT uiPos);
```

#### Parameters

**hFile:** (type: TcFileHandle) [in] refers to the prior opened file

**uiPos:** (type: UINT) [in] position to set to

#### Return Value

Type: HRESULT

If position could be set, S\_OK is returned.

Error codes of special interest:

- ADS\_E\_TIMEOUT when timeout (5 seconds) has elapsed.

Further [ADS status codes](#) [▶ 300] could occur.

#### Description

This method sets the position within the file for further actions

### 12.4.3.6 Method ITcFileAccess:FileTell

Retrieves position in file.

#### Syntax

```
virtual HRESULT TCOMAPI FileTell(TcFileHandle hFile, PUINT puiPos);
```

#### Parameters

**hFile:** (type: TcFileHandle) [in] refers to the prior opened file

**puiPos:** (type: PUINT) [out] location of the position to be returned

#### Return Value

Type: HRESULT

If position could be retrieved, S\_OK is returned.

Error codes of special interest:

- ADS\_E\_TIMEOUT when timeout (5 seconds) has elapsed.

Further [ADS status codes](#) [► 300] could occur.

#### Description

This method retrieves the position within the file, which is currently set.

### 12.4.3.7 Method ITcFileAccess:FileRename

Renames a file

#### Syntax

```
virtual HRESULT TCOMAPI FileRename(PCCH szOldName, PCCH szNewName);
```

#### Parameters

**szOldName:** (type: PCCH) [in] the filename to be renamed

**szNewName:** (type: PCCH) [in] the new filename

#### Return Value

Type: HRESULT

If file could be renamed, S\_OK is returned.

Error codes of special interest:

- ADS\_E\_TIMEOUT when timeout (5 seconds) has elapsed.

Further [ADS status codes](#) [► 300] could occur.

#### Description

This method renames a file from an old name to a new name.

### 12.4.3.8 Method ITcFileAccess:FileDelete

Deletes a file.

**Syntax**

```
virtual HRESULT TCOMAPI FileDelete(PCCH szFileName);
```

**Parameters**

**szFileName:** (type: PCCH) [in] the filename to be deleted

**Return Value**

Type: HRESULT

If file could be deleted, S\_OK is returned.

Error codes of special interest:

- ADS\_E\_TIMEOUT when timeout (5 seconds) has elapsed.

Further [ADS status codes](#) [▶ 300] could occur.

**Description**

This method deletes a file from file system

**12.4.3.9 Method ITcFileAccess:FileGetStatus**

Retrieves status of a file.

**Syntax**

```
virtual HRESULT TCOMAPI FileGetStatus(PCCH szFileName, PTcFileStatus pFileStatus);
```

**Parameters**

**szFileName:** (type: PCCH) [in] the filename of interest

**pFileStatus:** (type: PTcFileStatus) [out] the status of the file. Compare *TcFileAccessServices.h*.

**Return Value**

Type: HRESULT

If status could be returned, S\_OK is returned.

Error codes of special interest:

- ADS\_E\_TIMEOUT when timeout (5 seconds) has elapsed.

Further [ADS status codes](#) [▶ 300] could occur.

**Description**

This method retrieves Status information of 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;
```

### 12.4.3.10 Method ITcFileAccess:FileFindFirst

Capability to step through files of a directory.

#### Syntax

```
virtual HRESULT TCOMAPI FileFindFirst (PCCH szFileName, PTcFileFindData pFileFindData ,  
PTcFileFindHandle phFileFind);
```

#### Parameters

**szFileName:** (type: PCCH) [in] Directory or path, and the file name to find. The file name can include wildcard characters like an asterisk (\*) or a question mark (?).

**pFileFindData:** (type: PTcFileFindData) [out] the description of the first file. Compare *TcFileAccessServices.h*.

**phFileFind:** (type: PTcFileFindHandle) [out] handle to search further on with FileFindNext.

#### Return Value

Type: HRESULT

If any file could be found, S\_OK is returned.

Error codes of special interest:

- ADS\_E\_TIMEOUT when timeout (5 seconds) has elapsed.

Further [ADS status codes](#) [▶ 300] could occur.

#### Description

This method starts with finding files in a defined directory. The Method provides access to PTcFileFindData of the first found file, which contains 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;
```

### 12.4.3.11 Method ITcFileAccess:FileFindNext

Step further on through files of a directory.

#### Syntax

```
virtual HRESULT TCOMAPI FileFindNext (TcFileFindHandle hFileFind, PTcFileFindData pFileFindData);
```

#### Parameters

**hFileFind:** (type: PTcFileFindHandle) [in] handle to search further on with FileFindNext.

**pFileFindData:** (type: PTcFileFindData) [out] the description of the next file. Compare *TcFileAccessServices.h*.

### Return Value

Type: HRESULT

If any file could be found, S\_OK is returned.

Error codes of special interest:

- ADS\_E\_TIMEOUT when timeout (5 seconds) has elapsed.

Further [ADS status codes](#) [▶ 300] could occur.

### Description

This method finds next file in a directory. The Method provides access to PTcFileFindData of the found file, which contains 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;
```

### 12.4.3.12 Method ITcFileAccess:FileFindClose

Close finding files of a directory.

#### Syntax

```
virtual HRESULT TCOMAPI FileFindClose (TcFileFindHandle hFileFind);
```

#### Parameters

**hFileFind:** (type: PTcFileFindHandle) [in] handle to close searching

#### Return Value

Type: HRESULT

If any file search could be closed, S\_OK is returned.

Error codes of special interest:

- ADS\_E\_TIMEOUT when timeout (5 seconds) has elapsed.

Further [ADS status codes](#) [▶ 300] could occur.

#### Description

This method closes finding of files in a directory.

### 12.4.3.13 Method ITcFileAccess:Mkdir

Create a directory on the filesystem.

### Syntax

```
virtual HRESULT TCOMAPI Mkdir(PCCH szDir);
```

### Parameters

**szDir:** (type: PCCH) [in] directory to create

### Return Value

Type: HRESULT

If directory could be created, S\_OK is returned.

Error codes of special interest:

- ADS\_E\_TIMEOUT when timeout (5 seconds) has elapsed.

Further [ADS status codes](#) [▶ 300] could occur.

### Description

This method creates a directory as defined by the szDir parameter.

## 12.4.3.14 Method ITcFileAccess:Rmdir

Delete a directory from the filesystem.

### Syntax

```
virtual HRESULT TCOMAPI Rmdir(PCCH szDir);
```

### Parameters

**szDir:** (type: PCCH) [in] directory to be deleted

### Return Value

Type: HRESULT

If directory could be deleted, S\_OK is returned.

Error codes of special interest:

- ADS\_E\_TIMEOUT when timeout (5 seconds) has elapsed.

Further [ADS status codes](#) [▶ 300] could occur.

### Description

This method deletes a directory as defined by the szDir parameter.

## 12.4.4 Interface ITcFileAccessAsync

Asynchronous access to file operations.


This interface extends [ITcFileAccess](#) [▶ 143].

### Syntax




```
TCOM_DECL_INTERFACE("C04AC244-C126-466E-982E-93EC571F2277", ITcFileAccessAsync) struct  
_declspec(novtable) ITcFileAccessAsync: public ITcFileAccess
```

Required include: TcFileAccessInterfaces.h

**Methods**

Symbol	Name	Description
	<a href="#">C</a>   <a href="#">▶</a> <a href="#">_152</a>   <a href="#">heck</a>	Query the state of the file operation

**Interface parameters**

Sym- bol	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

**Remarks**

Interface can be obtained from module instance with class id CID\_TcFileAccessAsync.  
When using the asynchronous, interface methods inherited from the synchronous variant will return ADS\_E\_PENDING if request has been successfully submitted, but is not yet finished. If called while the previous request is still processed, the error code ADS\_E\_BUSY will be returned.

Description of module parameters:

- PID\_TcFileAccessAsyncAdsProvider: Object ID of a task providing the ADS interface.
- PID\_TcFileAccessAsyncNetId / PID\_TcFileAccessAsyncNetIdStr: AmsNetId of the system service use for file access. The “Str” variant takes the AmsNetId as String. Please use one.
- PID\_TcFileAccessAsyncTimeoutMs: Timeout used for a file access
- PID\_TcFileAccessAsyncSegmentSize: Read and write file access is fragmented using this segment size

Please have a look at [Sample20a: FileIO-Cyclic Read / Write](#) | [▶](#) [269](#)

**12.4.4.1 Method ITcFileAccessAsync::Check()**

Retrieve state of the file operation

**Syntax**

```
virtual HRESULT TCOMAPI Check();
```

**Parameters**

none

**Return Value**

Type: HRESULT

Returns S\_OK, if file operation is completed

Error codes of special interest:

- ADS\_E\_PENDING, if the file operation is not completed.
- ADS\_E\_TIMEOUT, if the timeout for the file operation elapsed.

Further [ADS statuscodes](#) | [▶](#) [300](#) could occur.

**Description**

This operation checks the state of the prior called file operation

**12.4.5 Interface ITcIoCyclic**



Interface is implemented by TwinCAT modules which should be called on input update and on 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**

Icon	Name	Description
	<a href="#">InputUpdate</a> [ <a href="#">▶ 153</a> ]	Called at the beginning of a task cycle if interface has been registered with a cyclic I/O caller.
	<a href="#">OutputUpdate</a> [ <a href="#">▶ 154</a> ]	Called at the end of a task cycle if interface has been registered with a cyclic I/O caller

**Remarks**

ITcIoCyclic can be used to implement a TwinCAT module which acts as a fieldbus driver or as a I/O filter module.

This interface is passed to method ITcIoCyclicCaller::AddIoDriver when a module registers itself with a task, typically as the last initialization step in the SafeOP to OP transition. After registration the methods InputUpdate() and OutputUpdate() of the module instance are called, each once per task cycle.

**12.4.5.1 Method ITcIoCyclic:InputUpdate**

The method InputUpdate is usually called by a TwinCAT task, after the interface has been registered.

**Syntax**

```
virtual HRESULT TCOMAPI InputUpdate(ITcTask* ipTask,
ITcUnknown* ipCaller, DWORD dwStateIn, ULONG_PTR context = 0)=0;
```

**Parameters**

**ipTask:** (type: ITcTask) refers to the current task context.

**ipCaller:** (type: ITcUnknown) refers to the calling instance.

**dwStateIn:** (type: DWORD) reserved for future extensions, currently this is always zero

**context:** (type: ULONG\_PTR) context contains the value which has been passed to method ITcIoCyclicCaller::AddIoDriver()

**Return Value**

It is recommended to always return S\_OK. Currently, the return value is ignored by TwinCAT Tasks.

## Description

In a task cycle the method `InputUpdate()` is first called for all registered module instances. Therefore this method must be used for updating the data areas of the type `Input-Source` of the module.

### 12.4.5.2 Method `ITcIoCyclic:OutputUpdate`

The method `OutputUpdate` is usually called by a TwinCAT task, after the interface has been registered.

#### Syntax

```
virtual HRESULT TCOMAPI OutputUpdate(ITcTask* ipTask, ITcUnknown* ipCaller,
PDWORD pdwStateOut = NULL, ULONG_PTR context = 0)=0;
```

#### Parameters

**ipTask:** (type: `ITcTask`) refers to the current task context.

**ipCaller:** (type: `ITcUnknown`) refers to the calling instance.

**pdwStateOut:** (type: `DWORD`) [out] reserved for future extensions, currently returned value is ignored.

**context:** (type: `ULONG_PTR`) context contains the value which has been passed to method `ITcCyclicCaller::AddIoDriver()`

#### Return Value

It is recommended to always return `S_OK`. Currently, the return value is ignored by TwinCAT Tasks.

#### Description

In a task cycle the method `OutputUpdate()` 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.

## 12.4.6 Interface `ITcIoCyclicCaller`

Interface to register or unregister a module's `ITcIoCyclic` interface with a TwinCAT task.

#### Syntax

```
TCOM_DECL_INTERFACE("0300001F-0000-0000-e000-000000000064", ITcIoCyclicCaller)
struct __declspec(novtable) ITcIoCyclicCaller : public ITcUnknown
```

Required include: `TcIoInterfaces.h`

#### Methods

Icon	Name	Description
	<a href="#">AddIoDriver [▶ 155]</a>	Register module which implements the <code>ITcIoCyclic</code> interface.
	<a href="#">RemoveIoDriver [▶ 155]</a>	Unregister the previously registered <code>ITcIoCyclic</code> interface of a module.

#### Remarks

The `ITcIoCyclicCaller` interface is implemented by TwinCAT tasks. A module uses this interface to register its `ITcIoCyclic` interface with a task, typically as the last initialization step in the `SafeOP` to `OP` transition. After registration the method `CycleUpdate()` of the module instance is called. The interface is also used to unregister the module from being called by a task.

### 12.4.6.1 Method ITcIoCyclicCaller:AddIoDriver

Register a module's ITcIoCyclic interface with cyclic I/O caller, i.e. a TwinCAT task

#### Syntax

```
virtual HRESULT TCOMAPI AddIoDriver(STcIoCyclicEntry*
pEntry, ITcIoCyclic* ipDrv, ULONG_PTR context=0, ULONG sortOrder=0);
```

#### Parameters

**pEntry:** (type: STcIoCyclicEntry) pointer to a list entry, which is inserted into the internal list of the cyclic I/O caller. See also description.

**ipDrv:** (type: ITcIoCyclic) [in] interface pointer which will be used by cyclic I/O caller

**context:** (type: ULONG\_PTR) [optional] a context value which is passed to the ITcIoCyclic::InputUpdate() and ITcIoCyclic::OutputUpdate method on each call.

**sortOrder:** (type: ULONG) [optional] the sort order can be used to control the order of execution if different module instances are executed by the same cyclic caller.

#### Return Value

Type: HRESULT

#### Description

A TwinCAT module class usually uses a smart pointer to refer to the 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 using the TwinCAT object server. In addition the smart pointer class already contains a list entry. Therefore the smart pointer can be used as first parameter for the AddIoDriver method.

The following sample code shows the registration 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;
}
}
}
```

### 12.4.6.2 Method ITcIoCyclicCaller:RemovelIoDriver

Unregister a module instance from being called by a cyclic I/O caller.

#### Syntax

```
virtual HRESULT TCOMAPI
RemoveIoDriver(STcIoCyclicEntry* pEntry)=0;
```

#### Parameters

**pEntry:** (type: STcIoCyclicEntry) refers to the list entry which should be removed from the internal list of the cyclic I/O caller.

**Return Value**

If the entry is not in the internal list, the method returns E\_FAIL.

**Description**

Similar to the method AddIoDriver() the smart pointer for the cyclic I/O caller is used as list entry when the module instance should be removed from cyclic I/O caller.

Declaration of smart pointer and usage:

```
ITcIoCyclicCallerInfoPtr
m_spIoCyclicCaller;

if ( m_spIoCyclicCaller )
{
m_spIoCyclicCaller->RemoveIoDriver(m_spIoCyclicCaller);
}
m_spCyclicCaller = NULL;
```











**12.4.7 Interface ITComObject**

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**

Sym- bol	Name	Description
	<a href="#">TcGetObjectId(OTCID&amp; objId)</a> <a href="#">[▶ 156]</a>	Saves the object ID with the help of the given OTCID reference.
	<a href="#">TcSetObjectId</a> <a href="#">[▶ 157]</a>	Sets the object ID of the object to the given OTCID
	<a href="#">TcGetObjectName</a> <a href="#">[▶ 157]</a>	Saves the object names in the buffer with the given length
	<a href="#">TcSetObjectName</a> <a href="#">[▶ 158]</a>	Sets the object name of the object to given CHAR*
	<a href="#">TcSetObjState</a> <a href="#">[▶ 158]</a>	Initializes a transition to a predefined state.
	<a href="#">TcGetObjState</a> <a href="#">[▶ 158]</a>	Queries the current state of the object.
	<a href="#">TcGetObjPara</a> <a href="#">[▶ 159]</a>	Queries an object parameter identified with its PTCID
	<a href="#">TcSetObjPara</a> <a href="#">[▶ 159]</a>	Sets an object parameter identified with its PTCID
	<a href="#">TcGetParentObjId</a> <a href="#">[▶ 160]</a>	Saves the parent object ID with the help of the given OTCID reference.
	<a href="#">TcSetParentObjId</a> <a href="#">[▶ 160]</a>	Sets the parent object ID to the given OTCID.

**Comments**

The ITComObject interface is implemented by every TwinCAT module. It makes functionalities available regarding the state machine and Information from/to the TwinCAT system.

**12.4.7.1 Method ITComObject:TcGetObjectId(OTCID& objId)**

The method saves the object ID with the help of the given OTCID reference.



## Syntax

```
HRESULT TcGetObjectId( OTCID& objId )
```

## Parameters

**objId:** (type: OTCID&) Reference to OTCID value

## Return Value

Indicates success of OTCID retrieval.

## Description

The method stores Object ID using given OTCID reference.

### 12.4.7.2 Method ITcComObject:TcSetObjectId

The method TcSetObjectId sets object's object ID to the given OTCID.

## Syntax

```
HRESULT TcSetObjectId( OTCID objId )
```

## Parameters

**objId:** (type: OTCID) The OTCID, which should be set.

## Return Value

It is recommended to always return S\_OK. Currently, the return value is ignored by TwinCAT tasks.

## Description

Indicates success of id change.

### 12.4.7.3 Method ITcComObject:TcGetObjectName

The method TcGetObjectName stores the Object name into buffer with given length.

## Syntax

```
HRESULT TcGetObjectName( CHAR* objName, ULONG nameLen );
```

## Parameters

**objName:** (type: CHAR\*) the name, which should be set.

**nameLen:** (type: ULONG) the maximum length to write.

## Return Value

Indicates success of name retrieval.

## Description

The method TcGetObjectName stores the Object name into buffer with given length.

#### 12.4.7.4 Method ITcComObject:TcSetObjectName

The method TcSetObjectName sets objects's Object Name to the given CHAR\*.

##### Syntax

```
HRESULT TcSetObjectName( CHAR* objName )
```

##### Parameters

**objName:** (type: CHAR\*) the name of the object to be set

##### Return Value

Indicates success of name change.

##### Description

The method TcSetObjectName sets objects's Object Name to the given CHAR\*.

#### 12.4.7.5 Method ITcComObject:TcSetObjState

The method TcSetObjState initializes a transition to given state.

##### Syntax

```
HRESULT TcSetObjState(TCOM_STATE state, ITComObjectServer* ipSrv, PTCComInitDataHdr pInitData);
```

##### Parameters

**state:** (type: TCOM\_STATE) represents the new state

**ipSrv:** (type: ITComObjectServer\*) handles the object

**pInitData:** (type: PTCComInitDataHdr) Points to a list of parameters (optional)

See macro IMPLEMENT\_ITCOMOBJECT\_EVALUATE\_INITDATA for an example how the list can be iterated.

##### Return Value

Indicates success of state change.

##### Description

The method TcSetObjState initializes a transition to given state.

#### 12.4.7.6 Method ITcComObject:TcGetObjState

The method TcGetObjState retrieves the current state of the object.

##### Syntax

```
HRESULT TcGetObjState(TCOM_STATE* pState)
```

##### Parameters

**pState:** (type: TCOM\_STATE\*) pointer to the state

**Return Value**

Indicates success of state retrieval.

**Description**

The method TcGetObjState retrieves the current state of the object.

**12.4.7.7 Method ITcComObject:TcGetObjPara**

The method TcGetObjPara retrieves a object parameter identified by its PTCID.

**Syntax**

```
HRESULT TcGetObjPara(PTCID pid, ULONG& nData, PVOID& pData, PTCGP pgp=0)
```

**Parameters**

**pid:** (type: PTCID) Parameter ID of the object parameter

**nData:** (type: ULONG&) max length of the data

**pData:** (type: PVOID&) Pointer to the data

**pgp:** (type: PTCGP) reserved for future extension, pass NULL

**Return Value**

Indicates success of object parameter retrieval.

**Description**

The method TcGetObjPara retrieves a object parameter identified by its PTCID.

**12.4.7.8 Method ITcComObject:TcSetObjPara**

The method TcSetObjPara sets a object parameter identified by its PTCID.

**Syntax**

```
HRESULT TcSetObjPara(PTCID pid, ULONG nData, PVOID pData, PTCGP pgp=0)
```

**Parameters**

**pid:** (type: PTCID) Parameter ID of the object parameter

**nData:** (type: ULONG) max length of the data

**pData:** (type: PVOID) Pointer to the data

**pgp:** (type: PTCGP) reserved for future extension, pass NULL

**Return Value**

Indicates success of object parameter retrieval.

**Description**

The method TcSetObjPara sets a object parameter identified by its PTCID.

### 12.4.7.9 Method ITcComObject:TcGetParentObjId

The method TcGetParentObjId stores Parent Object ID using given OTCID reference.

#### Syntax

```
HRESULT TcGetParentObjId( OTCID& objId )
```

#### Parameters

**objId:** (type: OTCID&) Reference to OTCID value

#### Return Value

Indicates success of parentObjId retrieval.

#### Description

The method TcGetParentObjId stores Parent Object ID using given OTCID reference.

### 12.4.7.10 Method ITcComObject:TcSetParentObjId

The method TcSetParentObjId sets Parent Object ID using given OTCID reference.

#### Syntax

```
HRESULT TcSetParentObjId( OTCID objId )
```

#### Parameters

**objId:** (type: OTCID) Reference to OTCID value

#### Return Value

It is recommended to always return S\_OK. Currently, the return value is ignored by TwinCAT Tasks.

#### Description

The method TcSetParentObjId sets Parent Object ID using given OTCID reference.

## 12.4.8 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

Symbol	Name	Description
	<code>OTCID TcGetObjectID</code> <a href="#">▶ 161</a>	Queries the object ID.
	<code>TcTryToReleaseOpState</code> <a href="#">▶ 161</a>	Releases resources; must be implemented

## Comments

Further methods exist, which are not itemized here.

This functionality is provided as standard by the module wizards.

### 12.4.8.1 TcGetObjectId method

The method queries the object ID.

#### Syntax

```
OTCID TcGetObjectId(void)
```

#### Parameters

#### Return Value

OTCID: Returns the OTCID of the object.

#### Description

The method TcGetObjectId retrieves the Object ID of the object.

### 12.4.8.2 TcTryToReleaseOpState method

The method TcTryToReleaseOpState releases resources, e.g. data pointer, in order to prepare for leaving the OP state.

#### Syntax

```
BOOL TcTryToReleaseOpState(void)
```

#### Parameters

#### Return Value

TRUE means success in releasing resources.

#### Description

The method TcTryToReleaseOpState releases resources, e.g. data pointer, in order to prepare for leaving the OP state. Should be implemented to resolve possible circular dependencies among module instances. See [sample 10](#) [[▶ 235](#)] for an example.

## 12.4.9 Interface ITcPostCyclic


Interface is implemented by TwinCAT modules which should be called once per task cycle after the output update (comparable to 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

Icon	Name	Description
	<a href="#">PostCycleUpdate</a> [ <a href="#">▶_162</a> ]	Called once per task cycle after the output update if interface has been registered with a cyclic caller.

## Remarks

The ITcPostCyclic interface is implemented by TwinCAT modules. This interface is passed to method ITcCyclicCaller::AddPostModule() when a module registers itself with a task, typically as the last initialization step in the SafeOP to OP transition. After registration the method PostCycleUpdate() of the module instance is called.

### 12.4.9.1 Method ITcPostCyclic:PostCyclicUpdate

The method PostCyclicUpdate usually called by a TwinCAT task after the output update, after the interface has been registered.

## Syntax

```
HRESULT TCOMAPI PostCycleUpdate(ITcTask* ipTask, ITcUnknown* ipCaller, ULONG_PTR context)
```

## Parameters

ipTask: (type: ITcTask) refers to the current task context.

ipCaller: (type: ITcUnknown) refers to the calling instance.

Context: (type: ULONG\_PTR) context contains the value which has been passed to method ITcPostCyclicCaller::AddPostModule()

## Return Value

It is recommended to always return S\_OK. Currently, the return value is ignored by TwinCAT Tasks.

## Description

Within a task cycle the method PostCycleUpdate() is called after OutputUpdate() has been for all registered module instances. Therefore, this method should be used to implement such cyclic processing.

## 12.4.10 Interface ITcPostCyclicCaller



Interface to register or unregister a module's ITcPostCyclic interface with 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

Icon	Name	Description
	<a href="#">AddPostModule</a> [ <a href="#">▶_163</a> ]	Register module which implements the ITcPostCyclic interface.
	<a href="#">RemovePostModule</a> [ <a href="#">▶_164</a> ]	Unregister the previously registered ITcPostCyclic interface of a module.

## Remarks

The ITcPostCyclicCaller interface is implemented by TwinCAT tasks. A module uses this interface to register its ITcPostCyclic interface with a task, typically as the last initialization step in the SafeOP to OP transition. After registration the method PostCycleUpdate() of the module instance is called. The interface is also used to unregister the module from being called by a task.

### 12.4.10.1 Method ITcPostCyclicCaller:AddPostModule

Register a module's ITcPostCyclic interface with cyclic caller, i.e. a TwinCAT task.

#### Syntax

```
virtual HRESULT TCOMAPI  
AddPostModule(STcPostCyclicEntry* pEntry, ITcPostCyclic* ipMod, ULONG_PTR  
context=0, ULONG sortOrder=0)=0;
```

#### Parameters

**pEntry:** (type: STcPostCyclicEntry) [in] pointer to a list entry, which is inserted into the internal list of the cyclic caller. See also description.

**ipMod:** (type: ITcPostCyclic) [in] interface pointer which will be used by cyclic caller

**context:** (type: ULONG\_PTR) [optional] a context value which is passed to the ITcPostCyclic::PostCyclicUpdate() method on each call.

**sortOrder:** (type: ULONG) [optional] the sort order can be used to control the order of execution if different module instances are executed by the same cyclic caller.

#### Return Value

Type: HRESULT

On success the method returns S\_OK. If cyclic caller, i.e. the TwinCAT task, is not in OP state, the error ADS\_E\_INVALIDSTATE is returned.

#### 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 point and a reference to the task can be obtained using the TwinCAT object server. In addition the smart pointer class already contains a list entry. Therefore the smart pointer can be used as first parameter for the AddPostModule method.

The following sample code shows the registration 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;  
  
}  
  
}  
  
}
```

### 12.4.10.2 Method ITcPostCyclicCaller:RemovePostModule

Unregister a module instance from being called by a cyclic caller.

#### Syntax

```
virtual HRESULT TCOMAPI
RemovePostModule(STcPostCyclicEntry* pEntry)=0;
```

#### Parameters

pEntry: (type: STcPostCyclicEntry) refers to the list entry which should be removed from the internal list of the cyclic caller.

#### Return Value

If the entry is not in the internal list, the method returns E\_FAIL.

#### Description

Similar to the method AddPostModule() the smart pointer for the cyclic caller is used as list entry when the module instance should be removed from cyclic caller.

Declaration and usage of smart pointer:

```
ITcPostCyclicCallerInfoPtr m_spPostCyclicCaller;
```

```
if (
m_spPostCyclicCaller ) {
m_spPostCyclicCaller->RemovePostModule(m_spPostCyclicCaller);
}

m_spPostCyclicCaller = NULL;
```

## 12.4.11 Interface ITcRTimeTask


Retrieve 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

Icon	Name	Description
	<a href="#">GetCpuAccount</a> [ <a href="#">▶ 164</a> ]	Retrieve the CPU account of a TwinCAT task.

#### Remarks

Retrieving and using TwinCAT task Information could be done by this interface.

Please have a look at [Sample30: Timing Measurement](#) [[▶ 277](#)]

### 12.4.11.1 Method ITcRTimeTask::GetCpuAccount()

Retrieve the CPU account of a TwinCAT task.



**Syntax**

```
virtual HRESULT TCOMAPI GetCpuAccount(PULONG pAccount)=0;
```

**Parameters**

**pAccount:** (type: PULONG) [out] TwinCAT task CPU account is stored in this parameter.

**Return Value**

E\_POINTER if parameter pAccount is NULL, otherwise S\_OK.

**Description**

The method GetCpuAccount() allows to retrieve the actual computation time used by the task.

Code fragment which shows the usage of GetCpuAccount() e.g. within method ITcCyclic::CycleUpdate():

```
// 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;
}
```

**12.4.12 Interface ITcTask**







Retrieve timestamps and task specific information from a TwinCAT task.

**Syntax**

```
TCOM_DECL_INTERFACE("02000002-0000-0000-e000-000000000064", ITcTask)
struct __declspec(novtable) ITcTask : public ITcUnknown
```

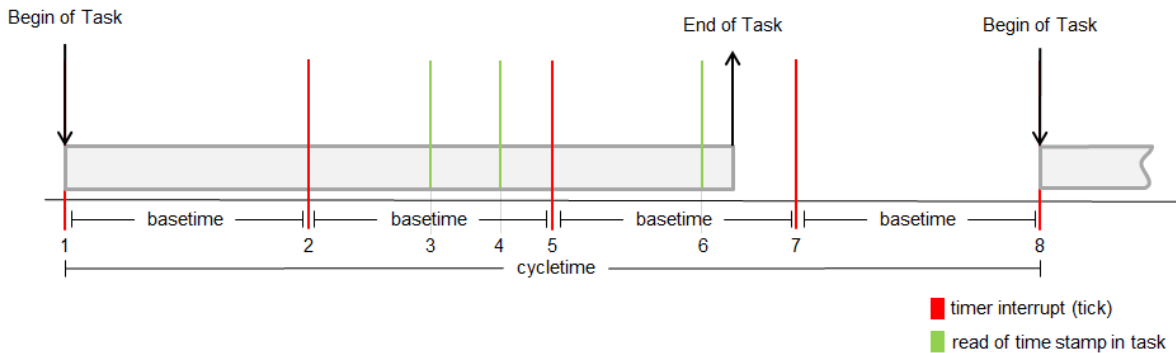
Required include: TcRtInterfaces.h

**Methods**

Icon	Name	Description
	<a href="#">GetCycleCounter</a> [ <a href="#">▶ 167</a> ]	Retrieve number of task cycles since task start
	<a href="#">GetCycleTime</a> [ <a href="#">▶ 168</a> ]	Retrieve the task cycle time in nanoseconds, i.e. the time between "begin of task" and next "begin of task"
	<a href="#">GetPriority</a> [ <a href="#">▶ 166</a> ]	Retrieve task priority
	<a href="#">GetCurrentSysTime</a> [ <a href="#">▶ 166</a> ]	Retrieve time at task cycle start in 100 nanoseconds intervals since 1. January 1601 (UTC)
	<a href="#">GetCurrentDcTime</a> [ <a href="#">▶ 167</a> ]	Retrieve distributed clock time at task cycle start in nanoseconds since 1. January 2000
	<a href="#">GetCurPentiumTime</a> [ <a href="#">▶ 167</a> ]	Retrieve time at method call in 100 nanoseconds intervals since 1. January 1601 (UTC)

**Remarks**

The ITcTask interface can be used to measure time within the RT 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

**12.4.12.1 Method ITcTask:GetPriority**

Retrieve task priority

**Syntax**

```
virtual HRESULT TCOMAPI GetPriority(PULONG pPriority)=0;
```

**Parameters**

**pPriority:** (type: PULONG) [out] task priority value is stored in this parameter.

**Return Value**

E\_POINTER if parameter pPriority is NULL, otherwise S\_OK.

**Description**

[Sample30: Timing Measurement \[▶ 277\]](#) shows usage of this method.

**12.4.12.2 Method ITcTask:GetCurrentSysTime**

Retrieve time at task cycle start in 100 nanoseconds intervals since 1. January 1601 (UTC)

**Syntax**

```
virtual HRESULT TCOMAPI GetCurrentSysTime(PLONGLONG pSysTime)=0;
```

**Parameters**

**pSysTime:** (type: PLONGLONG) [out] current system time at task cycle start is stored in this parameter.

**Return Value**

E\_POINTER if parameter pSysTime is NULL, otherwise S\_OK.

**Description**

[Sample30: Timing Measurement \[▶ 277\]](#) shows usage of this method.

**12.4.12.3 Method ITcTask:GetCurrentDcTime**

Retrieve distributed clock time at task cycle start in nanoseconds since 1. January 2000

**Syntax**

```
virtual HRESULT TCOMAPI GetCurrentDcTime(PLONGLONG  
pDcTime)=0;
```

**Parameters**

**pDcTime:** (type: PLONGLONG) [out] distributed clock time at task cycle start is stored in this parameter.

**Return Value**

E\_POINTER if parameter pDcTime is NULL, otherwise S\_OK.

**Description**

[Sample30: Timing Measurement \[▶ 277\]](#) shows usage of this method.

**12.4.12.4 Method ITcTask:GetCurPentiumTime**

Retrieve time at method call in 100 nanoseconds intervals since 1. January 1601 (UTC)

**Syntax**

```
virtual HRESULT TCOMAPI GetCurPentiumTime(PLONGLONG  
pCurTime)=0;
```

**Parameters**

**pCurTime:** (type: PLONGLONG) [out] current time (UTC) in 100 nanoseconds intervals since 1. January 1601 is stored in this parameter

**Return Value**

E\_POINTER if parameter pCurTime is NULL, otherwise S\_OK.

**Description**

[Sample30: Timing Measurement \[▶ 277\]](#) shows usage of this method.

**12.4.12.5 Method ITcTask:GetCycleCounter**

Retrieve number of task cycles since task start.

**Syntax**

```
virtual HRESULT TCOMAPI GetCycleCounter(PULONGLONG
pCnt)=0;
```

**Parameters**

**pCnt:** (type: PULONGLONG) [out] number of task cycles since task has been started is stored in this parameter

**Return Value**

E\_POINTER if parameter pCnt is NULL, otherwise S\_OK

**Description**

[Sample30: Timing Measurement \[▶ 277\]](#) shows usage of this method.

**12.4.12.6 Method ITcTask:GetCycleTime**

Retrieve the task cycle time in nanoseconds, i.e. the time between "begin of task" and next "begin of task"

**Syntax**

```
virtual HRESULT TCOMAPI GetCycleTime(PULONG
pCycleTimeNS)=0;
```

**Parameters**

**pCycleTimeNS:** (type: PULONG) [out] the configured task cycle time in nanoseconds is stored in this parameter.

**Return Value**

E\_POINTER if parameter pCycleTimeNS is NULL, otherwise S\_OK.

**Description**

[Sample30: Timing Measurement \[▶ 277\]](#) shows usage of this method.

**12.4.13 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**

Symbol	Name	Description
	NotifyCycleTimeExceeded	Called if the cycle time was exceeded.

**Remarks**

Please note that the callback will not take place within the calculations but on the end of the cycle. So this method does not provide a mechanism to immediately stop calculations.

**12.4.13.1 Method ITcTaskNotification::NotifyCycleTimeExceeded()**

Gets called if cycle time was exceeded beforehand

**Syntax**

```
virtual HRESULT TCOMAPI NotifyCycleTimeExceeded ();
```

**Parameters**

**ipTask:** (type: ITcTask) refers to the current task context.

**context:** (type: ULONG\_PTR) context

**Return Value**

Type: HRESULT

Return S\_OK, if file operation is completed

**Description**

Gets called if cycle time was exceeded beforehand. So not immediately on exceeded time, but afterwards.

**12.4.14 Interface ITcUnknown**

ITcUnknown defines the reference counting as well as querying a reference to a more specific interface.




**Syntax**

```
TCOM_DECL_INTERFACE("00000001-0000-0000-e000-000000000064", ITcUnknown)
```

Declared in: TcInterfaces.h

Required include: -

**Methods**

Icon	Name	Description
	<a href="#">TcAddRef [► 169]</a>	Increments the reference counter.
	<a href="#">TcQueryInterface [► 170]</a>	Query reference to an implemented interface by the IID
	<a href="#">TcRelease [► 171]</a>	Decrements the reference counter.

**Remarks**

Every TcCOM interface is directly or indirectly derived from ITcUnknown. As a consequence every TcCOM module class implements ITcUnknown, because it is derived from ITComObject.

The default implementation for ITcUnknown will delete the object if its last reference is released. Therefore an interface pointer must not be dereferenced after TcRelease() has been called.

**12.4.14.1 Method ITcUnknown:TcAddRef**

This method increments the reference counter.

**Syntax**

```
ULONG TcAddRef( )
```

**Return Value**

Resulting reference count value.

**Description**

Increments the reference counter and returns the new value..

**12.4.14.2 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 )
```

**iid:** (Type: RITCID) Interface IID

**pipItf:** (PPVOID Type) pointer to interface pointer. Is set when the requested interface type is available from the corresponding instance.

**Return Value**

A return value S\_OK indicates success.

If requested interface is not available the method will return ADS\_E\_NOINTERFACE.

**Description**

Query reference to an implemented interface by 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)
    {
        IComObjectPtr 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 parameter in TcQueryInterface. The operator “&” will return pointer to internal interface pointer member of the smart pointer. Variant 1 assumes that interface pointer is initialized if TcQueryInterface indicates success. If scope is left the destructor of the smart pointer spObj releases the reference.

**Variant 2:**

```
HRESULT GetTraceLevel(ITcUnkown* ip, TcTraceLevel& tl)
{
    HRESULT hr = S_OK;
    IComObjectPtr spObj = ip;
    if (spObj != NULL)
    {
        spObj->TcGetObjParam(PID_TcTraceLevel, &tl);
    }
}
```

```
else
{
hr = ADS_E_NOINTERFACE;
}
return hr;
}
```

When assigning interface pointer `ip` to smart pointer `spObj` method `TcQueryInterface` is implicitly called with `IID_ITComObject` on the instance `ip` refers to. This results in shorter code, however it loses the original return code of `TcQueryInterface`.

### 12.4.14.3 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 reference counter gets zero, object deletes itself.

## 12.5 Runtime Library (RtlR0.h)

TwinCAT has its own implementation of the runtime library. These functions are declared in `RtlR0.h`, a part of TwinCAT SDK.

**Methods provided**

	Name	Description
	abs	Calculates the absolute value.
	atof	Converts a string (char *buf) into a double.
	BitScanForward	Searches for a set bit (1) from LSB to MSB.
	BitScanReverse	Searches for a set bit (1) from MSB to LSB.
	labs	Calculates the absolute value.
	memcmp	Compares two buffers
	memcpy	Copies one buffer into another
	memcpy_byte	Copies one buffer into another (byte-wise)
	memset	Sets the bytes of a buffer to a value
	qsort	QuickSort for sorting a list
	snprintf	Writes formatted data into a character string.
	sprintf	Writes formatted data into a character string.
	sscanf	Reads data from a character string after specification of a format.
	strcat	Appends one character string to another.
	strchr	Searches for a character in a character string.
	strcmp	Compares two character strings.
	strcpy	Copies a character string.
	strlen	Determines the length of a character string.
	strncat	Appends one character string to another.
	strncmp	Compares two character strings.
	strncpy	Copies a character string.
	strstr	Searches for a character string within a character string.
	strtol	Converts a character string into an integer.
	strtoul	Converts a character string into an unsigned integer.
	swscanf	Reads data from a character string after specification of a format.
	tolower	Converts a letter into a lower-case letter.
	toupper	Converts a letter into an upper-case letter.
	vsprintf	Writes formatted data into a character string.
	vsprintf	Writes formatted data into a character string.

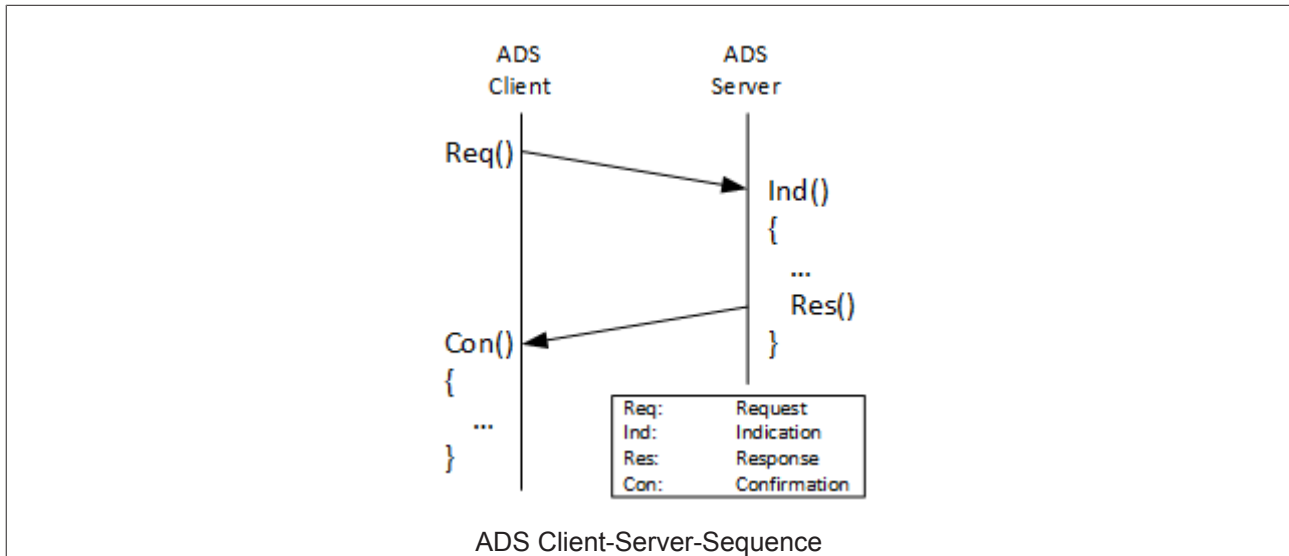
**Comments**

All functions are based on the C++ runtime library.



## 12.6 ADS Communication

ADS based on the Client-Server-principle (figure 1). An ADS request invokes the corresponding indication methods on the server side. The ADS response invokes the corresponding confirmation method on the client side.



This section describes outgoing as well as incoming ADS communication for TwinCAT 3 C++ Modules.

ADS Command Set	Description
<a href="#">AdsReadDeviceInfo [▶ 173]</a>	General device information can be read with this command..
<a href="#">AdsRead [▶ 175]</a>	ADS read command, to request data from an ADS device.
<a href="#">AdsWrite [▶ 177]</a>	ADS write command, to transfer data to an ADS device.
<a href="#">AdsReadState [▶ 181]</a>	ADS command to retrieve state of an ADS device.
<a href="#">AdsWriteControl [▶ 183]</a>	ADS control command to change the state of an ADS device.
<a href="#">AdsAddDeviceNotification [▶ 185]</a>	Observe a variable. The client will be notified by an event.
<a href="#">AdsDelDeviceNotification [▶ 187]</a>	Removes the variable that was connected before.
<a href="#">AdsDeviceNotification [▶ 189]</a>	Used to transmit the device notification event.
<a href="#">AdsReadWrite [▶ 179]</a>	ADS write/read command. Data is transmitted to an ADS device (write) and its response data read with one call.

The [ADS Return Codes \[▶ 300\]](#) apply to the whole ADS communication.

As a starting point, please have a look at [Sample07: Receiving ADS Notifications \[▶ 231\]](#)

### 12.6.1 AdsReadDeviceInfo

#### 12.6.1.1 AdsReadDeviceInfoReq

The method `AdsDeviceInfoReq` permits to send an ADS `DeviceInfo` command for reading the identification and version number of an ADS server.

The `AdsReadDeviceInfoCon` will be called on arrival of the answer.

#### Syntax

```
int AdsReadDeviceInfoReq( AmsAddr& rAddr, ULONG invokeId );
```

## Parameters

**rAddr:** (type: AmsAddr&) [in] structure with NetId and port number of the ADS server.

**invokeld:** (type: ULONG) [in] handle of the command, which is sent. The Invokeld is specified from the source device and serves to identify the commands.

## Return Value

Type: int

error code - see [AdsStatuscodes](#) [▶ 300]

### 12.6.1.2 AdsReadDeviceInfoInd

The method AdsDeviceInfoInd indicates an ADS DeviceInfo command for reading the identification and version number of an ADS server. The [AdsReadDeviceInfoRes](#) [▶ 174] must be called afterwards.

## Syntax

```
void AdsReadDeviceInfoInd( AmsAddr& rAddr, ULONG invokeId );
```

## Parameters

**rAddr:** (type: AmsAddr&) [in] structure with NetId and port number of the ADS server.

**invokeld:** (type: ULONG) [in] handle of the command, which is sent. The Invokeld is specified from the source device and serves to identify the commands.

## Return Value

void

### 12.6.1.3 AdsReadDeviceInfoRes

The method AdsReadDeviceInfoRes sends an ADS Read Device Info. [AdsReadDeviceInfoCon](#) [▶ 175] forms the counterpart and is subsequently called.

## Syntax

```
int AdsReadDeviceInfoRes( AmsAddr& rAddr, ULONG invokeId, ULONG nResult, CHAR  
name[ADS_FIXEDNAMESIZE], AdsVersion version );
```

## Parameter

**rAddr:** (type: AmsAddr&) [in] structure with NetId and port number of the responding ADS server

**invokeld:** (type: ULONG) [in] handle of the command that is sent. The Invokeld is specified by the source device and is used for the identification of the commands.

**nResult:** (type: ULONG) [in] contains the result of the ADS write command. See [AdsStatuscodes](#) [▶ 300].

**name:** (type: char[ADS\_FIXEDNAMESIZE]) [in] contains the name of the device.

**version:** (type: AdsVersion) [in] structure of build (int), revision (byte) and version (byte) of the device

## Return value

Type: int

Error code - see [AdsStatuscodes \[► 300\]](#).

### 12.6.1.4 AdsReadDeviceInfoCon

The method `AdsReadDeviceInfoCon` permits to receive an ADS read device info confirmation. The receiving module has to provide this method. The [AdsReadDeviceInfoReq \[► 173\]](#) is the counterpart and need to be called beforehand.

#### Syntax

```
void AdsReadDeviceInfoCon( AmsAddr& rAddr, ULONG invokeId, ULONG nResult,
CHAR name[ADS_FIXEDNAMESIZE], AdsVersion version );
```

#### Parameters

**rAddr:** (type: `AmsAddr&`) [in] structure with `NetId` and port number of the ADS server answering

**invokeld:** (type: `ULONG`) [in] handle of the command, which is sent. The `Invokeld` is specified from the source device and serves to identify the commands.

**nResult:** (type: `ULONG`) [in] contains the result of the ADS write command. See [AdsStatuscodes \[► 300\]](#)

**name:** (type: `char[ADS_FIXEDNAMESIZE]`) [in] contains the name of the device.

**version:** (type: `AdsVersion`) [in] struct of `Build` (int), `Revision` (byte) and `Version` (byte) of the device

#### Return Value

void

## 12.6.2 AdsRead

### 12.6.2.1 AdsReadReq

The method `AdsReadReq` permits to send an ADS read command, for the transfer of data from an ADS device.

The [AdsReadCon \[► 177\]](#) will be called on arrival of the answer.

#### Syntax

```
int AdsReadReq( AmsAddr& rAddr, ULONG invokeId, ULONG indexGroup, ULONG indexOffset, ULONG
cbLength );
```

#### Parameters

**rAddr:** (type: `AmsAddr&`) [in] structure with `NetId` and port number of the ADS server.

**invokeld:** (type: `ULONG`) [in] handle of the command, which is sent. The `Invokeld` is specified from the source device and serves to identify the commands.

**indexGroup:** (type: `ULONG`) [in] contains the index group number (32 bit, unsigned) of the requested ADS service.

**indexOffset:** (type: ULONG) [in] contains the index offset number (32 bit, unsigned) of the requested ADS service.

**cbLength:** (type: ULONG) [in] contains the length, in bytes, of the data (pData) to be read.

### Return Value

Type: int

error code - see [AdsStatuscodes \[▶ 300\]](#)

## 12.6.2.2 AdsReadInd

The method AdsReadInd permits to receive an ADS read request. The [AdsReadRes \[▶ 176\]](#) needs to be called for sending the result.

### Syntax

```
void AdsReadInd( AmsAddr& rAddr, ULONG invokeId, ULONG indexGroup, ULONG indexOffset, ULONG cbLength );
```

### Parameters

**rAddr:** (type: AmsAddr&) [in] structure with NetId and port number of the ADS server answering

**invokeId:** (type: ULONG) [in] handle of the command, which is sent. The InvokeId is specified from the source device and serves to identify the commands.

**indexGroup:** (type: ULONG) [in] contains the index group number (32 bit, unsigned) of the requested ADS service.

**indexOffset:** (type: ULONG) [in] contains the index offset number (32 bit, unsigned) of the requested ADS service.

**cbLength:** (type: ULONG) [in] contains the length, in bytes, of the data (pData) to be read.

### Return value

Type: int

ADS Return Code - see [AdsStatuscodes \[▶ 300\]](#).

## 12.6.2.3 AdsReadRes

The method AdsReadRes permits to send an ADS read response. [AdsReadCon \[▶ 177\]](#) is the counterpart and will be called afterwards.

### Syntax

```
int AdsReadRes( AmsAddr& rAddr, ULONG invokeId, ULONG nResult, ULONG cbLength, PVOID pData );
```

### Parameter

**rAddr:** (type: AmsAddr&) [in] structure with NetId and port number of the responding ADS server

**invokeId:** (type: ULONG) [in] handle of the command that is sent. The InvokeId is specified by the source device and is used for the identification of the commands.

**nResult:** (type: ULONG) [in] contains the result of the ADS read command. See [AdsStatuscodes \[▶ 300\]](#)

**cbLength:** (type: ULONG) [in] contains the length in bytes of the data that was read (pData).

**pData:** (type: PVOID) [in] pointer to the data buffer in which the data are located.

### Return value

Type: int

ADS Return Code - see [AdsStatuscodes \[▶ 300\]](#).

## 12.6.2.4 AdsReadCon

The method AdsReadCon permits to receive an ADS read confirmation. The receiving module has to provide this method.

The [AdsReadReq \[▶ 175\]](#) is the counterpart and need to be called beforehand.

### Syntax

```
void AdsReadCon( AmsAddr& rAddr, ULONG invokeId, ULONG nResult, ULONG cbLength, PVOID pData );
```

### Parameters

**rAddr:** (type: AmsAddr&) [in] structure with NetId and port number of the ADS server answering

**invokeId:** (type: ULONG) [in] handle of the command, which is sent. The InvokeId is specified from the source device and serves to identify the commands.

**nResult:** (type: ULONG) [in] contains the result of the ADS read command. See [AdsStatuscodes \[▶ 300\]](#)

**cbLength:** (type: ULONG) [in] contains the length, in bytes, of the data (pData) which was read

**pData:** (type: PVOID) [in] pointer of the data buffer, in which the data is located.

### Return Value

void

## 12.6.3 AdsWrite

### 12.6.3.1 AdsWriteReq

The method AdsWriteReq permits to send an ADS write command, for the transfer of data to an ADS device.

The [AdsWriteCon \[▶ 179\]](#) will be called on arrival of the answer.

### Syntax

```
int AdsWriteReq( AmsAddr& rAddr, ULONG invokeId, ULONG indexGroup, ULONG indexOffset, ULONG cbLength, PVOID pData );
```

### Parameters

**rAddr:** (type: AmsAddr&) [in] structure with NetId and port number of the ADS server.

**invokeId:** (type: ULONG) [in] handle of the command, which is sent. The InvokeId is specified from the source device and serves to identify the commands.

**indexGroup:** (type: ULONG) [in] contains the index group number (32 bit, unsigned) of the requested ADS service.

**indexOffset:** (type: ULONG) [in] contains the index offset number (32 bit, unsigned) of the requested ADS service.

**cbLength:** (type: ULONG) [in] contains the length, in bytes, of the data (pData) to be written

**pData:** (type: PVOID) [in] pointer of the data buffer, in which the data written is located.

### Return Value

Type: int

error code - see [AdsStatuscodes \[▶ 300\]](#)

### 12.6.3.2 AdsWriteInd

The method `AdsWriteInd` indicates an ADS write command, for the transfer of data to an ADS device. The [AdsWriteRes \[▶ 178\]](#) has to be called for confirming the operation.

#### Syntax

```
void AdsWriteInd( AmsAddr& rAddr, ULONG invokeId, ULONG indexGroup, ULONG indexOffset, ULONG cbLength, PVOID pData );
```

#### Parameters

**rAddr:** (type: AmsAddr&) [in] structure with NetId and port number of the ADS server.

**invokeId:** (type: ULONG) [in] handle of the command, which is sent. The `InvokeId` is specified from the source device and serves to identify the commands.

**indexGroup:** (type: ULONG) [in] contains the index group number (32 bit, unsigned) of the requested ADS service.

**indexOffset:** (type: ULONG) [in] contains the index offset number (32 bit, unsigned) of the requested ADS service.

**cbLength:** (type: ULONG) [in] contains the length, in bytes, of the data (pData) to be written

**pData:** (type: PVOID) [in] Pointer of the data buffer, in which the data written is located.

### Return Value

void

error code - see [AdsStatuscodes \[▶ 300\]](#)

### 12.6.3.3 AdsWriteRes

The method `AdsWriteRes` sends an ADS write response. [AdsWriteCon \[▶ 179\]](#) forms the counterpart and is subsequently called.

#### Syntax

```
int AdsWriteRes( AmsAddr& rAddr, ULONG invokeId, ULONG nResult );
```

## Parameters

**rAddr:** (type: AmsAddr&) [in] structure with NetId and port number of the ADS server answering

**invokeld:** (type: ULONG) [in] handle of the command, which is sent. The Invokeld is specified from the source device and serves to identify the commands.

**nResult:** (type: ULONG) [in] contains the result of the ADS write command. See [AdsStatuscodes \[▶ 300\]](#)

## Return Value

Type: int

ADS Return Code - see [AdsStatuscodes \[▶ 300\]](#)

### 12.6.3.4 AdsWriteCon

The method AdsWriteCon permits to receive an ADS write confirmation. The receiving module has to provide this method.

The [AdsWriteReq \[▶ 177\]](#) is the counterpart and need to be called beforehand.

## Syntax

```
void AdsWriteCon( AmsAddr& rAddr, ULONG invokeId, ULONG nResult );
```

## Parameters

**rAddr:** (type: AmsAddr&) [in] structure with NetId and port number of the ADS server answering

**invokeld:** (type: ULONG) [in] handle of the command, which is sent. The Invokeld is specified from the source device and serves to identify the commands.

**nResult:** (type: ULONG) [in] contains the result of the ADS write command. See [AdsStatuscodes \[▶ 300\]](#)

## Return Value

void

### 12.6.4 AdsReadWrite

#### 12.6.4.1 AdsReadWriteReq

The method AdsReadWriteReq permits to send an ADS readwrite command, for the transfer of data to and from an ADS device. The [AdsReadWriteCon \[▶ 181\]](#) will be called on arrival of the answer.

## Syntax

```
int AdsReadWriteReq( AmsAddr& rAddr, ULONG invokeId, ULONG indexGroup, ULONG indexOffset, ULONG cbReadLength, ULONG cbWriteLength, PVOID pData );
```

## Parameters

**rAddr:** (type: AmsAddr&) [in] structure with NetId and port number of the ADS server.

**invokeld:** (type: ULONG) [in] handle of the command, which is sent. The Invokeld is specified from the source device and serves to identify the commands.

**indexGroup:** (type: ULONG) [in] contains the index group number (32 bit, unsigned) of the requested ADS service.

**indexOffset:** (type: ULONG) [in] contains the index offset number (32 bit, unsigned) of the requested ADS service.

**cbReadLength:** (type: ULONG) [in] contains the length, in bytes, of the data (pData) to be read

**cbWriteLength:** (type: ULONG) [in] contains the length, in bytes, of the data (pData) to be written

**pData:** (type: PVOID) [in] pointer of the data buffer, in which the data written is located.

### Return Value

Type: int

error code - see [AdsStatuscodes](#) [► 300]

### 12.6.4.2 AdsReadWriteInd

The method `AdsReadWriteInd` indicates an ADS readwrite command, for the transfer of data to and from an ADS device. The [AdsReadWriteRes](#) [► 182] needs to be called for sending the result.

### Syntax

```
void AdsReadWriteInd( AmsAddr& rAddr, ULONG invokeId, ULONG indexGroup,
                    ULONG indexOffset, ULONG cbReadLength, ULONG cbWriteLength, PVOID pData );
```

### Parameters

**rAddr:** (type: AmsAddr&) [in] structure with NetId and port number of the ADS server.

**invokeId:** (type: ULONG) [in] handle of the command, which is sent. The `InvokeId` is specified from the source device and serves to identify the commands.

**indexGroup:** (type: ULONG) [in] contains the index group number (32 bit, unsigned) of the requested ADS service.

**indexOffset:** (type: ULONG) [in] contains the index offset number (32 bit, unsigned) of the requested ADS service.

**cbReadLength:** (type: ULONG) [in] contains the length, in bytes, of the data (pData) to be read

**cbWriteLength:** (type: ULONG) [in] contains the length, in bytes, of the data (pData) to be written

**pData:** (type: PVOID) [in] pointer of the data buffer, in which the data written is located.

### Return Value

void

### 12.6.4.3 AdsReadWriteRes

The method `AdsReadWriteRes` permits to receive an ADS read write confirmation. The [AdsReadWriteCon](#) [► 181] is the counterpart and will be called afterwards.



## Syntax

```
int AdsReadWriteRes( AmsAddr& rAddr, ULONG invokeId, ULONG nResult, ULONG cbLength, PVOID pData );
```

## Parameters

**rAddr:** (type: AmsAddr&) [in] structure with NetId and port number of the ADS server answering

**invokeId:** (type: ULONG) [in] handle of the command, which is sent. The InvokeId is specified from the source device and serves to identify the commands.

**nResult:** (type: ULONG) [in] contains the result of the ADS write command. See [AdsStatuscodes \[▶ 300\]](#)

**cbLength:** (type: ULONG) [in] contains the length, in bytes, of the data (pData) which was read

**pData:** (type: PVOID) [in] pointer of the data buffer, in which the data is located.

## Return value

Type: int

ADS Return Code - see [AdsStatuscodes \[▶ 300\]](#).

### 12.6.4.4 AdsReadWriteCon

The method AdsReadWriteCon permits to receive an ADS read write confirmation. The receiving module has to provide this method.

The [AdsReadWriteReq \[▶ 179\]](#) is the counterpart and need to be called beforehand.

## Syntax

```
void AdsReadWriteCon( AmsAddr& rAddr, ULONG invokeId, ULONG nResult, ULONG cbLength, PVOID pData );
```

## Parameters

**rAddr:** (tpe: AmsAddr&) [in] structure with NetId and port number of the ADS server answering

**invokeId:** (type: ULONG) [in] handle of the command, which is sent. The InvokeId is specified from the source device and serves to identify the commands.

**nResult:** (type: ULONG) [in] contains the result of the ADS write command. See [AdsStatuscodes \[▶ 300\]](#)

**cbLength:** (type: ULONG) [in] contains the length, in bytes, of the data (pData) which was read

**pData:** (type: PVOID) [in] pointer of the data buffer, in which the data is located.

## Return Value

void

### 12.6.5 AdsReadState

#### 12.6.5.1 AdsReadStateReq

The method AdsReadStateReq permits to send an ADS read state command for reading the ADS status and the device status from an ADS server. The [AdsReadStateCon \[▶ 183\]](#) will be called on arrival of the answer.

## Syntax

```
int AdsReadStateReq(AmsAddr& rAddr, ULONG invokeId);
```

## Parameters

**rAddr:** (type: AmsAddr) [in] structure with NetId and port number of the ADS server.

**invokeId:** (type: ULONG) [in] handle of the command, which is sent. The InvokeId is specified from the source device and serves to identify the commands.

## Return Value

Type: int

error code - see [AdsStatuscodes \[▶ 300\]](#)

### 12.6.5.2 AdsReadStateInd

The method AdsReadStateInd indicates an ADS read state command for reading the ADS status and the device status from an ADS device. The [AdsReadStateRes \[▶ 182\]](#) needs to be called for sending the result.

## Syntax

```
void AdsReadStateInd( AmsAddr& rAddr, ULONG invokeId );
```

## Parameters

**rAddr:** (type: AmsAddr) [in] structure with NetId and port number of the ADS server.

**invokeId:** (type: ULONG) [in] handle of the command, which is sent. The InvokeId is specified from the source device and serves to identify the commands.

## Return Value

void

### 12.6.5.3 AdsReadStateRes

The method AdsWriteRes enables the sending of an ADS status read response. [AdsReadStateCon \[▶ 183\]](#) forms the counterpart and is subsequently called.

## Syntax

```
int AdsReadStateRes( AmsAddr& rAddr, ULONG invokeId, ULONG nResult, USHORT adsState, USHORT deviceState );
```

## Parameter

**rAddr:** (type: AmsAddr&) [in] structure with NetId and port number of the responding ADS server

**invokeId:** (type: ULONG) [in] handle of the command that is sent. The InvokeId is specified by the source device and is used for the identification of the commands.

**nResult:** (type: ULONG) [in] contains the result of the ADS write command. See [AdsStatuscodes \[▶ 300\]](#)

**adsState:** (type: USHORT) [in] contains the ADS state of the device

**deviceState:** (type: USHORT) [in] contains the device status of the device

## Return value

Type: int

Error code - see [AdsStatuscodes \[▶ 300\]](#).

### 12.6.5.4 AdsReadStateCon

The method `AdsWriteCon` permits to receive an ADS read state confirmation. The receiving module has to provide this method.

The [AdsReadStateReq \[▶ 181\]](#) is the counterpart and needs to be called beforehand.

#### Syntax

```
void AdsReadStateCon( AmsAddr& rAddr, ULONG invokeId, ULONG nResult, USHORT adsState, USHORT deviceState );
```

#### Parameters

**rAddr:** (type: `AmsAddr&`) [in] structure with `NetId` and port number of the ADS server answering

**invokeld:** (type: `ULONG`) [in] handle of the command, which is sent. The `Invokeld` is specified from the source device and serves to identify the commands.

**nResult:** (type: `ULONG`) [in] contains the result of the ADS write command. See [AdsStatuscodes \[▶ 300\]](#)

**adsState:** (type: `USHORT`) [in] contains the ads state of the device

**deviceState:** (type: `USHORT`) [in] contains the device state of the device

#### Return Value

void

## 12.6.6 AdsWriteControl

### 12.6.6.1 AdsWriteControlReq

The method `AdsWriteControlReq` permits to send an ADS write control command for changing the ADS status and the device status of an ADS server. The [AdsWriteControlCon \[▶ 185\]](#) will be called on arrival of the answer.

#### Syntax

```
int AdsWriteControlReq( AmsAddr& rAddr, ULONG invokeId, USHORT adsState, USHORT deviceState, ULONG cbLength, PVOID pData );
```

#### Parameters

**rAddr:** (type: `AmsAddr&`) [in] structure with `NetId` and port number of the ADS server.

**invokeld:** (type: `ULONG`) [in] handle of the command, which is sent. The `Invokeld` is specified from the source device and serves to identify the commands.

**adsState:** (type: `USHORT`) [in] contains the index group number (32 bit, unsigned) of the requested ADS service.

**deviceState:** (type: `USHORT`) [in] contains the index offset number (32 bit, unsigned) of the requested ADS service.

**cbLength:** (type: ULONG) [in] contains the length, in bytes, of the data (pData)

**pData:** (type: PVOID) [in] pointer of the data buffer, in which the data written is located.

### Return Value

Type: int

error code - see [AdsStatuscodes](#) [▶ 300]

### 12.6.6.2 AdsWriteControlInd

The method `AdsWriteControlInd` permits to send an ADS write control command for changing the ADS status and the device status of an ADS device. The [AdsWriteControlRes](#) [▶ 184] has to be called for confirming the operation.

### Syntax

```
void AdsWriteControlInd( AmsAddr& rAddr, ULONG invokeId, USHORT adsState, USHORT deviceState,
ULONG cbLength, PVOID pDeviceData );
```

### Parameters

**rAddr:** (type: AmsAddr&) [in] structure with NetId and port number of the ADS server.

**invokeId:** (type: ULONG) [in] handle of the command, which is sent. The InvokeId is specified from the source device and serves to identify the commands.

**adsState:** (type: USHORT) [in] contains the index group number (32 bit, unsigned) of the requested ADS service.

**deviceState:** (type: USHORT) [in] contains the index offset number (32 bit, unsigned) of the requested ADS service.

**cbLength:** (type: ULONG) [in] contains the length, in bytes, of the data (pData)

**pData:** (type: PVOID) [in] pointer of the data buffer, in which the data written is located.

### Return Value

void

### 12.6.6.3 AdsWriteControlRes

The method `AdsWriteControlRes` permits to send an ADS write control response. The [AdsWriteControlCon](#) [▶ 185] is the counterpart and will be called afterwards.

### Syntax

```
int AdsWriteControlRes( AmsAddr& rAddr, ULONG invokeId, ULONG nResult );
```

### Parameters

**rAddr:** (type: AmsAddr&) [in] structure with NetId and port number of the ADS server answering

**invokeld:** (type: ULONG) [in] handle of the command, which is sent. The Invokeld is specified from the source device and serves to identify the commands.

**nResult:** (type: ULONG) [in] contains the result of the ADS write command. See [AdsStatuscodes \[▶ 300\]](#)

### Return Value

Type: int

ADS return code - see [AdsStatuscodes \[▶ 300\]](#)

## 12.6.6.4 AdsWriteControlCon

The method `AdsWriteCon` permits to receive an ADS write control confirmation. The receiving module has to provide this method.

The [AdsWriteControlReq \[▶ 183\]](#) is the counterpart and needs to be called beforehand

### Syntax

```
void AdsWriteControlCon( AmsAddr& rAddr, ULONG invokeId, ULONG nResult );
```

### Parameters

**rAddr:** (type: `AmsAddr&`) [in] tructure with NetId and port number of the ADS server answering

**invokeld:** (type: ULONG) [in] handle of the command, which is sent. The Invokeld is specified from the source device and serves to identify the commands.

**nResult:** (type: ULONG) [in] contains the result of the ADS write command. See [AdsStatuscodes \[▶ 300\]](#)

### Return Value

void

## 12.6.7 AdsAddDeviceNotification

### 12.6.7.1 AdsAddDeviceNotificationReq

The method `AdsAddDeviceNotificationReq` permits to send an ADS add device notification command, for adding a device notification to an ADS device. The [AdsAddDeviceNotificationCon \[▶ 187\]](#) will be called on arrival of the answer.

### Syntax

```
int AdsAddDeviceNotificationReq( AmsAddr& rAddr, ULONG invokeId, ULONG indexGroup, ULONG  
indexOffset,  
AdsNotificationAttrib noteAttrib);
```

### Parameters

**rAddr:** (type: `AmsAddr&`) [in] structure with NetId and port number of the ADS server.

**invokeld:** (type: ULONG) [in] handle of the command, which is sent. The Invokeld is specified from the source device and serves to identify the commands.

**indexGroup:** (type: ULONG) [in] contains the index group number (32 bit, unsigned) of the requested ADS service.

**indexOffset:** (type: ULONG) [in] contains the index offset number (32 bit, unsigned) of the requested ADS service.

**noteAttrib:** (type: AdsNotificationAttrib) [in] contains specification of the notification parameters (cbLength, TransMode, MaxDelay)

### Return Value

Type: int

error code - see [AdsStatuscodes](#) [▶ 300]

## 12.6.7.2 AdsAddDeviceNotificationInd

The method AdsAddDeviceNotificationInd should enable sending [AdsDeviceNotification](#) [▶ 189]. The [AdsAddDeviceNotificationRes](#) [▶ 186] has to be called for confirming the operation.

### Syntax

```
void AdsAddDeviceNotificationInd( AmsAddr& rAddr, ULONG invokeId, ULONG indexGroup, ULONG indexOffset, AdsNotificationAttrib noteAttrib );
```

### Parameters

**rAddr:** (type: AmsAddr&) [in] structure with NetId and port number of the ADS server answering

**invokeId:** (type: ULONG) [in] handle of the command, which is sent. The InvokeId is specified from the source device and serves to identify the commands.

**indexGroup:** (type: ULONG) [in] contains the index group number (32 bit, unsigned) of the requested ADS service.

**indexOffset:** (type: ULONG) [in] contains the index offset number (32 bit, unsigned) of the requested ADS service.

**noteAttrib:** (type: AdsNotificationAttrib) [in] contains specification of the notification parameters (cbLength, TransMode, MaxDelay).

### Return Value

void

## 12.6.7.3 AdsAddDeviceNotificationRes

The method AdsAddDeviceNotificationRes permits to send an ADS add device notification response. The [AdsAddDeviceNotificationCon](#) [▶ 187] is the counterpart and will be called afterwards.

### Syntax

```
void AdsAddDeviceNotificationCon( AmsAddr& rAddr, ULONG invokeId, ULONG nResult, ULONG handle );
```

### Parameters

**rAddr:** (type: AmsAddr&) [in] structure with NetId and port number of the ADS server answering

**invokeId:** (type: ULONG) [in] handle of the command, which is sent. The InvokeId is specified from the source device and serves to identify the commands.

**nResult:** (type: ULONG) [in] contains the result of the ADS write command. See [AdsStatuscodes](#) **handle:** (type: ULONG) [in] handle to the generated device notification

## Return Value

void

### 12.6.7.4 AdsAddDeviceNotificationCon

The method `AdsAddDeviceNotificationCon` confirms an ADS add device notification request. The `AdsAddDeviceNotificationReq` [► 185] is the counterpart and needs to be called beforehand.

#### Syntax

```
void AdsAddDeviceNotificationCon( AmsAddr& rAddr, ULONG invokeId, ULONG nResult, ULONG handle );
```

#### Parameters

**rAddr:** (type: `AmsAddr&`) [in] structure with `NetId` and port number of the ADS server answering

**invokeId:** (type: `ULONG`) [in] handle of the command, which is sent. The `InvokeId` is specified from the source device and serves to identify the commands.

**nResult:** (type: `ULONG`) [in] contains the result of the ADS write command. See `AdsStatuscodes` handle

**handle:** (type: `ULONG`) [in] handle to the generated device notification

## Return Value

void

## 12.6.8 AdsDelDeviceNotification

### 12.6.8.1 AdsDelDeviceNotificationReq

The method `AdsDelDeviceNotificationReq` permits to send an ADS delete device notification command, for removing a device notification from an ADS device. The `AdsDelDeviceNotificationCon` [► 188] will be called on arrival of the answer.

#### Syntax

```
int AdsDelDeviceNotificationReq( AmsAddr& rAddr, ULONG invokeId, ULONG hNotification );
```

#### Parameters

**rAddr:** (type: `AmsAddr&`) [in] structure with `NetId` and port number of the ADS server

**invokeId:** (type: `ULONG`) [in] handle of the command, which is sent. The `InvokeId` is specified from the source device and serves to identify the commands

**hNotification:** (type: `ULONG`) [in] contains the handle to the notification, which should be removed

## Return Value

Type: `int`

error code - see `AdsStatuscodes` [► 300]

### 12.6.8.2 AdsDelDeviceNotificationInd

The method `AdsAddDeviceNotificationCon` permits to receive an ADS delete device notification confirmation. The receiving module has to provide this method. The [AdsDelDeviceNotificationRes \[▶ 188\]](#) has to be called for confirming the operation.

#### Syntax

```
void AdsDelDeviceNotificationCon( AmsAddr& rAddr, ULONG invokeId, ULONG nResult );
```

#### Parameters

**rAddr:** (type: `AmsAddr&`) [in] structure with `NetId` and port number of the ADS server answering

**invokeId:** (type: `ULONG`) [in] handle of the command, which is sent. The `InvokeId` is specified from the source device and serves to identify the commands

**nResult:** (type: `ULONG`) [in] contains the result of the ADS write command. See [AdsStatuscodes \[▶ 300\]](#)

#### Return Value

void

### 12.6.8.3 AdsDelDeviceNotificationRes

The method `AdsAddDeviceNotificationRes` permits to receive an ADS delete device notification. The [AdsDelDeviceNotificationCon \[▶ 188\]](#) is the counterpart and will be called afterwards.

#### Syntax

```
int AdsDelDeviceNotificationRes( AmsAddr& rAddr, ULONG invokeId, ULONG nResult );
```

#### Parameters

**rAddr:** (type: `AmsAddr&`) [in] structure with `NetId` and port number of the ADS server answering

**invokeId:** (type: `ULONG`) [in] handle of the command, which is sent. The `InvokeId` is specified from the source device and serves to identify the commands.

**nResult:** (type: `ULONG`) [in] contains the result of the ADS command. See [AdsStatuscodes \[▶ 300\]](#)

#### Return Value

Int

Returns the result of the ADS command. See [AdsStatuscodes \[▶ 300\]](#)

### 12.6.8.4 AdsDelDeviceNotificationCon

The method `AdsAddDeviceNotificationCon` permits to receive an ADS delete device notification confirmation. The receiving module has to provide this method. The [AdsDelDeviceNotificationReq \[▶ 187\]](#) is the counterpart and need to be called beforehand.

#### Syntax

```
void AdsDelDeviceNotificationCon( AmsAddr& rAddr, ULONG invokeId, ULONG nResult );
```



## Parameters

**rAddr:** (type: AmsAddr&) [in] structure with NetId and port number of the ADS server answering

**invokeld:** (type: ULONG) [in] handle of the command, which is sent, the Invokeld is specified from the source device and serves to identify the commands

**nResult:** (type: ULONG) [in] contains the result of the ADS write command, see [AdsStatuscodes \[▶ 300\]](#)

## Return Value

void

## 12.6.9 AdsDeviceNotification

### 12.6.9.1 AdsDeviceNotificationReq

The method `AdsAddDeviceNotificationReq` permits to send an ADS device notification, to inform an ADS device. The [AdsDeviceNotificationInd \[▶ 189\]](#) will be called on the counterpart.

#### Syntax

```
int AdsDeviceNotificationReq( AmsAddr& rAddr, ULONG invokeId, ULONG cbLength,
AdsNotificationStream notifications[] );
```

#### Parameters

**rAddr:** (type: AmsAddr&) [in] structure with NetId and port number of the ADS server.

**invokeld:** (type: ULONG) [in] handle of the command, which is sent. The Invokeld is specified from the source device and serves to identify the commands.

**nResult:** (type: ULONG) [in] contains result of the device notification indication.

**notifications[]:** (type: AdsNotificationStream) [in] contains information of the device notification(s).

#### Return Value

Type: int

ADS return code - see [AdsStatuscodes \[▶ 300\]](#)

### 12.6.9.2 AdsDeviceNotificationInd

The method `AdsDeviceNotificationInd` enables receiving of information from an ADS device notification display. The receiving module must provide this method. There is no acknowledgment of receipt.

[AdsDeviceNotificationCon \[▶ 190\]](#) must be called by [AdsDeviceNotificationReq \[▶ 189\]](#) to check the transfer.

#### Syntax

```
void AdsDeviceNotificationInd( AmsAddr& rAddr, ULONG invokeId, ULONG cbLength,
AdsNotificationStream* pNotifications );
```

#### Parameters

**rAddr:** (type: AmsAddr&) [in] structure with NetId and port number of the ADS server answering

**invokeld:** (type: ULONG) [in] handle of the command, which is sent. The Invokeld is specified from the source device and serves to identify the commands.

**cbLength:** (type: ULONG) [in] contains the length of pNotifications

**pNotifications:** (type: AdsNotificationStream\*) [in] pointer to the Notifications. This array consists of AdsStampHeader, which contains notification handle and data via AdsNotificationSample.

### Return Value

void

### 12.6.9.3 AdsDeviceNotificationCon

The sender can use the method AdsAddDeviceNotificationCon to check the transfer of an ADS device notification.

AdsDeviceNotificationReq [► 189] must be called first.

### Syntax

```
void AdsDeviceNotificationCon( AmsAddr& rAddr, ULONG invokeId, ULONG nResult );
```

### Parameters

**rAddr:** (type: AmsAddr&) [in] structure with NetId and port number of the ADS server.

**invokeld:** (type: ULONG) [in] handle of the command, which is sent. The Invokeld is specified from the source device and serves to identify the commands.

**nResult:** (type: ULONG) [in] contains result of the device notification indication

### Return Value

void

## 12.7 Mathematical Functions

TwinCAT has its own mathematical functions implemented, because the math.h implementation provided by Microsoft is not real-time capable.































These functions are declared in TcMath.h, which is 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

**i** TwinCAT 3.1 4018 provides an fpu87.h with the same methods. This continues to exist and redirects to TcMath.h.

### Methods provided

Mathematical functions

	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.
	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.
	cubic_	Calculates the cube.
	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.
<b>Help Functions</b>		
	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 changes with each call.

## Comments

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.](#)

## 12.8 Time Functions

TwinCAT provides functions for time conversion.

The functions are declared in TcTimeConversion.h, which is part of the TwinCAT SDK.

	Name	Description
	TcDayOfWeek	Calculates the day of week (Sunday is 0)
	TcIsLeapYear	Calculates if the given year is a leap year
	TcDaysInYear	Calculates the number of days in given year
	TcDaysInMonth	Calculates the number of day in given month
	TcSystemTimeToFileTime(const SYSTEMTIME* lpSystemTime, FILETIME *lpFileTime);	Converts the given system time to a file time
	TcFileTimeToSystemTime(const FILETIME *lpFileTime, SYSTEMTIME* lpSystemTime);	Converts the given file time to a system time
	TcSystemTimeToFileTime(const SYSTEMTIME* lpSystemTime, ULONGLONG& ul64FileTime);	Converts the given system time to a file time (ULONGLONG format)
	TcFileTimeToSystemTime(const ULONGLONG& ul64FileTime, SYSTEMTIME* lpSystemTime);	Converts the given file time (ULONGLONG format) to a system time

## 12.9 STL / Containers

TwinCAT 3 C++ supports STL with regard to

- List
- Map
- Set
- Stack
- String
- Vector
- WString
- Algorithms (such as binary\_search)
  - See c:\TwinCAT\3.x\Sdk\Include\Stl\Stl\algorithm for a specific list of supported algorithms

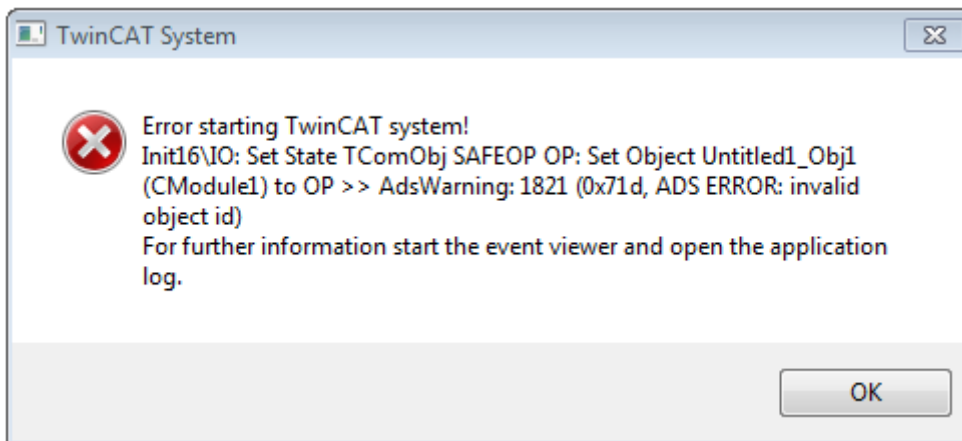
Please note:

- 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 [here](#) [▶ 138].

## 12.10 Error Messages - Comprehension

TwinCAT might come up with Error messages which seem to be overloaded.



They contain very detailed information about the occurred error. As an example the screenshot above does describe:

- The error occurred during the transition SAFE OP to OP
- The related object is “Untitled1\_Obj1 (CModule1)”
- The errorcode 1821 / 0x71d indicates that the object id is invalid

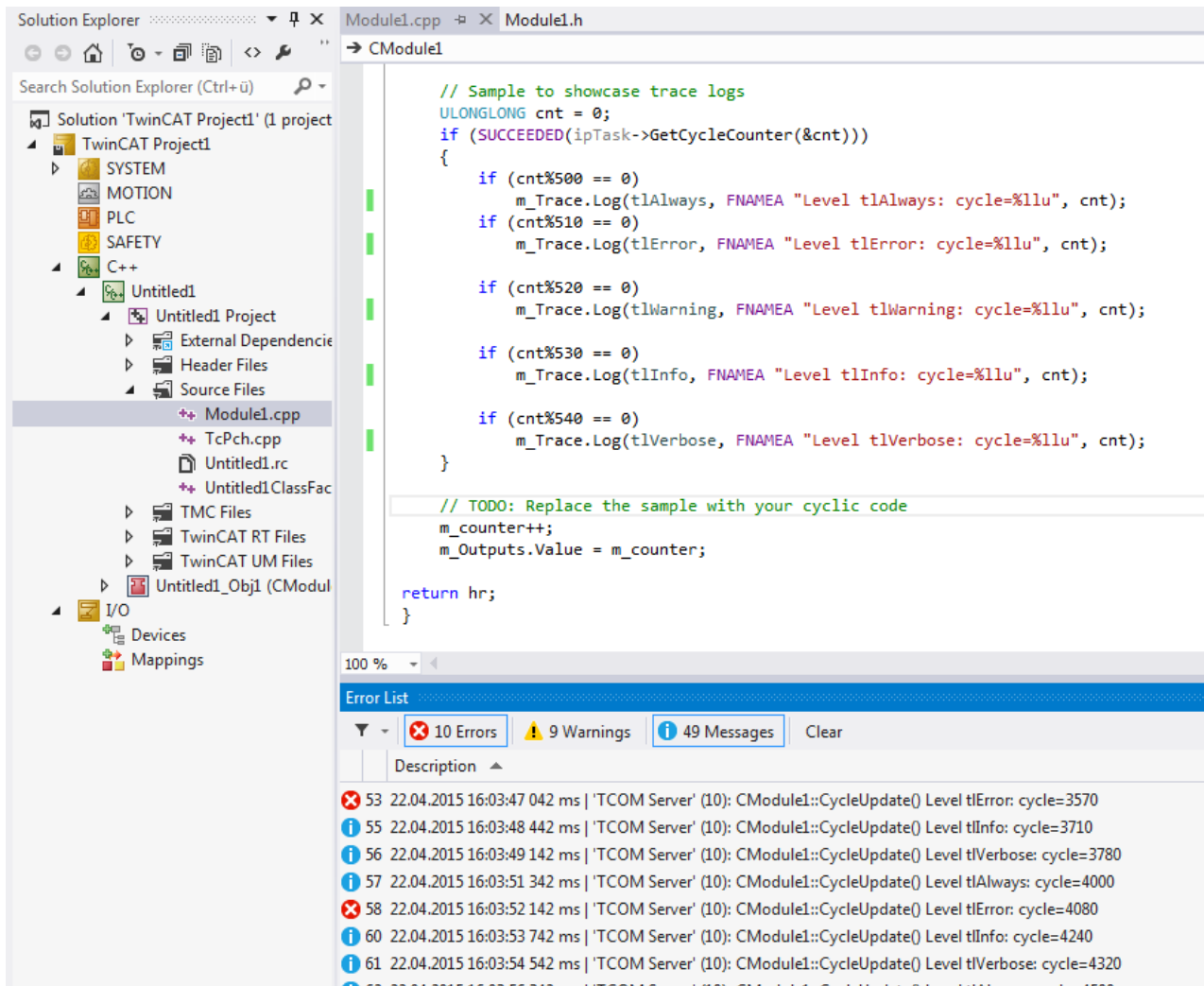
Conclusion should be that one needs to have a look at the “SetObjStateSP()” method, which is responsible to this transition. For the generated default code one could see that adding the module is done there.

The reason why this error occurred was that no task was assigned to this module – thus the module can’t have a task to be executed within.

## 12.11 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 of tracing messages is as follows:

```
m_Trace.Log(TLEVEL, FNMACRO"A message", ...);
```

With these properties:

- `TLEVEL` categorizes a message into one of five different levels. While tracing the higher level will always include the trace of the lower levels: E.g. a message traced to level "tlWarning" will occur with level "tlAlways", "tlError" and "tlWarning" - it will NOT trace the messages "tlInfo" and "tlVerbose".

Level 0	tlAlways
Level 1	tlError
Level 2	tlWarning
Level 3	tlInfo
Level 4	tlVerbose

- `FNMACRO` could be used to put function name in front of message to be printed
  - `FENTERA`: Used while entering a function; will print function name followed by „>>>“
  - `FNAMEA`: Used within a function; will print function name
  - `FLEAVEA`: Used upon leaving a function; will print function name followed by “<<<”
- Format specifier

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);
    }

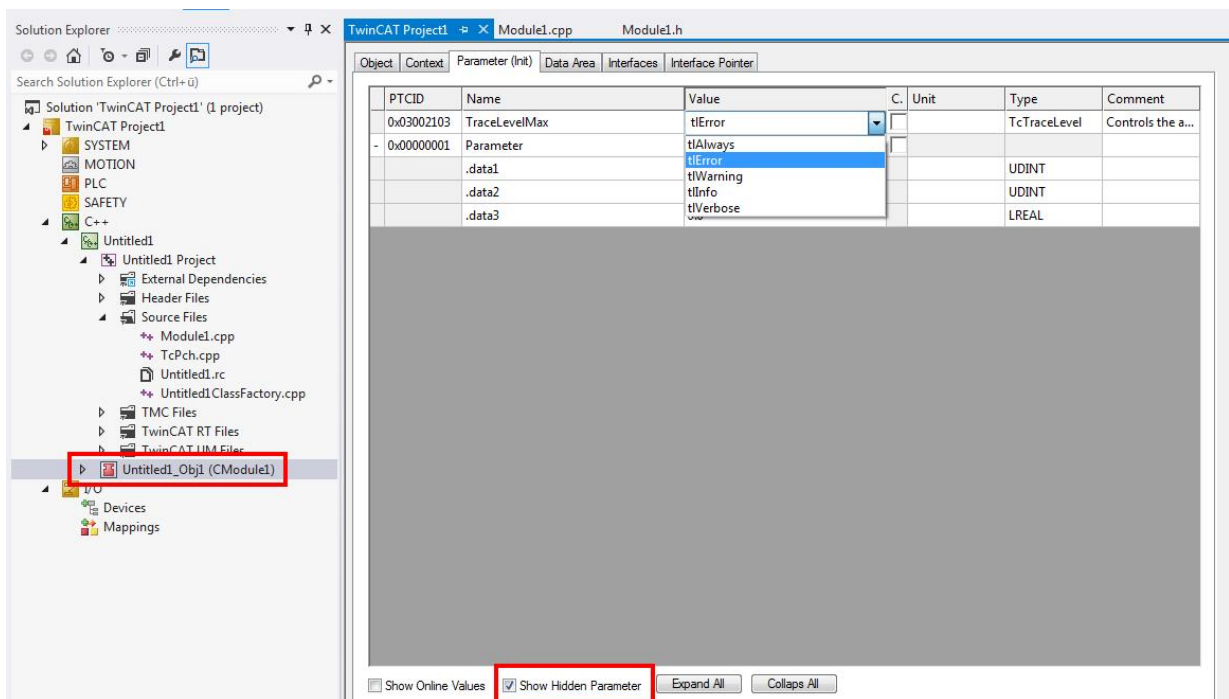
    // TODO: Replace the sample with your cyclic code
    m_counter++;
    m_Outputs.Value = m_counter;

    return hr;
}
```

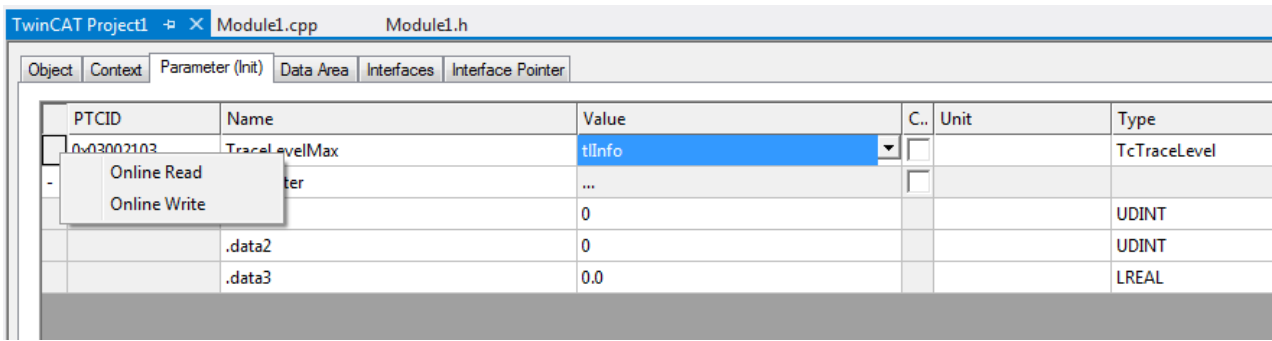
Using trace level

At module instance it's possible to pre configure the trace level.

1. Navigate to the instance of the module in the solution tree
2. Select tab "Parameter (Init)" on the right side.
3. Be aware to enable "Show Hidden Parameters"
4. Select the trace level
5. To test everything, select highest level "tlVerbose".



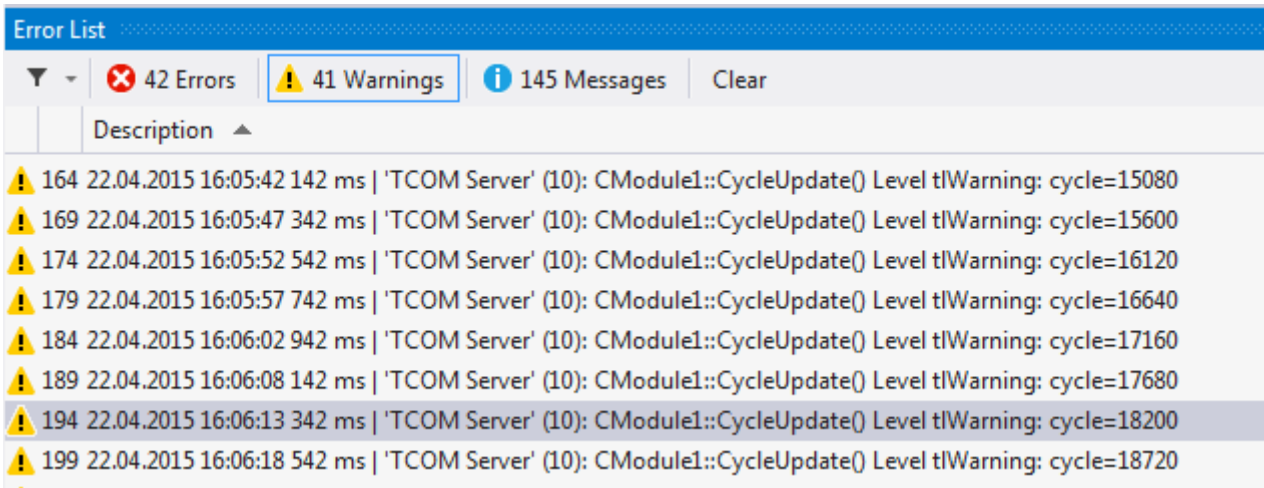
Alternatively, the trace level could also be changed during runtime by going to the instance, selecting a level at "Value" for the TraceLevelMax parameter, right-click in front on the first column and select "Online Write"



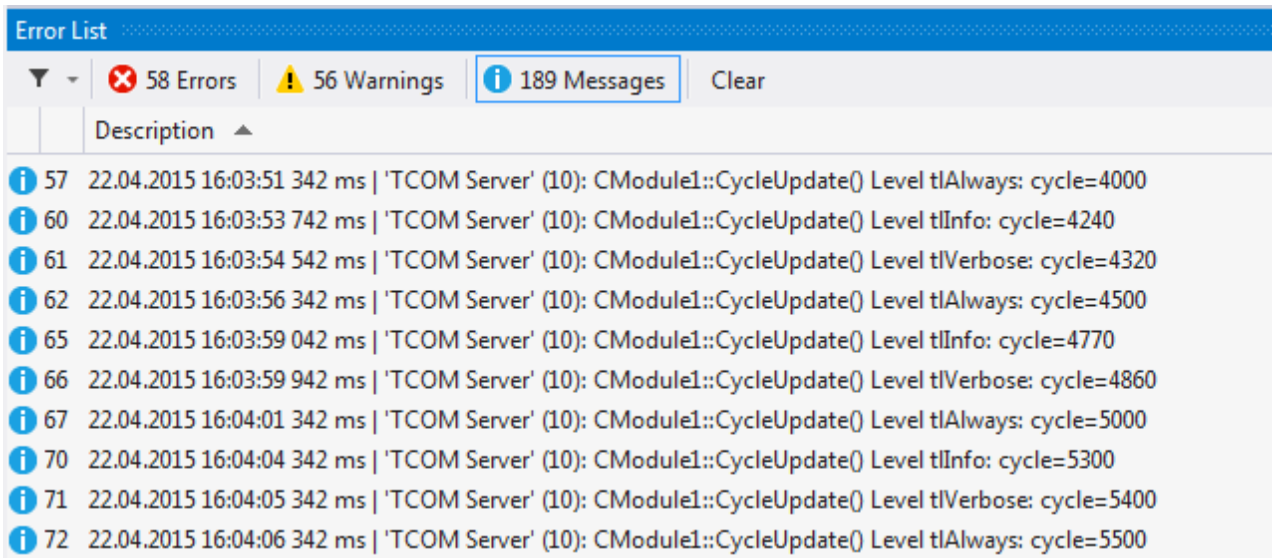
**Filter messages categories**

Visual Studio Error List allows to filter the entries with respect to their category. The three categories "Errors", "Warnings" and "Messages" can be enabled or disabled independently by just toggling the buttons.

In this screenshot only "Warnings" is enabled - but "Errors" and "Messages" are disabled:



In this screenshot only "Messages" is enabled - but "Errors" and "Warnings" are disabled to be displayed.





## 13 How to...?

This is a collection of frequently ask questions about common paradigms of coding as well as handling of TwinCAT C++ modules.

### 13.1 Using the Automation Interface

The Automation Interface can be used for C++ projects

This includes [creating projects \[▶ 76\]](#) and using the wizard for [creating module classes \[▶ 77\]](#).

In addition, the project properties can be set, and the TMC code generator and publishing of modules can be called. The corresponding [documentation \[▶ 307\]](#) is part of the Automation Interface.

Irrespective of the programming language, [access to and creation and handling of TcCOM modules \[▶ 311\]](#) 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 \[▶ 122\]](#). In later versions this is done implicitly when the target system uses Windows 10 as operating system.

### 13.3 Publishing of modules

The section [Export modules \[▶ 44\]](#) describes how TwinCAT modules are published, thus they could be transferred and [imported \[▶ 45\]](#) on any TwinCAT system.

The engineering system (XAE) does not have to be of the same platform type as the execution system. Therefore TwinCAT build during publishing all versions of the module.

Some use-cases require customizing the publishing procedure of modules:

- If working in a pure 32bit (x86) environment, the x64 builds could be skipped, thus no certificates are required.
- The User Mode (UM) builds could be skipped, if not used.



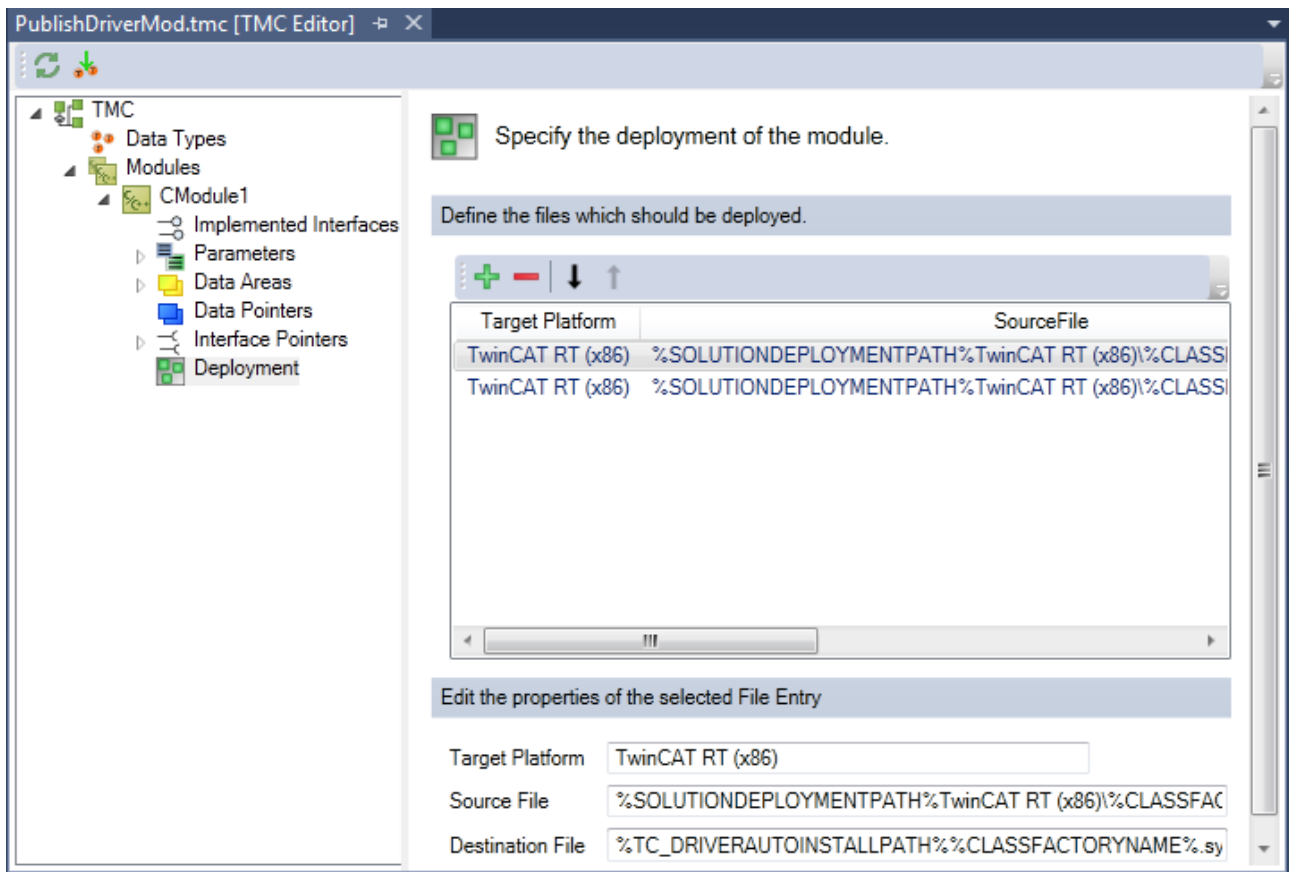
#### Migration

If a build is not included in the published module, the specific platform could not be used as execution system.

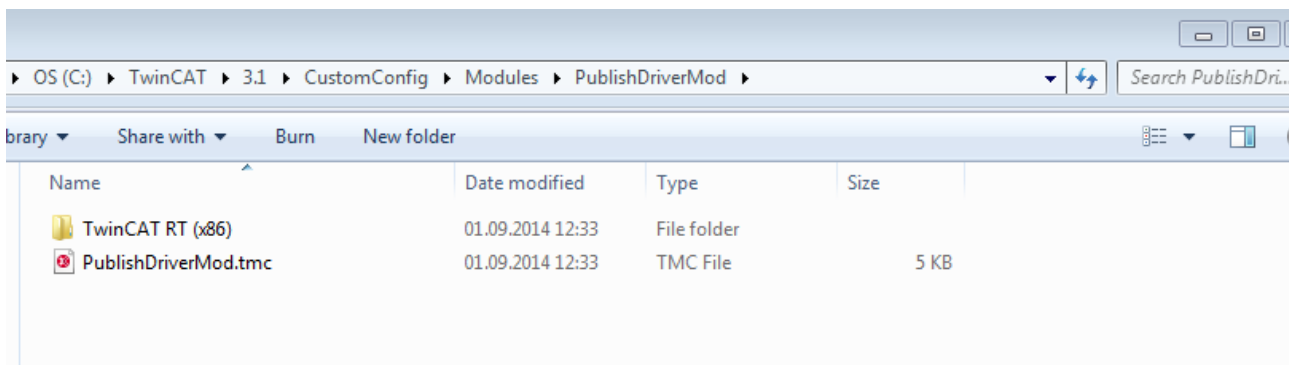
---

Please add / remove the corresponding Target Platforms from [Deployment \[▶ 122\]](#) of the TMC Editor.

For example this Deployment configuration:



Does only provide the TwinCAT RT (x86) build:



## 13.4 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:TcPublishModule /p:TcPublishDestinationBaseFolder=c:\temp
```

The `CppProject.vcxproj` parameter must be adapted according to the existing project file.

The `TcPublishDestinationBaseFolder` parameter is optional here. If it is not specified, the normal storage location will be used (`C:\TwinCAT\3.x\CustomConfig\Modules`).

## 13.5 Clone

The runtime data can be transferred from one machine to another by means of a file copy if both machines originate from the same platform and are connected with equivalent hardware equipment.

The following steps describe a simple procedure to transfer a binary configuration from one machine, "source", to another, "destination".

- ✓ Please empty the folder C:\TwinCAT\3.x\Boot on the source machine.
  - 1. Create (or activate) the module on the source machine.
  - 2. Transfer the folder C:\TwinCAT\3.x\Boot from the source to the destination.
  - 3. Transfer the driver itself from C:\TwinCAT\3.x\Driver\AutoInstall\MYDRIVER.sys.
  - 4. Optionally also transfer MYDRIVER.pdb.
  - 5. If drivers are new on a machine:  
TwinCAT must carry out a registration once. To do this, switch TwinCAT to RUN using SysTray (right-click->System->Start/Restart).  
The following call can alternatively be used (replace "%1" by the driver name):  

```
sc create %1 binPath= c:\TwinCAT\3.1\Driver\AutoInstall\%1.sys type= kernel
start= auto group= "file system" DisplayName= %1 error= normal
```
- ⇒ The destination machine can be started.

**i Handling licenses**

Note that licenses cannot be transferred in this manner. Please use pre-installed licenses, volume licenses or other mechanisms for providing licenses.

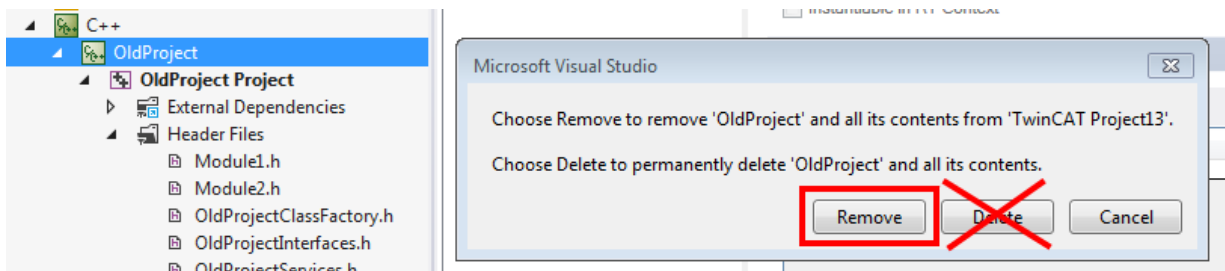
### 13.6 Renaming TwinCAT C++ projects

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.

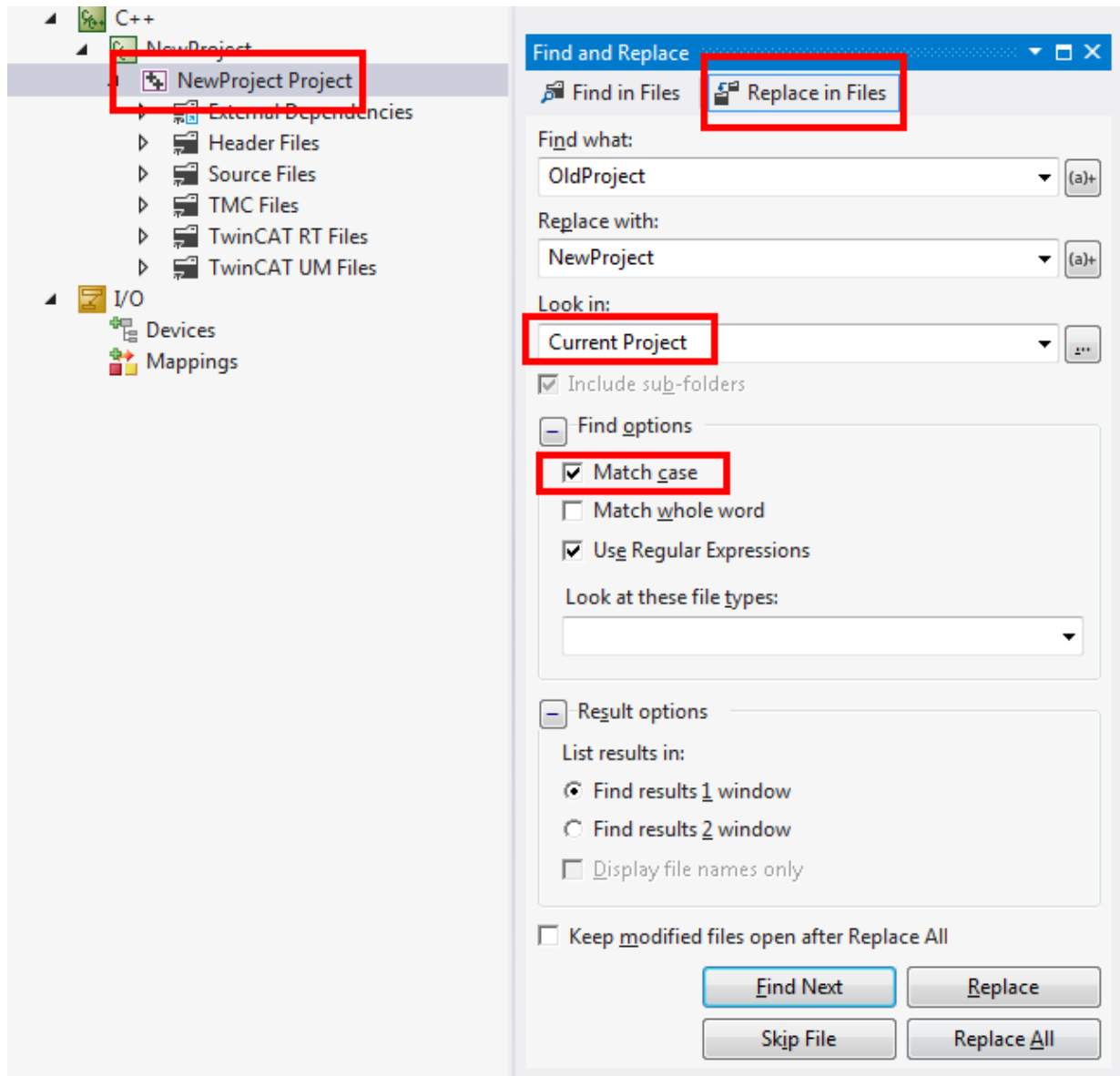
- ✓ A project, "OldProject", exists and is to be renamed "NewProject".
- 1. If TcCOM instances exist in the project and are to be retained along with their links, first move them by drag & drop out of the project into System->TcCOM Objects.
- 2. Remove the old project from the TwinCAT solution ("Remove").



- 3. Compilations of the "OldProject" can be deleted. To do this, delete the corresponding .sys/.pdb 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 version management 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 instances in the source code (case-sensitive):  
"OLDPROJECT" becomes "NEWPROJECT" and

"OldProject" becomes "NewProject".

The "Find and Replace" dialog in Visual Studio is useful for this: the "NewProject Project" must be selected in the Solution Explorer (cf. screenshot).



## NOTE

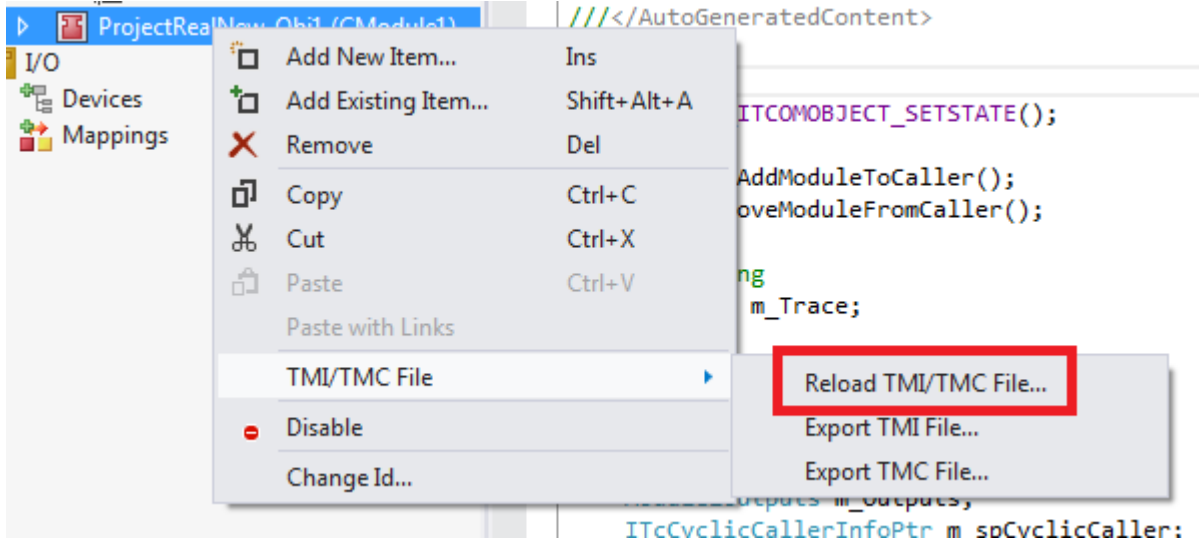
### Incorrect source code

The simple renaming of all instances 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:

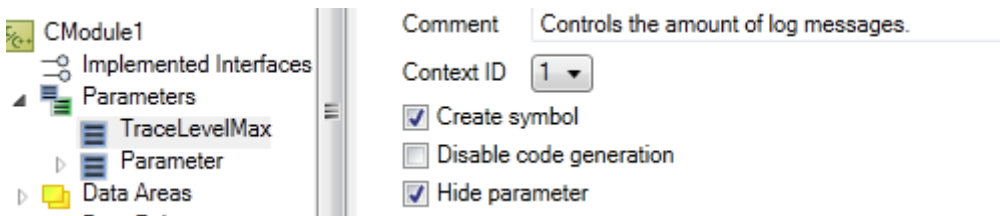
1. A) If instances from the project should exist, update them. To do this, right-click on the instance, select TMI/TMC File->Reload TMI/TMC File... and select the renamed new TMC file.



- 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).  
Delete the files OldProject.sys/.pdb in C:\TwinCAT\3.x\Driver\AutoInstall.
4. Test the project.

### 13.7 Access Variables via ADS

Variables of C++ modules could be accessed by ADS, if variable is marked as "Create Symbol" in TMC Editor:



The Name of the variable for access by ADS is constructed from the instance name.

So for the TraceLevelMax parameter it might be:

Untitled1\_Obj1 (CModule1).TraceLevelMax

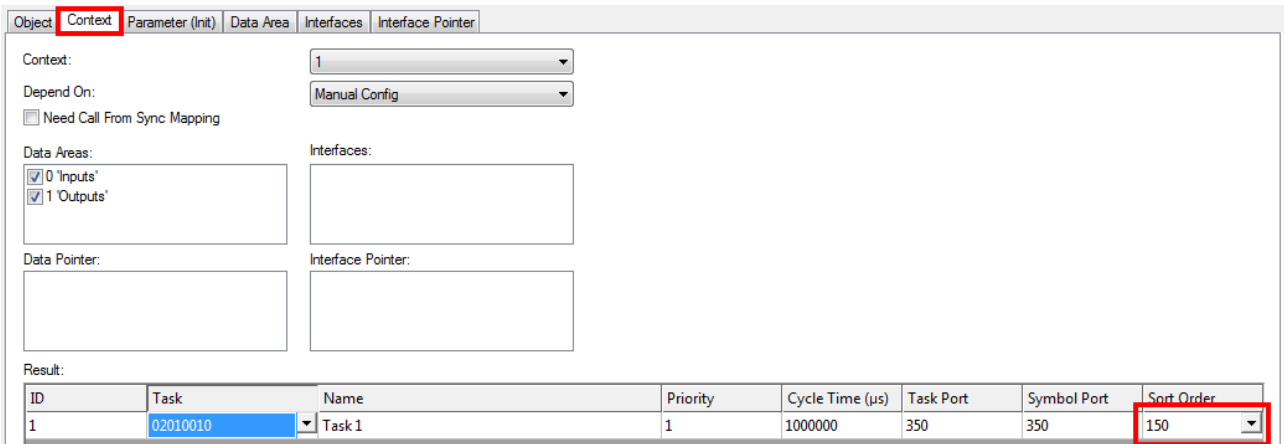
### 13.8 TcCallAfterOutputUpdate for C++ modules

Comparable to PLC Attribute TcCallAfterOutputUpdate C++ modules could be called after the output update. Equivalent to the [ITcCyclic](#) [▶ 140] Interface: Please make use of the [ITcPostCyclic](#) [▶ 161] Interface

### 13.9 Ordering Execution in one Task

Various module instances could be assigned to one task, thus customer needs a mechanism to determine the execution order within the task.

The keyword is "Sort Order", which is configured in the [Context \[▶ 126\]](#) of the [TwinCAT Module Instance Configurator \[▶ 124\]](#).



Please see [Sample26: Execution order at one task \[▶ 275\]](#) on how to implement this.

### 13.10 Use Stack Size > 4kB

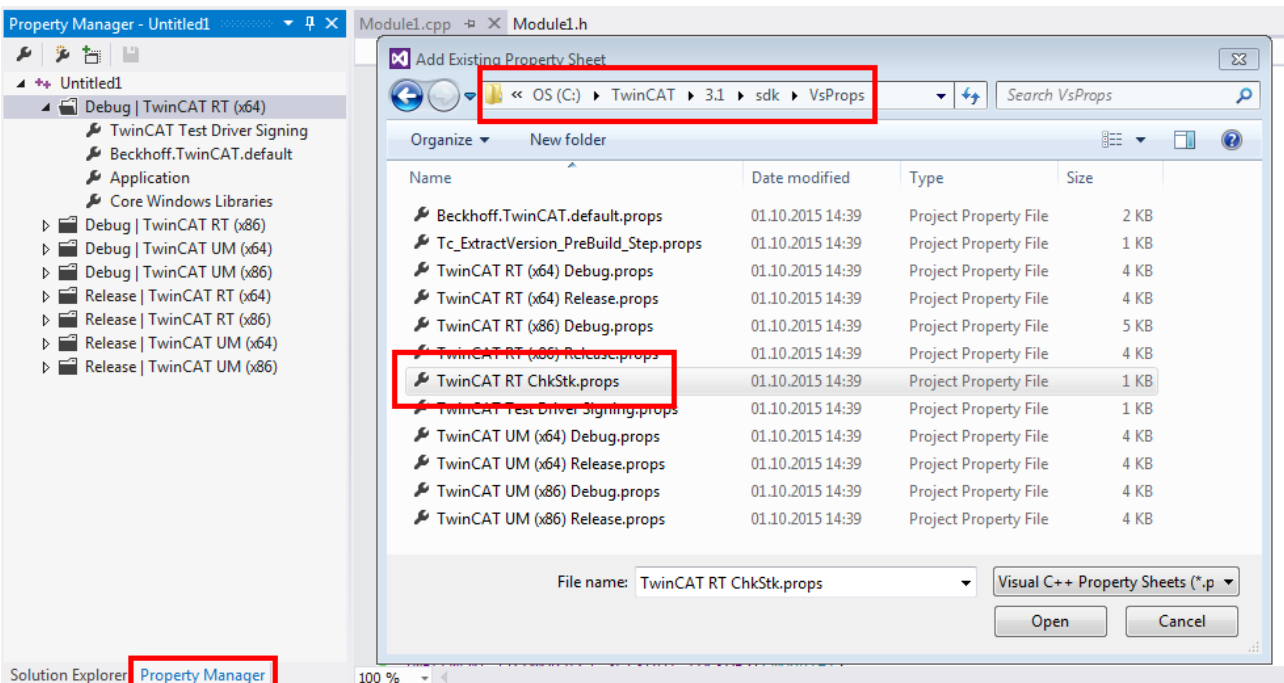
If a method uses a stack size larger than 4 KB, the `__chkstk` function will additionally be called in the case of real-time modules.

For this reason the error message

```
Error LNK2001: unresolved external symbol __chkstk
```

occurs.

You can use the library "TcChkStk.lib" provided by TwinCAT by adding the property file "TwinCAT RT ChkStk" (located by default in `c:\TwinCAT\3.x\sdk\VsProps\`) to the respective RT variants using the property manager.



The TwinCAT execution environment (XAR) has a limit of 64 KB for the stack size.

If (local) memory should be required, there are several possibilities:

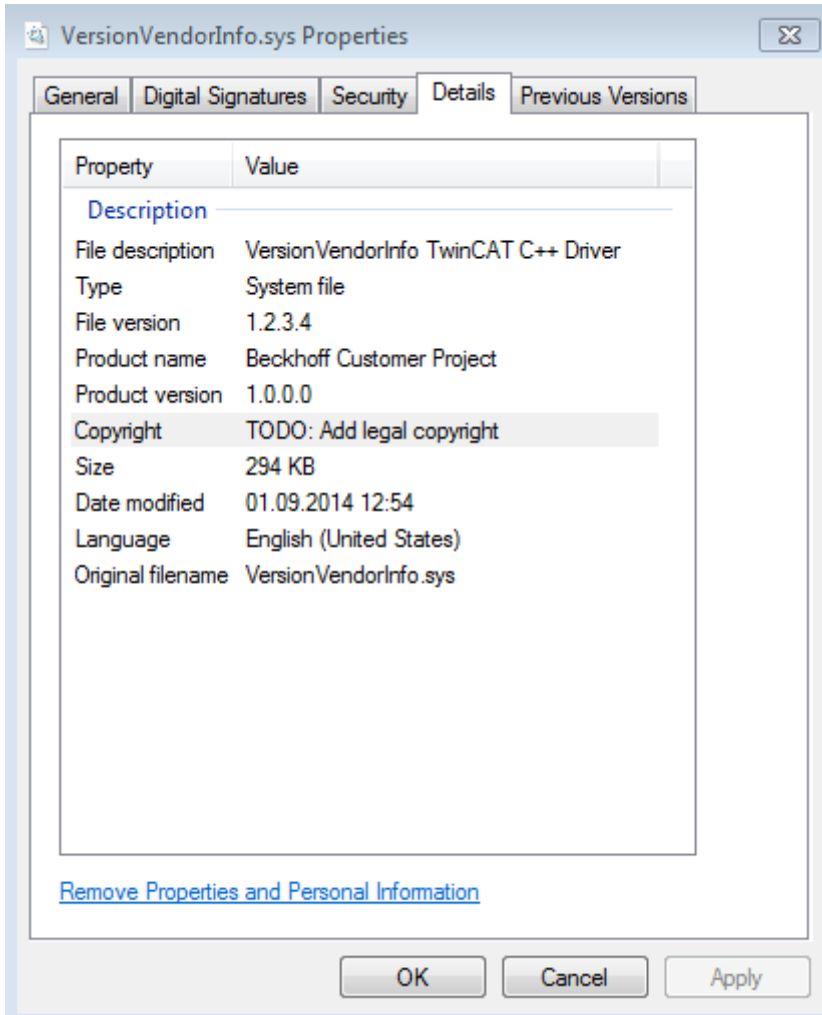
- Member variables (header file of the module)
- Symbols of the module ([TMC Editor \[▶ 79\]](#)), e.g. in an additional, locally used DataArea

- Dynamic allocation of memory using new() and delete() (cf. [Memory Allocation](#) [▶ 138])

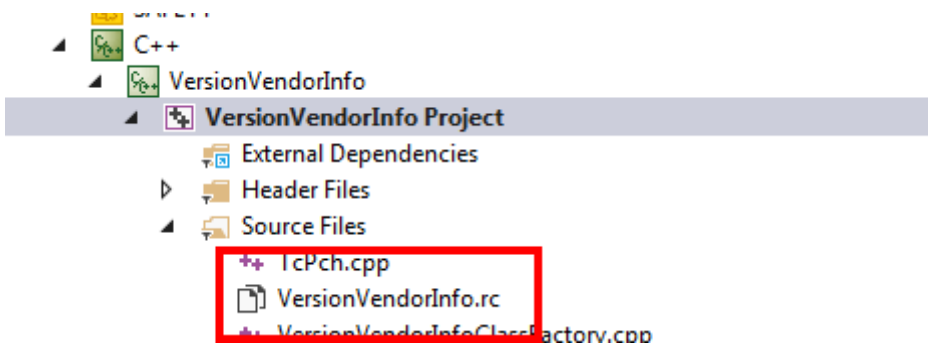
### 13.11 Setting version/vendor information

Windows provides a mechanism for retrieving vendor and version resources, which are defined during a .rc file for compile time.

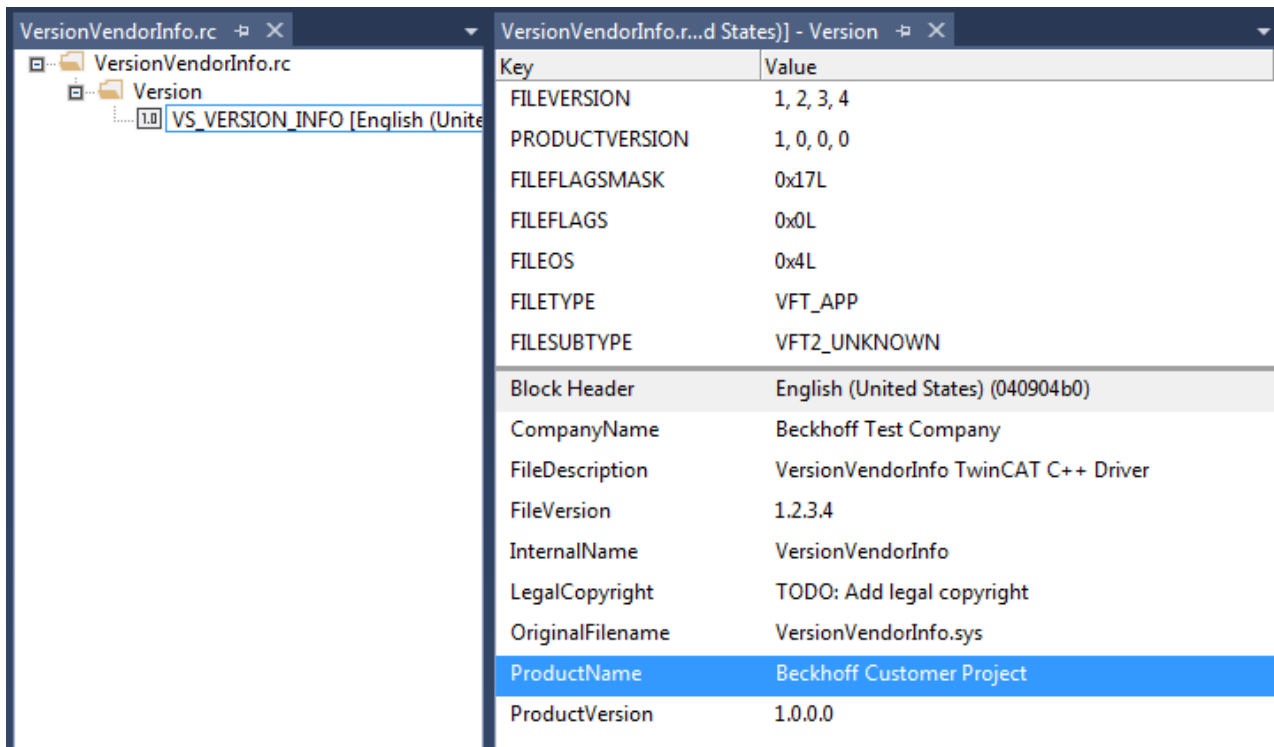
These are accessible e.g. via the Details tab of each file Properties:



TwinCAT does provide this behavior via the well-known Windows mechanisms of .rc Files, which are created during the TwinCAT C++ project creation.



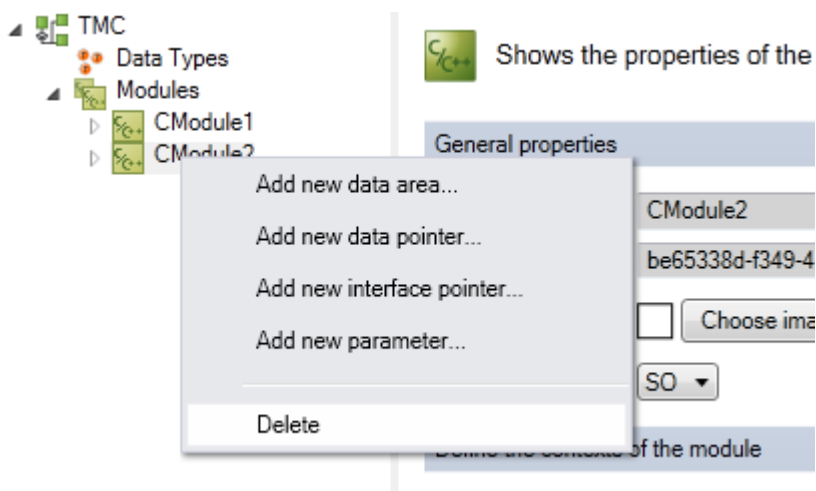
Please use the Resource Editor on the .rc-File in the Source Files folder for setting these properties:



## 13.12 Delete Module

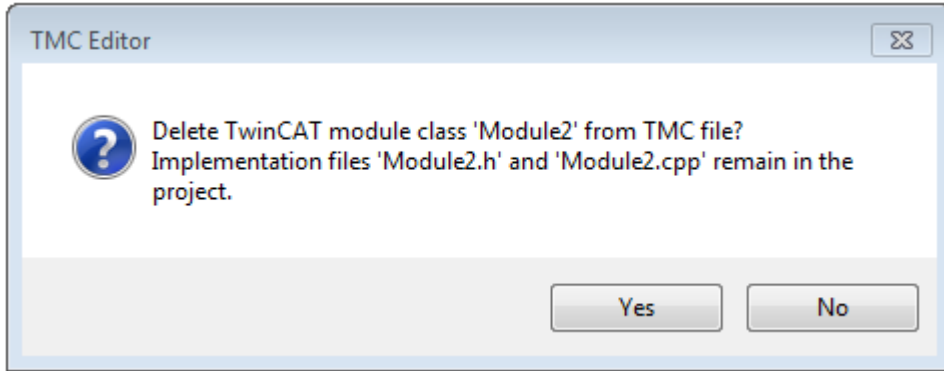
Deleting a TwinCAT C++ module from a C++ project is available via the TMC Editor.

1. Right-click on the module (here: CModule2)
2. Select "Delete"





3. Confirm deletion from TMC



4. Please note that the .cpp and .h files will remain – delete them manually, if required. Delete usage of other related components (i.e. header files, structs). Maybe see compiler error messages for assistance.

### 13.13 Initialization of TMC-member variables

All member variables of a TcCOM Module need to be initialized. The TMC Code generator supports this by

```
///

```

It is replaced by the TMC Code generator to:

```
///

```

Projects generated by TwinCAT C++ Wizard prior TwinCAT 3.1 Build 4018 does not use this feature but could easily be adopted by simply adding this line at the corresponding code (i.e. constructor):

```
///

```

### 13.14 Using PLC Strings as Method-Parameter

In order to pass a string from PLC to C++ as a method parameter use a pointer with length information while declaring the method in TMC:

Name	Type	Description
nStr	UDINT	Normal Type
pStr	SINT	Is Pointer

Such a method could be called by implementing a method within the wrapper functionblock:

```

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
    
```

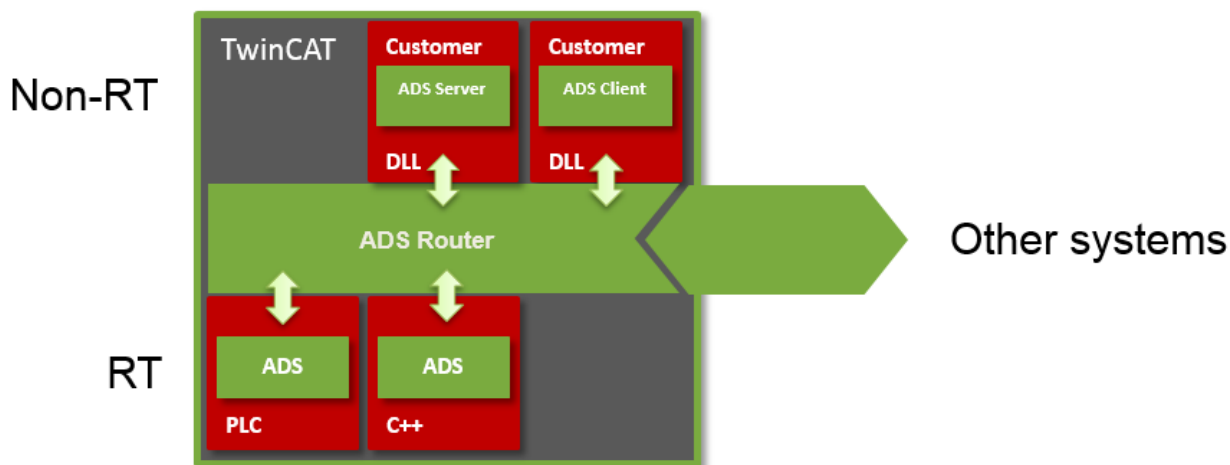
Reason is the different handling of method parameters from both worlds:

- PLC: Uses call by value for STRING(nn) datatypes
- TwinCAT C++ (TMC): Uses call by reference

### 13.15 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 User mode. 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 User-mode application that makes use of the third-party library that provides TwinCAT functions via ADS.



This action of an ADS component in User mode 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 User mode if necessary).

Such an ADS component in User mode can also be used in the same way from the PLC. In addition, ADS can communicate beyond device limits.

The following examples illustrate the use of ADS in C++ modules:

[Sample03: C++ as ADS server \[▶ 216\]](#)

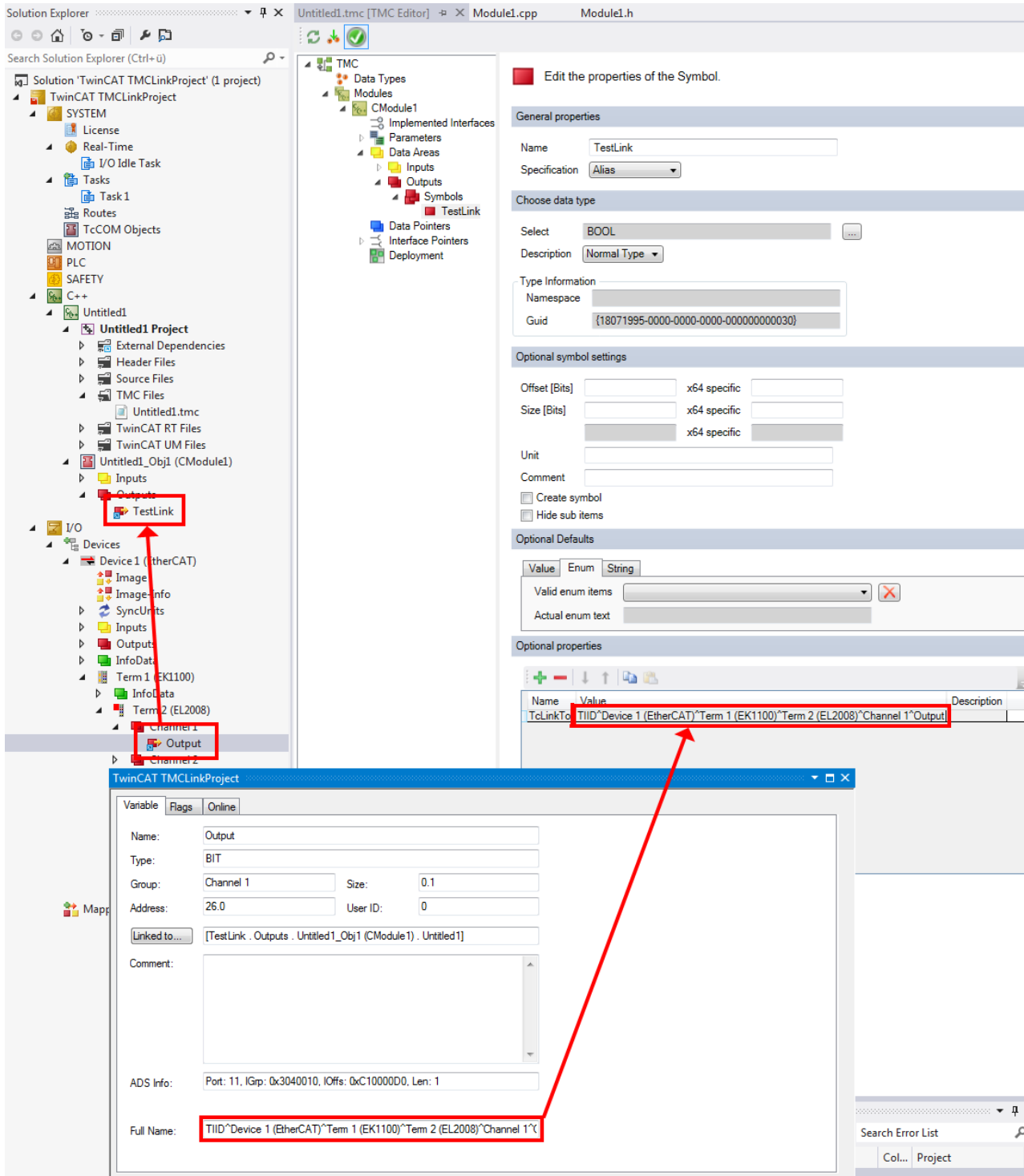
[Sample07: Receiving ADS Notifications \[▶ 231\]](#)

[Sample08: provision of ADS-RPC \[▶ 232\]](#)

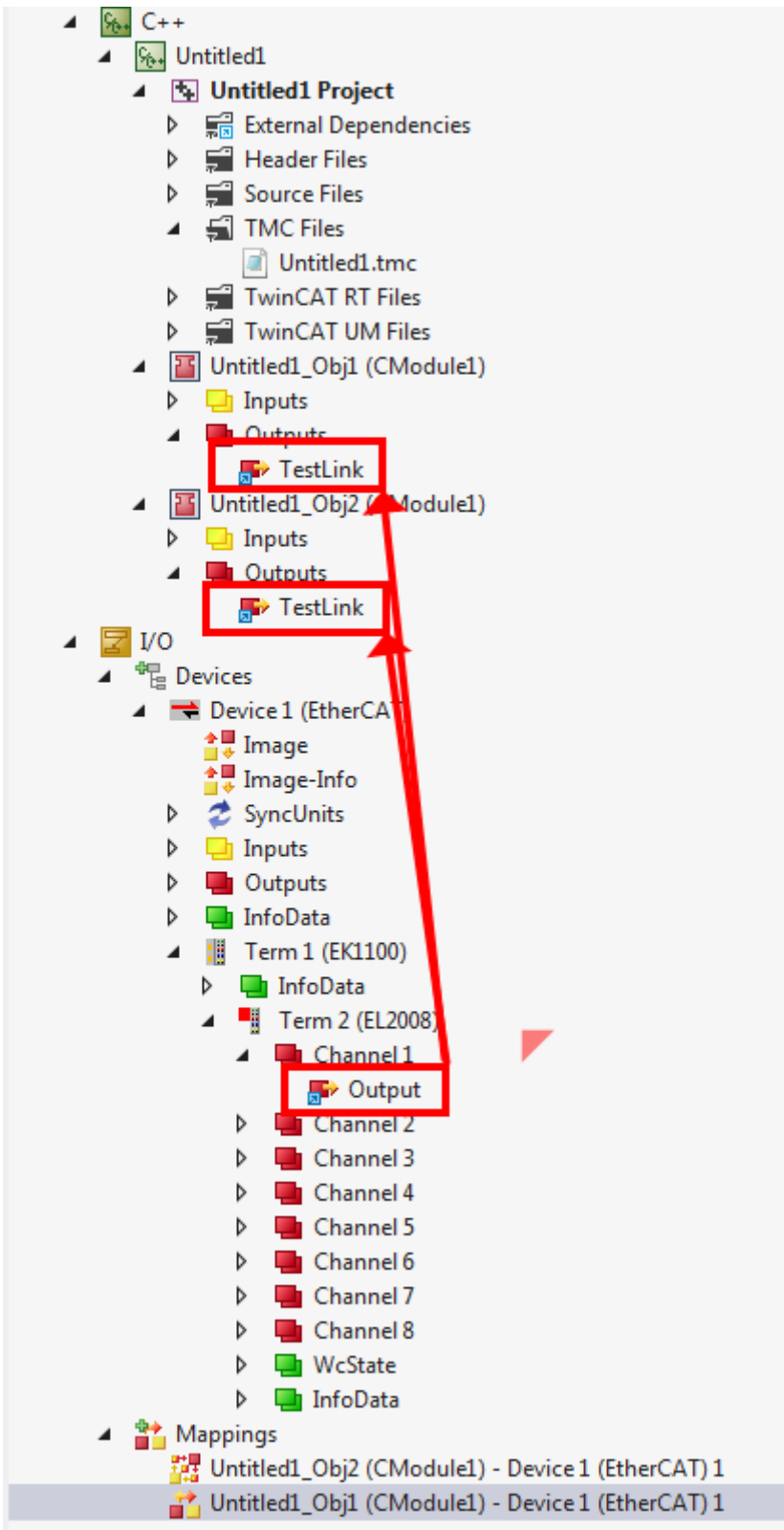
### 13.16 Linking via TMC editor (TcLinkTo)

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 property "TcLinkTo" 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:

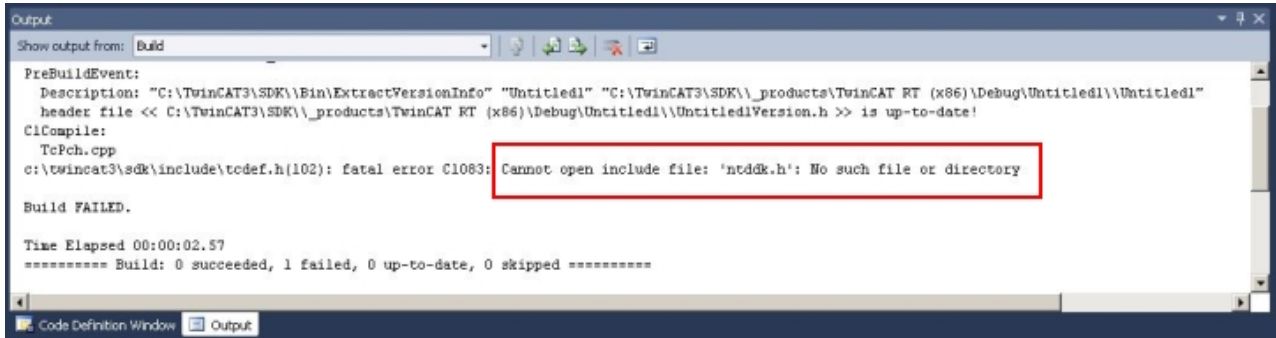


# 14 Troubleshooting

This is a list of pitfalls and glitches within the handling of TwinCAT C++ modules.

## 14.1 Build - "Cannot open include file ntddk.h"

When building a TwinCAT C++ project, this error message indicates that there is a problem with the WinDDK installation on your engineering computer.

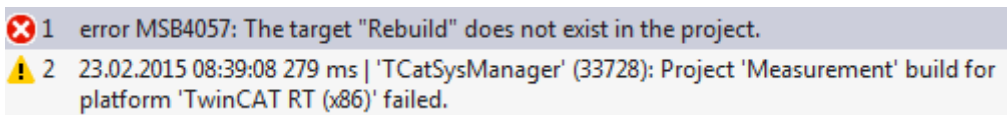


In case this error message is shown, please check the following according to the [WinDDK installation manual](#) [p. 20]:

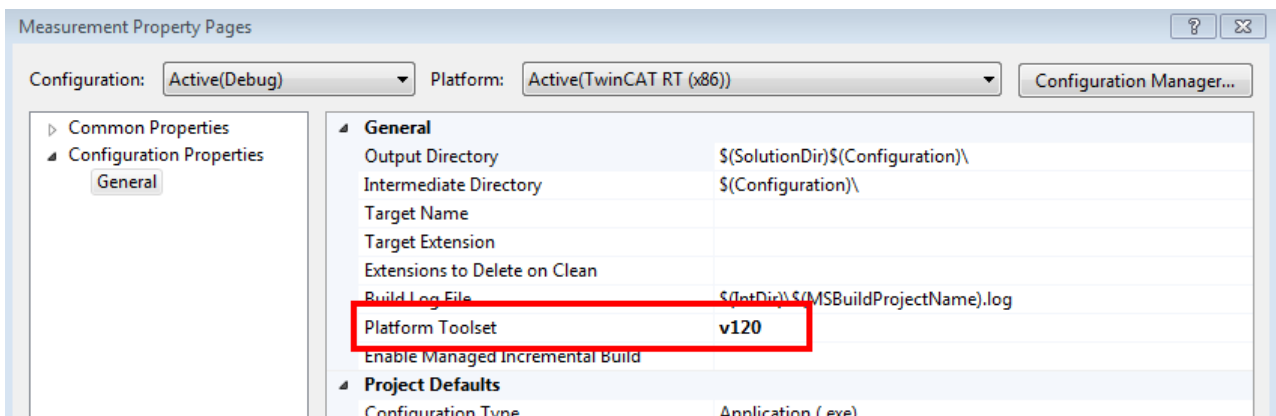
- Check if you have installed WinDDK
- Check the existence of the environment variable "WinDDK7" and its configured value, as described in the document above. The value must be equal to the path where WinDDK is installed, including the first sub-folder. After changing this value a reboot is needed.

## 14.2 Build - "The target ... does not exist in the project"

Especially when transferring a TwinCAT solution from one machine to another, Visual Studio might come up with error messages that all targets (like Build, Rebuild, Clean) do not exist in the project.

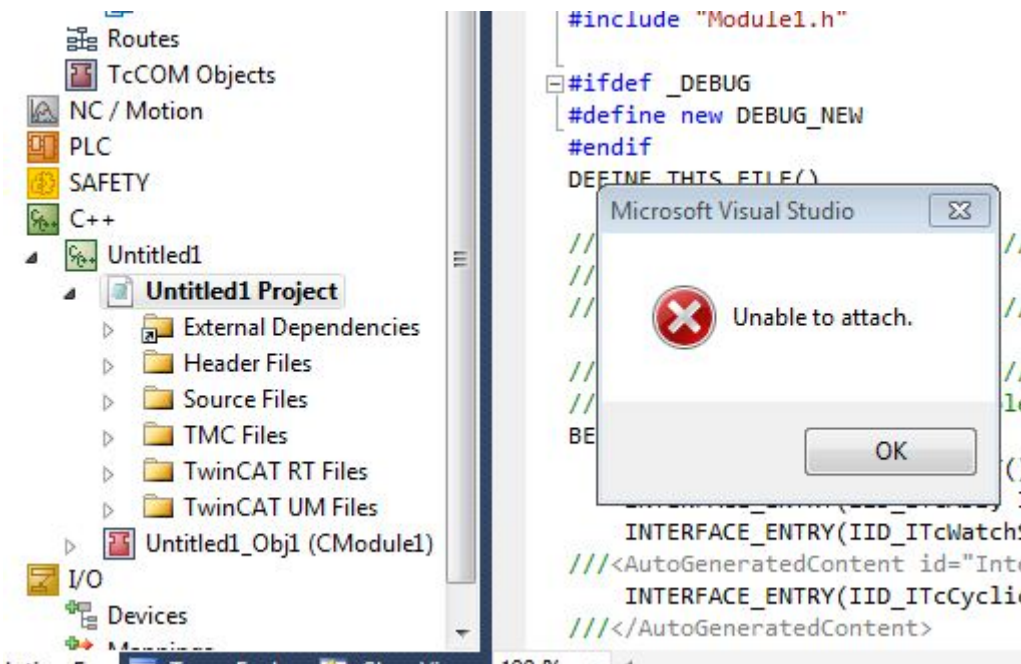


Please check the "platform toolset" configuration of the C++ project – it need to be reconfigured, if solutions are migrated from one to another Visual Studio version:

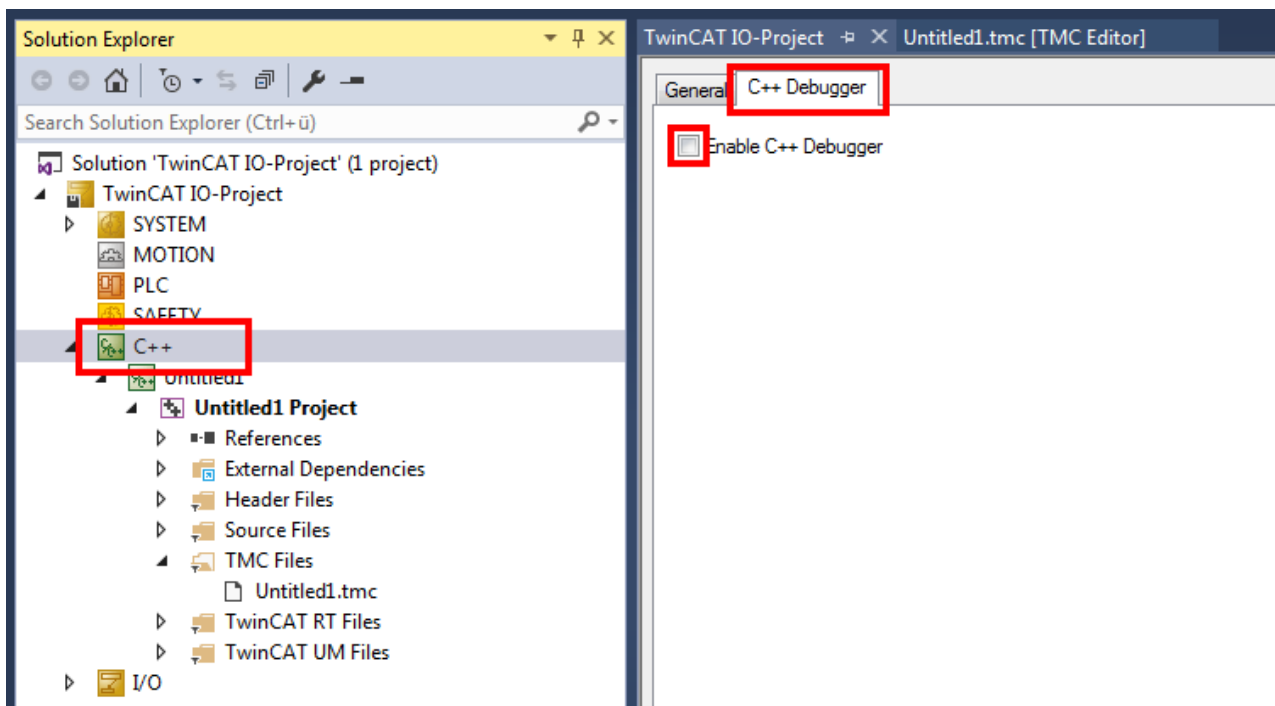


### 14.3 Debug - "Unable to attach"

When starting the debugger to debug a TwinCAT C++ project, this error message indicates that there is a missing configuration step:

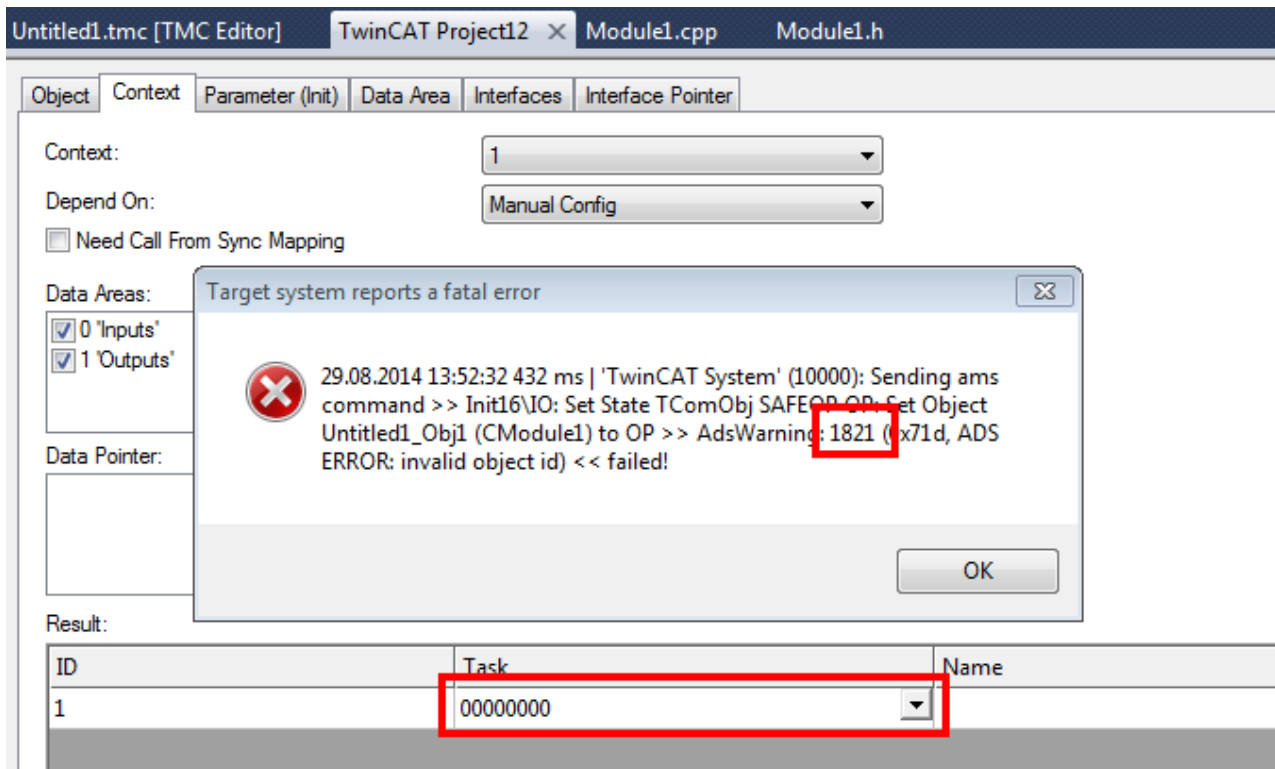


In case this error message is shown, please navigate to "System -> Real-Time" and select tab "C++ Debugger" and activate the option "Enable C++ Debugger"



## 14.4 Activation – “invalid object id” (1821/0x71d)

If the ADS return code 1821 / 0x71d is reported during startup, please check the context of the module instance like described in the [Quick Start \[▶ 59\]](#).



## 14.5 Error Message – VS2010 and LNK1123/COFF

During compilation of a TwinCAT C++ module the error message

```
LINK : fatal error LNK1123: failure during conversion to COFF: file invalid or corrupt
```

indicates that a Visual Studio 2010 is used, but without Service Pack 1, which is [required \[▶ 18\]](#) for TwinCAT C++ modules.

Please download the [installer program](#) for the service pack from Microsoft.

## 14.6 Using C++ classes in TwinCAT C++ module

When adding (non TwinCAT) C++ classes using the Visual Studio Add->Class... context menu, the compiler / linker complains

```
Error 4 error C1010: unexpected end of file while looking for precompiled header. Did you forget to add '#include ""' to your source?
```

Please add the following lines to the very beginning of your generated class file:

```
#include "TpCh.h"
#pragma hdrstop
```

## 14.7 Using afxres.h

In some templates afxres.h is included, which on some systems is not provided.

The header file can be replaced with winres.h.

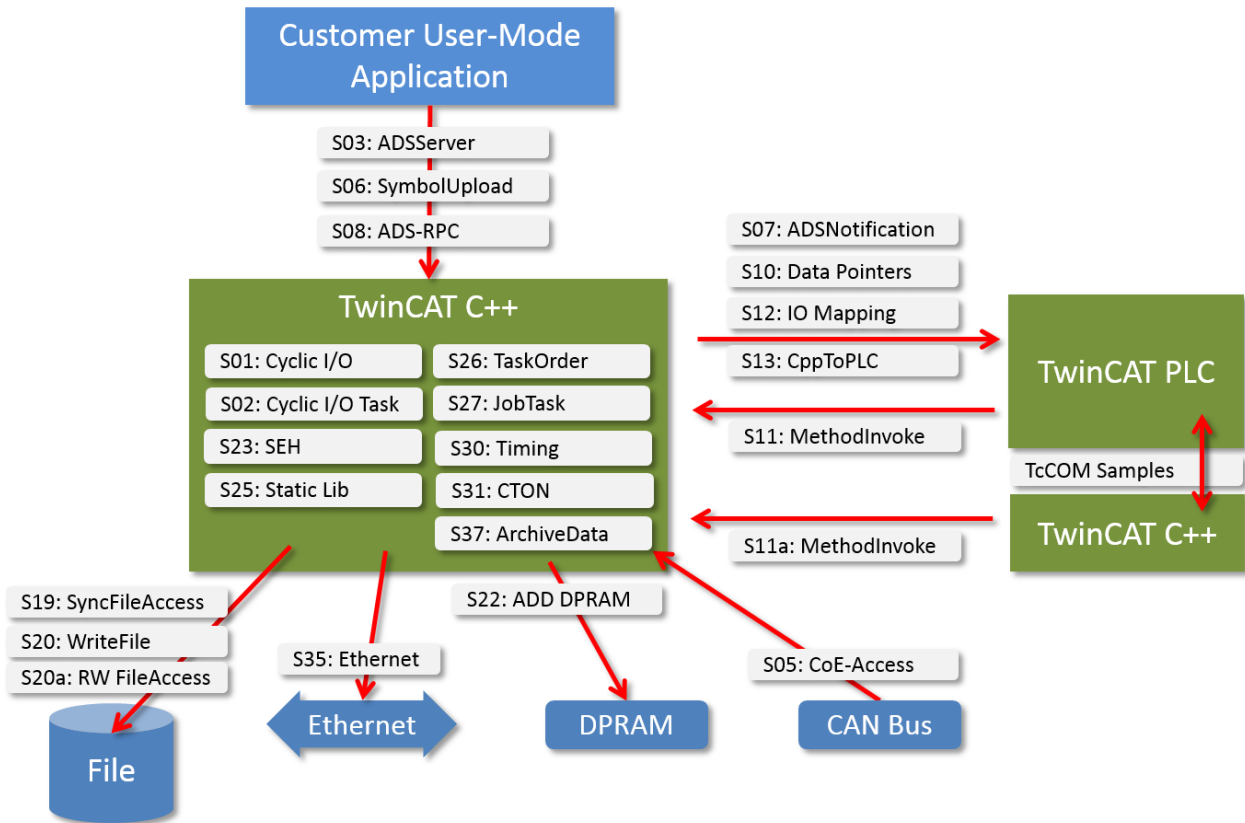


# 15 C++-samples

## 15.1 Overview

Numerous samples are available – further samples follow.

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	<a href="#">Sample01: cyclic with IO module</a> [▶ 215]	This article describes the implementation of a TC3 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 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.
02	<a href="#">Sample02: cyclic with IO task</a> [▶ 216]	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	<a href="#">Sample03: ADS Server Client</a> [▶ 216]	Describes the design and implementation of one's own ADS interface in a C++ module. The sample contains two parts: - ADS Server implemented in TC3 C++ with user-specific ADS interface. - ADS Client UI implemented in C#, which transmits user-specific ADS messages to the ADS server
05	<a href="#">Sample05: CoE access over ADS</a> [▶ 225]	Shows how CoE registers of EtherCAT devices can be accessed over ADS
06	<a href="#">Sample06: ADS C# client uploads ADS symbols</a> [▶ 226]	Shows how symbols in an ADS server can be accessed via the ADS interface. C# ADS client connects with a module implemented in PLC/C++/Matlab; uploading of the available symbol information and read/write subscription for process values.
07	<a href="#">Sample07: Receiving ADS Notifications</a> [▶ 231]	Describes the implementation of a TC3 C++ module that receives ADS notifications regarding data changes on other modules.
08	<a href="#">Sample08: provision of ADS-RPC</a> [▶ 232]	Describes the implementation of methods that can be called by ADS via the task.
10	<a href="#">Sample10: Module communication: Use of data pointers</a> [▶ 235]	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	<a href="#">Sample11: Module communication: PLC module calls a method of a C-module</a> [▶ 236]	This sample contains two parts <ul style="list-style-type: none"> <li>• 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.</li> <li>• Second PLC module for interacting with the first module by calling methods from the C++ module</li> </ul>
11a	<a href="#">Sample11a: Module communication: C-module cites a method in the C-module</a> [▶ 263]	This sample contains two classes in one driver (can also be done between two drivers) <ul style="list-style-type: none"> <li>• One module that provides a calculation method. Access is protected through a Critical section.</li> <li>• A second module that acts as the caller in order to use the methods in the other module</li> </ul>
12	<a href="#">Sample12: Module communication: IO mapping used</a> [▶ 264]	<ul style="list-style-type: none"> <li>• 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.</li> </ul>


13	<a href="#">Sample13: Module communication: C-module calls PLC methods [▶ 265]</a>	• Describes how a TwinCAT C++ module calls methods of a PLC function block via TcCOM interface.
19	<a href="#">Sample19: Synchronous File Access [▶ 268]</a>	Describes how the File IO function can be used in a synchronized manner with C++ modules. The sample writes process values in 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. The status of the routine for decoupled writing to the file can be checked.
20	<a href="#">Sample20: FileIO-Write [▶ 269]</a>	Describes how the File IO function can be used with C++ modules. The sample writes process values in 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. The status of the routine for decoupled writing to the file can be checked.
20a	<a href="#">Sample20a: FileIO-Cyclic Read / Write [▶ 269]</a>	A more extensive sample than S20 and S19. It describes the cyclic read and/or write access to files from a TC3 C++ module.
22	<a href="#">Sample22: Automation Device Driver (ADD): Access DPRAM [▶ 270]</a>	Describes how the TwinCAT Automation Device Driver (ADD) is to be written for access to the DPRAM.
25	<a href="#">Sample25: Static Library [▶ 274]</a>	Describes how to use the TC3 C++ static library contained in another TC3 C++ module.
26	<a href="#">Sample26: Execution order at one task [▶ 275]</a>	Describes the determination of the task execution sequence, if a task is assigned to more than one module.
27	<a href="#">Sample27: Using the JobTask</a>	Describes the use of a JobTask by means of four variants.
30	<a href="#">Sample30: Timing Measurement [▶ 277]</a>	Describes the measurement of the TC3 C++ cycle or execution time.
31	<a href="#">Sample31: Functionblock TON in TwinCAT3 C++ [▶ 278]</a>	Describes the implementation of a behavior in C++, which is comparable to a TON function block of PLC / 61131.
37	<a href="#">Sample37: Archive data [▶ 280]</a>	Describes the loading and saving of the state of an object during the initialization and de-initialization.
TcCOM	<a href="#">TcCOM samples [▶ 281]</a>	Several samples are provided to illustrate the module communication between PLC and C++.

## 15.2 Sample01: Cyclic module with IO

This article describes how to implement a TC3 C++ module which is using the module IO mapped to physical IO

### Download

Here you can access the [source code for this sample](#).

1. Unpack the downloaded ZIP file.
2. Open the zip file that it contains in TwinCAT 3 by clicking on "Open Project...".
3. Select your target system.
4. Build the sample on your local machine (e.g. Build->Build Solution).
5. Activate the configuration by clicking on  .  
⇒ The sample is ready for operation.

## Description

This demo describes the jump start to create a C++ module incrementing a counter each cyclic and assign the counter in the logical output "Value" in the data-area.  
The data-area can be mapped to physical IO or to another logical input of another module instance.


This sample is step by step described as quick start [here \[▶ 50\]](#).

## 15.3 Sample02: Cyclic C++ logic using IO from IO-task

This article describes how to implement a TC3 C++ module which is using an image of an IO-task.

### Download

Here you can access the [source code](#) for this sample.

1. Unpack the downloaded ZIP file.
2. Open the zip file that it contains in TwinCAT 3 by clicking on "Open Project...".
3. Select your target system.
4. Build the sample on your local machine (e.g. Build->Build Solution).
5. Activate the configuration by clicking on  .  
⇒ The sample is ready for operation.

Source code, which is not automatically generated by the wizard, is identified with a start flag "//sample code" and end flag "//sample code end".

In this way you can search for these strings in the files, in order 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 "TcloTaskImageAccessDrv" with a module instance "TcloTaskImageAccessDrv\_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.

## 15.4 Sample03: C++ as ADS server

This article describes how to

- Create a TC3-C++ module acting as an ADS-server.  
The server will provide an ADS interface to start / stop / reset a counter variable inside the C++ module. The counter is available as module output and can be mapped to an output terminal (analog

or number of digital IO's)

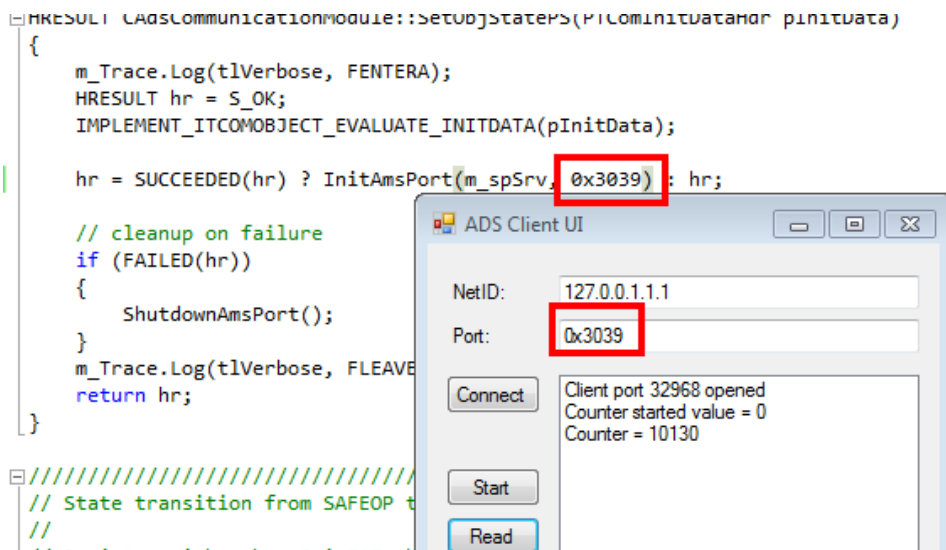
[How to implement the TC3 ADS-Server functionality written in C++ \[▶ 217\]](#)

- Create a C#-ADS-Client to interact with C++-ADS-Server.  
The client provides a UI to connect locally or over network to ADS-Server with ADS-interface to count.  
The UI allows to start / stop / read / overwrite and reset the counter  
[Sample code: ADS Client UI written in C# \[▶ 221\]](#)

### Understanding the Sample

The sample uses capabilities to automatically determine an ADS port. This has the drawback that the client needs to be configured every startup to access the correct ADS Port.

Alternatively, one can hard-code the ADS port in module like shown below.  
Drawback here: The C++ module can't be instantiated more than once since sharing an ADS port is not possible.




## 15.4.1 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:

1. Unpack the downloaded ZIP file.
  2. Open the zip file that it contains in TwinCAT 3 by clicking on "Open Project...".
  3. Select your target system.
  4. Build the sample on your local machine (e.g. Build->Build Solution).
  5. Activate the configuration by clicking on .
- ⇒ The sample is ready for operation.

### Description

This sample contains one C++ module acting as a ADS-server providing access to a counter which could be started, stopped and read.

The header file of the module defines the counter variable "m\_bCount" and the corresponding .cpp file initialises the value within the constructor as well as implements the logic within the "CycleUpdate" method.

The method "AdsReadWriteInd" of the .cpp parses incoming messages and sends back the return values. For one additionally added message type a define where added to the header file.

Details like the definition of the ADS message types are described within the following Cookbook, where you can assemble 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](#) [► 50]

### 2. Create a C++ project with ADS port

Follow the steps to [create a new TwinCAT 3 C++ project](#) [► 51].

In the dialog "Class templates" select "TwinCAT Module Class with ADS port".

### 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 IComObject
    , 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 data "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". Add the blue code lines 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
};
```

Add the blue code lines in your <MyClass>::AdsReadWriteInd() method (in this case 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. Add sample logic to the cyclic part

The method <MyClass>::CycleUpdate() is cyclically called - this is the place to modify the logic.

```

// 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 to 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 \[► 56\]](#) in order to provide the inputs/outputs for the module.
  2. Save the project.
  3. [Compile \[► 56\]](#) 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 \[► 62\]](#) the TwinCAT project.
- ⇒ The sample is ready for operation.

## 5. Determine the ADS port of the module instance

Generally the ADS port may be

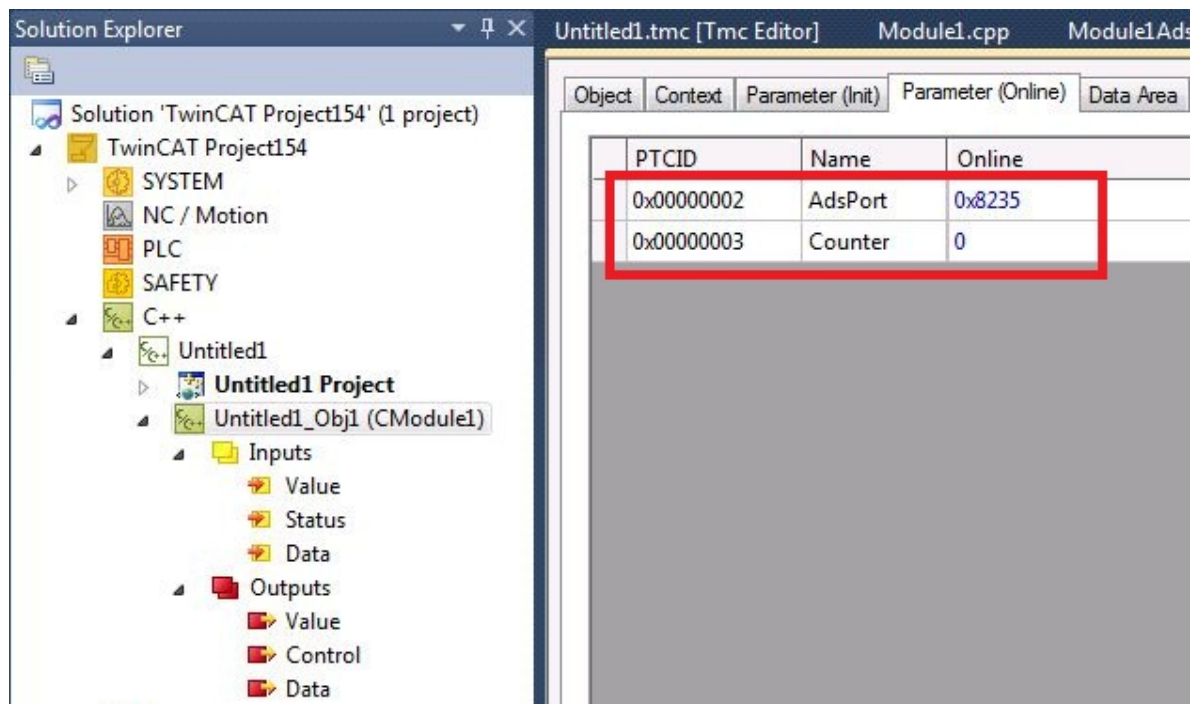
- pre-numbered, so that the same port is always used for this module instance



- 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](#) [▶ 221].

#### Also see about this

- 📄 [Create TwinCAT 3 C++ Module instance](#) [▶ 57]
- 📄 [Create a TwinCAT task and apply it to the module instance](#) [▶ 59]

## 15.4.2 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.**

- ✓ This code requires .NET Framework 3.5 or higher!
1. Unpack the downloaded ZIP file
  2. Open the included sln file with Visual Studio
  3. **Create the sample on your local machine** (right-click on the project and click on "Build")
  4. Start the program, i.e. 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 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)

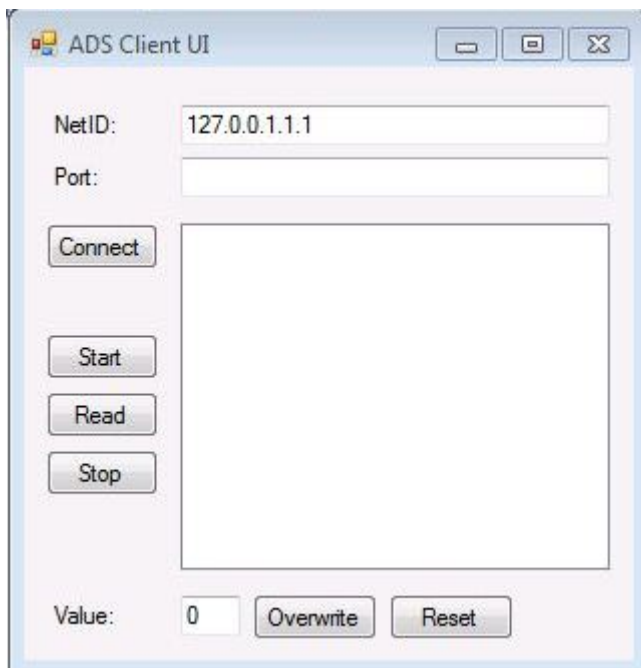
Enter another NetID, if you want to communicate with an ADS partner connected to another ADS router via the network.

First you have to create an ADS route between your device and the remote device.

- AdsPort  
Enter the 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 access to symbols (this is provided automatically, without the need for user intervention).  
Find the assigned AdsPort [► 217], in this sample the AdsPort was 0x8235 (dec 33333).

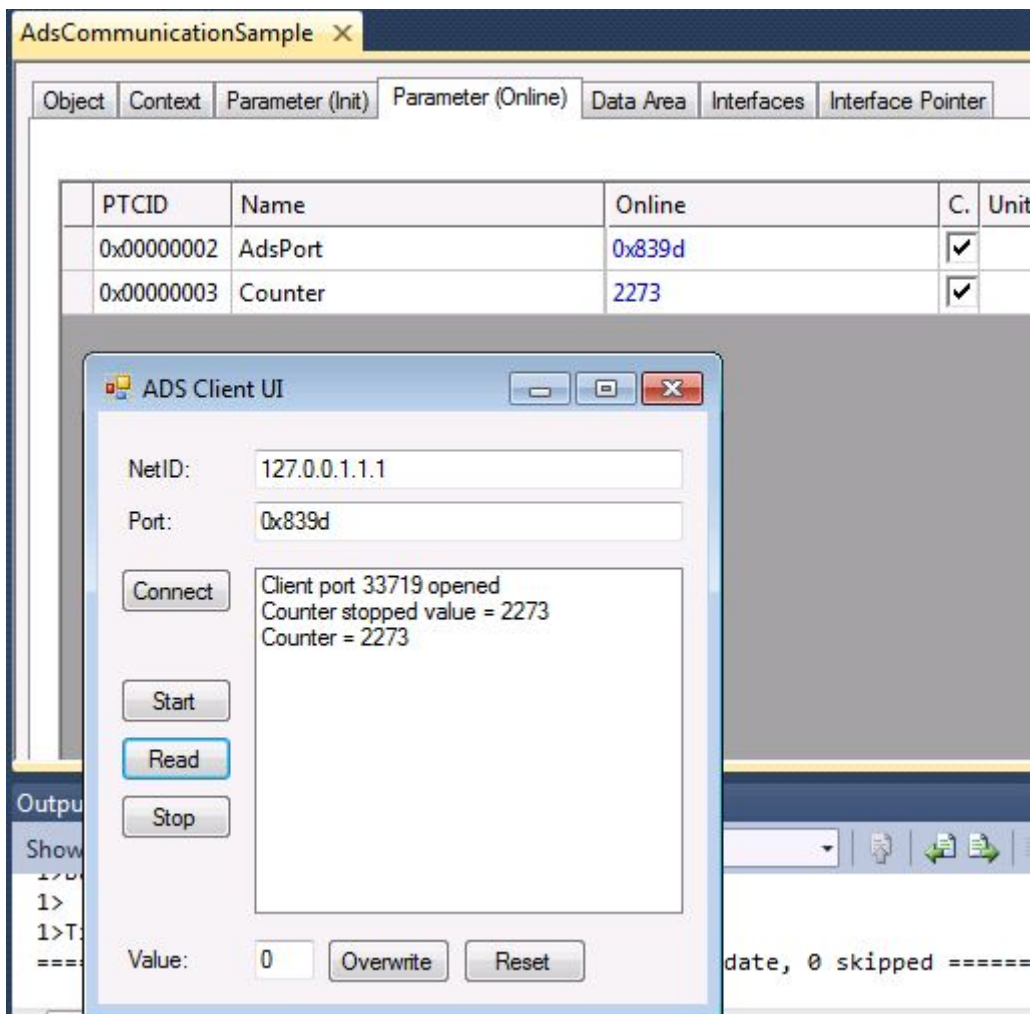
### Create link with communication partner

Click on "Connect" to call the method TcAdsClient.Connect for the purpose of creating a link with the configured port.



Use the buttons Start / Read / Stop / Overwrite / Reset to send ADS messages to the ADS server. The specific indexGroup / indexOffset commands were already designed in the ADS interface of the ADS server [► 217].

The result of clicking on the command buttons can also be seen in the module instance in the tab "Parameters (online)".



## C# program

This is the "core" code of ADS client for the GUI etc.; please download the ZIP file shown 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);
    }
}
}
```

## 15.5 Sample05: C++ CoE access via ADS

This article describes how to implement a TC3 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](#).

1. Unpack the downloaded ZIP file.
2. Open the zip file that it contains in TwinCAT 3 by clicking on "Open Project...".
3. Select your target system.
4. Build the sample on your local machine (e.g. Build->Build Solution).
5. Note the actions listed on this page under **Configuration**.
6. Activate the configuration by clicking on  .  
⇒ The sample is ready for operation.

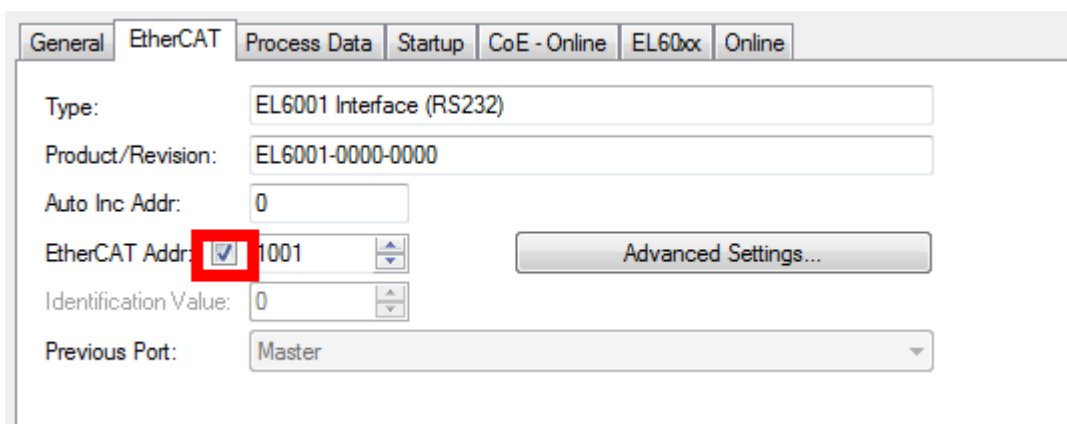
### Description

This sample describes access to an EtherCAT Terminal, which reads the manufacturer ID and specifies the baud rate for serial communication.

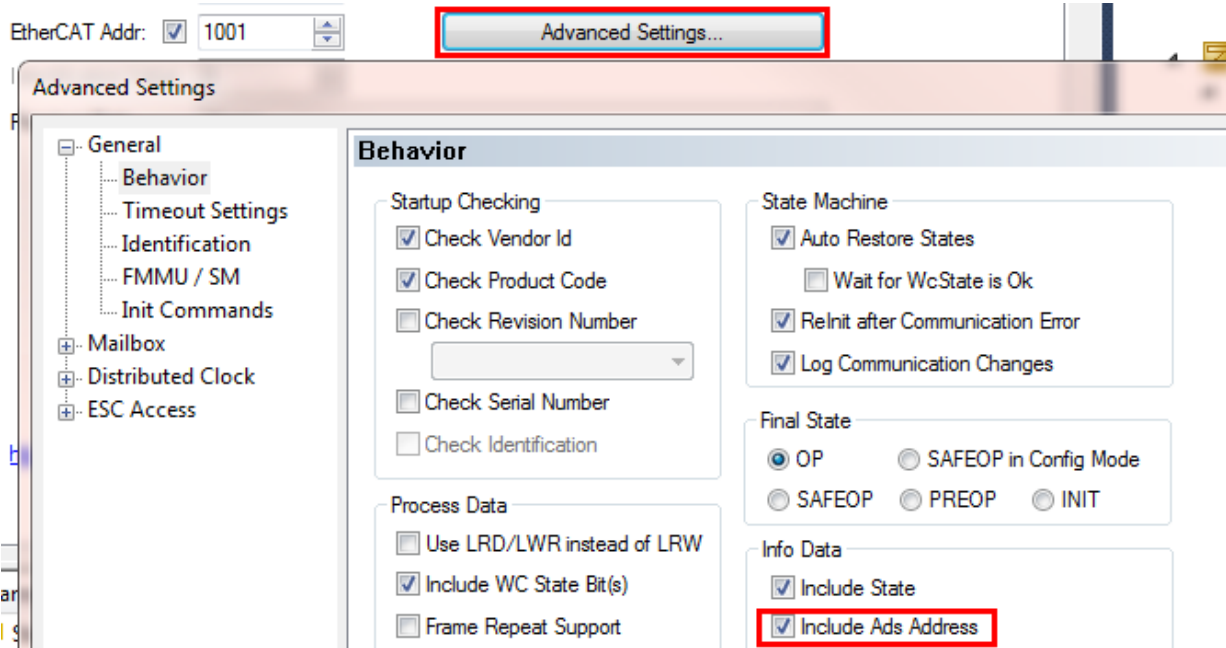
This sample describes the quick start for the purpose of creating a C++ module that increments a counter on each cycle and assigns the counter to the logical output "Value" in the data area.

### Configuration

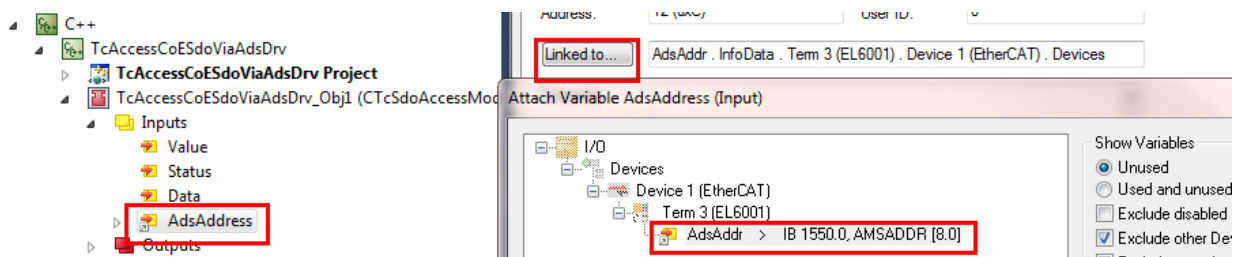
1. Activate the EtherCAT address of the terminal concerned and assign it.



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 AdsAddress:



4. The module parameters are read out and displayed by the sample code during the course of the

initialization:

Object	Context	Parameter (Init)	Parameter (Online)	Data Area	Interfaces	Interface Po
		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

## 15.6 Sample06: UI-C#-ADS client uploading the symbolic from module

This article describes the implementation of an ADS client for

- communicating with an ADS server, which provides a process image (data area). The connection can be local or remote via the network.

- Upload symbol information
- Read / write data synchronously
- Subscribe to symbols, in order to obtain values "on change" as callback.

## Download

**Access the** [source code](#) for this client sample:

- ✓ This code requires .NET Framework 3.5 or higher!
1. Unpack the downloaded ZIP file
  2. Open the included sln file with Visual Studio
  3. **Create the sample on your local machine** (right-click on the project and click on "Build")
  4. Start the program, i.e. right-click on Project, Debug->Start new instance

The client sample should be used with example 03 "C++ as ADS server".

Please open [sample 03 \[► 217\]](#) before starting with this client-side sample!

## Description

This sample illustrates the ADS options.

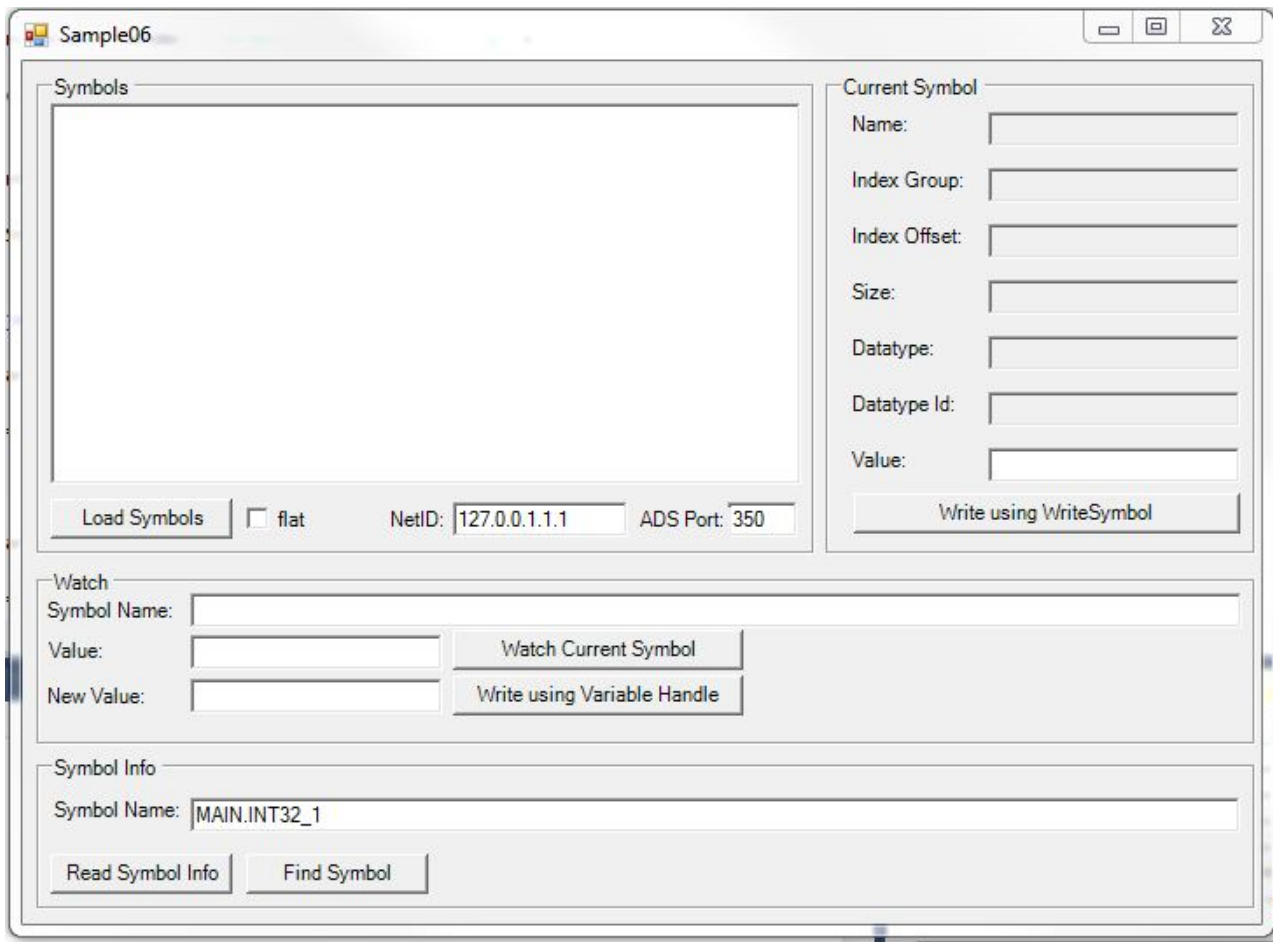
The details of the implementation are described in "Form1.cs", which is included in the download. The connection via ADS with the target system is established in the "btnLoad\_Click" method, which is called on clicking on the "Load Symbols" button. From there you can explore the different GUI functions.

## Background information:

For this ADS client it is irrelevant whether the ADS server is based on TwinCAT 2 or TwinCAT 3. Also, it does not matter whether the server is a C++ module, a PLC module or an IO task without 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, in order 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)

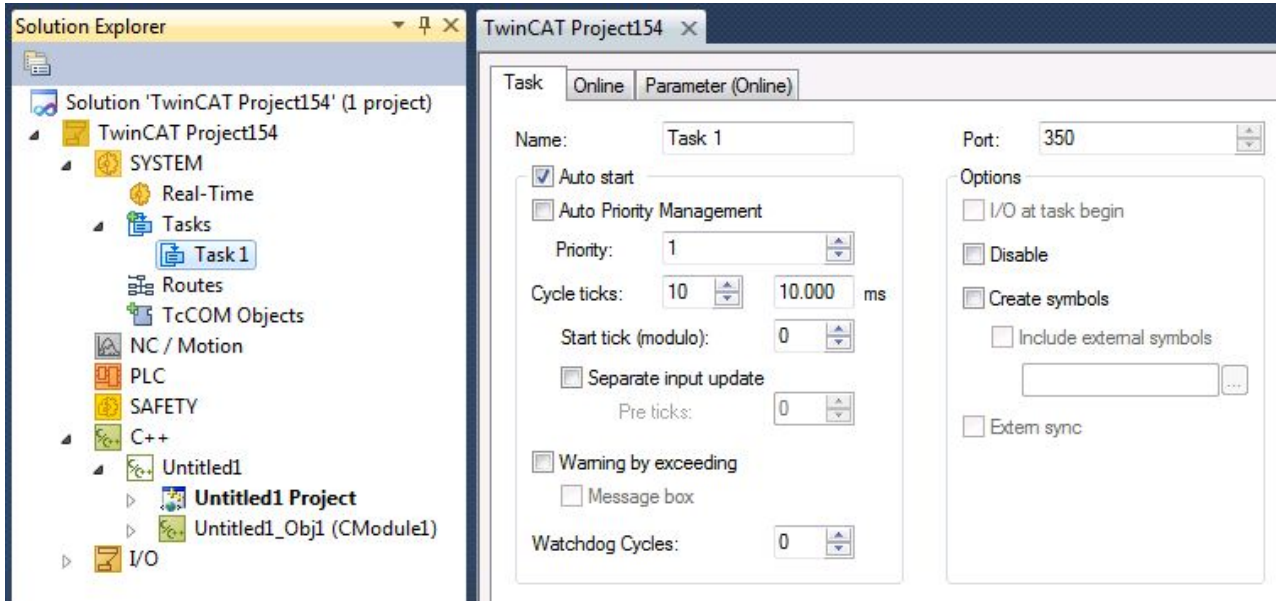
### **i** Do not confuse the ADS server port with the regular ADS port.

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):

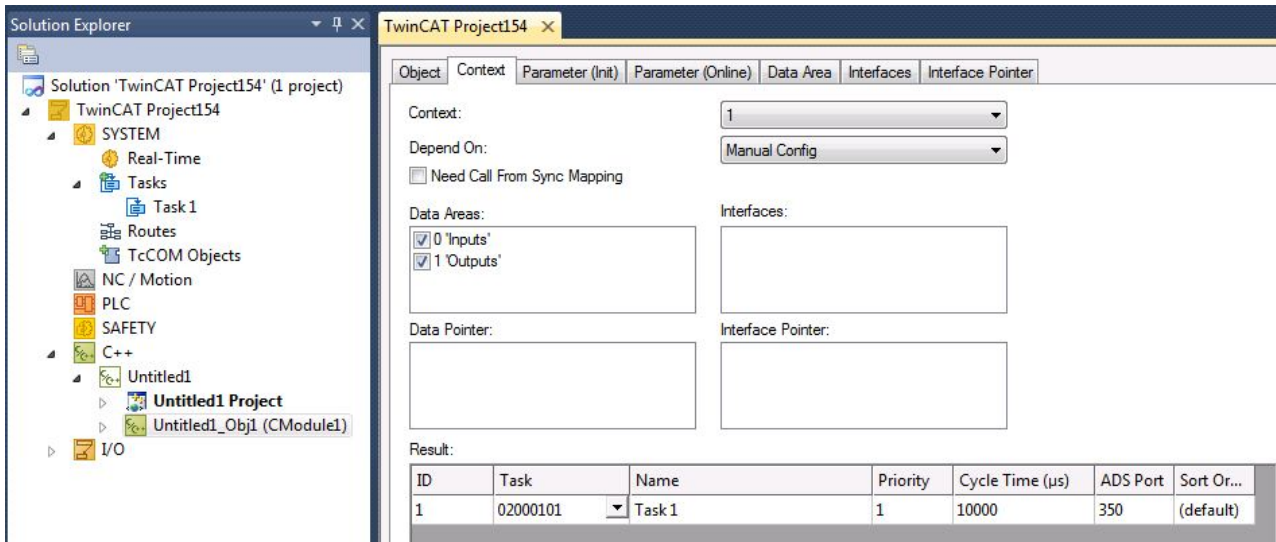
The regular ADS port is required to access symbols. You can find the AdsPort for the IO task of your instance or the module instance yourself (since the module is executed in the context of the IO task).



Navigate to IO task "Task1" and note the value of the port: 350



Since the C++ module instance is executed in the context of task 1, the ADS port is also 350.



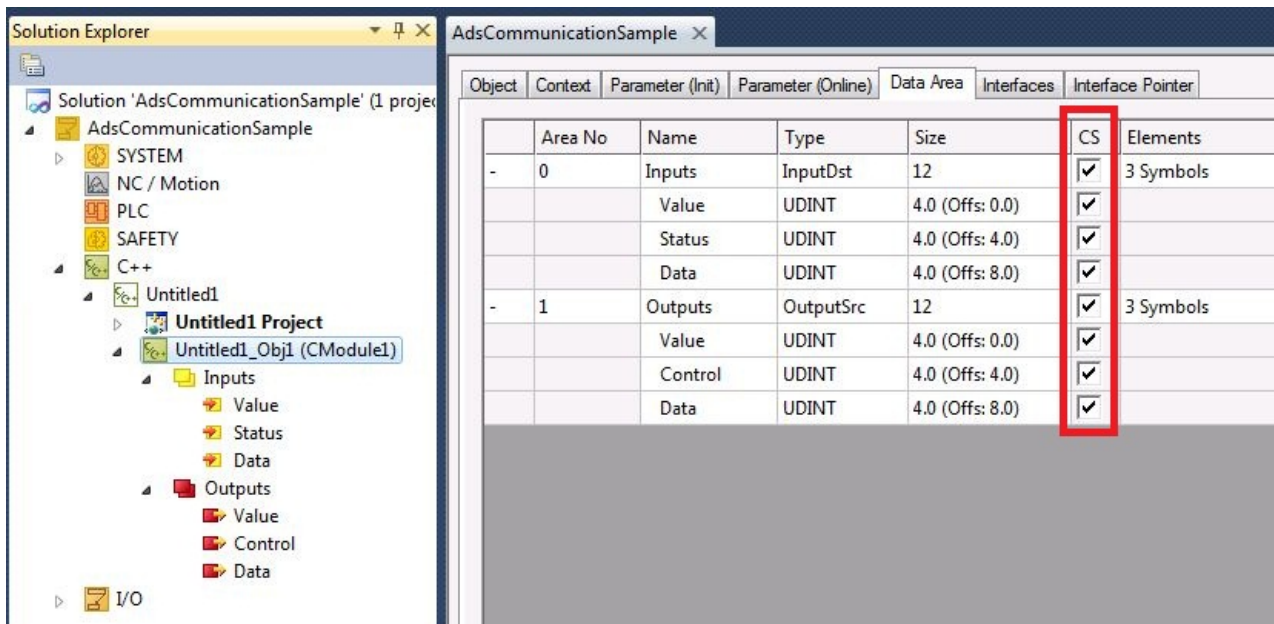
**Enabled symbols for access available via ADS**

Individual symbols or whole data areas can be provided for access via ADS, or they can be deliberately not provided.

Navigate to the "Data Area" tab of your instance and activate/deactivate the column "C/S".

In this sample all symbols are marked and therefore available for ADS access.

After the modifications please select "Activate configuration".



### Load symbols

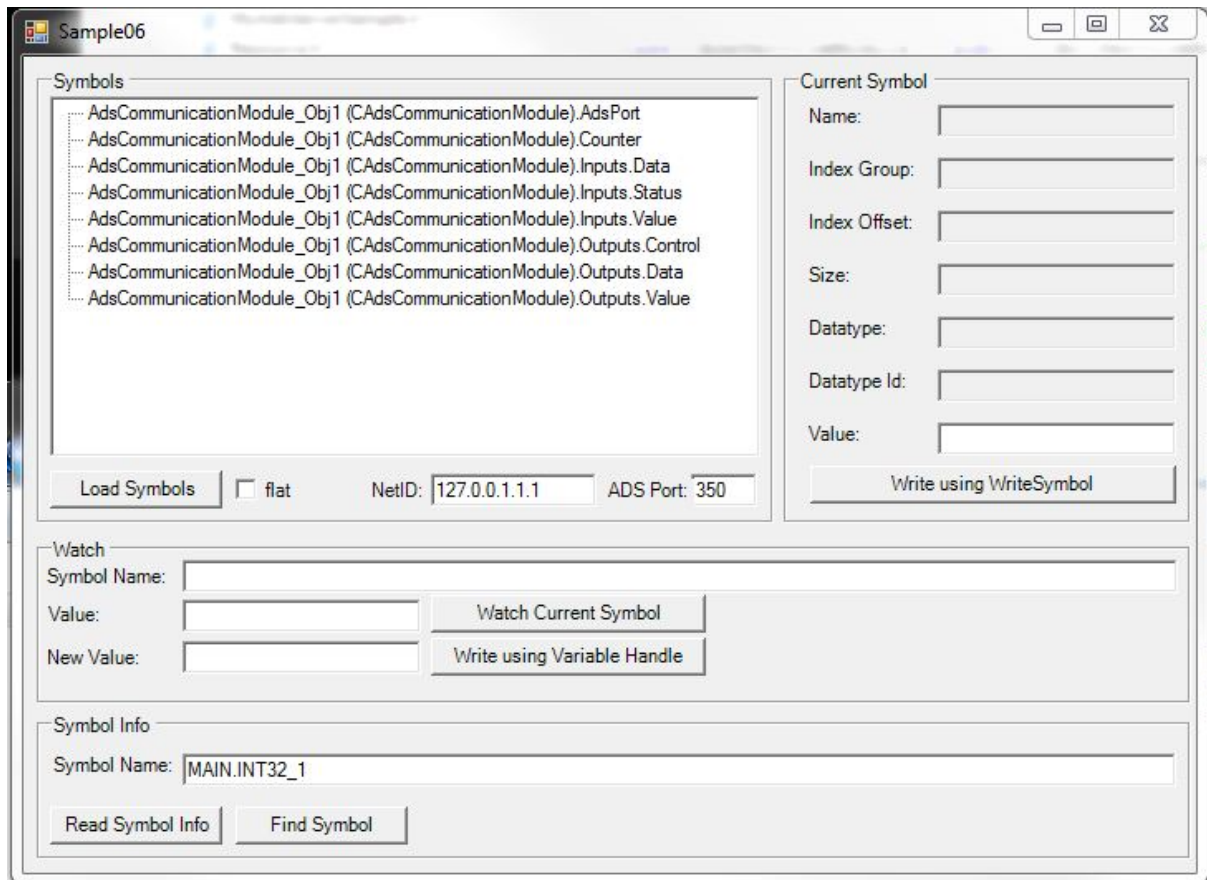
Once the NetID and the ADS port have been set up, click on the "Load Symbols" button to make a connection with the target system and load the symbols.

All available symbols are then visible. You can then:

- Write a new value:  
 Select a symbol in the tree on the left, e.g. "Counter"  
 Enter a new value in the "Value" field on the right and click on "Write using WriteSymbol".  
 The new value is written to the ADS server.

After writing a new value with "Write using WriteSymbol", the C# application is assigned a callback with the new value.

- Subscribe in order to obtain callback when the value changes.  
Select a symbol in the tree on the left, e.g. "Counter"  
Click on "Watch Current Symbol"




## 15.7 Sample07: Receiving ADS Notifications

This article describes how to implement a TC3 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 C++ modules.

### Download

Here you can access the [source code for this sample](#):

1. Unpack the downloaded ZIP file.
2. Open the zip file that it contains in TwinCAT 3 by clicking on "Open Project...".
3. Select your target system.
4. Build the sample on your local machine (e.g. Build->Build Solution).
5. Activate the configuration by clicking on  .  
⇒ The sample is ready for operation.

### Description

This sample describes the reception of ADS notifications in a TwinCAT C++ module.

The solution contains two modules for this purpose.

- A C++ module, which registers for querying ADS notifications of a variable.

- Put simply: a PLC program, which 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 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 \[► 173\]](#)

## 15.8 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](#).

1. Unpack the downloaded ZIP file.
  2. Open the zip file that it contains in TwinCAT 3 by clicking on "Open Project...".
  3. Select your target system.
  4. Build the sample on your local machine (e.g. Build->Build Solution).
  5. Activate the configuration by clicking on .
- ⇒ 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

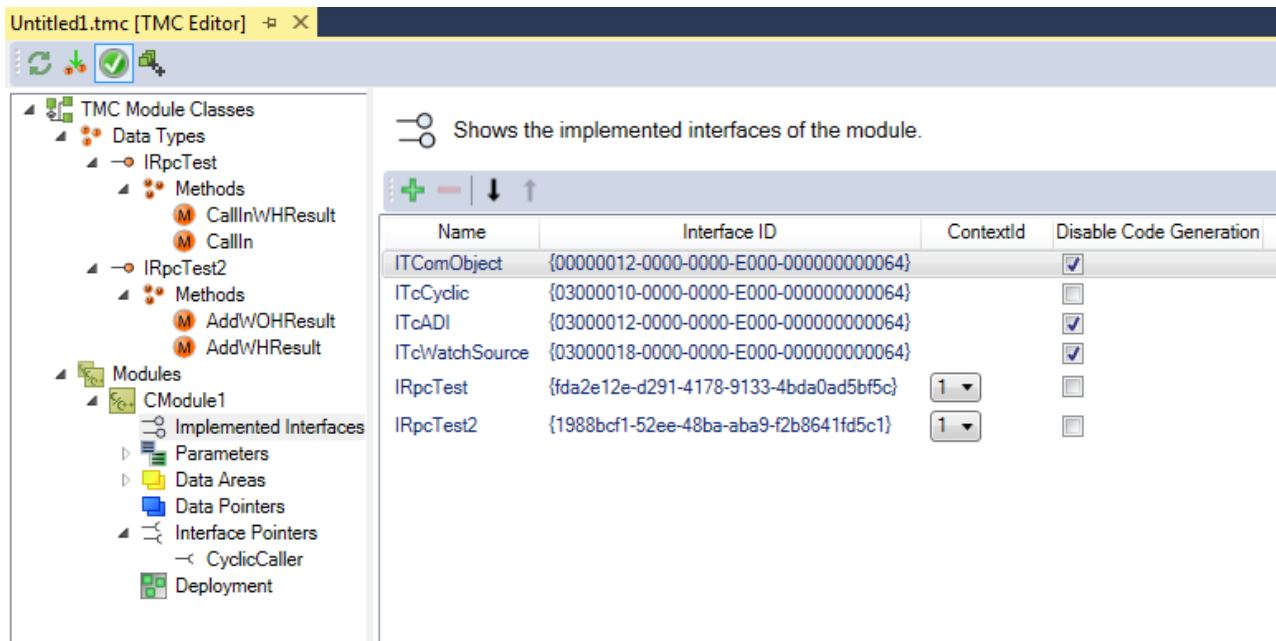
The sample consists of the TwinCAT C++ module, which offers the RPC methods and a C++ sample program that calls them.

#### 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 greater detail in the [Description of the TMC Editor \[▶ 79\]](#).

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.

Their ADS symbol name is formed according to the pattern: `ModuleInstance.Interface#MethodName`. Particularly important in the implementing module is the `ContextId`, which defines the context for the execution.

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()

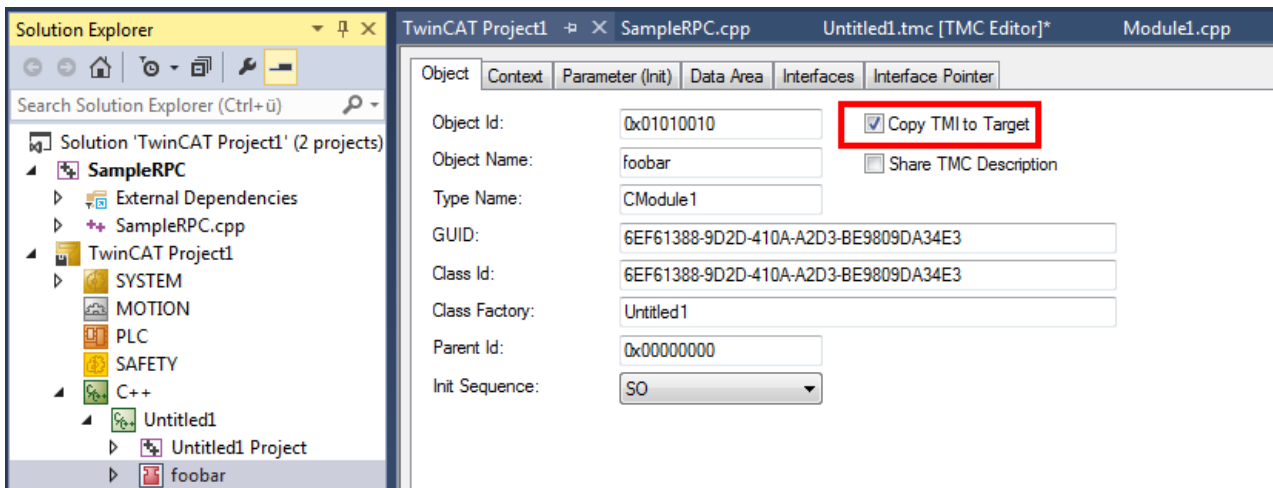
    ///<AutoGeneratedContent id="ImplementationOf_IRpcTest">
    HRESULT CModule1::CallInWHResult(LONG in)
    {
        HRESULT hr = S_OK;
        return hr;
    }

    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 also call these methods by OPC-UA – the TMI files are required on the target system for this.

### C++ example 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
```

## 15.9 Sample10: module communication: Using data pointer

This article describes the implementation of two TC3 C++ modules, which communicate via a data pointer.

### Download

Here you can access the [source code](#) for this sample:

1. Unpack the downloaded ZIP file.
  2. Open the zip file that it contains in TwinCAT 3 by clicking on "Open Project...".
  3. Select your target system.
  4. Build the sample on your local machine (e.g. Build->Build Solution).
  5. Activate the configuration by clicking on  .
- ⇒ 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 the input "Bit1" is set, the output "Switch1" is set accordingly.
- When the input "Bit2" is set, the output "Switch2" is 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 ITcyclic 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 execution order is defined by the "Sort Order" in the "Context" tab of the respective module. This is provided as parameter "SortOrder" in the SmartPointer (m\_SpCyclicCaller), which also includes the ObjectID.


## 15.10 Sample11: module communication: PLC module invokes method of C-module

This article describes the implementation:

- [of a C++ module \[► 237\]](#) that provides methods for controlling a state machine. Follow this step-by-step introduction with regard to the implementation of a C++ module that provides an interface to the state machine.
- [of a PLC module \[► 251\]](#) 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 advantage. 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

**Get the [source code for this sample](#):**

1. Unpack the downloaded ZIP file.
  2. Open the zip file that it contains in TwinCAT 3 by clicking on "Open Project...".
  3. Select your target system.
  4. Build the sample on your local machine (e.g. Build->Build Solution).
  5. Activate the configuration by clicking on  .
- ⇒ The sample is ready for operation.



## 15.10.1 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.

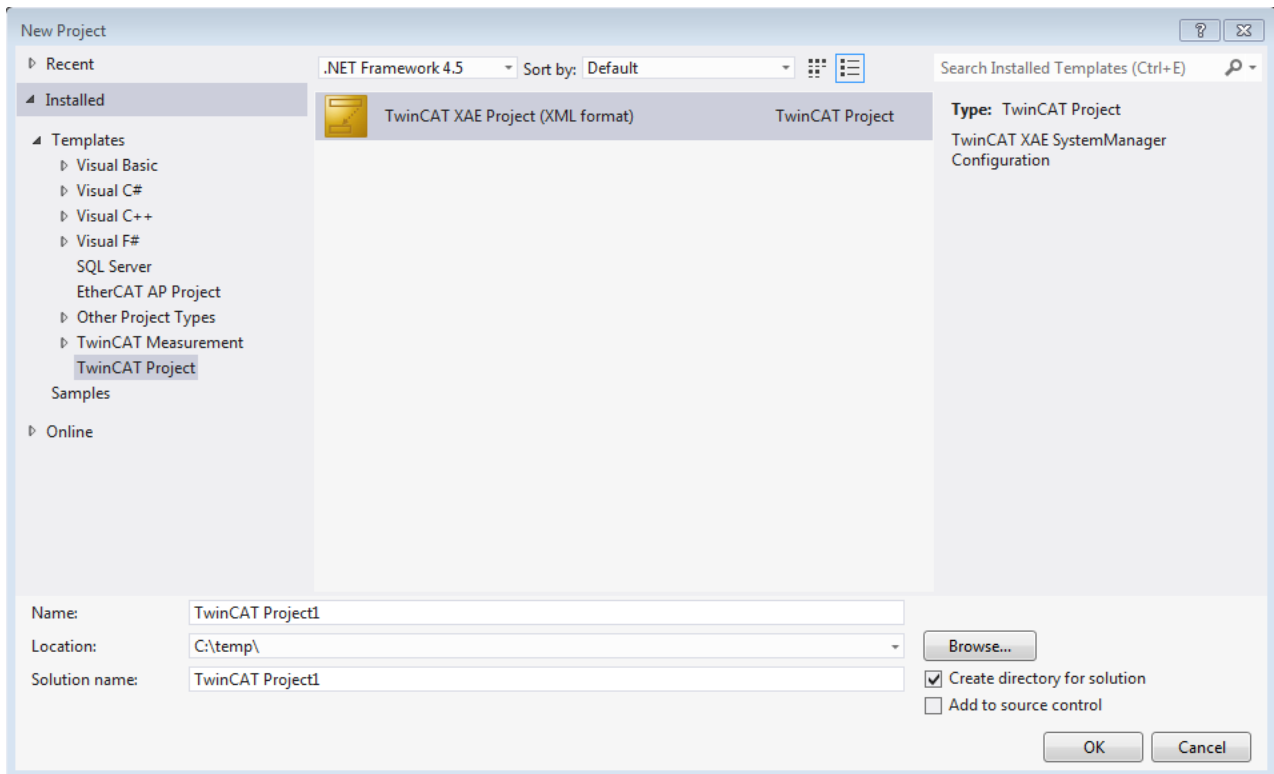
- [Calling the function from the PLC logic \[► 236\]](#) - i.e. affecting the C++ code from the PLC
- [Calling the function from the C++ logic \[► 263\]](#) - i.e. interaction between two C++ modules

This article describes:

- Step 1: [create a new TwinCAT 3 project \[► 237\]](#)
- Step 2: [create a new TwinCAT 3 C++ driver \[► 238\]](#)
- Step 3: [generate a new TwinCAT 3 interface \[► 240\]](#)
- Step 4: [add methods to the interface \[► 241\]](#)
- Step 5: [add a new interface to the module \[► 244\]](#)
- Step 6: [start the TwinCAT TMC code generator to generate code for the module class description \[► 246\]](#)
- Step 7: [implementation of the member variables and the constructor \[► 246\]](#)
- Step 8: [implementation of methods \[► 247\]](#)
- Step 9: [implementation of a cyclic update \[► 248\]](#)
- Step 10: [compilation of code \[► 249\]](#)
- Step 11: [creating an instance of the C++ module \[► 250\]](#)
- Step 12: [finished; check results \[► 251\]](#)

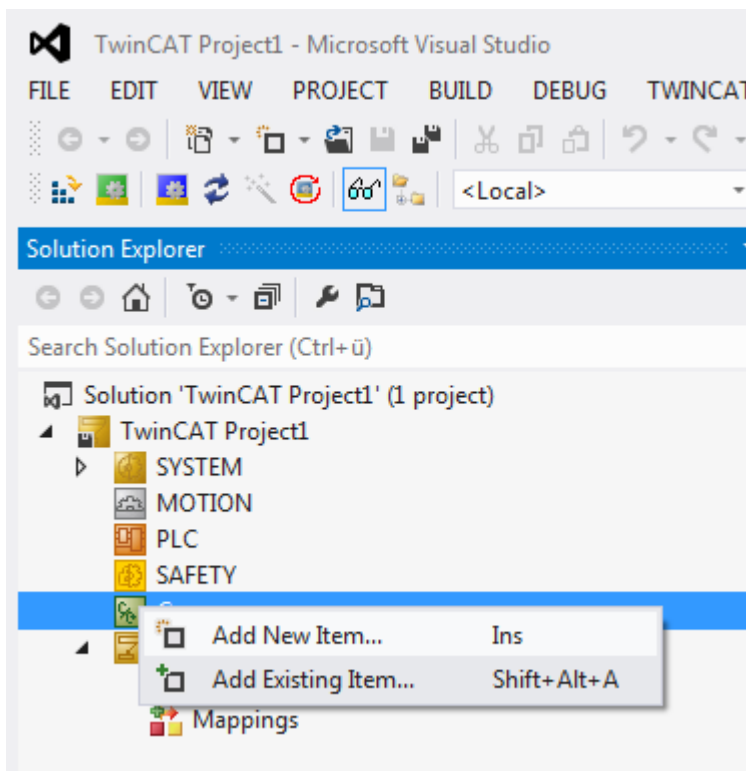
### Step 1: create a new TwinCAT 3 project

First of all, create a TwinCAT project as usual.

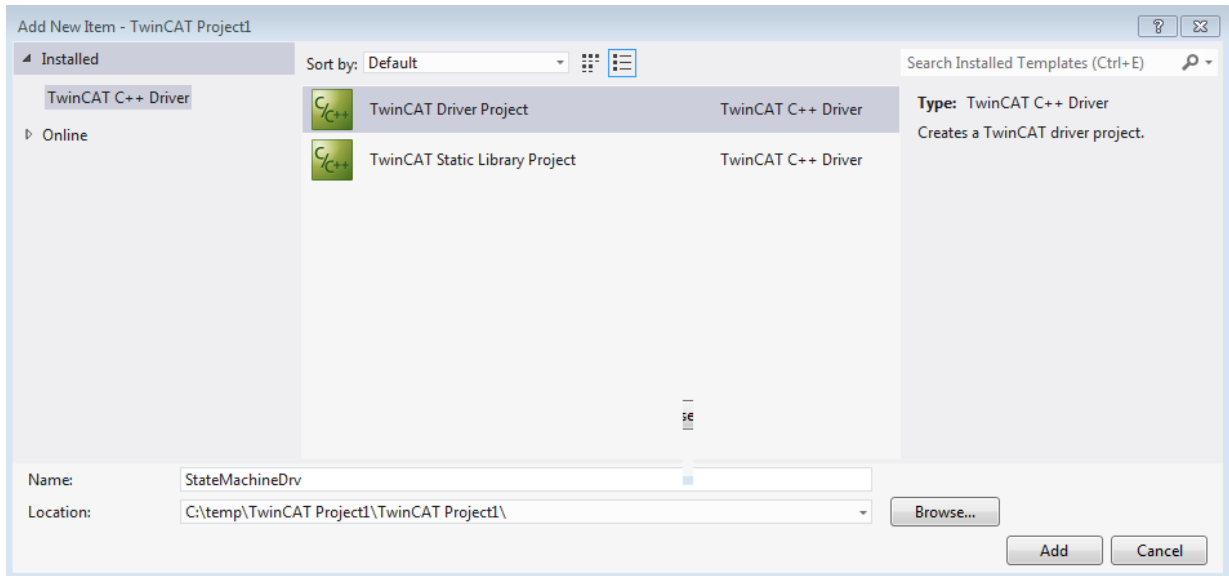


## Step 2: create a new TwinCAT 3 C++ driver

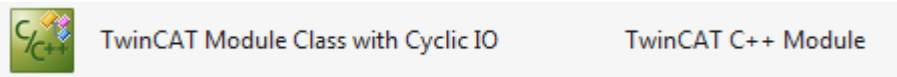
1. Right-click on **C++** and Add New Item...



2. Select the template "TwinCAT Driver Project" and enter a driver name, "StateMachineDrv" in this sample. Click on **Add** to continue.

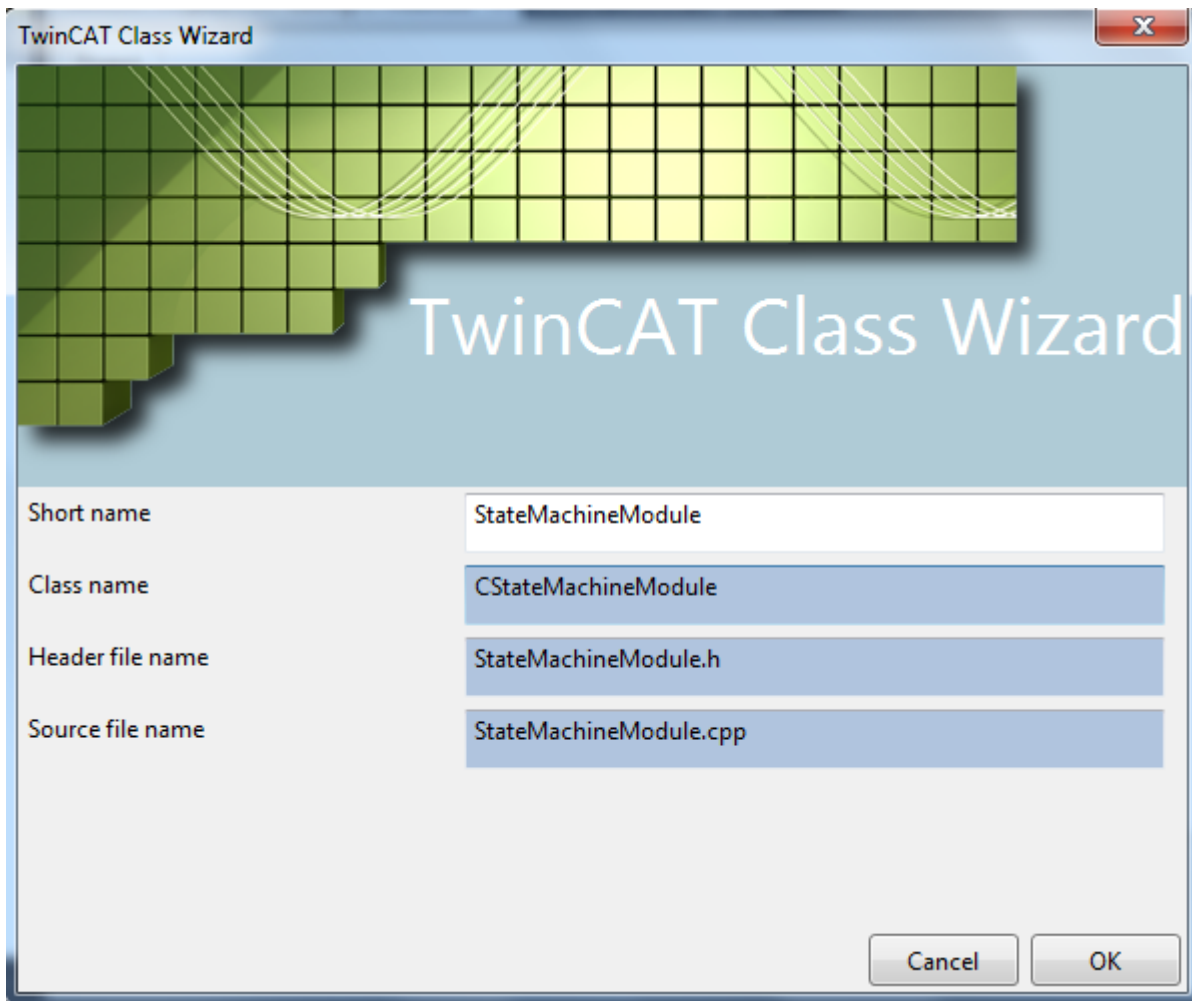


3. Select a template to be used for this new driver. In this sample "TwinCAT Module Class with Cyclic IO" is selected, since the internal counter of the state machine is available for assigning to the IO.
4. Click on **Add** to continue.



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

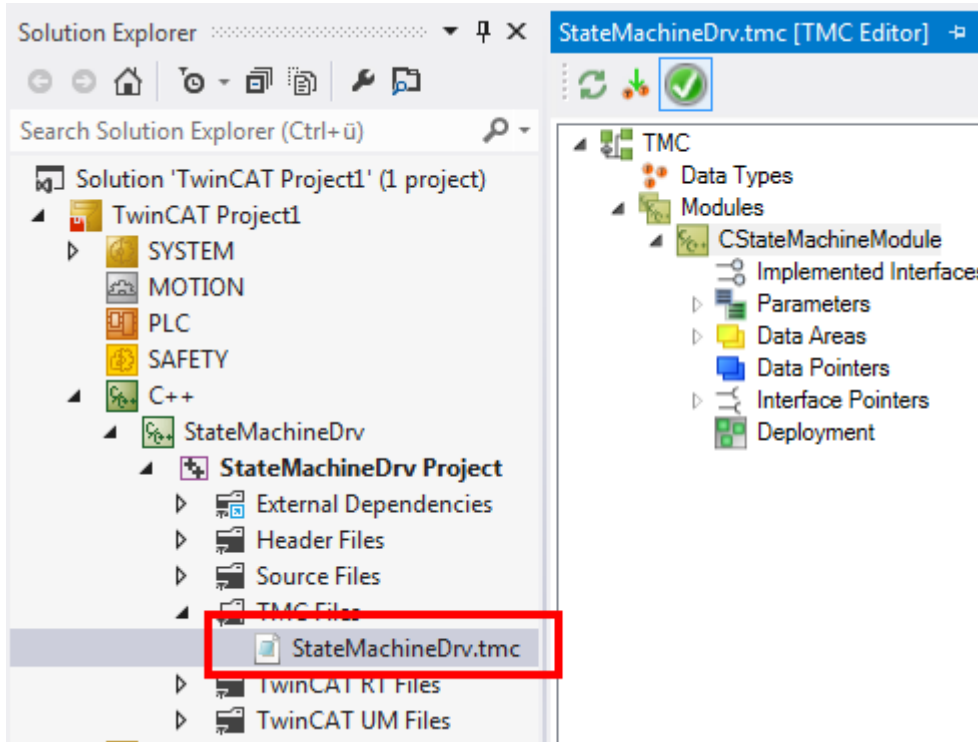
#### **NOTE**

##### **Name conflict**

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

- Please do not use keywords as names that are reserved for the PLC.

1. Start the TMC editor by double-clicking on **StateMachineDrv.tmc**.



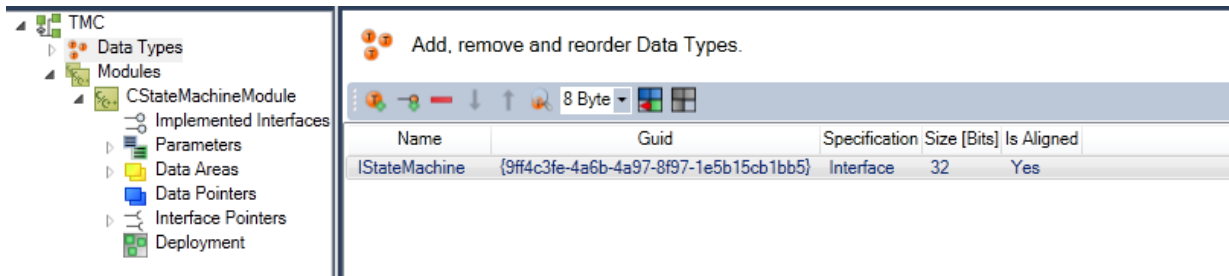
2. Select **Data Types** within the TMC editor

3. Add a new interface by clicking on **Add a new interface** 

⇒ A new entry **Interface1** is then listed.

4. Open **Interface1** 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 generate a new default method, *Method1*.

2. Replace the default name Method1 by a more meaningful name, in this sample Start.

The screenshot shows the TMC software interface. On the left is a tree view with the following structure:

- TMC
  - Data Types
    - IStateMachine
      - Methods
        - Start (highlighted with an 'M' icon)
  - Modules
    - CStateMachineModule
      - Implemented Interfaces
      - Parameters
      - Data Areas
      - Data Pointers
      - Interface Pointers
      - Deployment

The main panel on the right is titled "Edit the properties of the method." and contains the following sections:

- General properties**: Name: Start
- Define the data type**:
  - Select: HRESULT
  - Description: Normal Type
  - Type Information:
    - Name: HRESULT
    - Namespace: (empty)
    - Guid: {18071995-0000-0000-0000-000000000019}
    - Resolve Type button
- Define the parameters of the method**:
 

Name	Type	Description	Default Value

3. Add a second method and name it Stop.

The screenshot shows the TMC software interface. On the left is a tree view with the following structure:

- TMC
  - Data Types
    - IStateMachine
      - Methods
        - Start (highlighted with an 'M' icon)
        - Stop (highlighted with an 'M' icon)
  - Modules
    - CStateMachineModule
      - Implemented Interface
      - Parameters
      - Data Areas
      - Data Pointers
      - Interface Pointers
      - Deployment

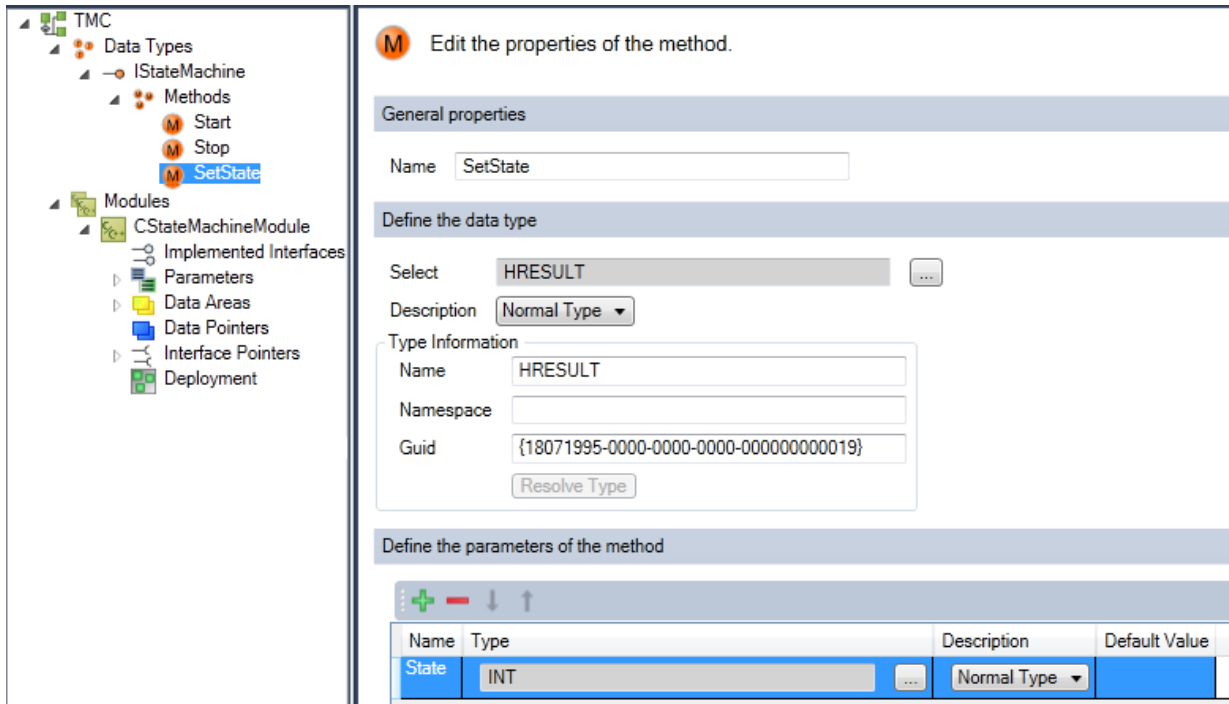
The main panel on the right is titled "Edit the properties of the method." and contains the following sections:

- General properties**: Name: Stop
- Define the data type**:
  - Select: HRESULT
  - Description: Normal Type
  - Type Information:
    - Name: HRESULT
    - Namespace: (empty)
    - Guid: {18071995-0000-0000-0000-000000000019}
    - Resolve Type button
- Define the parameters of the method**:
 

Name	Type	Description	Default Value

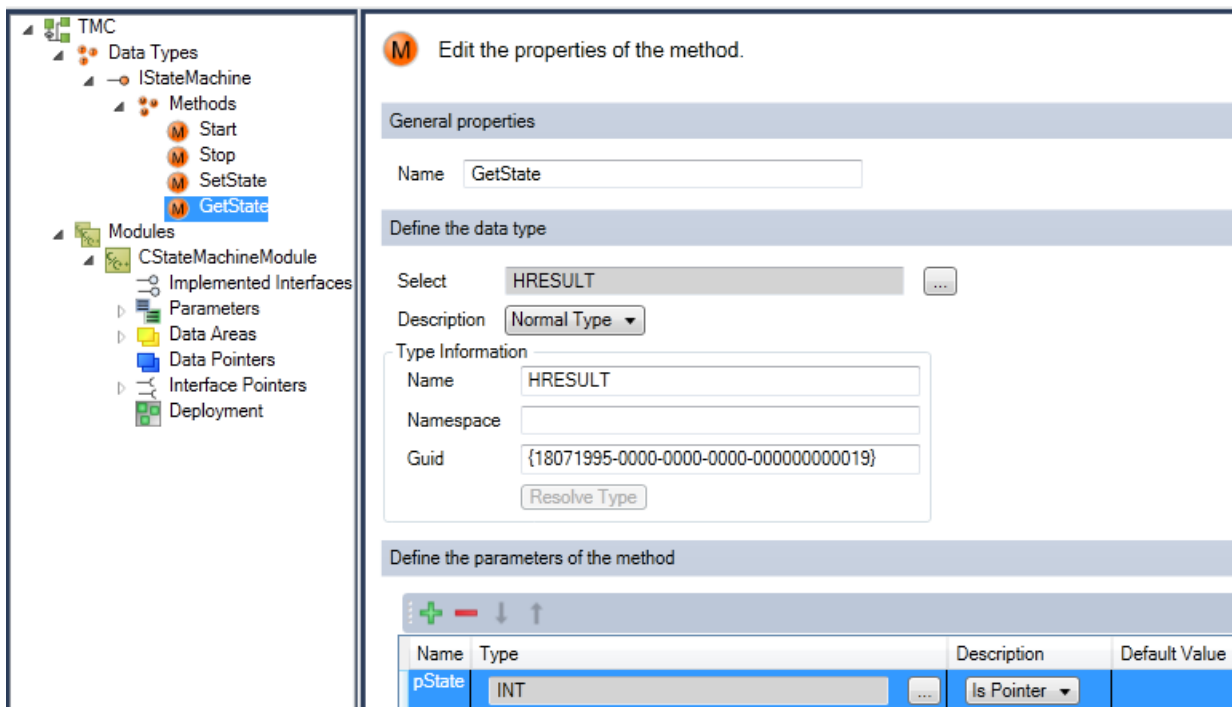
4. Add a third method and name it SetState.

- 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**.

- Click on the name Parameter1 and change the name in the editing field to State.
- After Start, Stop and SetState have been defined, define a further method.
- Rename it "GetState".
- Add a parameter and call it pState (which is conceived to become a pointer later on).
- 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.

Return Type	Name
HRESULT	Start
HRESULT	Stop
HRESULT	SetState
HRESULT	GetState

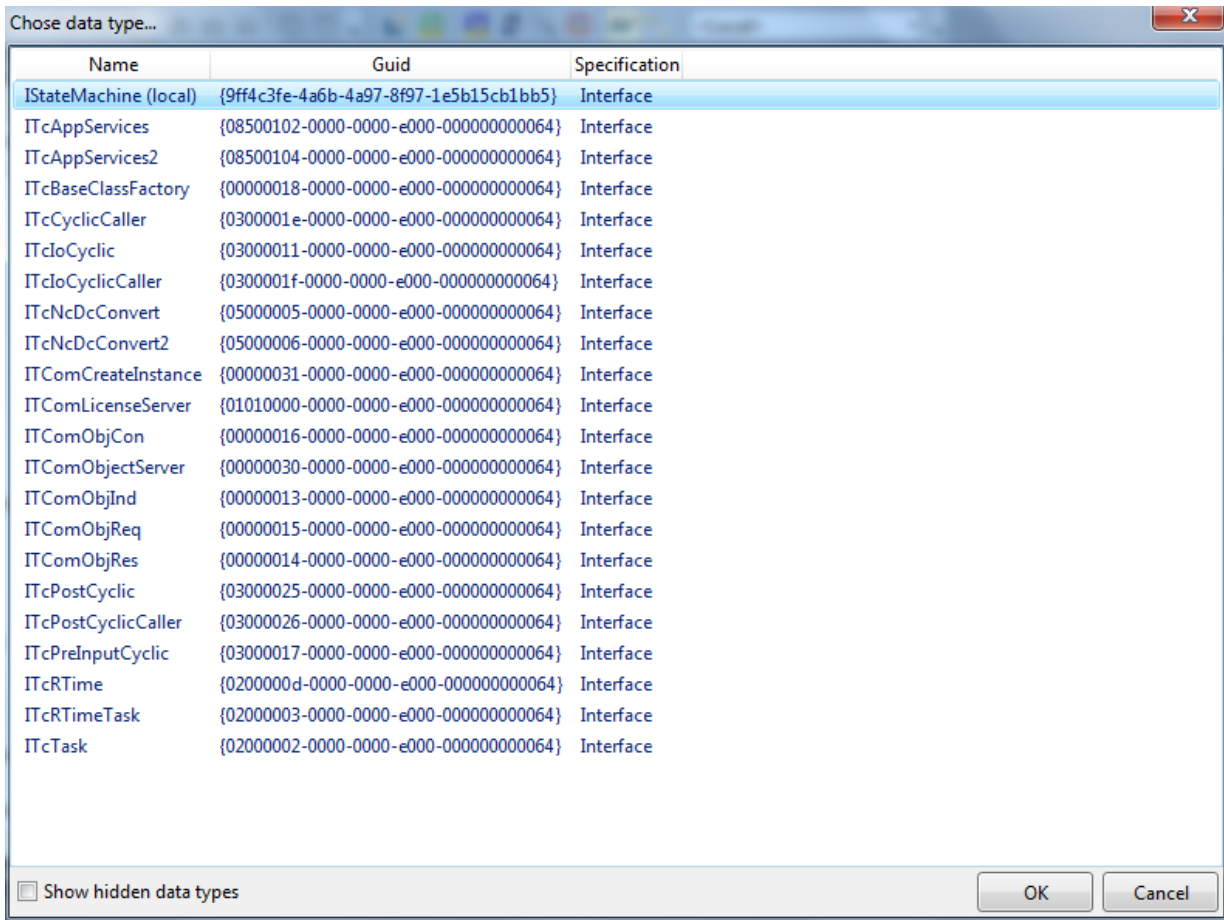
**Step 5: add a new interface to the module**

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.

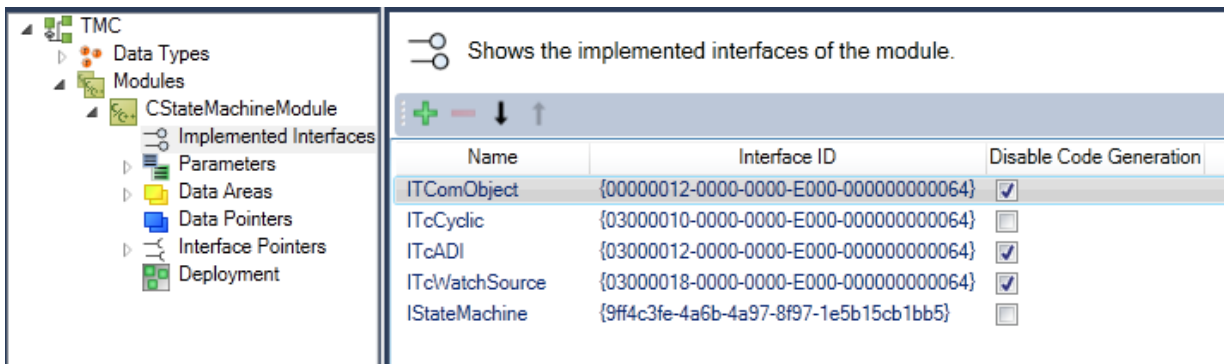
Name	Interface ID	Disable Code Generation
ITComObject	{00000012-0000-0000-E000-000000000064}	<input checked="" type="checkbox"/>
ITcCyclic	{03000010-0000-0000-E000-000000000064}	<input type="checkbox"/>
ITcADI	{03000012-0000-0000-E000-000000000064}	<input checked="" type="checkbox"/>
ITcWatchSource	{03000018-0000-0000-E000-000000000064}	<input checked="" type="checkbox"/>



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

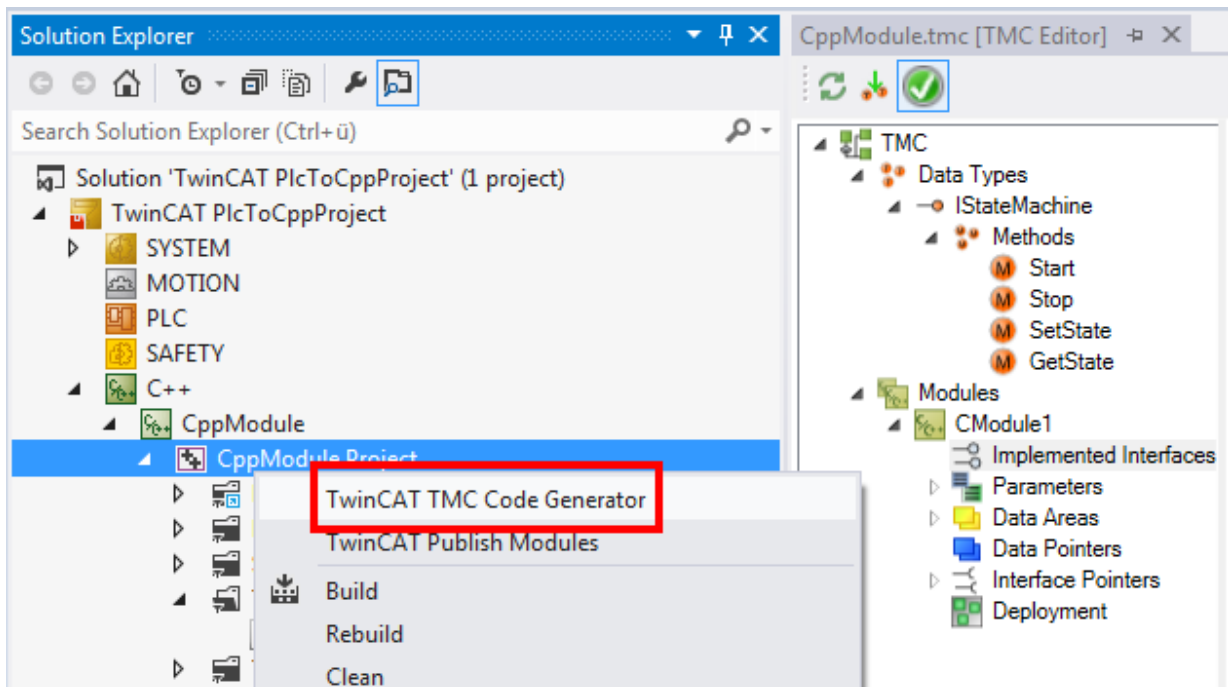


⇒ The new interface IStateMachine is part of the module description.



**Step 6: Start TwinCAT TMC code generator**

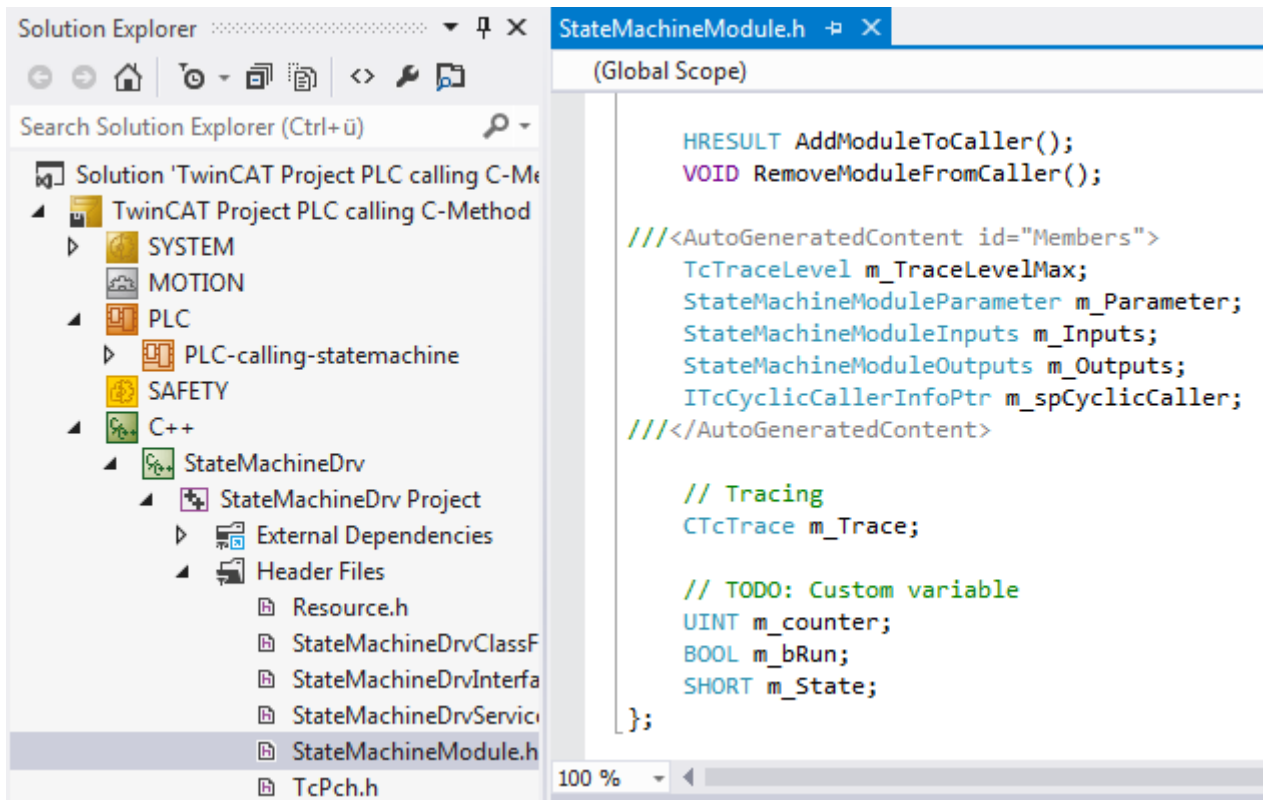
1. In order to generate the C/C++ code on the basis of this module, right-click in the C/C++ project and then select the **TwinCAT TMC Code Generator**.



- ⇒ The module StateMachineModule.cpp now contains the new interfaces
- CModule1: Start()
  - CModule1: Stop()
  - CModule1: SetState(SHORT State)
  - CModule1: GetState(SHORT\* pState).

**Step 7: implementation of the member variables and the constructor**

Add the member variables to the header file StateMachineModule.h.



### Step 8: implementation of 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;
}

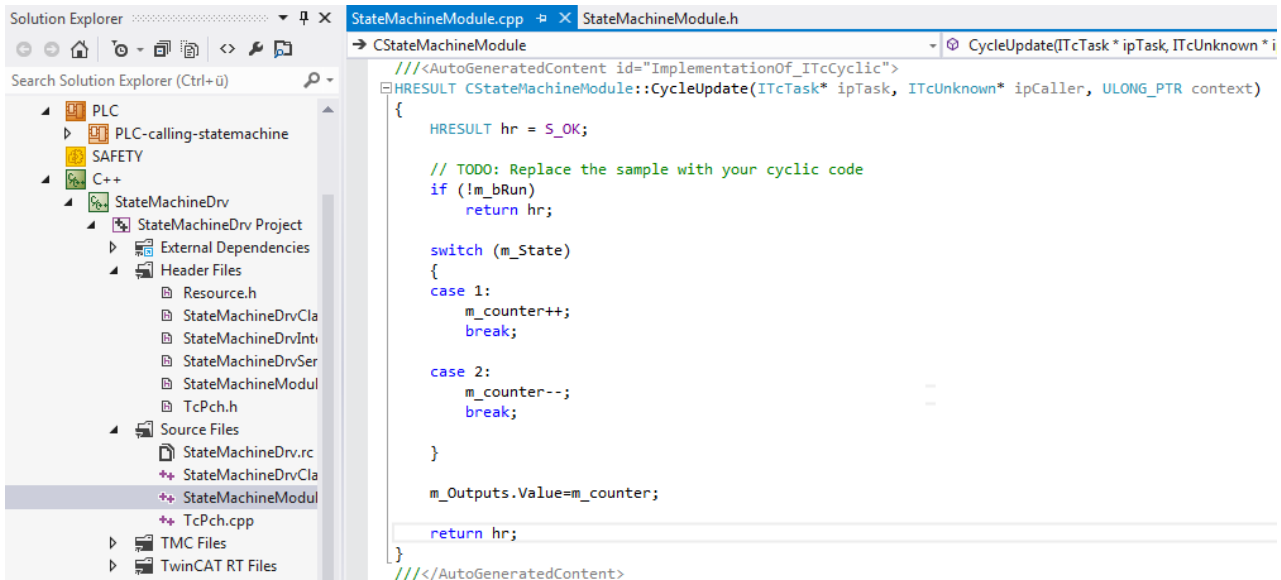
HRESULT CModule1::SetState(SHORT State)
{
    HRESULT hr = S_OK;
    m_State = State;
    return hr;
}

HRESULT CModule1::GetState(SHORT* pState)
{
    HRESULT hr = S_OK;
    *pState = m_State;
    return hr;
}
///</AutoGeneratedContent>
```

### Step 9: implementation of a cyclic update

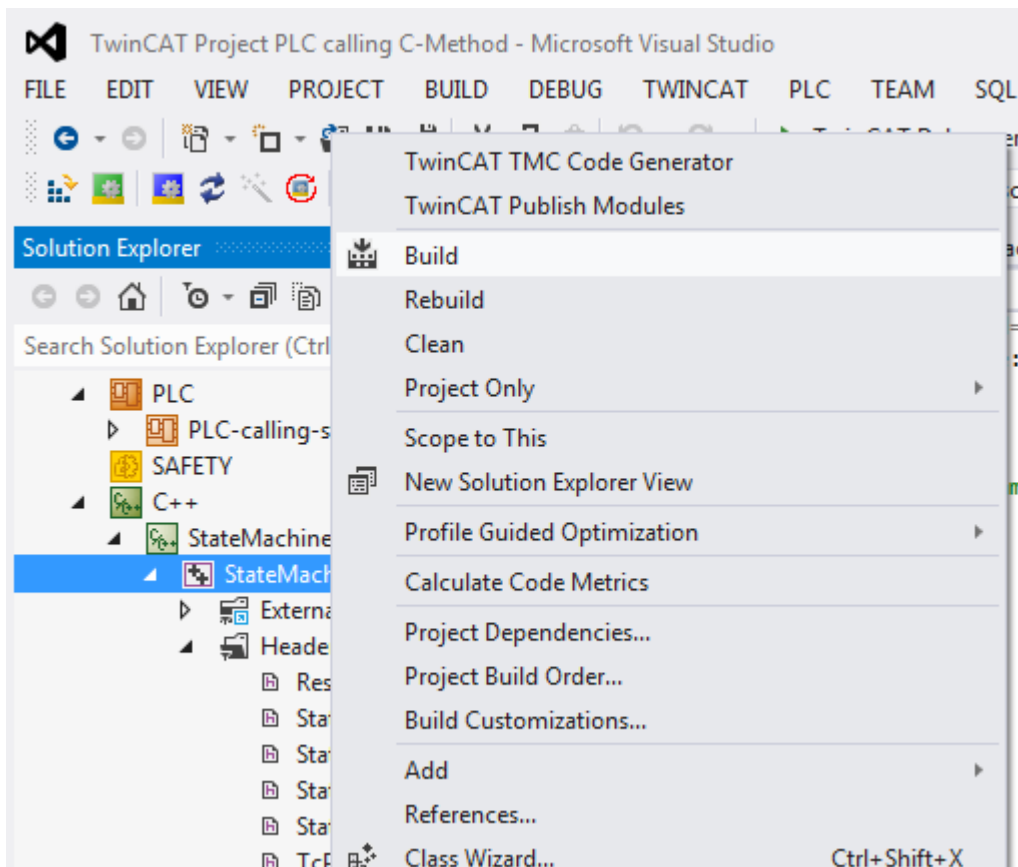
The C++ module instance is cyclically called, even if the internal state machine is in Stop mode.

- 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 quit.
- 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.

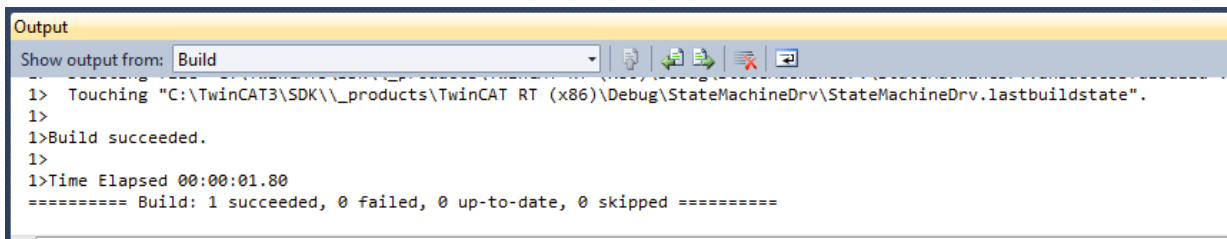


**Step 10: compilation of code**

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:



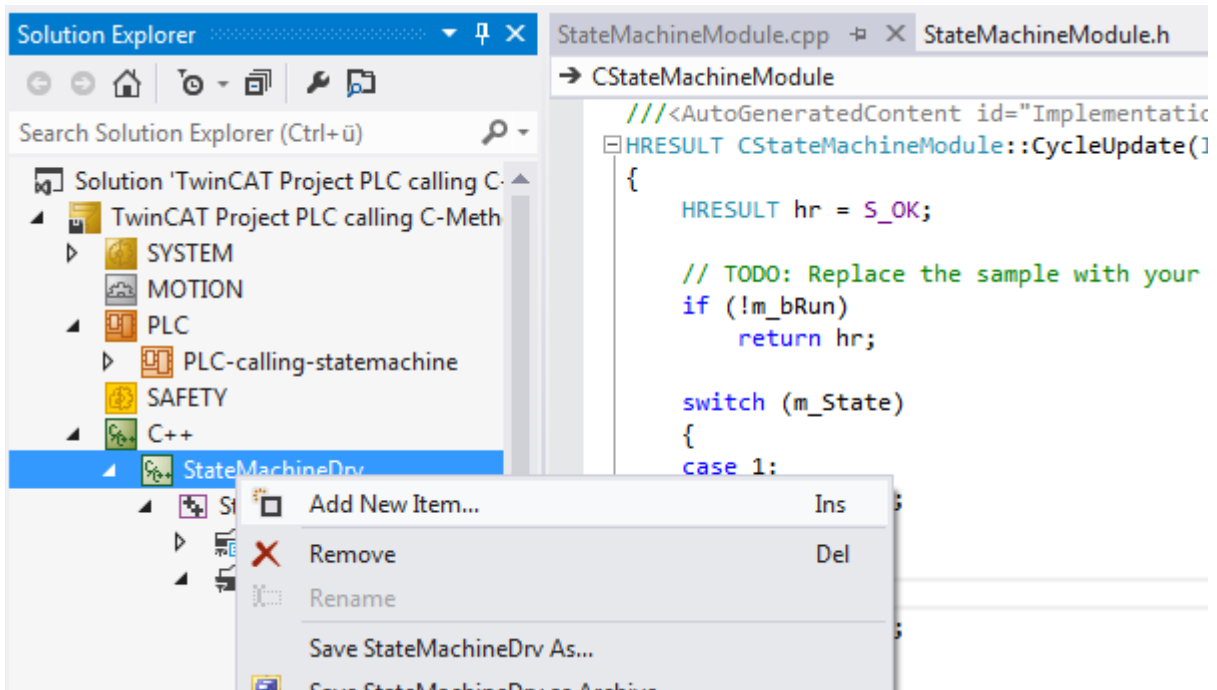
```

Output
Show output from: Build
1> Touching "C:\TwinCAT3\SDK\*_products\TwinCAT RT (x86)\Debug\StateMachineDrv\StateMachineDrv.lastbuildstate".
1>
1>Build succeeded.
1>
1>Time Elapsed 00:00:01.80
===== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped =====

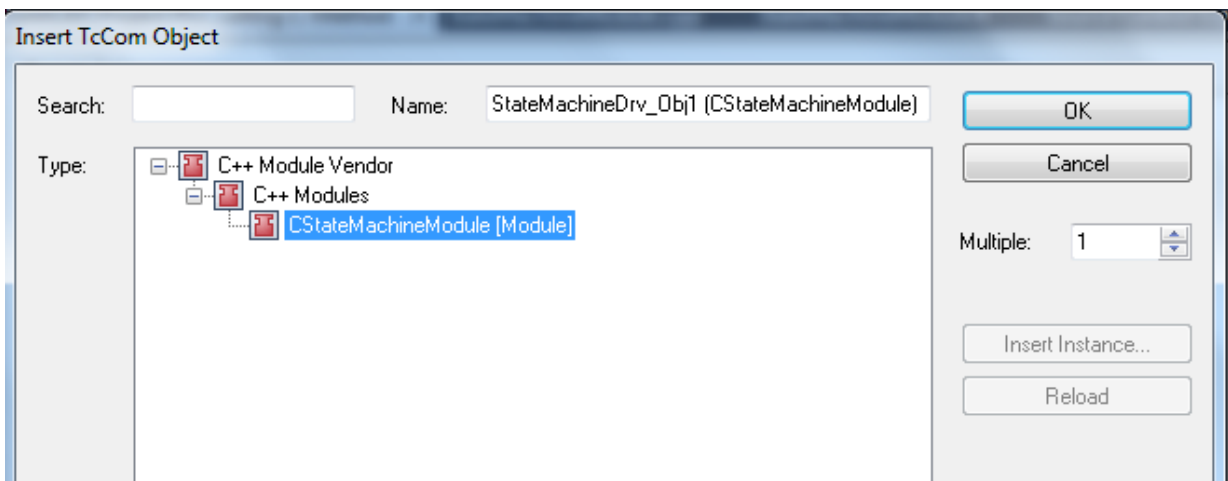
```

### Step 11: creating an instance of the C++ module

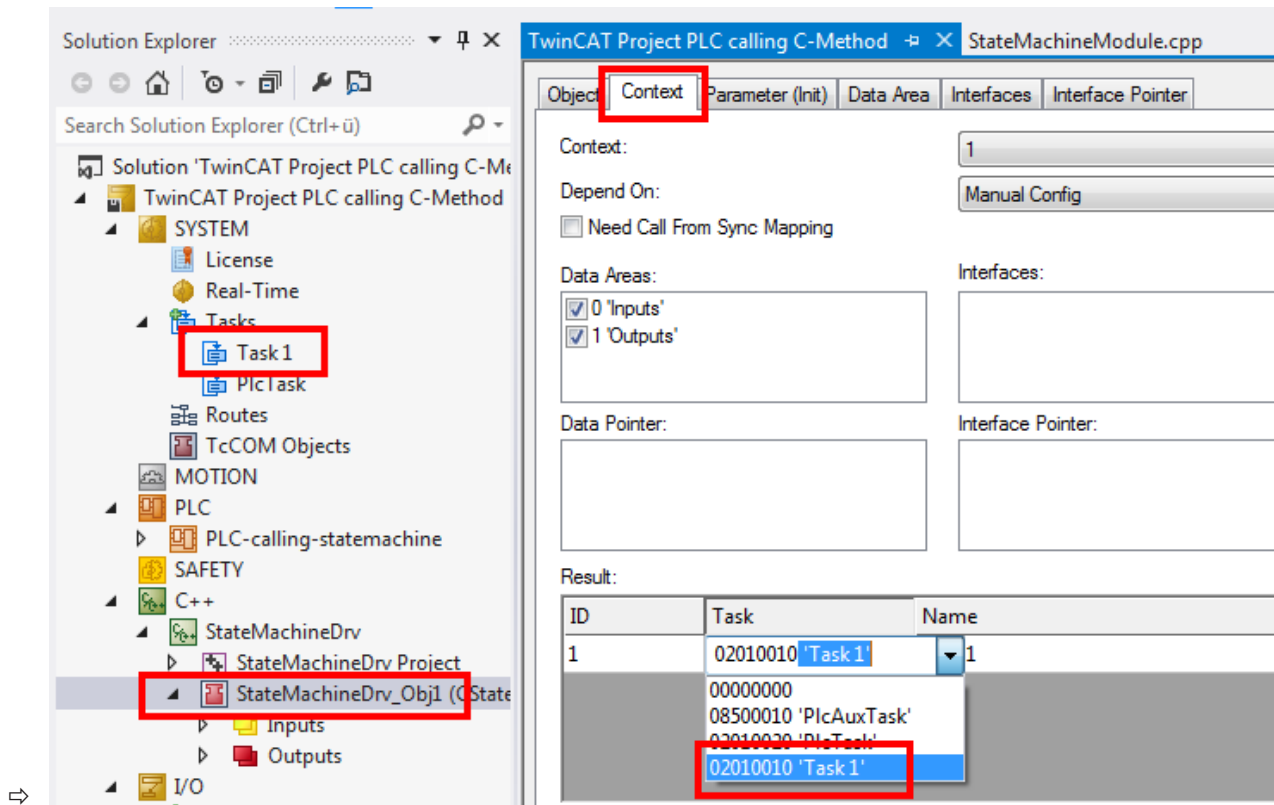
1. Right-click on the C++ project and select **Add New Item...** to create a new module instance.



2. Select the module that is to be added as a new instance – in this case `CStateMachineModule`.

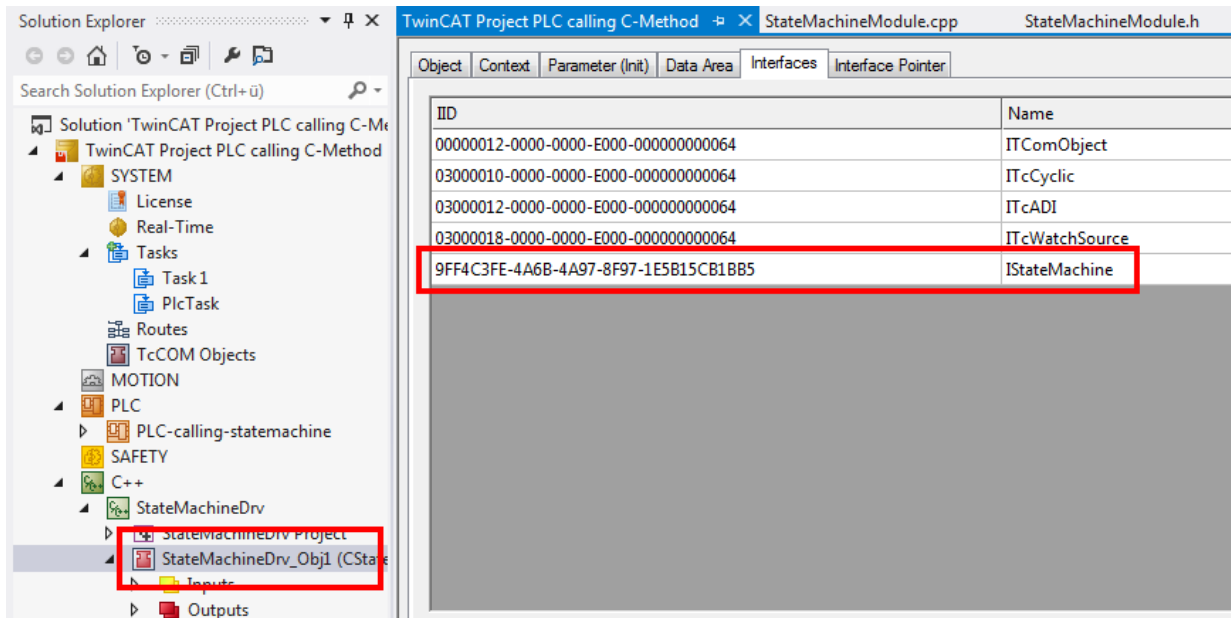


3. Assign the instance to a task:



**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



**15.10.2 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 available interfaces](#) [▶ 252]
- Step 2: [create a new PLC project](#) [▶ 253]

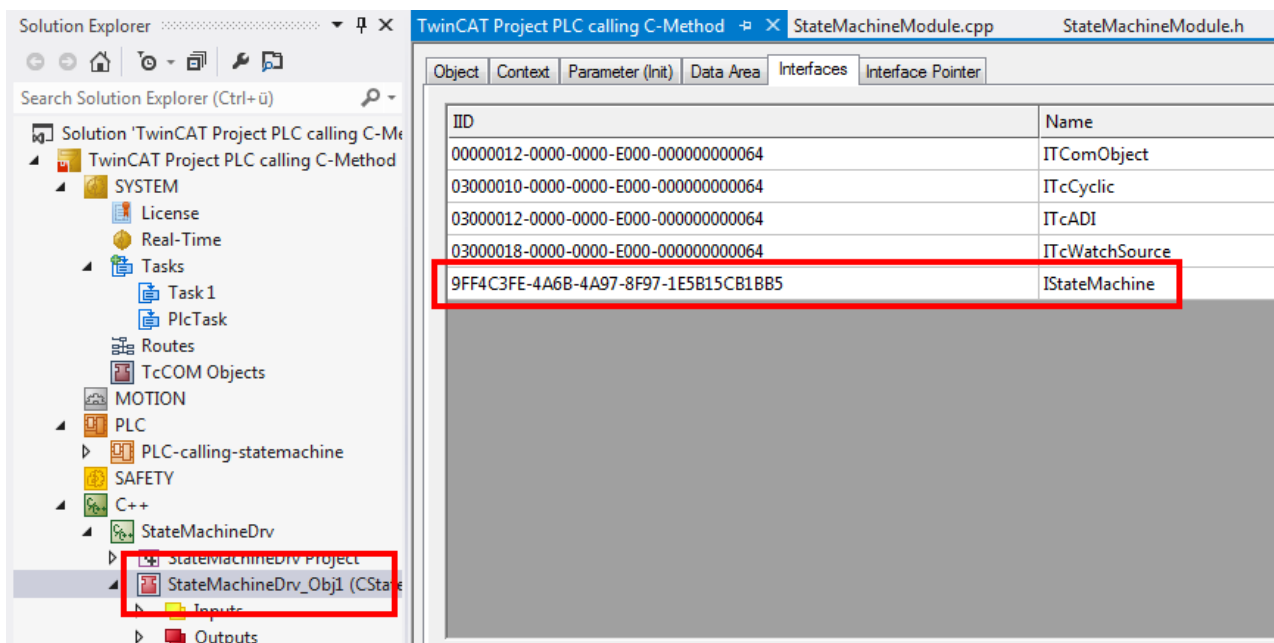
- Step 3: [add a new FB-StateMachine \[▶ 253\]](#) (which acts as the proxy that calls the C++ module methods)
- Step 4: [clean up the function block interface \[▶ 256\]](#)
- Step 5: [add FB methods "FB\\_init" and "exit" \[▶ 257\]](#)
- Step 6: [implement FB methods \[▶ 258\]](#)
- Step 7: [call FB-StateMachine in the PLC \[▶ 261\]](#)
- Step 8: [compile PLC code \[▶ 262\]](#)
- Step 9: [link PLC FB with C++ instance \[▶ 262\]](#)
- Step 10: [observe the execution of the two modules, PLC and C++ \[▶ 263\]](#)

### Step 1: check available interfaces

Option 1:

1. navigate to C++ module instance
2. Select the **Interfaces** tab.

⇒ The **IStateMachine** interface is in the list with its specific IID (Interface ID)

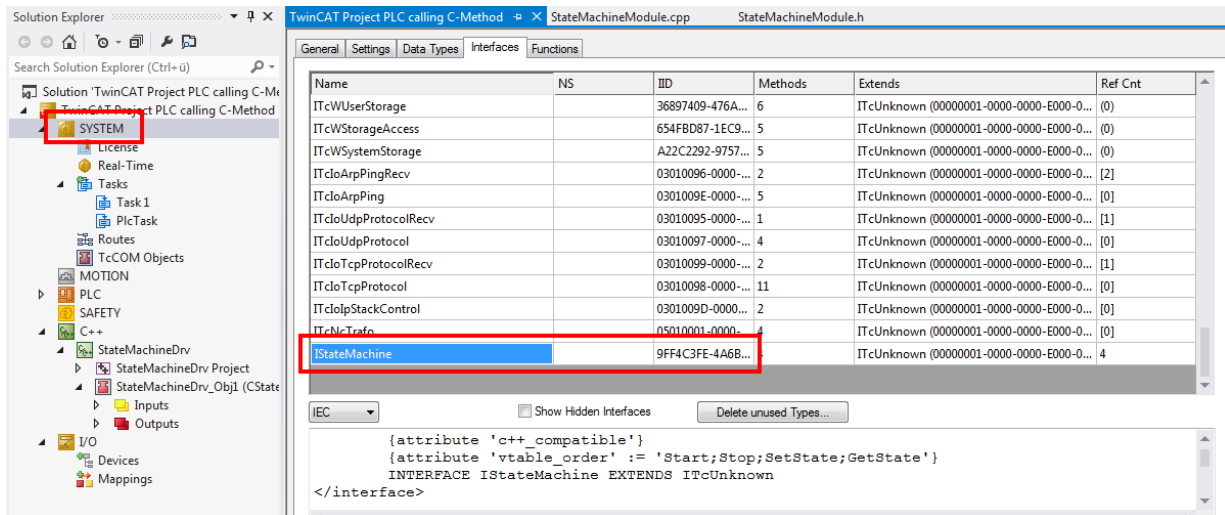


Option 2:

1. navigate to **System**
2. Select the **Interfaces** tab.



⇒ The **IStateMachine** interface is in the list with its specific IID (Interface ID)

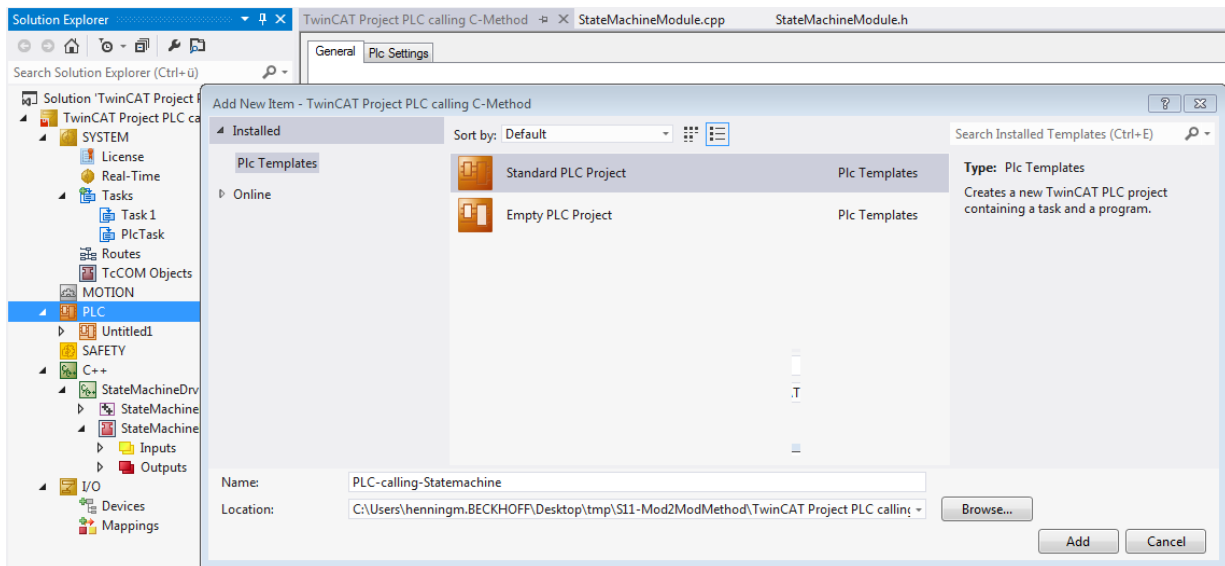


The lower section shows the stored code in different programming languages.

**Step 2: Creating a new PLC project**

A "standard PLC project" called "PLC-calling state machine" is created.

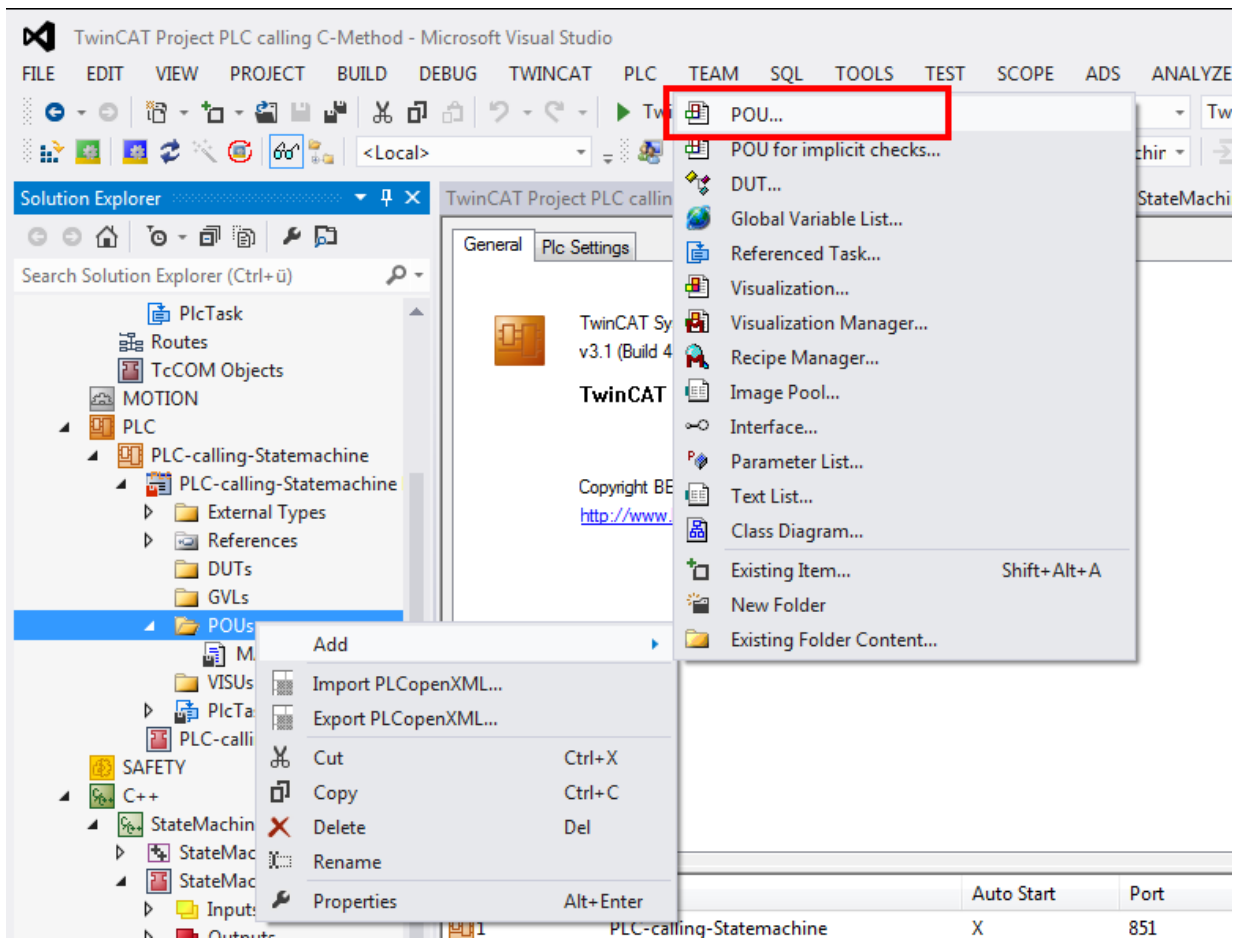
1. Right-click on the PLC node.
2. Select **Standard PLC Project**.
3. Adapt the name.



⇒ The project has been successfully created.

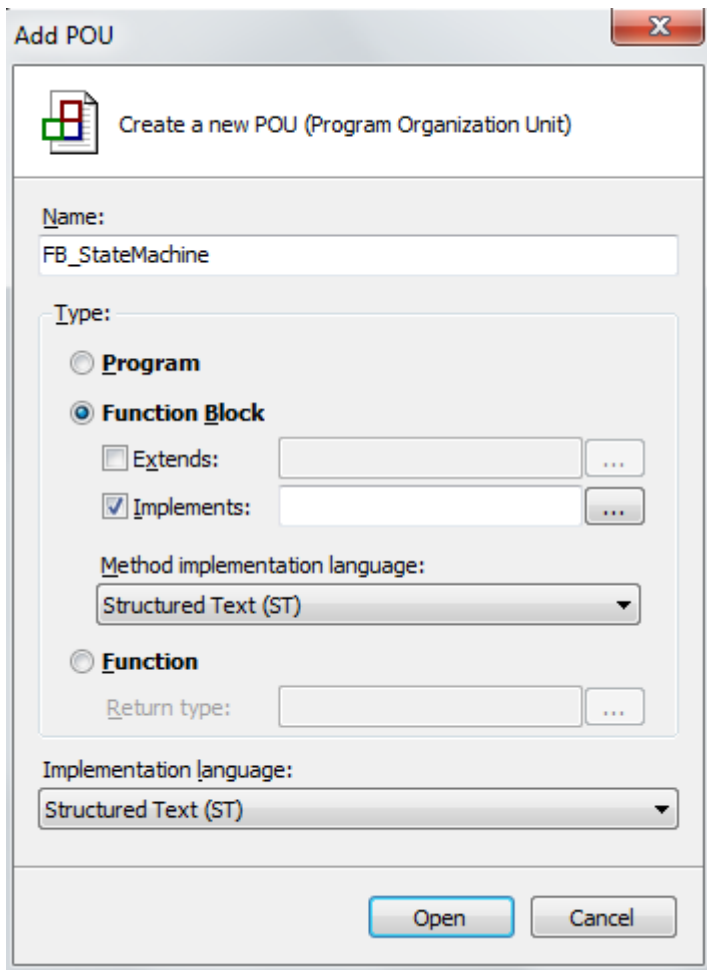
**Step 3: Add a function block (FB) (which serves as the proxy for calling the C++ module methods)**

1. Right-click on POU's

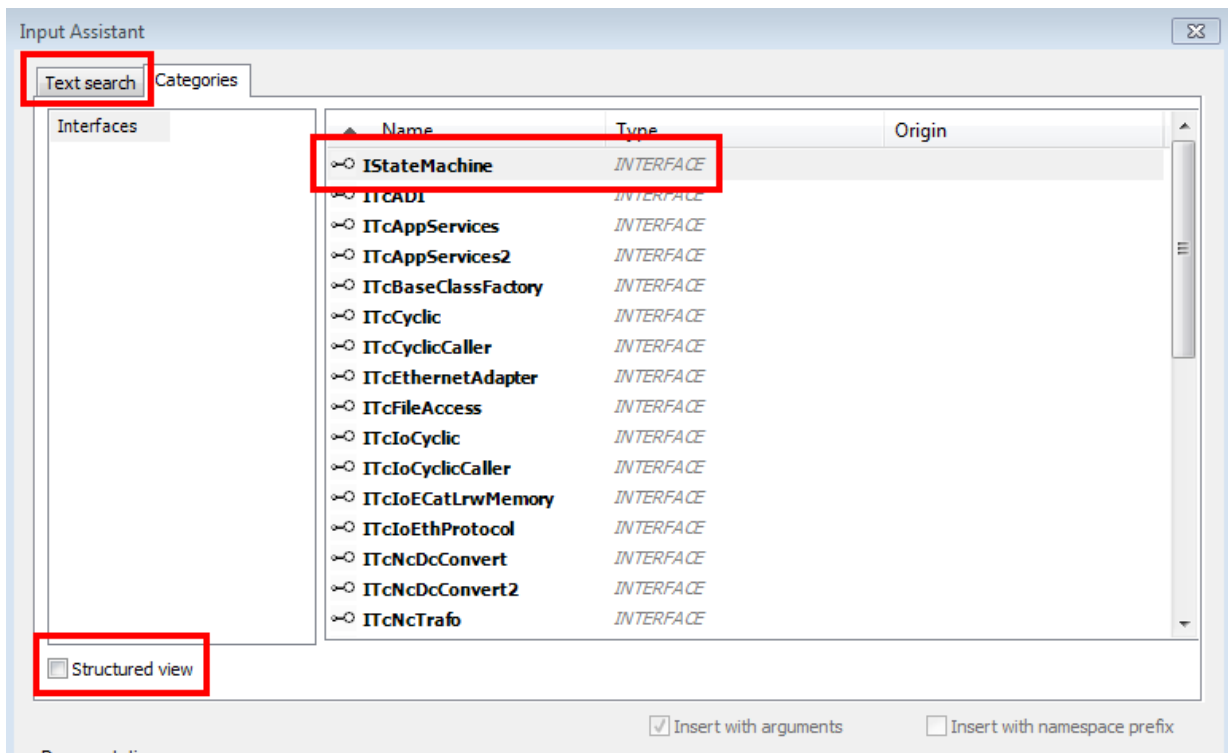
2. Select **Add->POU....**

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.



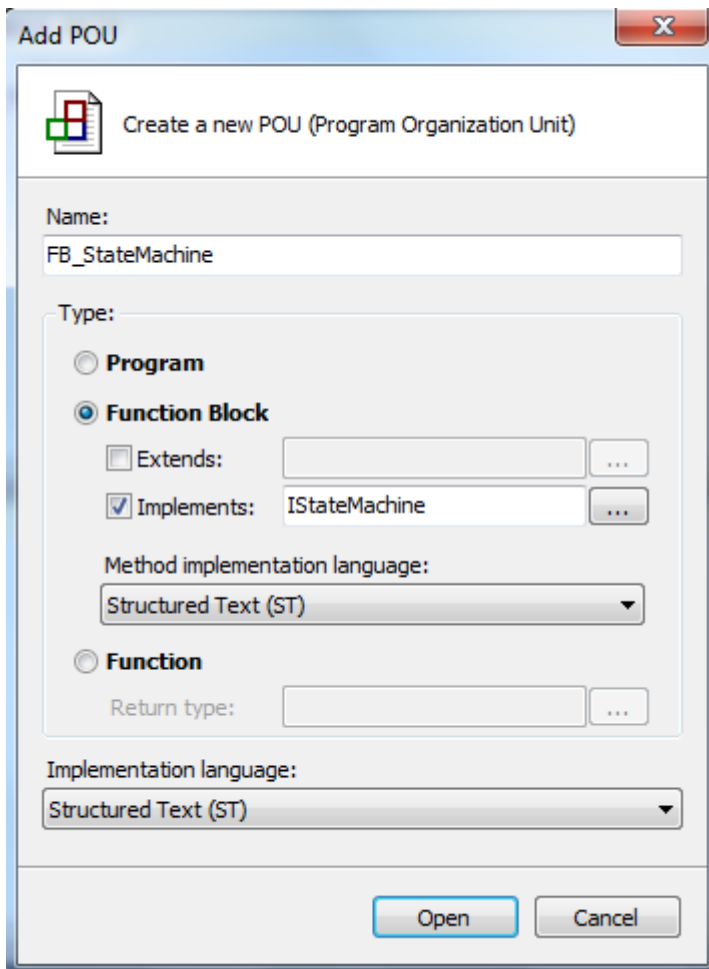
5. Select the interface either via the **Text Search** tab or the **Categories** tab by deselecting **Structured View**:



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: Customizing the function block interface

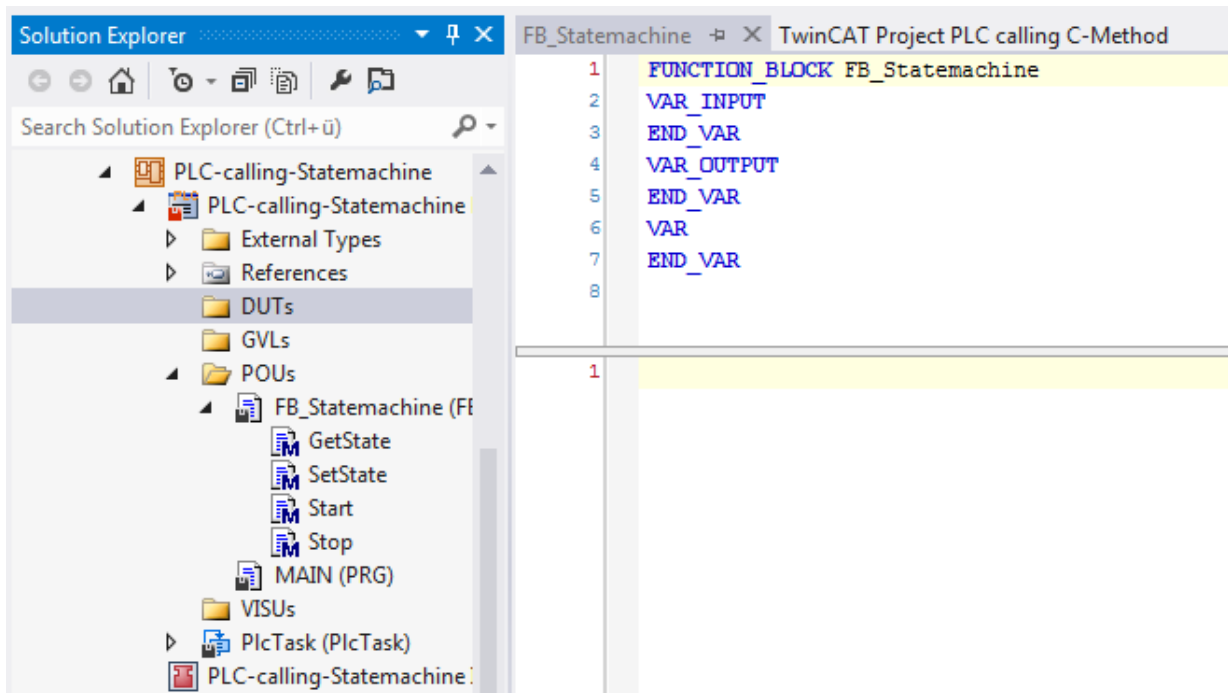
As a result of creating an FB that implements the IStateMachine interface, the wizard will create an FB with corresponding methods.

The FB\_StateMachine makes 4 methods available:

- GetState
- SetState
- Start
- Stop

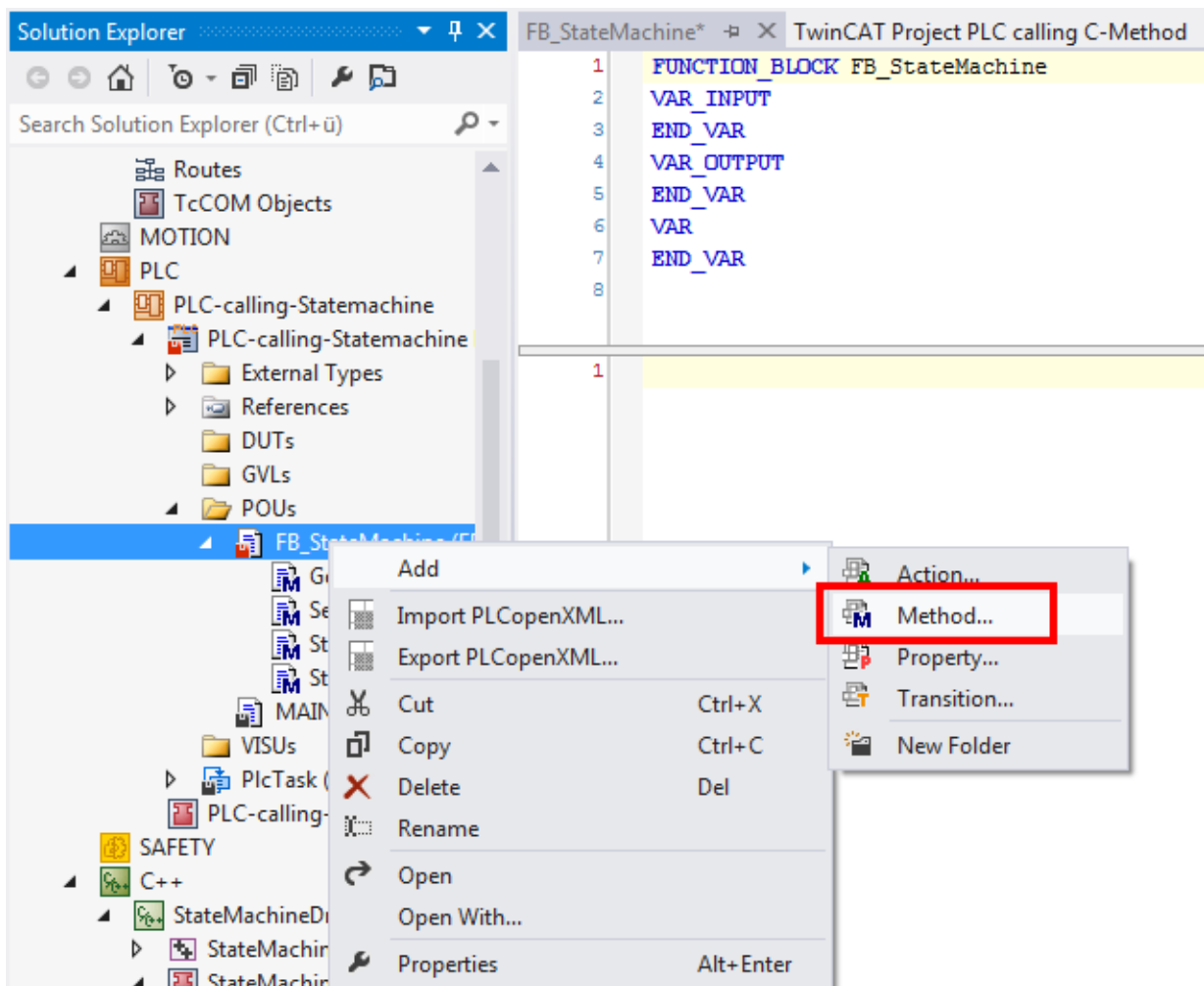
1. Delete Implements IStateMachine. Since the function block should act as proxy, it does not implement the interface itself. Therefore, it can be deleted.
2. Delete the methods TcAddRef, TcQueryInterface and TcRelease. They are not required for a proxy function block.

⇒ The result is:

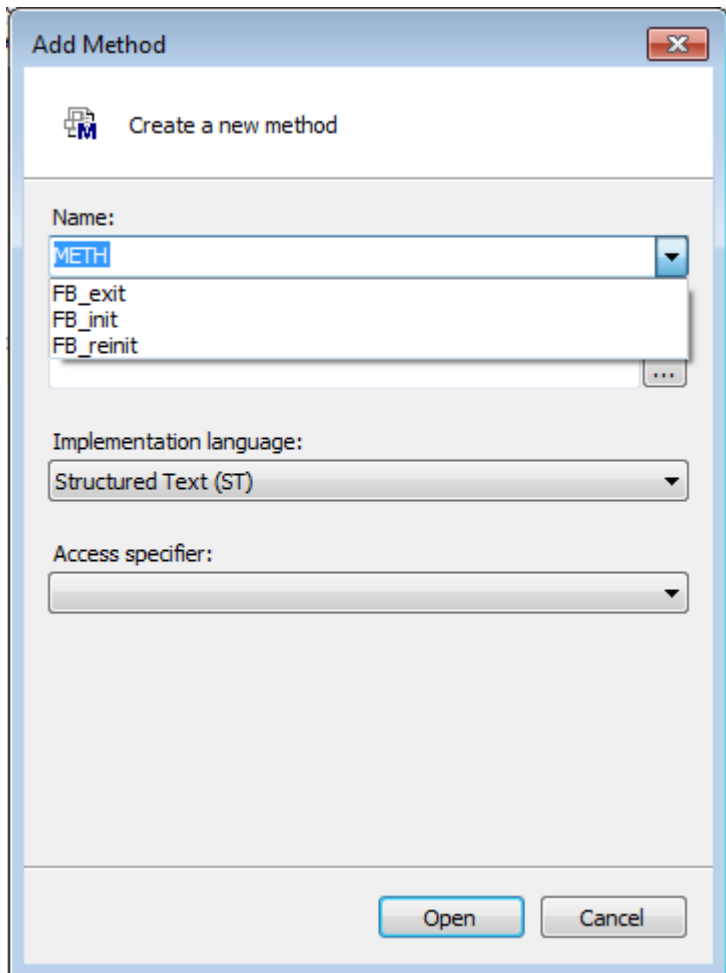


**Step 5: Add FB methods "FB\_init" (constructor) and "FB\_exit" (destructor)**

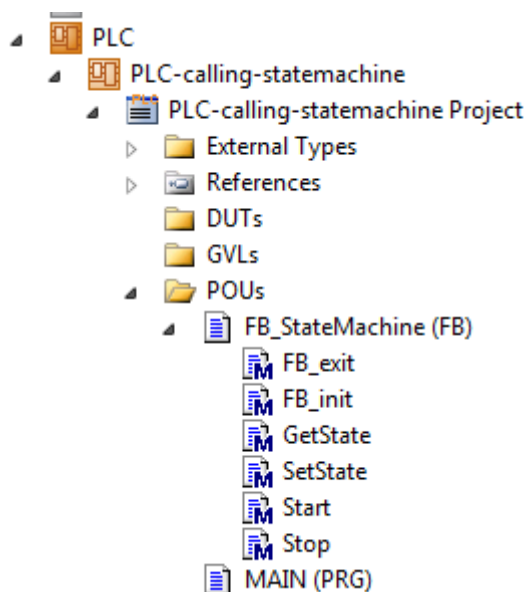
1. Right-click on **FB\_StateMachine** in the tree and select **Add / Method...**



- Add the methods `FB_exit` and `FB_init` - both with Structured Text (ST) as the implementation language. They are available as predefined name.



- 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.



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*
1  METHOD GetState : HRESULT
2  VAR_INPUT
3      pState : POINTER TO INT;
4  END_VAR
5
1  IF (ipStateMachine <> 0) THEN
2      GetState:= ipStateMachine.GetState(pState);
3  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*
1  METHOD SetState : HRESULT
2  VAR_INPUT
3      State : INT;
4  END_VAR
5
1  IF (ipStateMachine <> 0) THEN
2      SetState:= ipStateMachine.SetState(State);
3  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*
1  METHOD Start : HRESULT
2
1  IF (ipStateMachine <> 0) THEN
2      Start:= ipStateMachine.Start();
3  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*
1  METHOD Stop : HRESULT
2
1  IF (ipStateMachine <> 0) THEN
2      Stop:= ipStateMachine.Stop();
3  END_IF

```



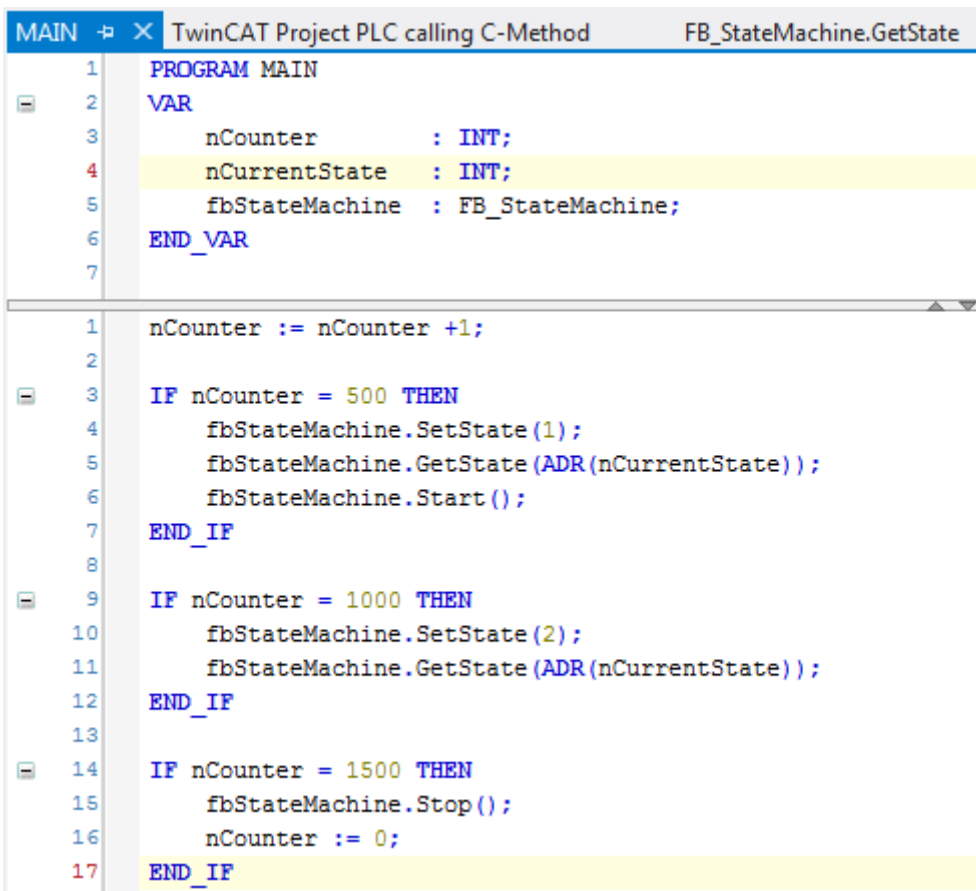
⇒ The implementation of the "FB\_StateMachine", which acts as the proxy for calling the C++ module instance, is completed.

### Step 7: call FB in the PLC

The FB\_StateMachine is now called in the POU MAIN.

This simple sample acts as follows:

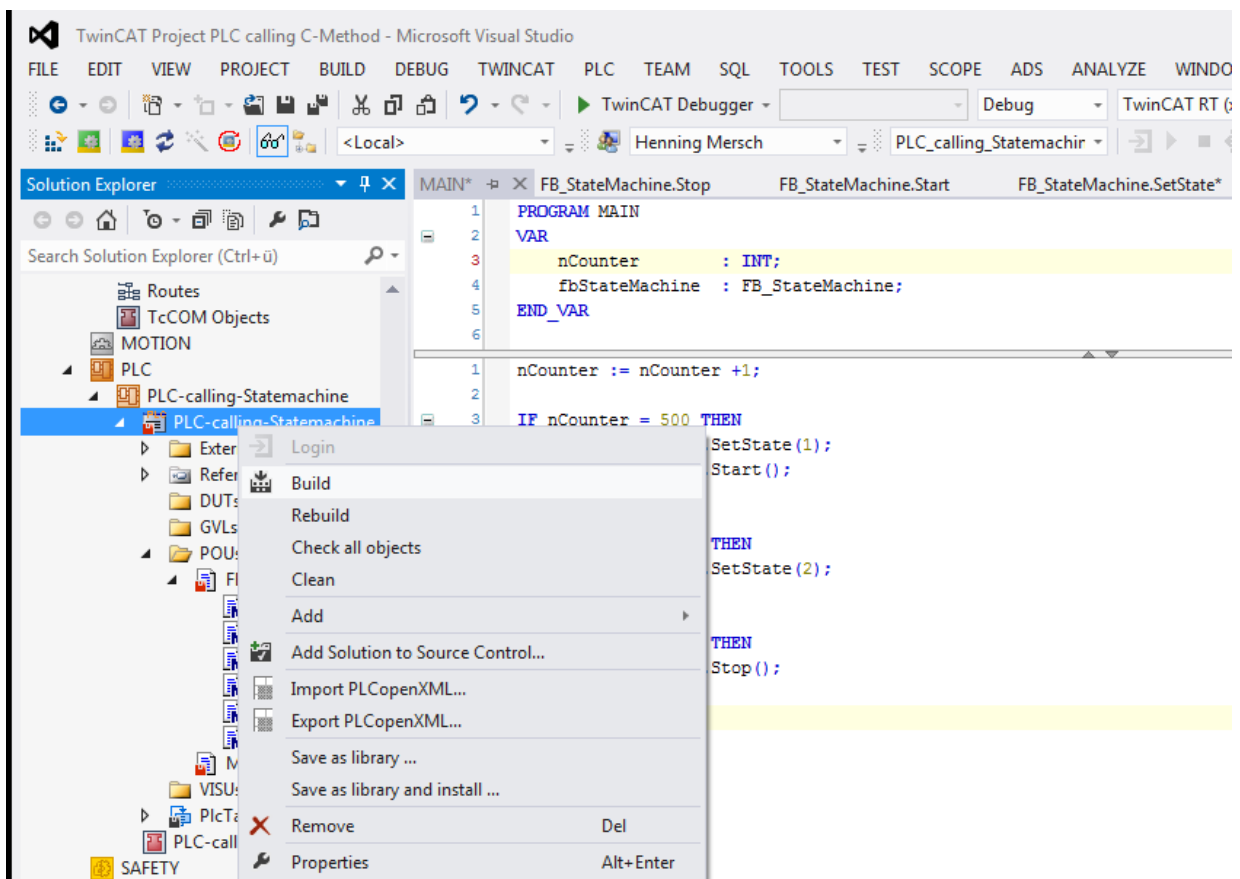
- Cyclic incrementation of a PLC counter nCounter
- If nCounter = 500, the C++ StateMachine is started with the state "1" in order to increment its internal C++ counter. Then read the state of C++ using GetState().
- If nCounter = 1000, the C++ state machine is set to the state "2" in order to decrement its internal C++ counter. Then read the state of C++ using GetState().
- If nCounter = 1500, the C++ StateMachine is stopped. The PLC counter is also set to 0, so that everything starts again from the beginning.



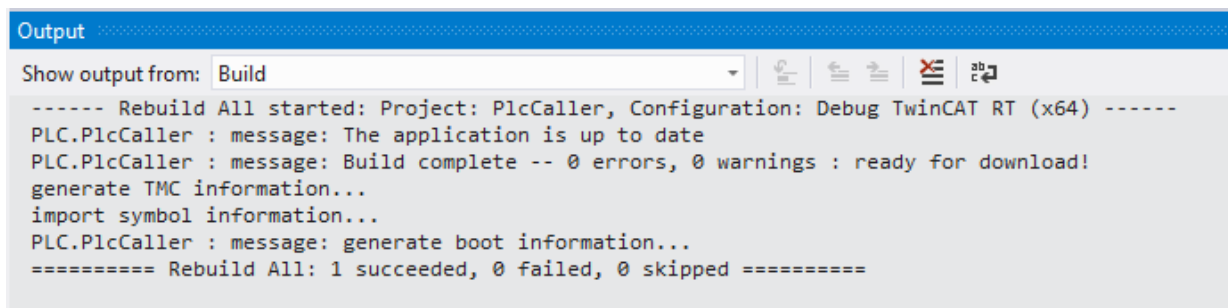
```
MAIN  ▸ X  TwinCAT Project PLC calling C-Method  FB_StateMachine.GetState
1  PROGRAM MAIN
2  VAR
3      nCounter      : INT;
4      nCurrentState : INT;
5      fbStateMachine : FB_StateMachine;
6  END_VAR
7
8
9  nCounter := nCounter + 1;
10
11 IF nCounter = 500 THEN
12     fbStateMachine.SetState(1);
13     fbStateMachine.GetState(ADR(nCurrentState));
14     fbStateMachine.Start();
15 END_IF
16
17 IF nCounter = 1000 THEN
18     fbStateMachine.SetState(2);
19     fbStateMachine.GetState(ADR(nCurrentState));
20 END_IF
21
22 IF nCounter = 1500 THEN
23     fbStateMachine.Stop();
24     nCounter := 0;
25 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".



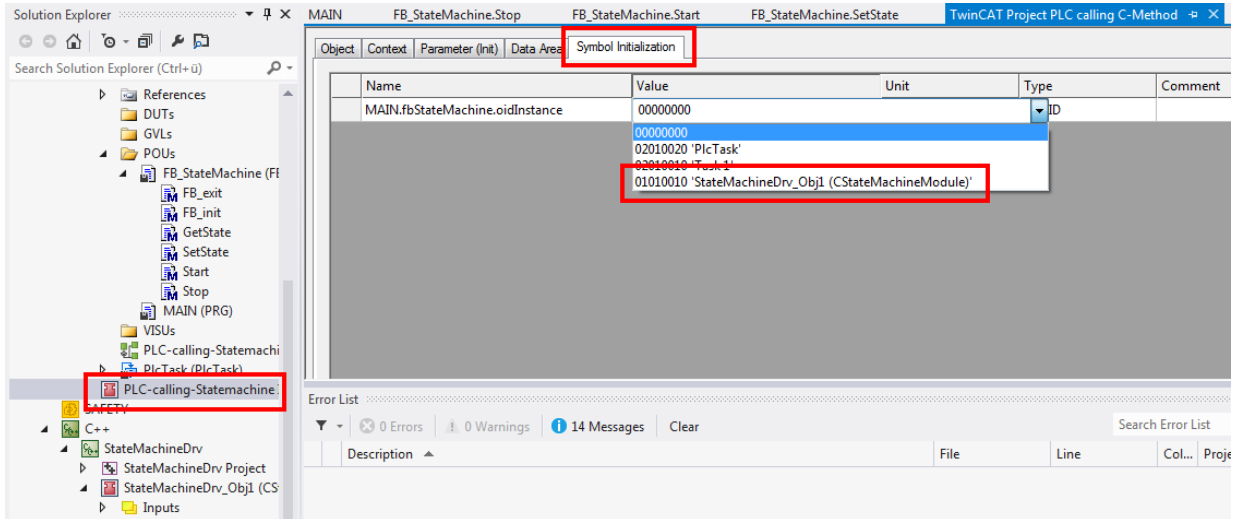
## Step 9: link PLC FB with C++ instance

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 example 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.

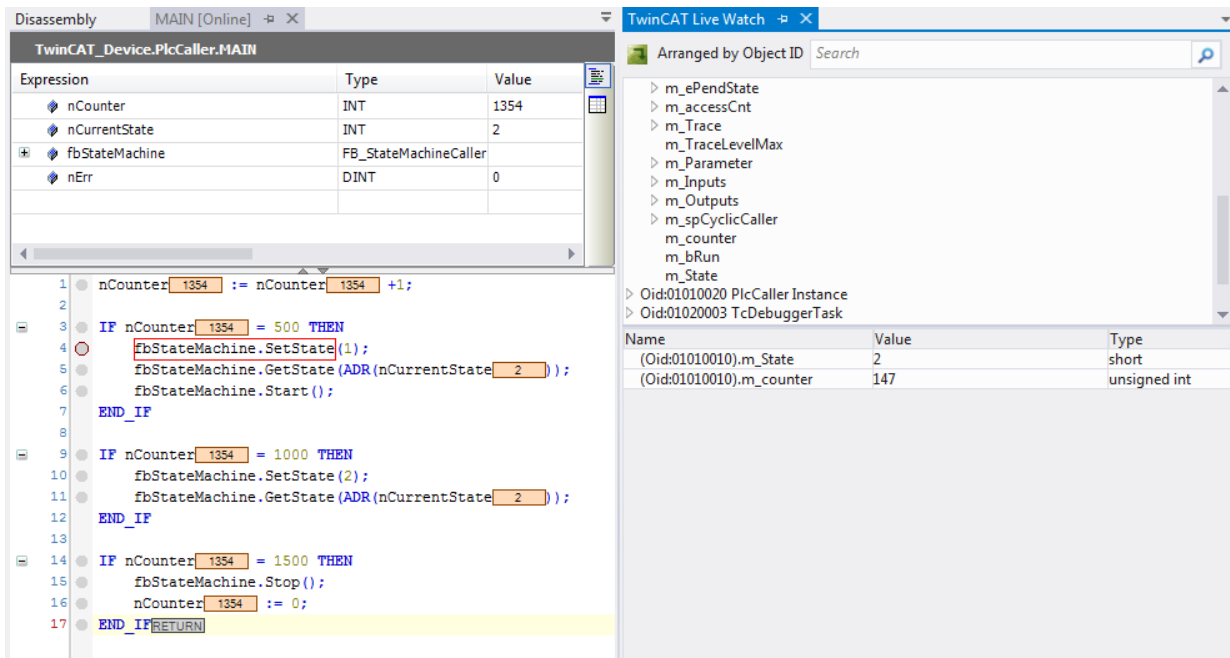


⇒ PLC and C++ module are connected to each other.

**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 [▶ 61] and follow the steps in the quick start [▶ 50] in order to start the debugging (right-hand side of the following illustration).




## 15.11 Sample11a: Module communication: C module calls a method of another C module

This article describes how TC3 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.

1. Unpack the downloaded ZIP file.
2. Open the zip file that it contains in TwinCAT 3 by clicking on "Open Project...".
3. Select your target system.
4. Build the sample on your local machine (e.g. Build->Build Solution).
5. Activate the configuration by clicking on  .  
⇒ 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 "DataWriteCounterModulo" 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.

## 15.12 Sample12: module communication: Using IO mapping

This article describes how two TC3 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.

1. Unpack the downloaded ZIP file.
2. Open the zip file that it contains in TwinCAT 3 by clicking on "Open Project...".
3. Select your target system.
4. Build the sample on your local machine (e.g. Build->Build Solution).
5. Activate the configuration by clicking on  .  
⇒ 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 to (linked with) the input "Value" of the

instance "Back".

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 \[► 63\]](#))

Ultimately, the user can specify the input "Value" of the "Front" instance and monitor the output "Value" of "Back".


## 15.13 Sample13: Module communication: C-module calls PLC methods

This article describes how a TwinCAT C++ module calls a methods of a PLC function block via the TcCOM interface.

### Download

System requirements: TwinCAT 3.1 Build 4020

Here you can access the [source code](#) for this sample.

1. Unpack the downloaded ZIP file.
  2. Open the zip file that it contains in TwinCAT 3 by clicking on "Open Project...".
  3. Select your target system.
  4. Build the sample on your local machine (e.g. Build->Build Solution).
  5. Activate the configuration by clicking on  .
- ⇒ The sample is ready for operation.

### Description

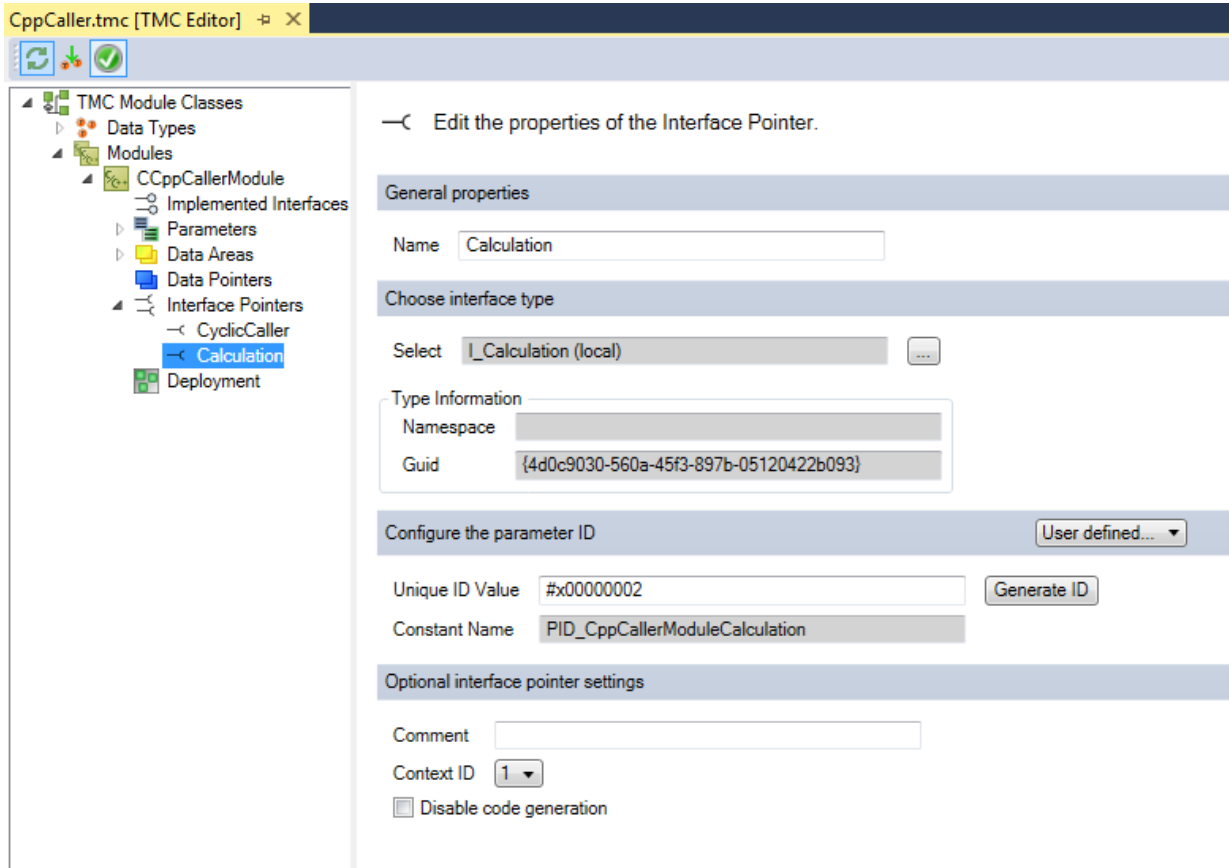
This sample provides for communication from a C++ module to a function block of a PLC by means of method call. To this end a TcCOM interface is defined that is offered by the PLC and used by the C++ module.

The PLC page as a provider in the process corresponds to the corresponding project of the [TcCOM Sample 01 \[► 281\]](#), where an PLC is considered after PLC communication. Here a Caller is now provided in C++, which uses the same interface.

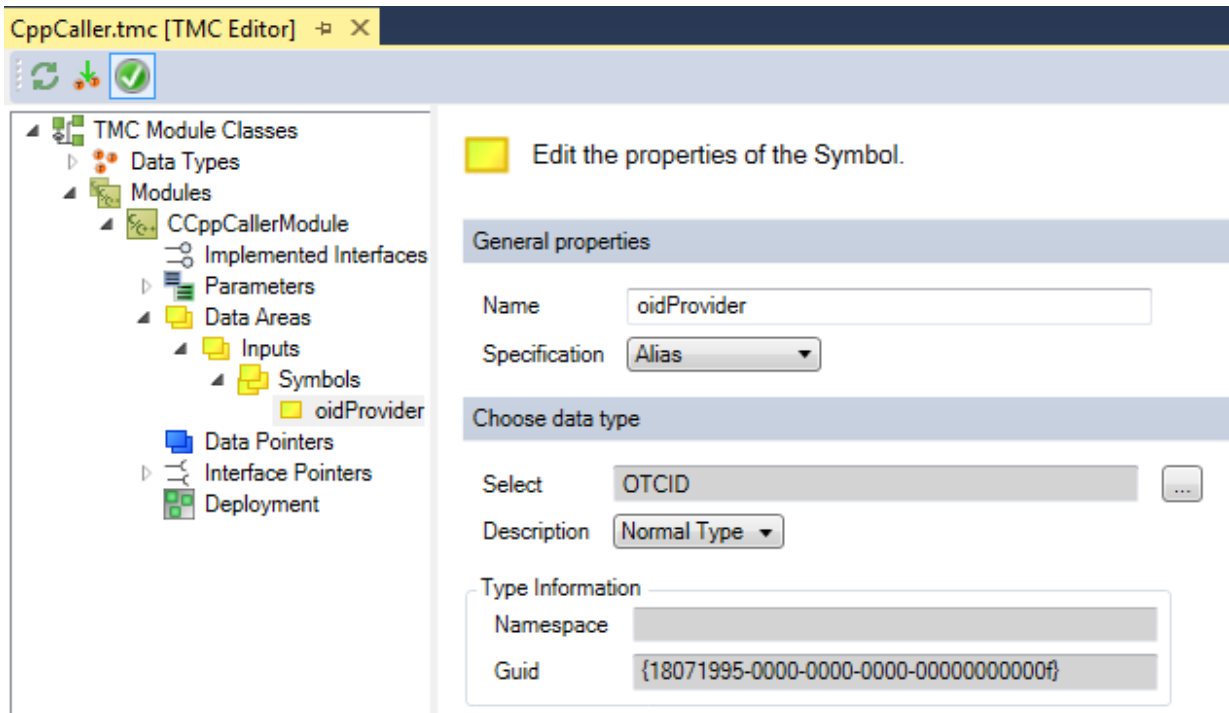
The PLC page adopted by [TcCOM Sample 01 \[► 281\]](#). The function block registered there as TcCOM module offers the object ID allocated to it as an output variable. It is the C++ module's task to make the offered interface of this function block accessible.

- ✓ 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 `Calculation`). 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.

```

CppClassModuleInputs m_Inputs;
ITcCyclicCallerInfoPtr m_spCyclicCaller;
I_CalculationPtr m_spCalculation;
///

```

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.

```

HRESULT CCppCallerModule::SetObjStateOS()
{
    m_Trace.Log(tlVerbose, FENTERA);
    HRESULT hr = S_OK;

    RemoveModuleFromCaller();

    // TODO: Add any additional deinitialization
    m_spCalculation = NULL;

    m_Trace.Log(tlVerbose, FLEAVEA "hr=0x%08x", hr);
    return hr;
}

// ...
HRESULT CCppCallerModule::SetObjStateSP() { ... }

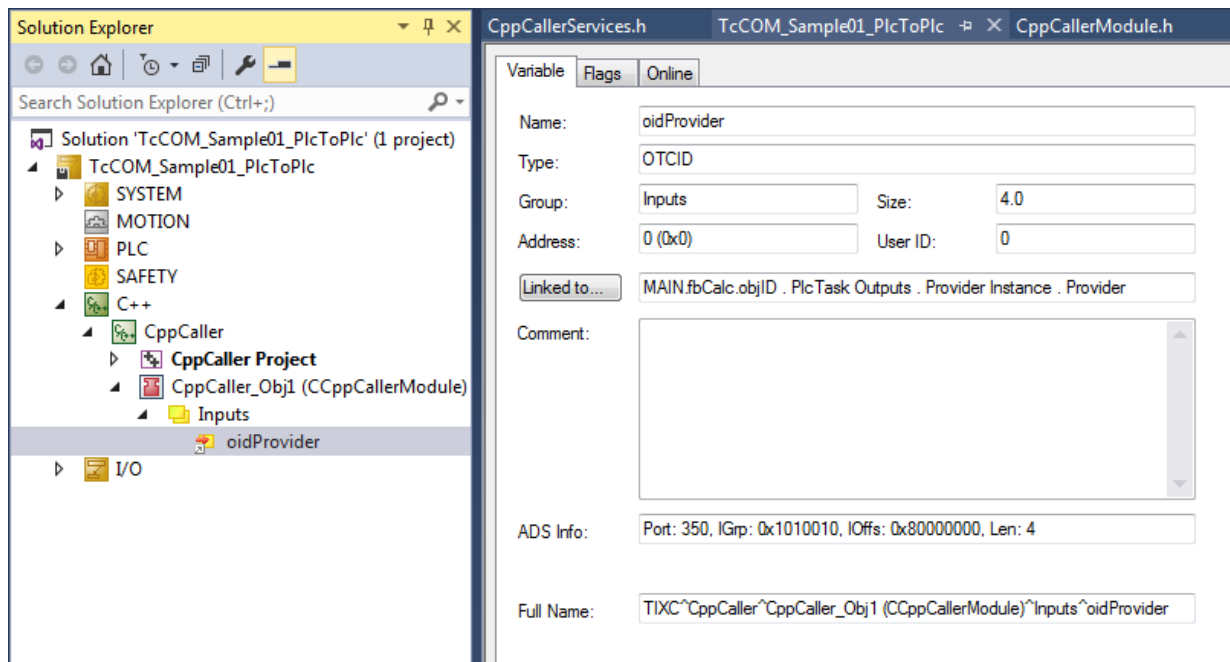
///

```

In addition, as can be seen above, the interface pointer is cleared when the program shuts down. This happens in the SetObjStateOS method.

- Now build the C++ project.
- 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.

## 15.14 Sample19: Synchronous File Access

This article describes how to implement a TC3 C++ module which accesses files on the hard disk within the startup of a module, thus within the real-time environment.

### Download

Get the [source code](#) for this sample

1. Unpack the downloaded ZIP file.
2. Open the zip file that it contains in TwinCAT 3 by clicking on "Open Project...".
3. Select your target system.
4. Build the sample on your local machine (e.g. Build->Build Solution).
5. Activate the configuration by clicking on .

⇒ The sample is ready for operation.

The whole source code, which is not automatically generated by the wizard, is identified with the comment start flag `///  
sample code` and the comment end flag `///  
sample code end`.

In this way you can search for these strings in the files, in order 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 window of TwinCAT 3. Then, a file `"%TC_TARGETPATH%DefaultConfig.xml"` (normally `"C:\TwinCAT\3.x\Target\DefaultConfig.xml"`) is copied to `"%TC_TARGETPATH%DefaultConfig.xml.bak"`.



For access to the log entries, see tab "Error List" of the TwinCAT 3 output window. The amount of information can be set by changing the variable TraceLevelMax in the instance "TcFileTestDrv\_obj1" in tab "Parameter (Init)".


## 15.15 Sample20: FileIO-Write

This article describes the implementation of TC3 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

Here you can access the [source code for this sample](#)

1. Unpack the downloaded ZIP file.
2. Open the zip file that it contains in TwinCAT 3 by clicking on "Open Project...".
3. Select your target system.
4. Build the sample on your local machine (e.g. Build->Build Solution).
5. Activate the configuration by clicking on  .  
⇒ The sample is ready for operation.

### Description

The sample includes an instance of "TcAsyncWritingModule", which writes data to the file "AsyncTest.txt" in directory BOOTPRJPATH (usually C:\TwinCAT\3.x\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.

## 15.16 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 TC3-C++ module.

### Download

Here you can access the [source code for this sample](#)

1. Unpack the downloaded ZIP file.
2. Open the zip file that it contains in TwinCAT 3 by clicking on "Open Project...".
3. Select your target system.
4. Build the sample on your local machine (e.g. Build->Build Solution).
5. Activate the configuration by clicking on  .  
⇒ The sample is ready for operation.

### Description

This sample shows how to access files reading and/or writing from the CycleUpdate method, thus in a cyclic way.

This sample contains the following projects and module instances.

- A static library (TcAsyncFileIo) providing the file access.  
Code for file access could be shared, thus this code is located in a static library used by the driver projects.
- One driver (TcAsyncBufferReadingDrv) providing two instances
  - ReadingModule: Uses the static library to read the file AsyncTest.txt
  - WriteDetectModule: Detecting write operations and initializing reads
- One driver (TcAsyncBufferWritingDrv) providing one instance
  - WriteModule: Uses the static library to write the file AsyncTest.txt

When starting the sample, the writing module starts writing data to the file, which will be located in the Bootproject-Path (usually C:\TwinCAT\3.x\Boot\AsyncTest.txt). The input variable “bDisableWriting” could be used in inhibit writing.

The objects are connected to each other: After writing is done the WritingModule triggers the DetectModule of the TcAsyncBufferReadingDrv. This initiates a read operation by the ReadingModule.

Please monitor the “nBytesWritten” / “nBytesRead” output variables of the WritingModule / ReadingModule. Additionally, log messages are generated on level “verbose”. As usual, these could be configured using the TraceLevelMax Parameter of the modules.

- One driver (TcAsyncFileFindDrv) providing one instance
  - FileFindModule: Uses the static library to list files of a directory

Use the input variable “bExecute” to trigger the action. The Parameter “FilePath” holds the directory to list (default: “c:\TwinCAT\3.1\Boot\\*”).

Please monitor the trace (Loglevel “Verbose”) for list of found files.

### 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.

To read up on file access, please start with the TcFsmFileReader and TcFsmFileWriter classes.

If too many history tracking messages occur, which hamper understanding of the sample, you can disable modules!

### See also

[Sample S19 \[► 268\]](#)

[Sample S20 \[► 269\]](#)

[Sample S25 \[► 274\]](#)


[Interface ITcFileAccess \[► 143\]](#) / [Interface ITcFileAccessAsync \[► 151\]](#)

## 15.17 Sample22: Automation Device Driver (ADD): Access DPRAM

This article describes how to implement a TC3 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](#).

1. Read the configuration details below before activation
2. Unpack the downloaded ZIP file.
3. Open the zip file that it contains in TwinCAT 3 by clicking on "Open Project...".
4. Select your target system.
5. Build the sample on your local machine (e.g. Build->Build Solution).
6. Note the actions listed on this page under **Configuration**.
7. Activate the configuration by clicking on  .  
⇒ The sample is ready for operation.

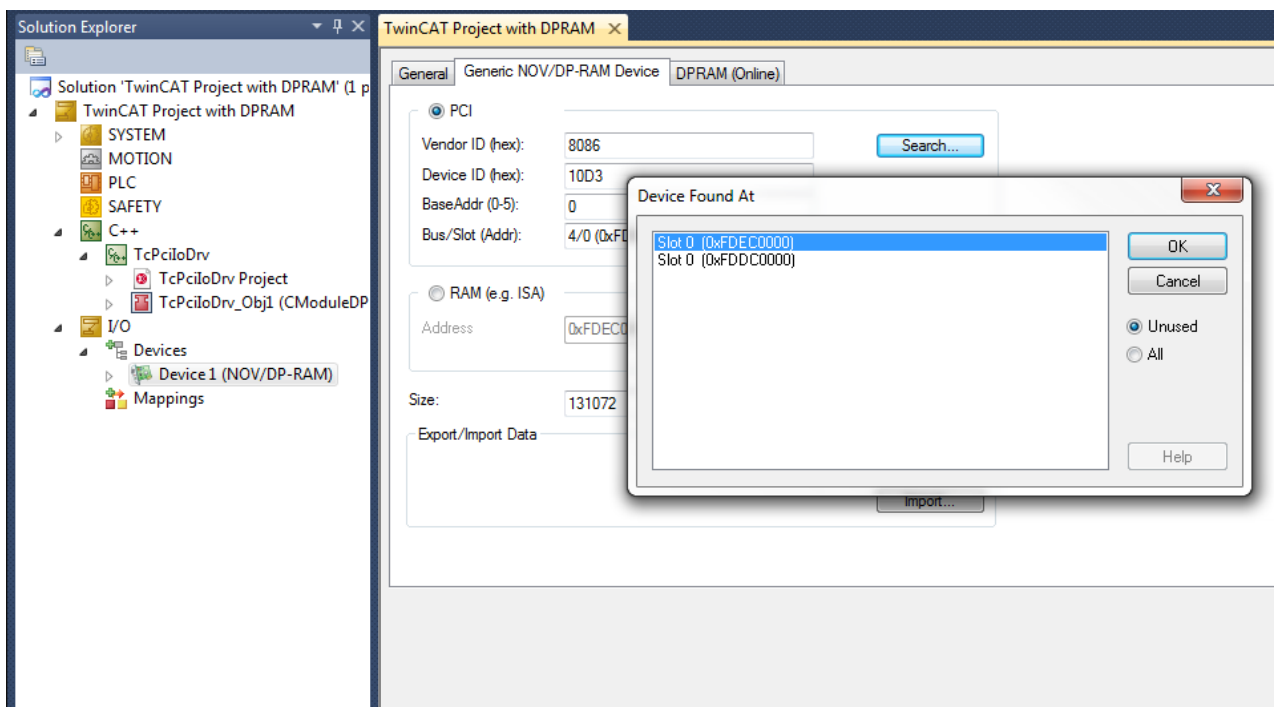
## Description

This sample is prepared to switch the network adapter's "Link Detect Bit" (i.e. of a CX5010) on and off, cyclically.

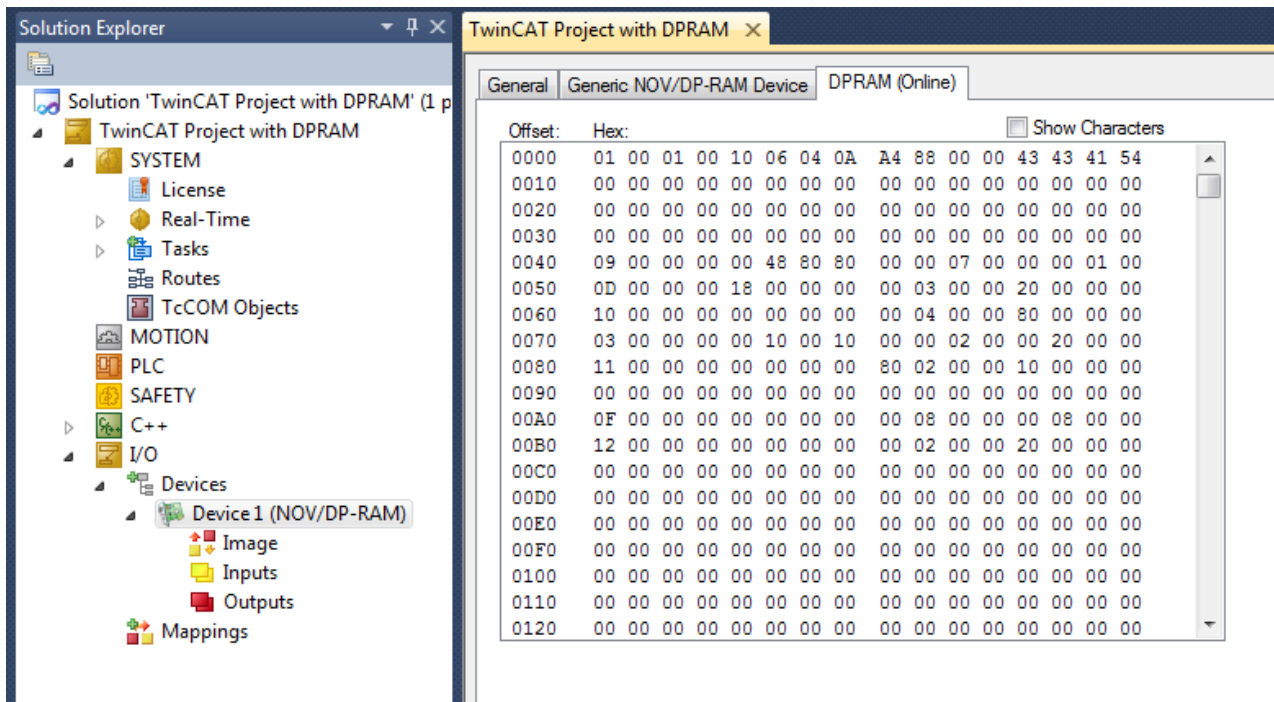
The C++ module is connected to the NOV/DP-RAM Device via the Interface Pointer "PciDeviceAdi" of the C++ module.

## 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/DP-RAM is set up correctly, use the DPRAM (Online) view:




## 15.18 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.

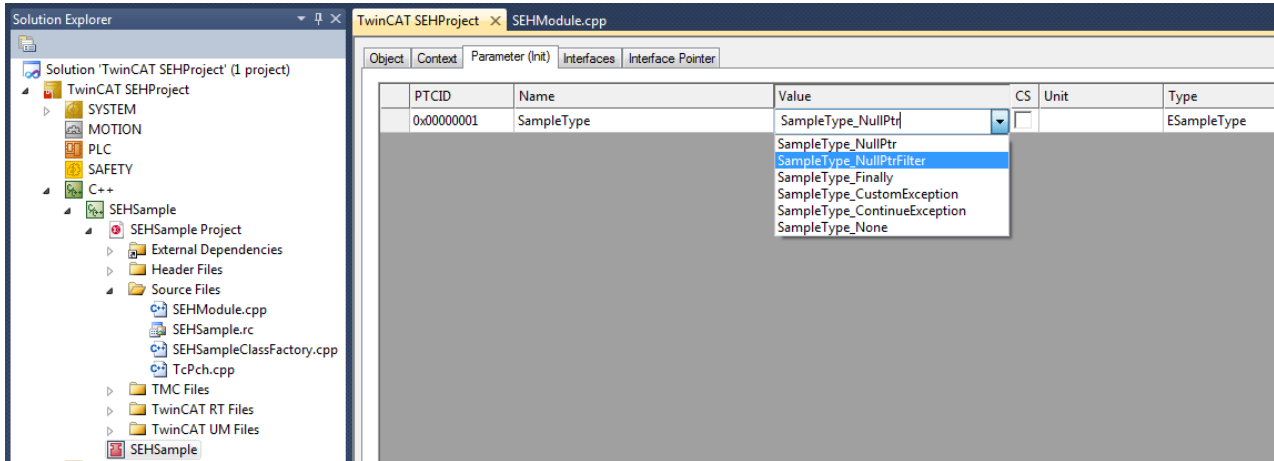
1. Unpack the downloaded ZIP file.
  2. Open the zip file that it contains in TwinCAT 3 by clicking on "Open Project...".
  3. Select your target system.
  4. Build the sample on your local machine (e.g. Build->Build Solution).
  5. Activate the configuration by clicking on  .
- ⇒ The sample is ready for operation.

### Description

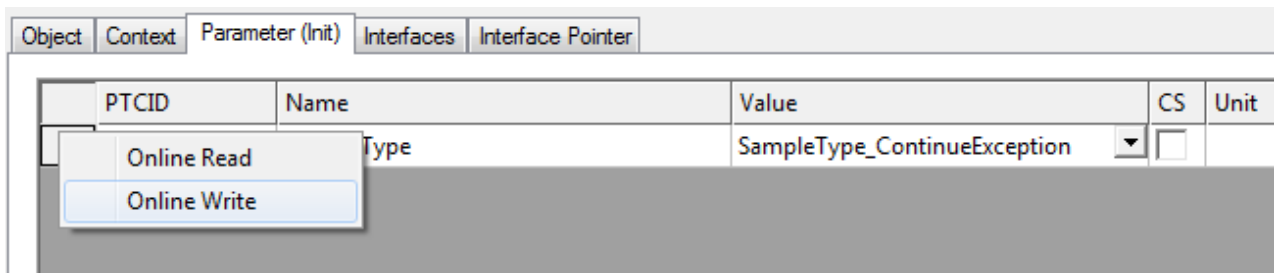
The sample contains five variants that demonstrate the use of SEH in TwinCAT C++:

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

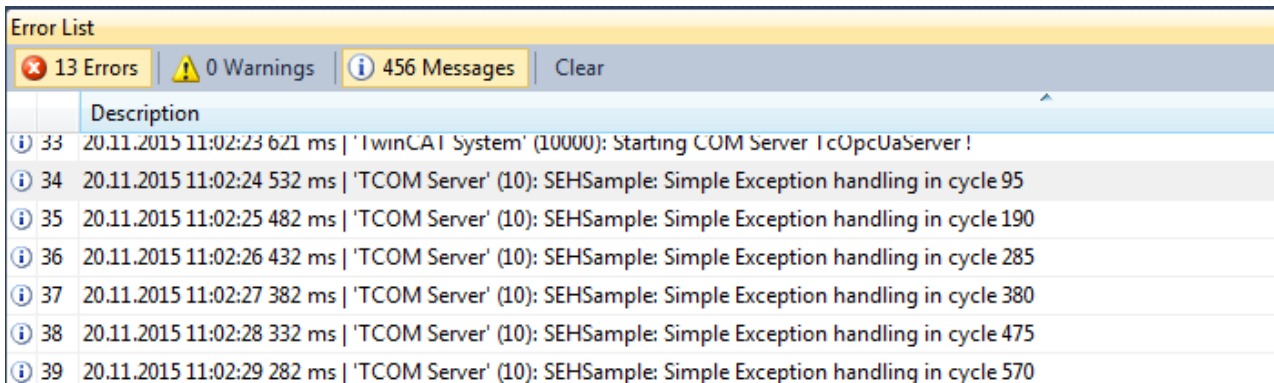
All of these variants can be selected via a drop-down box at the instance of the C++:



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  
Once again a NULL pointer access, 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  
Once again a NULL pointer access with `TcExcept{}`; however, this time the exception is forwarded after handling in the `FilterException()` method so that the further `TcExcept{}` also handles the exception.

## 15.19 Sample25: Static Library

This article describes how to implement a TC3 C++ static library module and you to make use of that module.

### Download

Here you can access the [source code for this sample](#).

1. Unpack the downloaded ZIP file.
2. Open the zip file that it contains in TwinCAT 3 by clicking on "Open Project...".
3. Select your target system.
4. Build the sample on your local machine (e.g. Build->Build Solution).
5. Activate the configuration by clicking on  .  
⇒ The sample is ready for operation.

### Description

The sample contains two projects: The project "DriverUsingStaticLib" uses the static content of the project "StaticLib".

**StaticLib:** On the one hand "StaticLib" offers a function "ComputeSomething" in `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

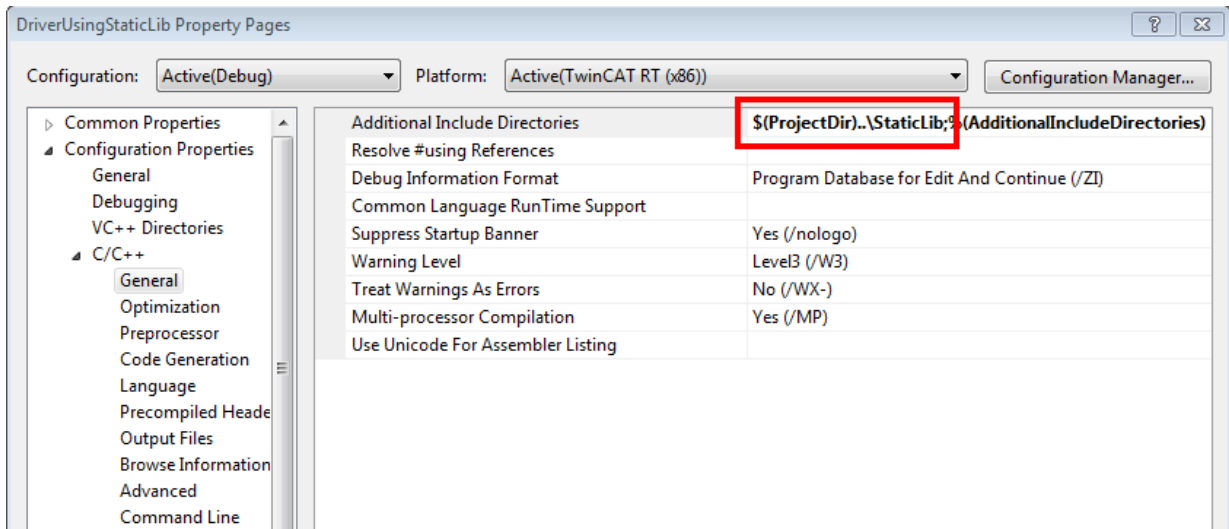
Follow the steps below to create and use a static library.

#### ● Manual recompilation

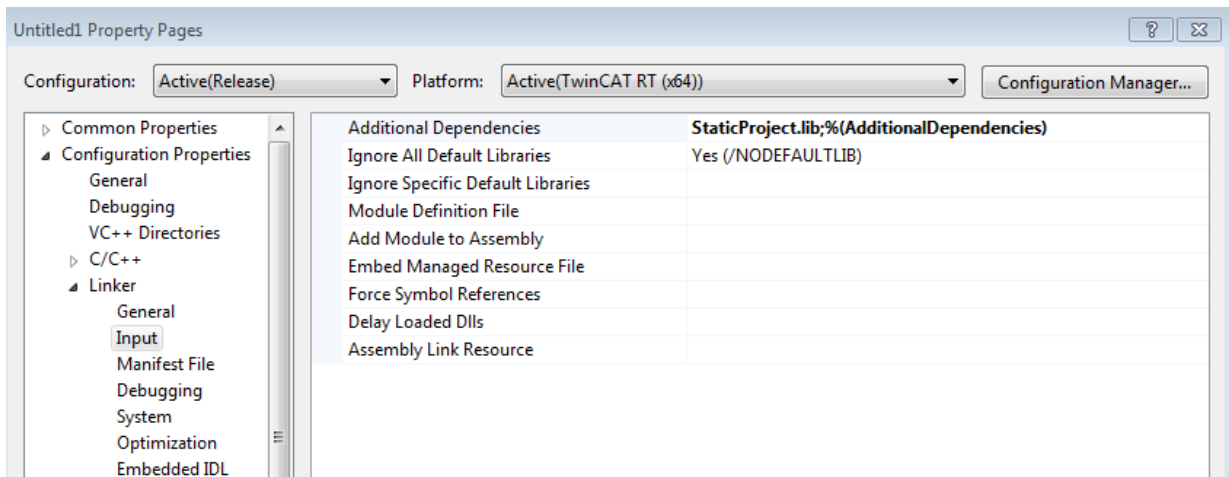
**i** Note that Visual Studio does not automatically recompile the static library during driver development. Do that manually.

- ✓ 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 `%(AdditionalIncludeDirectories)` 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: Execution order at one 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.

1. Unpack the downloaded ZIP file.
  2. Open the zip file that it contains in TwinCAT 3 by clicking on "Open Project...".
  3. Select your target system.
  4. Build the sample on your local machine (e.g. Build->Build Solution).
  5. Activate the configuration by clicking on  .
- ⇒ The sample is ready for operation.



## Description

The sample contains one Module "SortOrderModule" which is instantiated two times. The "Sort Order" determines the execution order, which could be configured via the [TwinCAT Module Instance Configurator](#) [► 124].

For demonstration purpose the "CycleUpdate" method traces Object-Name and -ID together with the sort order of that module. Within the console window one could see the execution order:

```

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

```

Within the sample one instance is configured with Sort Order 150 and one with 170 while they are both assigned to one 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:

The screenshot shows the 'Edit the properties of the parameter' dialog for 'TaskSortOrderContext1Parameter'. The following fields are highlighted with red boxes:

- Name: TaskSortOrderContext1Parameter
- Specification: Alias
- Choose data type: UDINT
- Type Information: Namespace (empty), Guid: {18071995-0000-0000-0000-000000000008}
- Configure the parameter ID: Context based...
- Select the property which should be taken from the given context: Sort order of task
- Select the context: 1
- ID Value: #x030020B0
- Constant Name: PID\_Ctx\_TaskSortOrder

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.

The screenshot shows the 'Optional parameter settings' dialog for 'TaskSortOrderContext1Parameter'. The following fields are highlighted with red boxes:

- Size [Bits]: (empty), x64 specific: (empty)
- Comment: (empty)
- Context ID: 1
- Disable code generation
- Create symbol
- Hide parameter
- Hide sub items
- Online parameter
- Read-only

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()

```

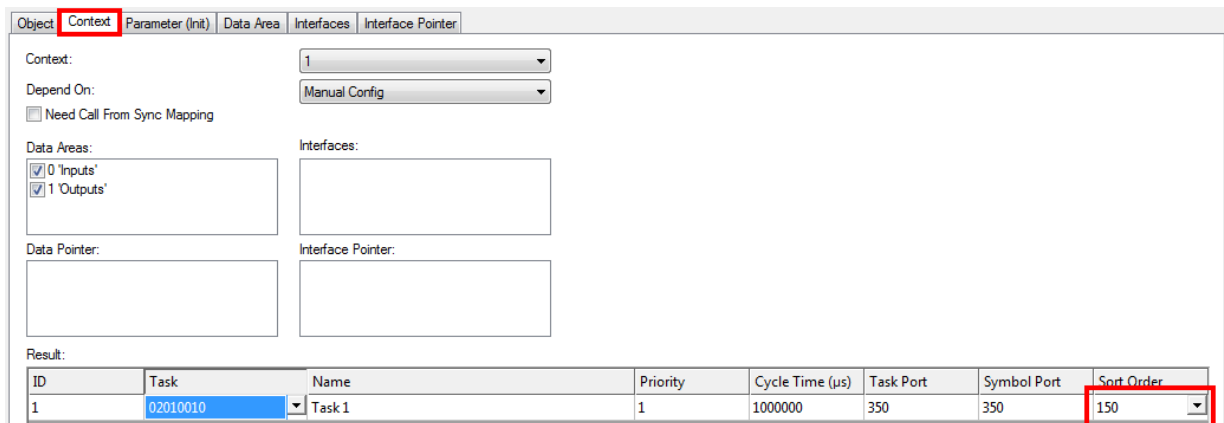
5. In this sample the object name, ID and sort order are tracked cyclically:

```

// 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](#) [▶ 124]:




## 15.21 Sample30: Timing Measurement

This article describes how to implement a TC3 C++ module which contains time measurement functionalities.

### Download

Here you can access the [source code](#) for this sample.

1. Unpack the downloaded ZIP file.
2. Open the zip file that it contains in TwinCAT 3 by clicking on "Open Project...".
3. Select your target system.
4. Build the sample on your local machine (e.g. Build->Build Solution).

5. Activate the configuration by clicking on  .  
⇒ 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 from 1 January 1601 (UTC).
- Querying the distributed clock time when the task cycle starts in nanoseconds since 1 January 2000.
- Querying the time when the method is called at intervals of 100 nanoseconds since 1 January 1601 (UTC).

### See also


[ITcTask interface \[► 165\]](#)

## 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.

1. Unpack the downloaded ZIP file.
2. Open the zip file that it contains in TwinCAT 3 by clicking on "Open Project...".
3. Select your target system.
4. Build the sample on your local machine (e.g. Build->Build Solution).
5. Activate the configuration by clicking on  .  
⇒ The sample is ready for operation.

### Description

The behavior of the module is comparable with a module created by the "Cyclic IO" wizard. `m_input.Value` is added to the `m_Output.Value`. In contrast to the "Cyclic IO" one, this one only adds `m_input.Value` to `m_Output.Value`, if a defined timespan (1000ms) is elapsed.

This is achieved by a class `CTON`, which is comparable to the TON functionblock of PLC / 61131.

### Understanding the Sample

The C++ class `CTON` (`TON.h/.cpp`) provides the behavior of a TON functionblock of the PLC / 61131. The method `Update()` is comparable to the body of the functionblock, which needs to be triggered regularly.

The Method `Update()` gets two "in" parameters:

- IN1: starts timer with rising edge, resets timer with falling edge
- PT: time to pass, before Q is set

And two "out" parameters:

- Q: is TRUE, PT seconds after IN had a rising edge
- ET: elapsed time

Additionally, the ITcTask needs to be provided for retrieving the time base.

### See also

[Sample30: Timing Measurement \[▶ 277\]](#)

[ITcTask interface \[▶ 165\]](#)


## 15.23 Sample35: Access Ethernet

This article describes the implementation of TC3 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.

1. Unpack the downloaded ZIP file.
2. Open the zip file that it contains in TwinCAT 3 by clicking on "Open Project...".
3. Select your target system.
4. Build the sample on your local machine (e.g. Build->Build Solution).
5. Note the actions listed on this page under **Configuration**.
6. Activate the configuration by clicking on  .  
⇒ The sample is ready for operation.

### Description

The sample contains one instance of "TcEthernetSample" module, which sends out and retrieves ARP packages for determining a remote hardware (MAC) address.

The CycleUpdate method implements a rudimentary state machine for sending ARP packages and waiting for an answer with a timeout.

The sample uses two Ethernet related components of TwinCAT:

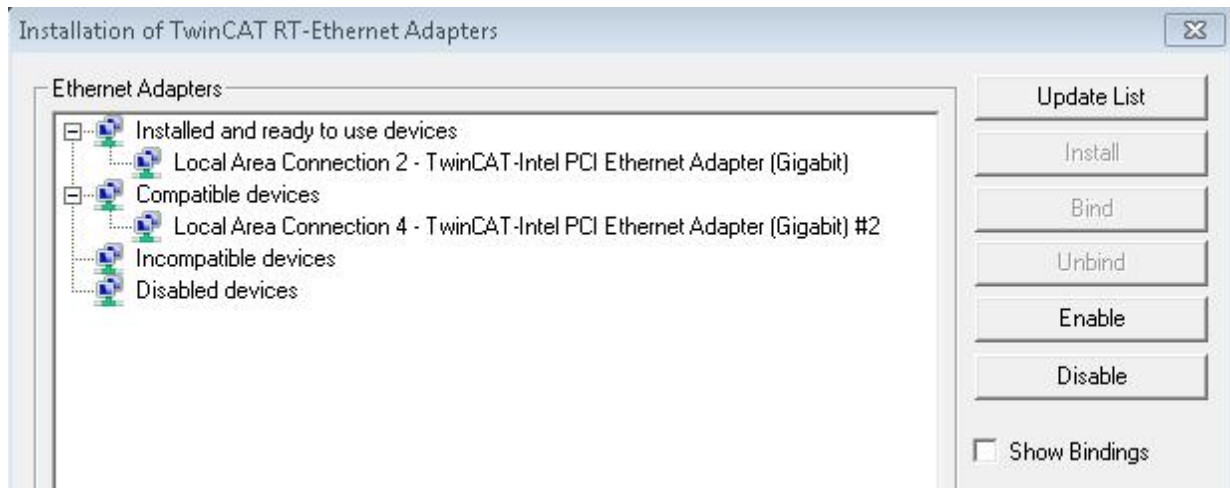
1. An **ITcEthernetAdapter** (instance name in sample m\_spEthernetAdapter) represents an RT Ethernet Adapter. It provides access to adapter parameters like hardware MAC address, link speed, link errors. It can be used to send Ethernet frames and it allows a module instance to register as an ITcloEthProtocol via the registerProtocol method.
2. The **ITclOoEthProtocol** is extended by the sample module, which provides to be notified on Ethernet events by the **ITcEthernetAdapter**.

### Configuration

The downloaded TwinCAT project must be configured for the execution in the 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 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.

### **i** TwinCAT supports retain data

TwinCAT also supports retain data, in order to utilize the NOVRAM of a device to make data persistent.

### Download

Get the [source code](#) for this sample.

1. Unpack the downloaded ZIP file.
2. Open the zip file that it contains in TwinCAT 3 by clicking on "Open Project...".
3. Select your target system.
4. Build the sample on your local machine (e.g. Build->Build Solution).
5. Activate the configuration by clicking on  .  
 ⇒ The sample is ready for operation.

### Description

The TcCOM Object Archive sample shows how to restore and store the state of an object during initialization and deinitialization. The state of the sample class CModuleArchive is the value of the counter CModuleArchive::m\_counter.

In the transition from PREOP to SAFEOP, i.e. method CModuleArchive::SetObjStatePS(), the object archive server (ITComObjArchiveServer) is used to create an object archive for read, which is accessed through interface ITComArchiveOp. This interface provides overloads of operator>>() in order to read from the archive.

In the transition from SAFEOP to PREOP, i.e. method `CModuleArchive::SetObjStateSP()`, the TCOM object archive server is used create an object archive for write, which is accessed through interface `ITComArchiveOp`. This interface provides overloads of `operator<<()` in order to write to the archive.

## 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 \[► 281\]](#) 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 \[► 291\]](#) shows how a PLC application can use functionalities of an existing instance of a TwinCAT C++ class. In this way separate algorithms written C++ (or Matlab) 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.

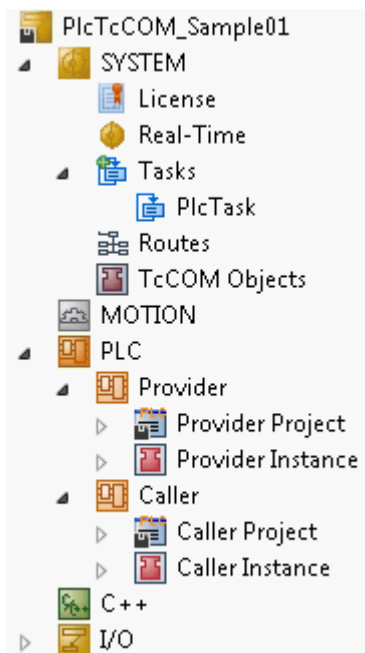
The [TcCOM\\_Sample03 sample \[► 295\]](#) shows how a PLC application uses functionalities of a TwinCAT C++ class by generating an instance of C++ class at the same time. In comparison to the previous sample this can offer increased flexibility.

### 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.

The two PLCs have to be in one 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:

1. [Creating an FB which provides its functionality globally in the first PLC \[► 282\]](#)

2. [Creating an FB which likewise offers this functionality there as a simple proxy in the second PLC \[► 287\]](#)

3. [Execution of the sample project \[► 289\]](#)

Downloading the sample: [https://infosys.beckhoff.com/content/1033/TC3\\_C/Resources/zip/2343046667.zip](https://infosys.beckhoff.com/content/1033/TC3_C/Resources/zip/2343046667.zip)

### NOTE

#### 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.

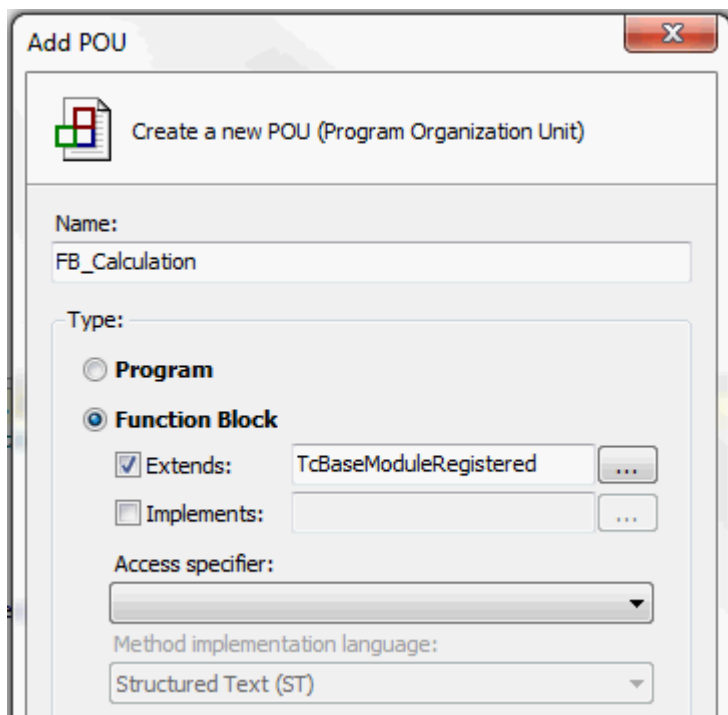
#### System requirements

TwinCAT version	Hardware	Libraries to be integrated
TwinCAT 3.1, Build 4020	x86, x64, ARM	Tc3_Module

### 15.25.1.1 Creating an FB which provides its functionality globally in the first PLC

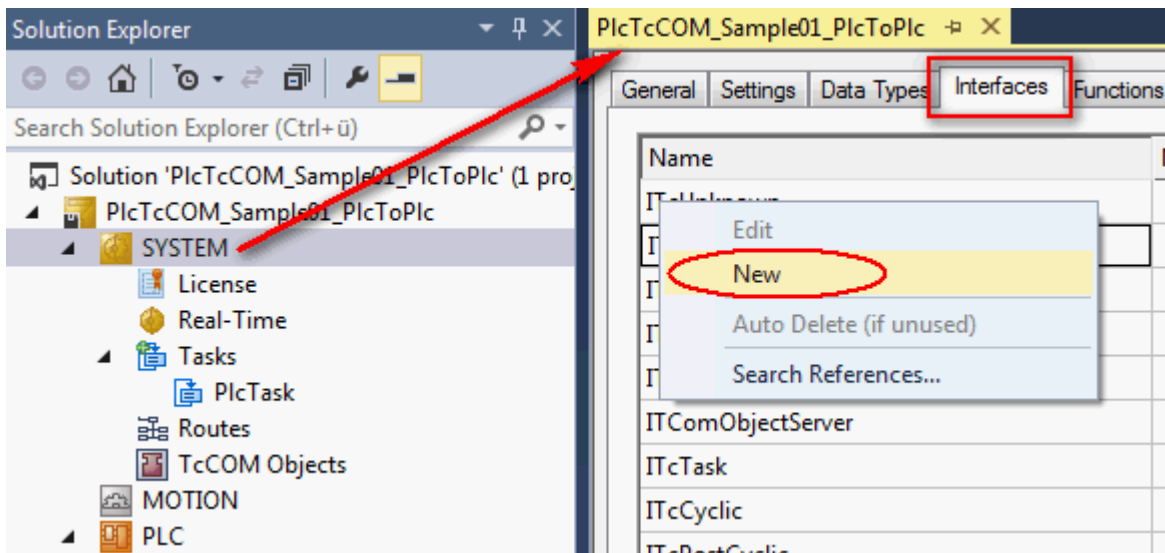
1. Create a PLC and prepare a new function block (FB) (here: FB\_Calculation). Derive the function block from the TcBaseModuleRegistered class, so that an instance of this function block is not only available in the same PLC, but can also be reached from a second.

**Note:** as an alternative you can also modify an FB in an existing PLC.

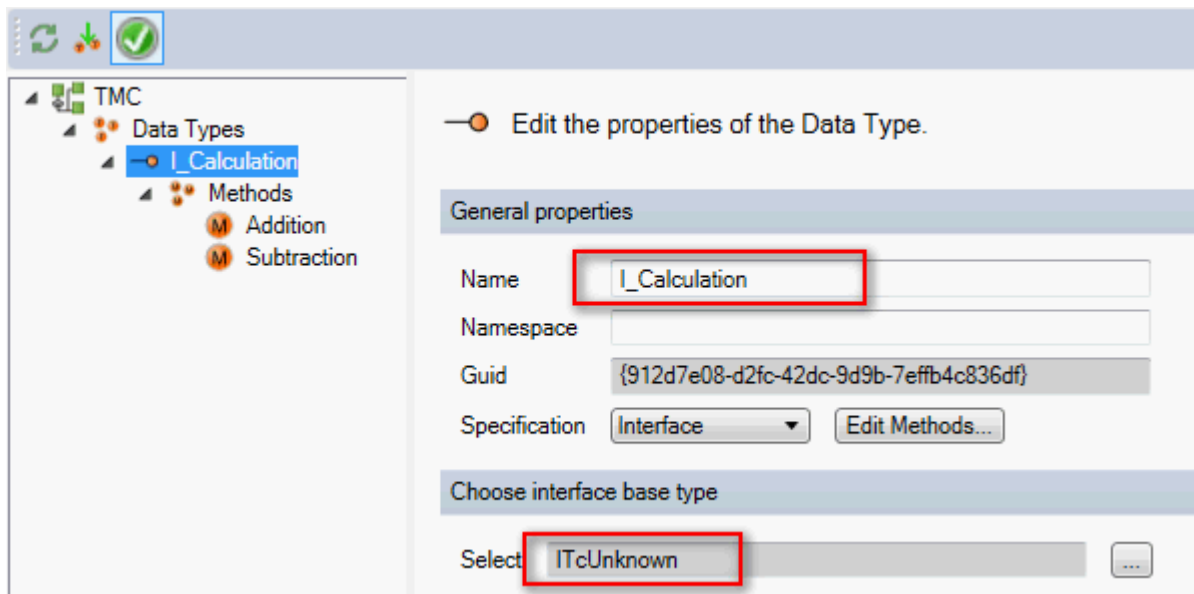


2. The function block must offer its functionality by means of methods. These are defined in a global interface, whose type is system-wide and known regardless of programming language. To create a global interface, open the Context menu in the "Interface" tab of System Properties and choose the option "New".

⇒ The TMC Editor opens, which provides you with support in creating a global interface.



3. Specify the name (here: I\_Calculation) and append the desired methods. The interface is automatically derived from ITcUnknown, in order to fulfill the TwinCAT TcCOM module concept.



4. Specify the name of the methods analogously (here: Addition() and Subtraction()) and select HRESULT as return data type. This return type is mandatory if this type of TcCOM communication should be implemented.
5. Specify the method parameters last and then close the TMC Editor.

**M** Edit the properties of the method.

**General properties**

Name: Addition

**RPC**

Enable  
 Include Return Value

**Choose return data type**

Select: HRESULT  
Description: Normal Type

**Type Information**

Namespace:   
Guid: {18071995-0000-0000-0000-000000000019}

**Define the parameters of the method**

Name	Type	Description	Default
nIn1	INT	Normal Type	
nIn2	INT	Normal Type	
nRes	INT	Is Reference	

6. Now implement the I\_Calculation interface in the FB\_Calculation function block and append the c++\_compatible attribute.

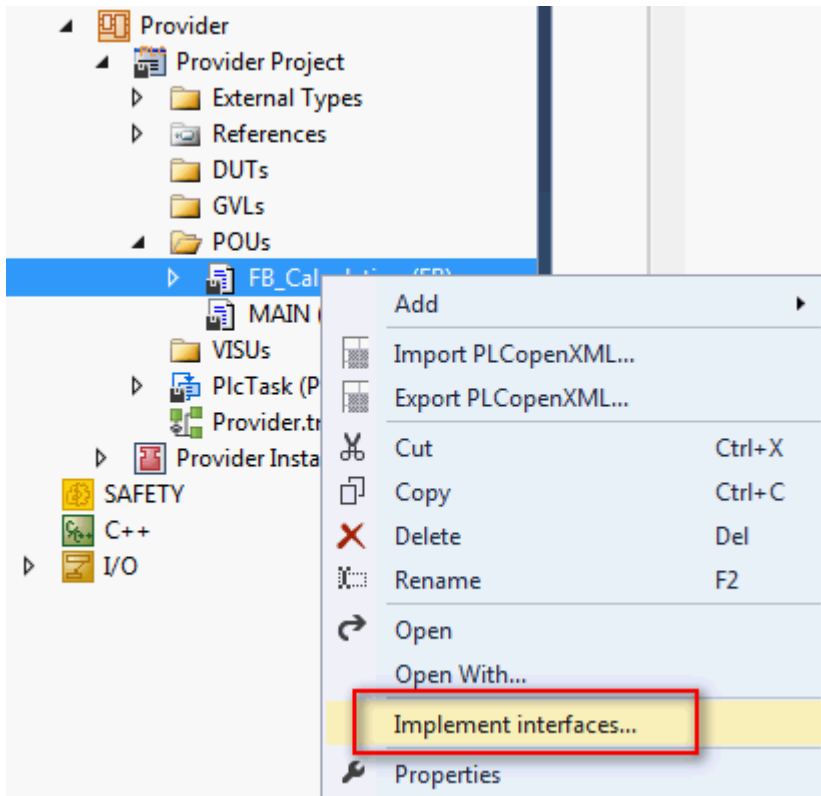
```

4   {attribute 'c++_compatible'}
5   FUNCTION_BLOCK FB_Calculation EXTENDS TcBaseModuleRegistered IMPLEMENTS I_Calculation
6
7   VAR
8   END_VAR
9

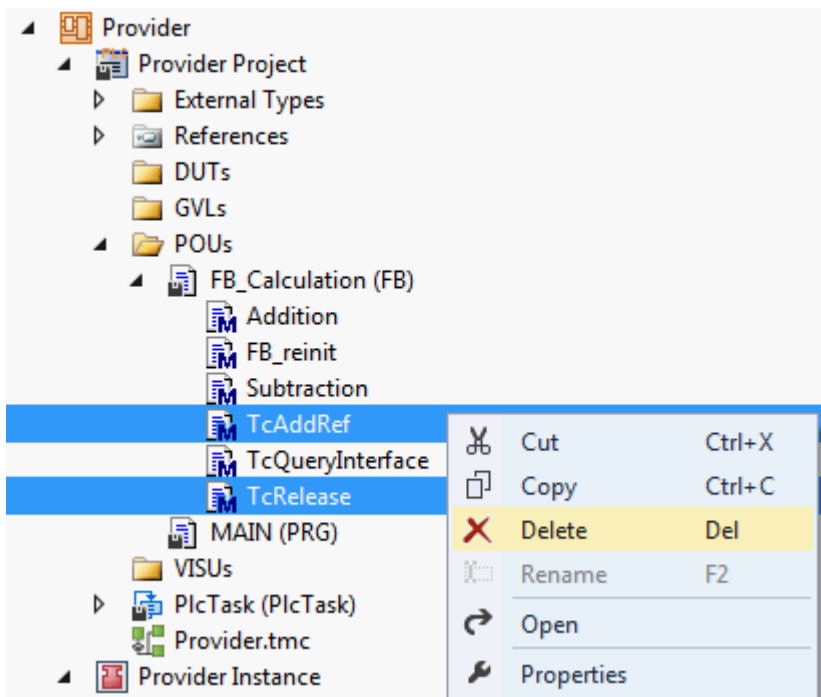
```

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 should 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 [► 169]. Via this method it is possible for other TwinCAT components to obtain an interface pointer to an instance of this function block and thus actuate method calls. The call for TcQueryInterface is successful if the function block or its base class provides the interface queried by means of iid (Interface ID). For this case the handed over interface pointer is allocated the address to the function block type-changed and the reference counter is incremented by means of TcAddRef().

```

FB_Calculation.TcQueryInterface  ▸ ×
1  {attribute 'object_name' := 'TcQueryInterface'}
2  {attribute 'c++_compatible'}
3  {attribute 'signature_flag' := '33554688'}
4  {attribute 'pack_mode' := '4'}
5  {attribute 'show'}
6  {attribute 'minimal_input_size' := '4'}
7  {attribute 'checksuperglobal'}
8  METHOD TcQueryInterface : HRESULT
9  VAR_INPUT
10     iid : REFERENCE TO IID;
11     pipItf : POINTER TO PVOID;
12 END_VAR
13
14 VAR
15     ipCalc : I_Calculation;
16 END_VAR
17
18 IF GuidEqual(ADR(iid), ADR(TC_GLOBAL_IID_LIST.IID_I_Calculation)) THEN
19     ipCalc := THIS^; // cast to interface pointer
20     pipItf^ := ITCUNKNOWN_TO_PVOID(ipCalc);
21     TcAddRef();
22     TcQueryInterface := S_OK;
23 ELSE
24     TcQueryInterface := SUPER^.TcQueryInterface(iid, pipItf);
25 END_IF
26

```

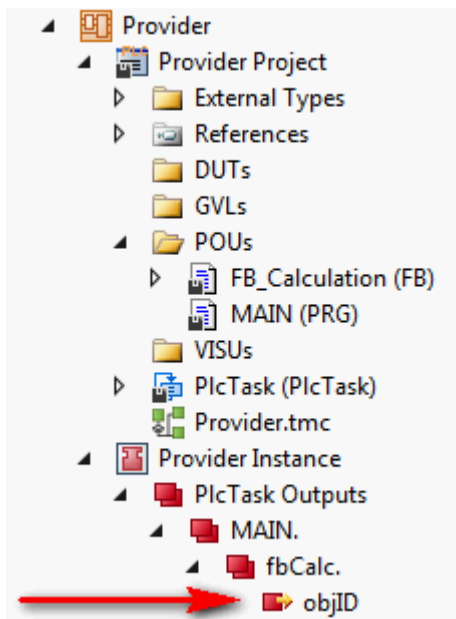
11. Fill the two methods Addition() and Subtraction() with the corresponding code to produce the functionality:  $nRes := nIn1 + nIn2$  and  $nRes := nIn1 - nIn2$
12. Add one or more instances of this function block in the MAIN program module or in a global variable list.  
 ⇒ The implementation in the first PLC is complete.

```

1  PROGRAM MAIN
2  VAR
3      m : UDINT;
4
5      fbCalc : FB_Calculation('MAIN.fbCalc');
6  END_VAR
7

```

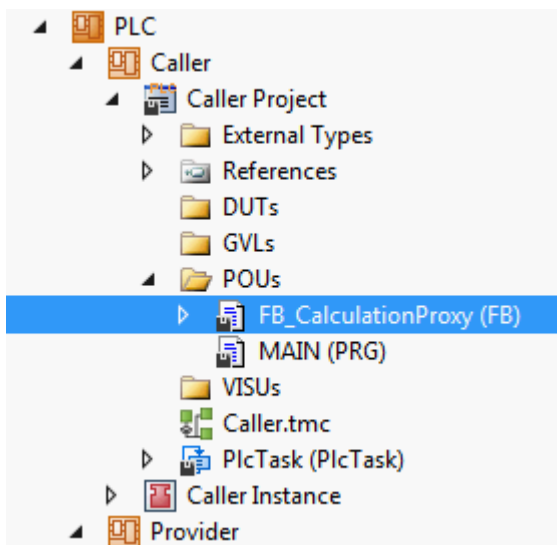
⇒ After compiling the PLC, the object ID of the TcCOM object which represents the instance of FB\_Calculation is available as an outlet in the in the process image.



### 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.



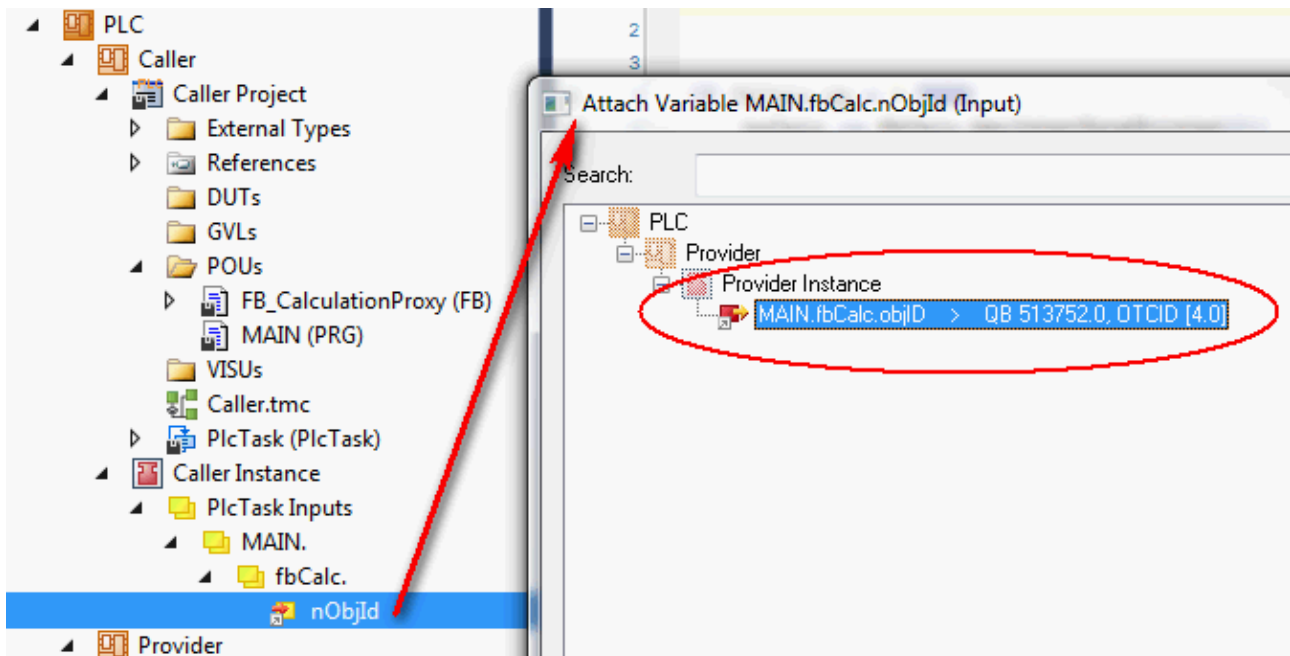
2. In the declaration part of the function block declare as an output an interface pointer to the global interface which later provides the functionality outward.

```

FB_CalculationProxy
1  FUNCTION_BLOCK FB_CalculationProxy
2  VAR_OUTPUT
3      ip : I_Calculation;
4  END_VAR
5
6  VAR
7      {attribute 'displaymode':='hex'}
8      nObjId AT%I* : OTCID; // Instance configured to be retrieved
9      iid : IID := TC_GLOBAL_IID_LIST.IID_I_Calculation;
10 END_VAR
11

```

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
1  METHOD GetInterfacePointer : HRESULT
2  VAR
3  END_VAR
4
5  IF nObjID <> 0 THEN
6      IF (ip = 0) THEN // only get interface pointer if it is not already existing
7          GetInterfacePointer := FW_ObjMgr_GetObjectInstance(oid:=nObjID, iid:=iid, pipUnk:=ADR(ip));
8      ELSE
9          GetInterfacePointer := E_HRESULTAdsErr.EXISTS;
10     END_IF
11 ELSE
12     GetInterfacePointer := E_HRESULTAdsErr.INVALIDOBJID;
13 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
1  |[attribute 'hide']
2  METHOD FB_exit : BOOL
3  VAR_INPUT
4      bInCopyCode : BOOL; // if TRUE, the exit method is c
5  END_VAR

1  IF NOT bInCopyCode THEN // if not online change
2      FW_SafeRelease(ADR(ip));
3  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*  FB_CalculationProxy
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

**i** 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.

The screenshot shows the 'Online Objects' table with the following data:

OTCID	Name	CTCID	State	RefCnt
03000000	IO	03000000-0000-0000-F00...	OP	2
08500000		08500000-0000-0000-F00...	OP	9
08500010	PlcAuxTask	02000002-0000-0000-F00...	OP	7
01010010	Caller Instance	08500001-0000-0000-F00...	OP	11
01010020	Provider Instance	08500001-0000-0000-F00...	OP	11
01010021	Provider_PlcTask	08500004-0000-0000-F00...	OP	4
71010000	MAIN.fbCalc	00000000-0000-0000-000...	OP	4
02000000	RTime	02000000-0000-0000-F00...	OP	47
02010020	PlcTask	01020001-0000-0000-F00...	OP	5
01000000	Router	01000000-0000-0000-F00...	OP	16
01000010	TComServerTask	01000010-0000-0000-F00...	OP	3
01000070	TcEventLogger	01000070-0000-0000-F00...	OP	2

Below the table, the 'MAIN [Online]' variable declaration is shown:

Expression	Type	Value	Prepared value	Add
m	UDINT	23735		
fbCalc	FB_Calculation			
m_objName	STRING	'MAIN.fbCalc'		
m_classId	GUID	{00000000-0000-0000-0000-...		
objID	OTCID	71010000		
hrComObjInit	HRESULT	00000000		
hrComObjExit	HRESULT	00000000		
hrComObjReinit	HRESULT	00000000		

⇒ 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.

The screenshot shows the 'MAIN [Online]' variable declaration in the 'Caller' project:

Expression	Type	Value	Prepared value
fbCalc	FB_CalculationWrapper		
ip	I_Calculation	16#FFFFFFA800AF99E00	
nObjId	OTCID	71010000	
iid	IID	{4D0C9030-560A-45F3-897...	
hrCalc	HRESULT	00000000	

Below the table, the code in the 'Provider' project is shown:

```

4 IF fbCalc.ip[16#FFFFFFA800AF99E00] = 0 THEN
5   hrCalc := fbCalc.QueryInterface();
6 END_IF
7 IF fbCalc.ip[16#FFFFFFA800AF99E00] <> 0 THEN
8   hrCalc := fbCalc.ip.Addition(a_10, b_7, nSum_17);
9   hrCalc := fbCalc.ip.Subtraction(a_10, b_7, nDiff_3);
10 END_IF RETURN
    
```

The code in the 'Provider' project is also shown:

```

1 m := 38186 := m 38186 + 1; RETURN
    
```

## 15.25.2 TcCOM\_Sample02\_PlcToCpp

This example 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++ 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. [Instantiating a TwinCAT++ class as a TwinCAT TcCOM Object \[▶ 291\]](#)
2. [Creating an FB in the PLC, which as a simple wrapper offers the functionality of the C++ object \[▶ 292\]](#)
3. [Execution of the sample project \[▶ 294\]](#)

Downloading the sample: [https://infosys.beckhoff.com/content/1033/TC3\\_C/Resources/zip/2343048971.zip](https://infosys.beckhoff.com/content/1033/TC3_C/Resources/zip/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

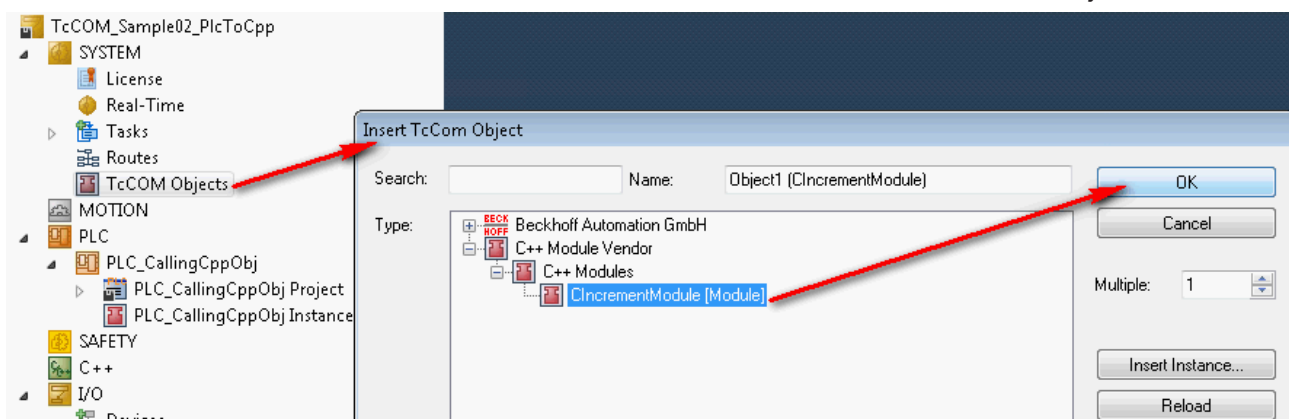
The TwinCAT C++ driver must be available on the destination system. TwinCAT offers a deployment for this purpose, so that the components only have to be stored properly on the development computer.

The existing TwinCAT C++ driver as well as its TMC description file(s) is available as a driver archive. This archive (IncrementerCpp.zip) is unpacked in the following folder:  
 C:\TwinCAT\3.1\CustomConfig\Modules\IncrementerCpp\

The TwinCAT deployment copies the file(s) later in the following folder upon the activation of a configuration on the destination system:

C:\TwinCAT\3.1\Driver\AutoInstall\

1. Open a TwinCAT project or create a new project.
2. Add an instance of Class CIncrementModule in the solution under the node "TcCOM Objects".



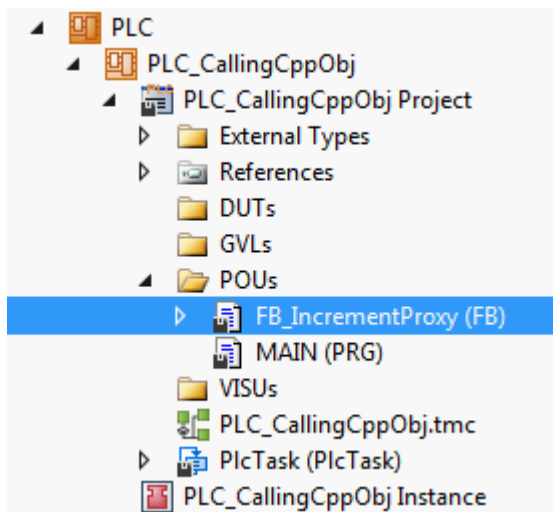
#### **i** Creation of the C++ driver

In the [documentation for TwinCAT C++ \[▶ 9\]](#) there is a detailed explanation on how C++ drivers for TwinCAT are created.

To create the aforementioned driver archive, when creating the driver select "Publish TwinCAT Modules" from the C++ project context as the last step.

### 15.25.2.2 Creating an FB in the PLC which offers as a simple proxy the functionality of the C++ object

1. Create a PLC and append a new function block there.



⇒ This proxy 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 outlet 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.

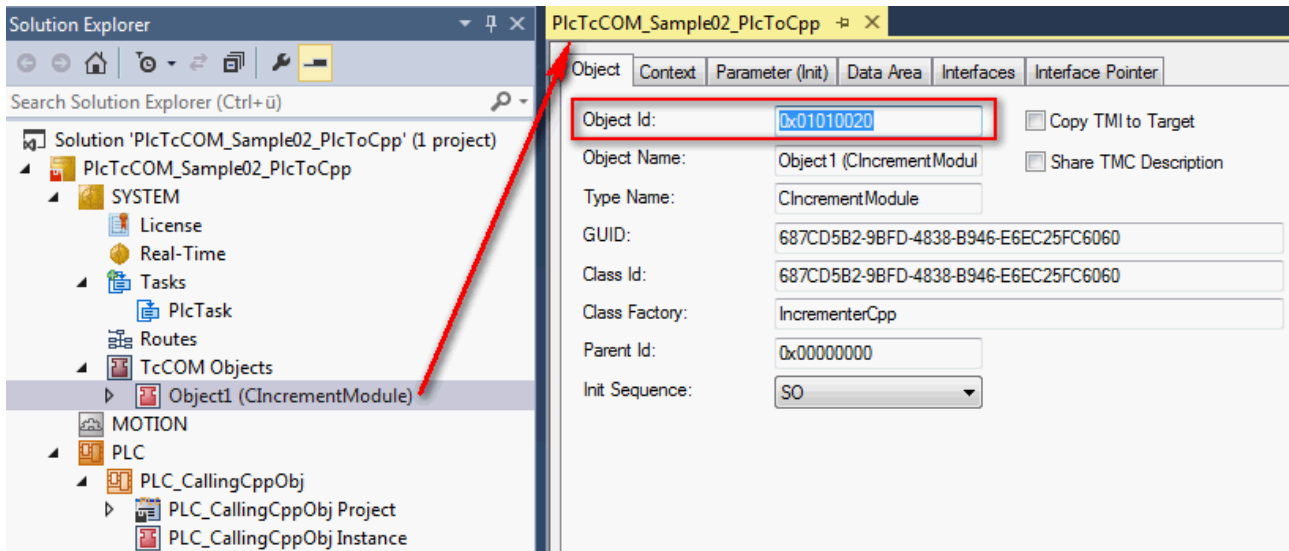
```

FB_IncrementProxy  [X]
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

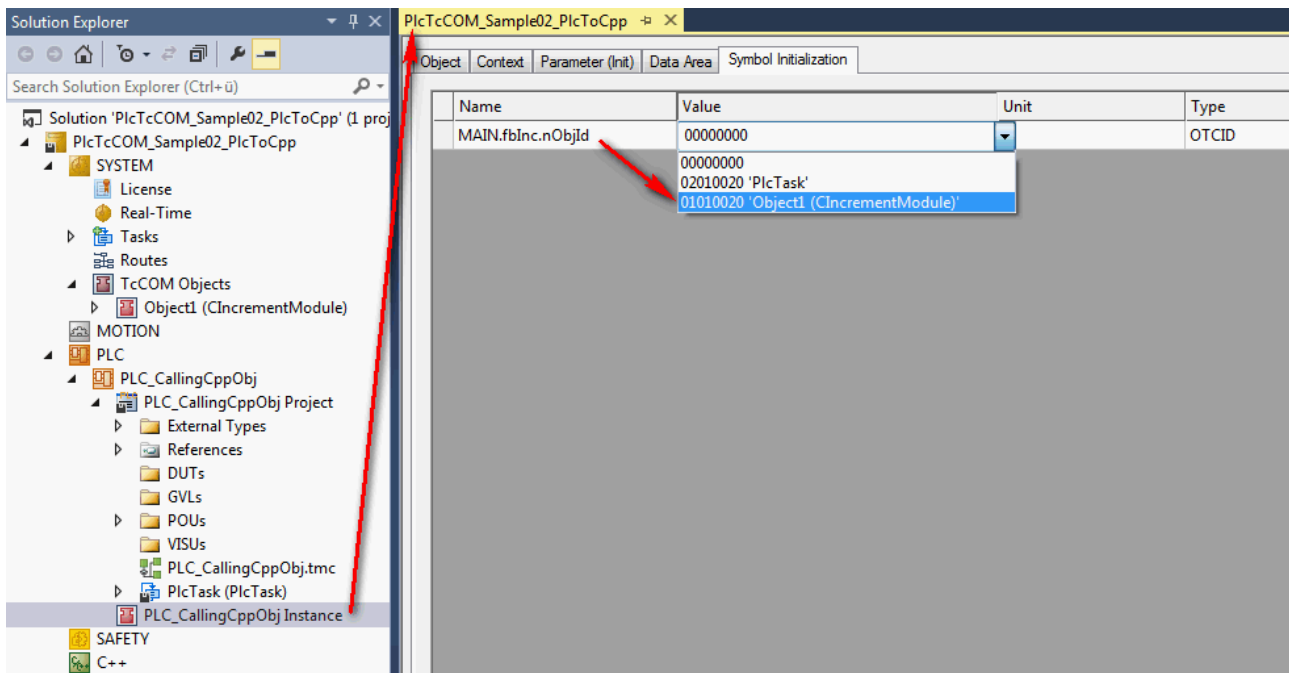
```

⇒ 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. Assign an existing object ID to the symbol name of the variables here by means of DropDown. 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



**Note:** 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 [▶ 281]).

4. Implement the PLC proxy 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  [X]
1  {attribute 'hide'}
2  METHOD FB_init : BOOL
3  VAR_INPUT
4      bInitRetains : BOOL; // if TRUE, the retain variables are initialized (warm start / cold start)
5      bInCopyCode : BOOL; // if TRUE, the instance afterwards gets moved into the copy code (online
6  END_VAR
7
1  IF NOT bInCopyCode THEN // if not online change
2      IF nObjID <> 0 THEN
3          hrInit := FW_ObjMgr_GetObjectInstance(oid:=nObjID, iid:=iid, pipUnk:=ADR(ip));
4      ELSE
5          hrInit := E_HRESULTAdsErr.INVALIDOBJID;
6      END_IF
7  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_IncrementProxy.FB_exit  [X]  FB_IncrementProxy.FB_init  [X]
1  {attribute 'hide'}
2  METHOD FB_exit : BOOL
3  VAR_INPUT
4      bInCopyCode : BOOL; // if TRUE, the exit method is called for
5  END_VAR
1  IF NOT bInCopyCode THEN // if not online change
2      FW_SafeRelease(ADR(ip));
3  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 outlet 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.

```

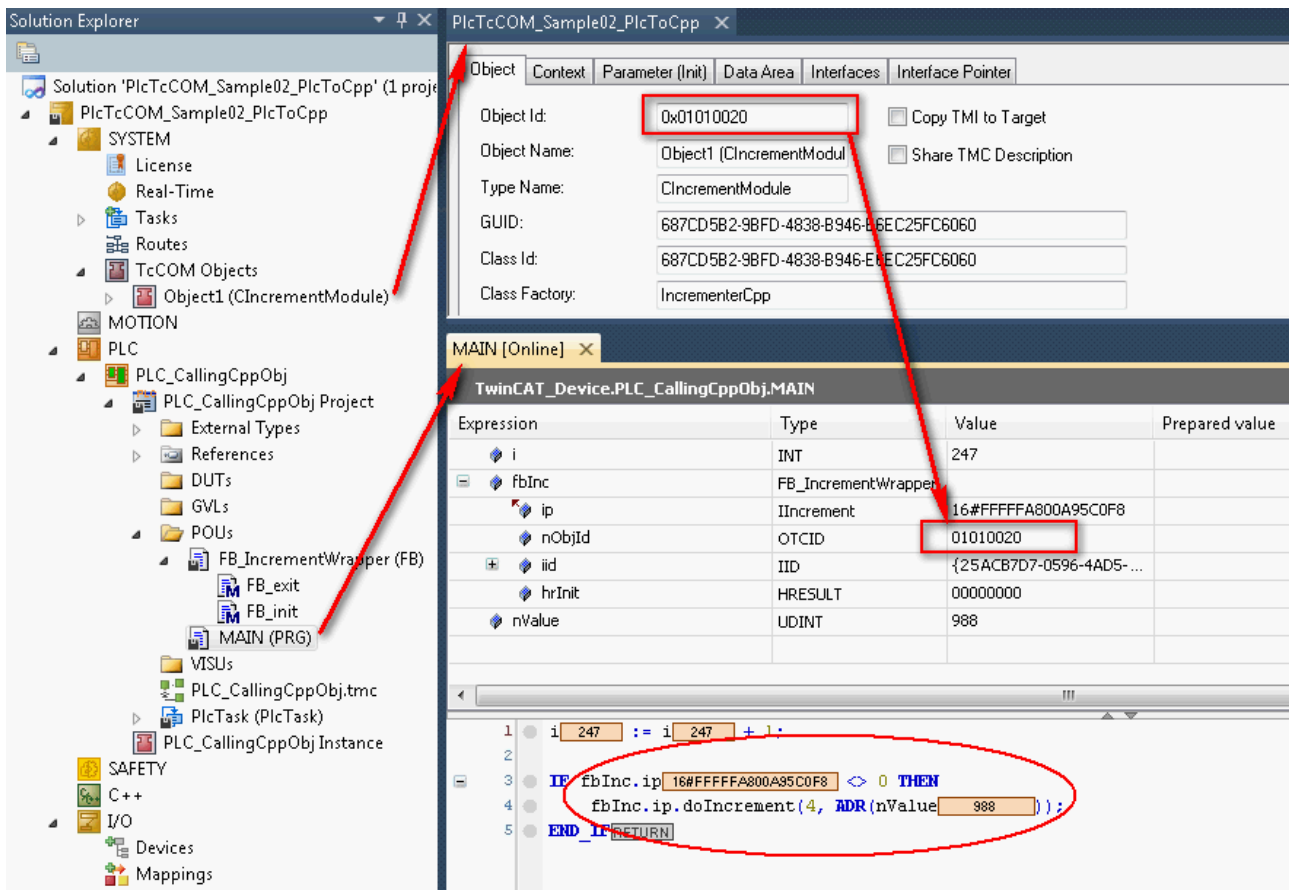
MAIN*  [X]
1  PROGRAM MAIN
2  VAR
3      fbInc : FB_IncrementProxy;
4      nValue : UDINT;
5  END_VAR
6
1  IF fbInc.ip <> 0 THEN
2      fbInc.ip.doIncrement(4, ADR(nValue));
3  END_IF
4

```

⇒ The sample is ready for testing.

### 15.25.2.3 Execution of the sample project

1. Select the destination system and compile the project.
  2. Enable the TwinCAT configuration and execute a log-in as well as starting the PLC.
- ⇒ In the online view of the PLC application the assigned object ID of the C++ object in the PLC Proxy function block can be seen. 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 \[► 296\]](#)
2. [Creating an FB in the PLC that creates the C++ object and offers its functionality \[► 296\]](#)
3. [Execution of the sample project \[► 298\]](#)

Downloading the sample: [https://infosys.beckhoff.com/content/1033/TC3\\_C/Resources/zip/2343051531.zip](https://infosys.beckhoff.com/content/1033/TC3_C/Resources/zip/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 a TwinCAT C++ driver and its classes

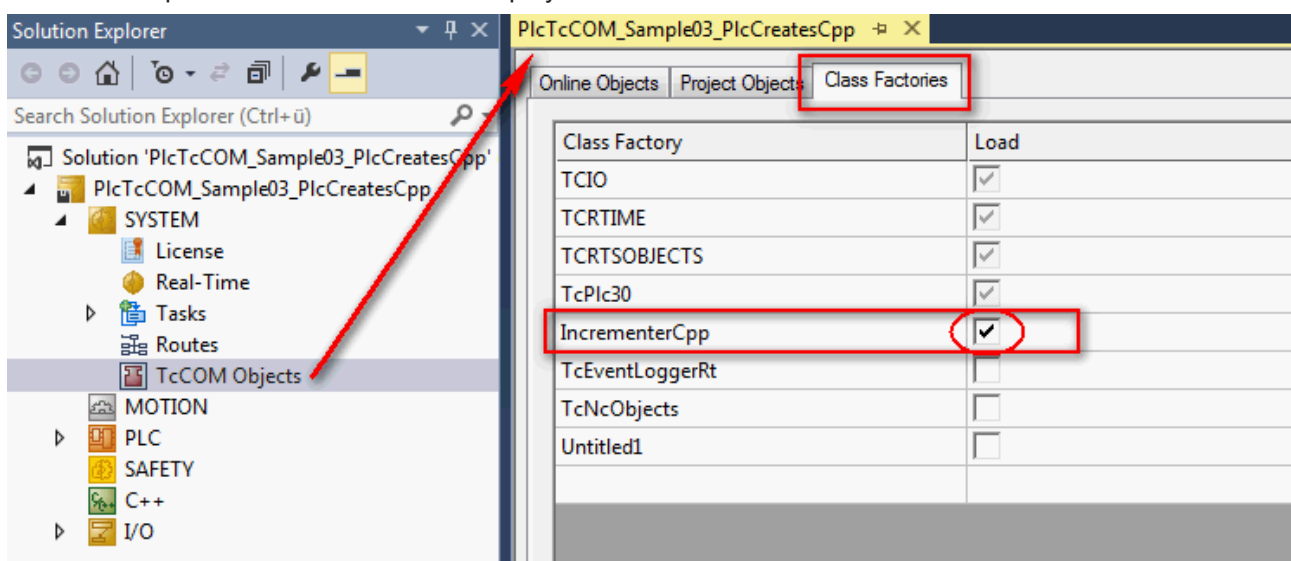
The TwinCAT C++ driver must be available on the destination system. TwinCAT offers a deployment for this purpose, so that the components only have to be stored properly on the development computer.

The existing TwinCAT C++ driver as well as its TMC description file(s) is available as a driver archive. This archive (IncrementerCpp.zip) is unpacked in the following folder:  
C:\TwinCAT\3.1\CustomConfig\Modules\IncrementerCpp\

The TwinCAT deployment copies the file(s) later in the following folder upon the activation of a configuration in the destination system:

C:\TwinCAT\3.1\Driver\AutoInstall\

1. Open a TwinCAT project or create a new project.
  2. Select the required C++ driver in the solution under the TcCOM Objects node in the “Class Factories” tab.
- ⇒ This ensures that the driver is loaded on the destination system when TwinCAT starts up. In addition this selection provides for the described deployment.



#### **i** Creation of the C++ driver

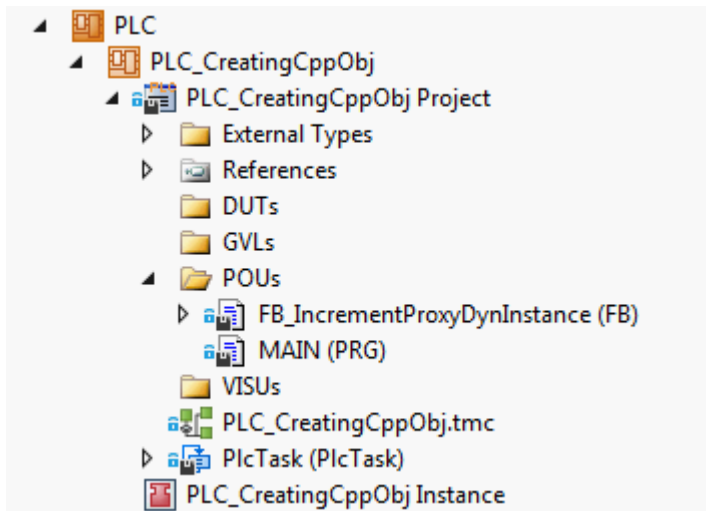
In the [documentation for TwinCAT C++ \[► 9\]](#) there is a detailed explanation on how C++ drivers for TwinCAT are created.

For Sample03 it is important to note that TwinCAT C++ drivers whose classes are supposed to be dynamically instantiated must be defined as “TwinCAT Module Class for RT Context”. The C++ Wizard offers a special template for this purpose.

In addition this sample uses a TwinCAT C++ class which manages without TcCOMinitialization 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 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 outlet an interface pointer to the (IIncrement) interface which later provides the functionality outward.

```

FB_IncrementProxyDynInstance*  [X]
1  FUNCTION_BLOCK FB_IncrementProxyDynInstance
2  VAR_OUTPUT
3      ip : IIncrement;
4  END_VAR
5
6  VAR
7      classId : CLSID := STRING_TO_GUID('687cd5b2-9bfd-4838-b946-e6ec25fc6060');
8      iid : IID := TC_GLOBAL_IID_LIST.IID_IIncrement;
9      hrInit : HRESULT;
10 END_VAR
11
1
    
```

3. Create class ID and the interface ID as member variables. While the interface ID is already available via a global list, the class IDs, provided they are not yet supposed to be known, are determined by other means. When you open the TMC description file of the associated C++ driver you will find the corresponding GUID there.

```

13 <Modules>
14 <Module GUID="{687cd5b2-9bfd-4838-b946-e6ec25fc6060}" Group="C++">
15 <Name:CIncrementModule</Name>
16 <CLSID ClassFactory="IncrementerCpp">{687cd5b2-9bfd-4838-b946-e6ec25fc6060}</CLSID>
17 <Licenses>
18 <License>
    
```

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. In the process the used FW\_ObjMgr\_CreateAndInitInstance() function is also given the name and the destination 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 manages without TcCOM initialization data and without TcCOM parameters. In the case of this function call the object itself increments a reference counter.

```

FB_IncrementProxyDynInstance.FB_init  X
1  METHOD FB_init : BOOL
2  VAR_INPUT
3      bInitRetains : BOOL; // if TRUE, the retain variables are initialized (warm start / cold start)
4      bInCopyCode : BOOL; // if TRUE, the instance afterwards gets moved into the copy code (online change)
5
6      sObjName : STRING; // object name to be set for this instance (optional)
7      eObjState : TCOM_STATE; // target object state (usually TCOM_STATE.TCOM_STATE_OP)
8  END_VAR

1  IF NOT bInCopyCode THEN // if not online change
2      objName := sObjName;
3      hrInit := FW_ObjMgr_CreateAndInitInstance(  clsId      := classId,
4                                                  iid        := iid,
5                                                  pipUnk     := ADR(ip),
6                                                  objId      := OTCID_CreateNewId,
7                                                  parentId   := TwinCAT_SystemInfoVarList._AppInfo.ObjId, //
8                                                  name       := sObjName,
9                                                  state      := eObjState,
10                                                 pInitData  := 0 );
11  END_IF
12

```

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  X  FB_IncrementProxyDynInstance.FB_init
1  {attribute 'hide'}
2  METHOD FB_exit : BOOL
3  VAR_INPUT
4      bInCopyCode : BOOL; // if TRUE, the exit method is called for exiting an instan
5  END_VAR

1  IF NOT bInCopyCode THEN // if not online change
2      FW_ObjMgr_DeleteInstance(ADR(ip));
3  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 outlet 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*  X
1  PROGRAM MAIN
2  VAR
3      fbInc : FB_IncrementProxyDynInstance(  sObjName:='CIncrementModule:fbInc',
4                                              eObjState:=TCOM_STATE.TCOM_STATE_OP);
5      nValue : UDINT;
6  END_VAR
7  |

1  IF fbInc.ip <> 0 THEN
2      fbInc.ip.doIncrement(100, ADR(nValue));
3  END_IF
4

```

⇒ The sample is ready for testing.

### 15.25.3.3 Execution of the sample project

1. Select the destination system and compile the project.

2. Enable the TwinCAT configuration and execute a log-in as well as starting the PLC.

⇒ In the online view of the PLC application the desired TcCOM object name in the PLC Proxy function block can be seen. The project node TcCOM-Objects keeps the generated object with the generated ID and the desired name in his list. The interface pointer has a valid value and the method will be executed.

The screenshot displays the TwinCAT IDE interface. On the left, the Solution Explorer shows the project 'PlcTcCOM\_Sample03\_PlcCreatesCpp' with a tree view including 'SYSTEM', 'MOTION', and 'PLC'. Under 'PLC', there is a 'PLC\_CreatingCppObj' folder containing 'PLC\_CreatingCppObj Project', 'External Types', 'References', 'DUTs', 'GVLs', 'POUs', and 'FB\_IncrementWrapperEx'. The 'FB\_IncrementWrapperEx' folder contains 'FB\_exit', 'FB\_init', 'MAIN (PRG)', 'VISUs', 'PLC\_CreatingCppObj.tmc', 'PlcTask (PlcTask)', and 'PLC\_CreatingCppObj Instance'.

The 'Online Objects' table in the center shows the following data:

OTCID	Name	CTCID	State	RefCnt
03000000	IO	03000000-0000-0000...	OP	2
08500000		08500000-0000-0000...	OP	8
08500010	PlcAuxTask	02000002-0000-0000...	OP	5
01010010	PLC_CreatingCppObj Instance	08500001-0000-0000...	OP	8
02000000	RTime	02000000-0000-0000...	OP	41
02010020	PlcTask	01020001-0000-0000...	OP	4
01000000	Router	01000000-0000-0000...	OP	15
01000010	TComServerTask	01000010-0000-0000...	OP	3
01000070	TcEventLogger	01000070-0000-0000...	OP	2
71010000	CIncrementModule:fbInc	687CD5B2-9BFD-48...	OP	3

The 'MAIN [Online]' window shows the variable declaration and the execution of the 'fbInc' function block. The variable 'i' is declared as an INT with a value of 8598. The 'fbInc' function block is called with the object name 'CIncrementModule:fbInc' and its GUID '16#FFFFFFA800AF99418'. The code snippet is as follows:

```

1 i_8598 := i_8598 + 1;
2
3
4 IF fbInc.ip_16#FFFFFFA800AF99418 <> 0 THEN
5     fbInc.ip.doIncrement(100, ADR(nValue_859800));
6 END_IF
7 RETURN
    
```



## 16 Appendix

- The [ADS return codes \[▶ 300\]](#) are important across TwinCAT 3, particularly if [ADS communication \[▶ 173\]](#) itself is implemented.
- The [Retain data \[▶ 304\]](#) (in NOVRAM memory) can be used in a similar way from the PLC and also C++.
- In addition to the [TcCOM Module \[▶ 30\]](#) 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 \[▶ 307\]](#)
  - [Creating and handling TcCOM modules \[▶ 311\]](#)

### 16.1 ADS Return Codes

Error codes: [0x000 \[▶ 300\]](#)..., [0x500 \[▶ 301\]](#)..., [0x700 \[▶ 302\]](#)..., [0x1000 \[▶ 304\]](#)...

#### Global Error Codes

Hex	Dec	Description
0x0	0	no error
0x1	1	Internal error
0x2	2	No Rtime
0x3	3	Allocation locked memory error
0x4	4	Insert mailbox error
0x5	5	Wrong receive HMSG
0x6	6	target port not found
0x7	7	target machine not found
0x8	8	Unknown command ID
0x9	9	Bad task ID
0xA	10	No IO
0xB	11	Unknown ADS command
0xC	12	Win 32 error
0xD	13	Port not connected
0xE	14	Invalid ADS length
0xF	15	Invalid AMS Net ID
0x10	16	Low Installation level
0x11	17	No debug available
0x12	18	Port disabled
0x13	19	Port already connected
0x14	20	ADS Sync Win32 error
0x15	21	ADS Sync Timeout
0x16	22	ADS Sync AMS error
0x17	23	ADS Sync no index map
0x18	24	Invalid ADS port
0x19	25	No memory
0x1A	26	TCP send error
0x1B	27	Host unreachable
0x1C	28	Invalid AMS fragment



**Router Error Codes**

Hex	Dec	Name	Description
0x500	1280	ROUTERERR_NOLOCKEDMEMORY	No locked memory can be allocated
0x501	1281	ROUTERERR_RESIZEMEMORY	The size of the router memory could not be changed
0x502	1282	ROUTERERR_MAILBOXFULL	The mailbox has reached the maximum number of possible messages. The current sent message was rejected
0x503	1283	ROUTERERR_DEBUGBOXFULL	The mailbox has reached the maximum number of possible messages. The sent message will not be displayed in the debug monitor
0x504	1284	ROUTERERR_UNKNOWNPORTTYPE	Unknown port type
0x505	1285	ROUTERERR_NOTINITIALIZED	Router is not initialized
0x506	1286	ROUTERERR_PORTALREADYINUSE	The desired port number is already assigned
0x507	1287	ROUTERERR_NOTREGISTERED	Port not registered
0x508	1288	ROUTERERR_NOMOREQUEUEUES	The maximum number of Ports reached
0x509	1289	ROUTERERR_INVALIDPORT	Invalid port
0x50A	1290	ROUTERERR_NOTACTIVATED	TwinCAT Router not active

**General ADS Error Codes**

Hex	Dec	Name	Description
0x700	1792	ADSERR_DEVICE_ERROR	error class <device error>
0x701	1793	ADSERR_DEVICE_SRVNOTSUPP	Service is not supported by server
0x702	1794	ADSERR_DEVICE_INVALIDGRP	invalid index group
0x703	1795	ADSERR_DEVICE_INVALIDOFFSET	invalid index offset
0x704	1796	ADSERR_DEVICE_INVALIDACCESS	reading/writing not permitted
0x705	1797	ADSERR_DEVICE_INVALIDSIZE	parameter size not correct
0x706	1798	ADSERR_DEVICE_INVALIDDATA	invalid parameter value(s)
0x707	1799	ADSERR_DEVICE_NOTREADY	device is not in a ready state
0x708	1800	ADSERR_DEVICE_BUSY	device is busy
0x709	1801	ADSERR_DEVICE_INVALIDCONTEXT	invalid context (must be in Windows)
0x70A	1802	ADSERR_DEVICE_NOMEMORY	out of memory
0x70B	1803	ADSERR_DEVICE_INVALIDPARM	invalid parameter value(s)
0x70C	1804	ADSERR_DEVICE_NOTFOUND	not found (files, ...)
0x70D	1805	ADSERR_DEVICE_SYNTAX	syntax error in command or file
0x70E	1806	ADSERR_DEVICE_INCOMPATIBLE	objects do not match
0x70F	1807	ADSERR_DEVICE_EXISTS	object already exists
0x710	1808	ADSERR_DEVICE_SYMBOLNOTFOUND	symbol not found
0x711	1809	ADSERR_DEVICE_SYMBOLVERSIONINVAL	symbol version invalid
0x712	1810	ADSERR_DEVICE_INVALIDSTATE	server is in invalid state
0x713	1811	ADSERR_DEVICE_TRANSMODENOTSUPP	AdsTransMode not supported
0x714	1812	ADSERR_DEVICE_NOTIFYHNDINVALID	Notification handle is invalid
0x715	1813	ADSERR_DEVICE_CLIENTUNKNOWN	Notification client not registered
0x716	1814	ADSERR_DEVICE_NOMOREHDLS	no more notification handles
0x717	1815	ADSERR_DEVICE_INVALIDWATCHSIZE	size for watch too big
0x718	1816	ADSERR_DEVICE_NOTINIT	device not initialized
0x719	1817	ADSERR_DEVICE_TIMEOUT	device has a timeout
0x71A	1818	ADSERR_DEVICE_NOINTERFACE	query interface failed
0x71B	1819	ADSERR_DEVICE_INVALIDINTERFACE	wrong interface required
0x71C	1820	ADSERR_DEVICE_INVALIDCLSID	class ID is invalid
0x71D	1821	ADSERR_DEVICE_INVALIDOBJID	object ID is invalid
0x71E	1822	ADSERR_DEVICE_PENDING	request is pending
0x71F	1823	ADSERR_DEVICE_ABORTED	request is aborted
0x720	1824	ADSERR_DEVICE_WARNING	signal warning
0x721	1825	ADSERR_DEVICE_INVALIDARRAYIDX	invalid array index
0x722	1826	ADSERR_DEVICE_SYMBOLNOTACTIVE	symbol not active
0x723	1827	ADSERR_DEVICE_ACCESSDENIED	access denied
0x724	1828	ADSERR_DEVICE_LICENSENOTFOUND	missing license
0x725	1829	ADSERR_DEVICE_LICENSEEXPIRED	license expired
0x726	1830	ADSERR_DEVICE_LICENSEEXCEEDED	license exceeded
0x727	1831	ADSERR_DEVICE_LICENSEINVALID	license invalid
0x728	1832	ADSERR_DEVICE_LICENSESYSTEMID	license invalid system id
0x729	1833	ADSERR_DEVICE_LICENSENOTIMELIMIT	license not time limited
0x72A	1834	ADSERR_DEVICE_LICENSEFUTUREISSUE	license issue time in the future
0x72B	1835	ADSERR_DEVICE_LICENSETIMETOLONG	license time period to long
0x72c	1836	ADSERR_DEVICE_EXCEPTION	exception occurred during system start
0x72D	1837	ADSERR_DEVICE_LICENSEDUPLICATED	License file read twice
0x72E	1838	ADSERR_DEVICE_SIGNATUREINVALID	invalid signature
0x72F	1839	ADSERR_DEVICE_CERTIFICATEINVALID	public key certificate
0x740	1856	ADSERR_CLIENT_ERROR	Error class <client error>
0x741	1857	ADSERR_CLIENT_INVALIDPARM	invalid parameter at service
0x742	1858	ADSERR_CLIENT_LISTEMPTY	polling list is empty
0x743	1859	ADSERR_CLIENT_VARUSED	var connection already in use
0x744	1860	ADSERR_CLIENT_DUPLINVOKEID	invoke ID in use
0x745	1861	ADSERR_CLIENT_SYNCTIMEOUT	timeout elapsed
0x746	1862	ADSERR_CLIENT_W32ERROR	error in win32 subsystem
0x747	1863	ADSERR_CLIENT_TIMEOUTINVALID	Invalid client timeout value

Hex	Dec	Name	Description
0x748	1864	ADSERR_CLIENT_PORTNOTOPEN	ads-port not opened
0x750	1872	ADSERR_CLIENT_NOAMSADDR	internal error in ads sync
0x751	1873	ADSERR_CLIENT_SYNCINTERNAL	hash table overflow
0x752	1874	ADSERR_CLIENT_ADDHASH	key not found in hash
0x753	1875	ADSERR_CLIENT_REMOVEHASH	no more symbols in cache
0x754	1876	ADSERR_CLIENT_NOMORESVM	invalid response received
0x755	1877	ADSERR_CLIENT_SYNCRESINVALID	sync port is locked

### RTime Error Codes

Hex	Dec	Name	Description
0x1000	4096	RTERR_INTERNAL	Internal fatal error in the TwinCAT real-time system
0x1001	4097	RTERR_BADTIMERPERIODS	Timer value not valid
0x1002	4098	RTERR_INVALIDTASKPTR	Task pointer has the invalid value ZERO
0x1003	4099	RTERR_INVALIDSTACKPTR	Task stack pointer has the invalid value ZERO
0x1004	4100	RTERR_PRIOEXISTS	The demand task priority is already assigned
0x1005	4101	RTERR_NOMORETCB	No more free TCB (Task Control Block) available. Maximum number of TCBs is 64
0x1006	4102	RTERR_NOMORESEMAS	No more free semaphores available. Maximum number of semaphores is 64
0x1007	4103	RTERR_NOMOREQUEUES	No more free queue available. Maximum number of queue is 64
0x100D	4109	RTERR_EXTIRQALREADYDEF	An external synchronization interrupt is already applied
0x100E	4110	RTERR_EXTIRQNOTDEF	No external synchronization interrupt applied
0x100F	4111	RTERR_EXTIRQINSTALLFAILED	The apply of the external synchronization interrupt failed
0x1010	4112	RTERR_IRQNOTLESSORQUAL	Call of a service function in the wrong context
0x1017	4119	RTERR_VMXNOTSUPPORTED	Intel VT-x extension is not supported
0x1018	4120	RTERR_VMXDISABLED	Intel VT-x extension is not enabled in system BIOS
0x1019	4121	RTERR_VMXCONTROLSMISSING	Missing function in Intel VT-x extension
0x101A	4122	RTERR_VMXENABLEFAILS	Enabling Intel VT-x fails

### TCP Winsock Error Codes

Hex	Dec	Description
0x274d	10061	A connection attempt failed because the connected party did not properly respond after a period of time, or established connection failed because connected host has failed to respond.
0x2751	10065	No connection could be made because the target machine actively refused it. This error normally occurs when you try to connect to a service which is inactive on a different host - a service without a server application.
0x274c	10060	No route to a host. A socket operation was attempted to an unreachable host
		Further Winsock error codes: Win32 Error Codes

## 16.2 Retain data

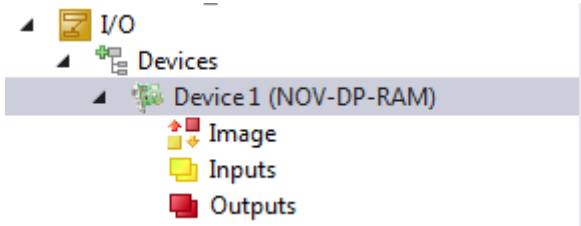
This section describes the option to make data available even after an ordered or spontaneous system restart. The NOV-RAM of a device is used for this purpose.

The following section describes the retain handler, which stores data and makes them available again, and the application of the different TwinCAT 3 programming languages.

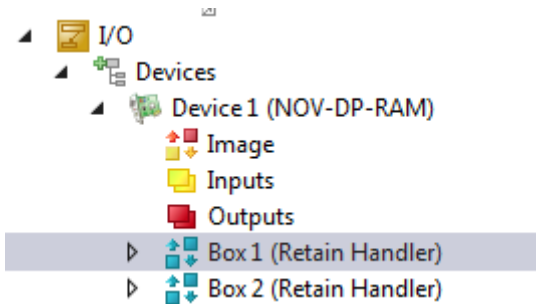
### Configuring a retain device

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.

A NOV-RAM DP device is created in the IO section of the solution for this purpose.

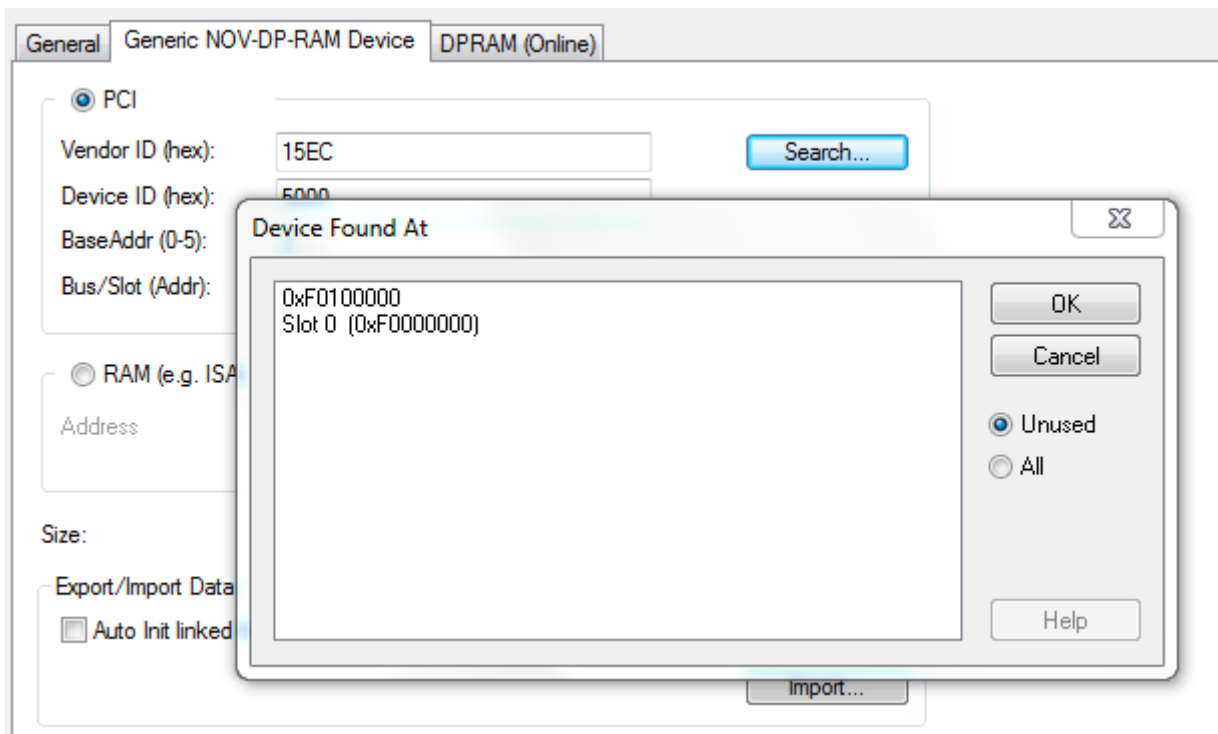


One or several retain handlers can be created under this device



**Storage location: NOV-RAM**

The NOV-DP-RAM device must be configured. The range to be used can be defined in the "Generic NOV-DP-RAM device" tab via "Search...".



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 section or identified with the attribute TcRetain.

```

PROGRAM MAIN
VAR RETAIN
  l: UINT;
  k AT %Q* : 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".

The screenshot shows the TwinCAT IDE interface. On the left, the Solution Explorer displays a project structure with 'Unbenannt1 Instance' expanded to show 'PlcTask Retains' containing 'MAIN.l', 'MAIN.k', and 'MAIN.m'. On the right, the 'Data Area' table is visible, showing three entries for 'PlcTask' components.

Area No	Name	Type	Size	CS	Elements	Retain Hdl
+ 1 (0)	PlcTask Outputs	OutputSrc	589824	<input checked="" type="checkbox"/>	1 Symbols	
+ 3 (0)	PlcTask Internal	Internal	589824	<input checked="" type="checkbox"/>	13 Symbols	
+ 4 (0)	PlcTask Retains	RetainSrc	589824	<input checked="" type="checkbox"/>	3 Symbols	03020001 'Box 1 (Retain H...

If self-generated data types (DUTs) are used as retains, the data types must exist in the TwinCAT type system. You can either use the option "Convert to Global Type" or you can create structures directly as `STRUCT RETAIN`, which will handle all occurrences of the structure via the retain handler.

### 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.

The screenshot shows the 'Edit the properties of the Data Area' dialog box. The left pane shows the 'TMC Module Classes' tree with 'CModule1' expanded to 'Data Areas' > 'RetainSample' > 'Symbols' containing 'x', 'y', and 'z'. The right pane shows the configuration for the 'RetainSample' data area.

**Edit the properties of the Data Area.**

**General properties**

Number:

Type:

Name:

**Optional data area settings**

Size [Bytes]:  x64 specific

Comment:

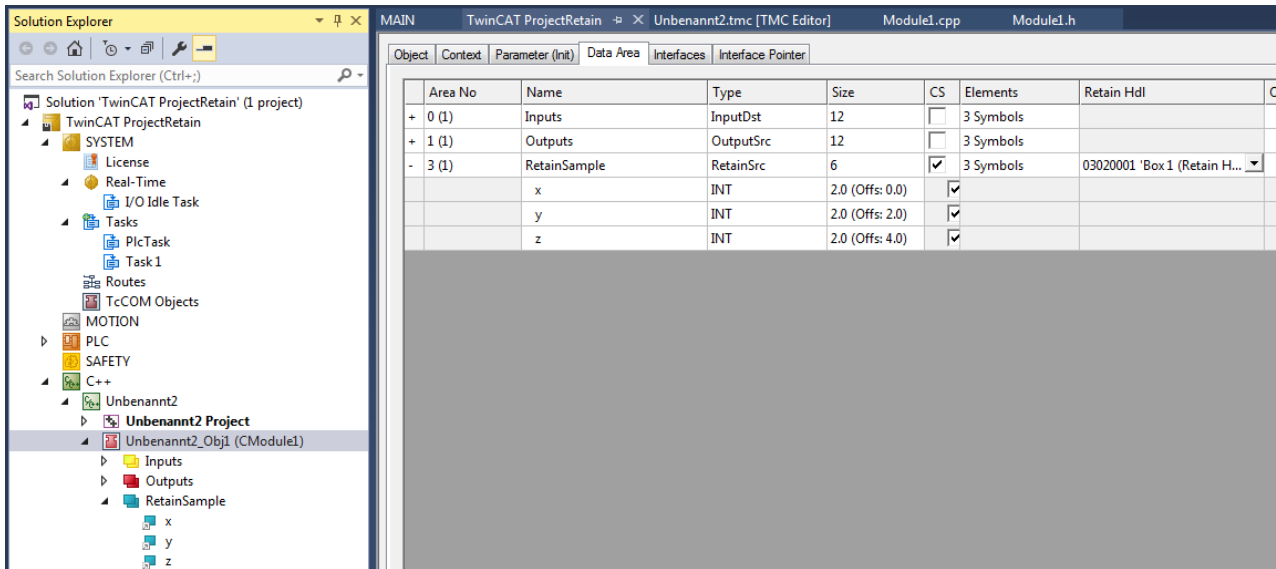
Context ID:

Datatype name:

Create symbols

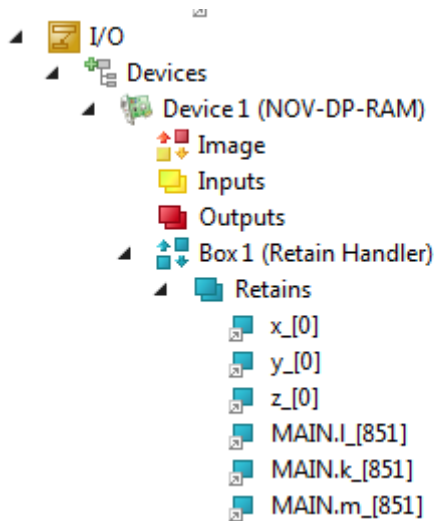
Disable code generation

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 \[▶ 76\]](#).

The customer could define own templates, which is documented at [the corresponding sub-section if C++ Section / Wizards \[▶ 128\]](#).

## Creating C++ projects

To create a new C++ project via Automation Interface, you need to navigate to the C++ node and then execute the `CreateChild()` method with the corresponding 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)
```

For instantiating a driver project please use "TcDriverWizard" 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`.

## Opening existing C++ projects

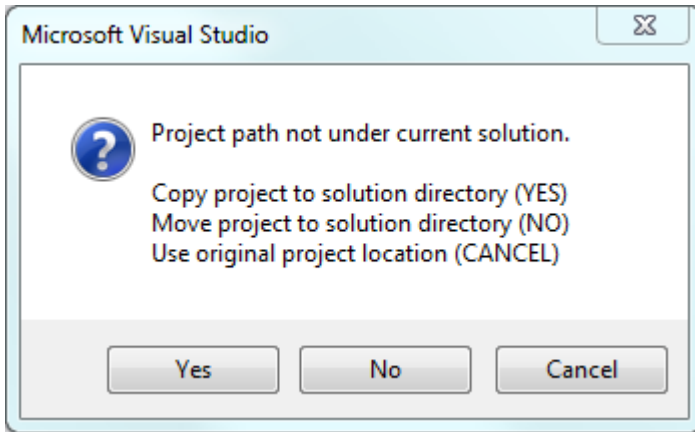
To open an existing C++-Project via Automation Interface, you need to navigate to the C++ node and then execute the `CreateChild()` method with the path to the corresponding C++ project file as a parameter.

You can use three different values as `SubType`:

- 0: Copy project to solution directory
- 1: Move project to solution directory
- 2: Use original project location (specify "" as `NameOfProject` parameter)



Basically, these values represent the functionalities (Yes, No, Cancel) from the following MessageBox in TwinCAT XAE:



In place of the template file you need to use the path to the C++ project (to its vcxproj file) that needs to be added. As an alternative, you can also use a C++ project archive (tzip file).

#### Code snippet (C#):

```
ITcSmTreeItem cpp = systemManager.LookupTreeItem("TIXC");
ITcSmTreeItem newProject = cpp.CreateChild("NameOfProject", 1, "", pathToProjectOrTzipFile);
```

#### Code snippet (Powershell):

```
$cpp = $systemManager.LookupTreeItem("TIXC")
$newProject = $cpp.CreateChild("NameOfProject", 1, "", $pathToProjectOrTzipFile)
```

Please note that C++ projects can't be renamed, thus the original project name needs to be specified. (cmp. [Renaming TwinCAT C++ projects](#) [► 199])

#### Creating module instances

TcCOM Modules could be created at the System -> TcCOM Modules node. Please [see documentation there](#) [► 311].

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 TMC Code Generator

TMC Code generator could be called to generate C++ code after changes at the TMC file of 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 Publish Modules command

Publishing includes building the project for all platforms. The compiled module will be provided for Export like described in the [Module-Handling section of C++ \[▶ 44\]](#).

#### 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 provide different options for the build and deployment process. These are settable by 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)
```

For the `BootProjectEncryption` the values "None" and "Target" are valid. Both other settings are "false" and "true" values.

## Building project

To build the project or solution you can use the corresponding classes and methods of the Visual Studio API, which are documented here.

## 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. 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 or programatically, which will be explained later in this chapter.

Object	Context	Parameter (Init)	Parameter (Online)	Data Area	Interfaces	Block Diagram
Object Id:		<input type="text" value="0x01010020"/>			<input type="checkbox"/>	Copy TMI to Target
Object Name:		<input type="text" value="Object1 (TempContr)"/>				
Type Name:		<input type="text" value="TempContr"/>				
GUID:		<input type="text" value="8F5FDCFF-EE4B-4EE5-80B1-25EB23BD1B45"/>				
Class Id:		<input type="text" value="8F5FDCFF-EE4B-4EE5-80B1-25EB23BD1B45"/>				
Class Factory:		<input type="text" value="TempContr"/>				
Parent Id:		<input type="text" value="0x00000000"/>				
Init Sequence:		<input type="text" value="PSO"/>				

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.

#### 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#):

```
ITcModuleManager2 moduleManager = (ITcModuleManager2)systemManager.GetModuleManager();
foreach (ITcModuleManager2 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 = destModule.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 = dataAreaNo.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:

The screenshot shows the configuration window for a task in TwinCAT XAE. The 'Context' dropdown is set to '0'. Below it, 'Depend On:' is set to 'Manual Config'. The 'Data Areas' section has four checked items: '0 'Input'', '1 'Output'', '21 'BlockIO'', and '22 'Cont.State''. The 'Data Pointer' and 'Interface Pointer' fields are empty. At the bottom, a 'Result' table shows the following data:

ID	Task	Name	Priority	Cycle Tim...	Task Port	Symbol Port	Sort Order
0	02010010	AdditionalTask1	1	10000	350	350	0

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 = taskObjectIdNode.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()`.