Manual

TC3 C++

TwinCAT 3

Version: 1.9
Date: 2019-11-11
Order No.: TC1300
# Table of contents

1 Foreword ......................................................................................................................... 9  
  1.1 Notes on the documentation ......................................................................................... 9  
  1.2 Safety instructions ......................................................................................................... 10  
2 Overview ........................................................................................................................................... 11  
3 Introduction ............................................................................................................................ 12  
  3.1 From conventional user mode programming to real-time programming in TwinCAT .... 14  
4 Requirements ......................................................................................................................... 20  
5 Preparation - just once! ............................................................................................................. 22  
  5.1 Installation "Microsoft Windows Driver Kit 7 (WDK)" .................................................. 22  
  5.2 Visual Studio - TwinCAT XAE Base toolbar .................................................................. 25  
  5.3 Prepare Visual Studio - Configuration and Platform toolbar ....................................... 26  
  5.4 Driver signing ................................................................................................................ 26  
    5.4.1 Operating system ....................................................................................................... 27  
    5.4.2 TwinCAT .................................................................................................................. 32  
6 Modules ...................................................................................................................................... 36  
  6.1 The TwinCAT Component Object Model (TcCOM) concept ....................................... 36  
    6.1.1 TwinCAT module properties .................................................................................... 38  
    6.1.2 TwinCAT module state machine ............................................................................ 45  
  6.2 Module-to-module communication ................................................................................. 47  
7 Modules - Handling ................................................................................................................. 50  
  7.1 Versioned C++ Projects ................................................................................................. 50  
  7.2 Non-versioned C++ projects ........................................................................................... 50  
    7.2.1 Export to TwinCAT 3.1 4022.xx .............................................................................. 51  
    7.2.2 Import up to TwinCAT 3.1 4022.xx ........................................................................ 52  
  7.3 Starting Modules .............................................................................................................. 53  
  7.4 TwinCAT Loader .............................................................................................................. 54  
    7.4.1 Test signing ............................................................................................................... 54  
    7.4.2 Encrypting Modules ................................................................................................. 56  
    7.4.3 Return Codes ............................................................................................................ 58  
    7.4.4 TcSignTool - Storage of the certificate password outside the project .................... 58  
8 TwinCAT C++ development ...................................................................................................... 59  
9 Quick start ............................................................................................................................... 61  
  9.1 Compiling/building a TwinCAT 3 C++ project ............................................................... 61  
  9.2 Create TwinCAT 3 project .............................................................................................. 62  
  9.3 Create TwinCAT 3 C++ project ...................................................................................... 63  
  9.4 Implement TwinCAT 3 C++ project ............................................................................... 67  
  9.5 Create TwinCAT 3 C++ Module instance .................................................................... 68  
  9.6 Create a TwinCAT task and apply it to the module instance ........................................ 70  
  9.7 TwinCAT 3 enable C++ debugger .................................................................................. 72  
  9.8 Activating a TwinCAT 3 project ..................................................................................... 73  
  9.9 Debug TwinCAT 3 C++ project ...................................................................................... 75  
10 Quick start with Online Change .......................................................................................... 80
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.1 Create TwinCAT 3 project</td>
<td>80</td>
</tr>
<tr>
<td>10.2 Create TwinCAT 3 C++ project</td>
<td>81</td>
</tr>
<tr>
<td>10.3 TwinCAT 3 C++ Configure project</td>
<td>84</td>
</tr>
<tr>
<td>10.4 Implement TwinCAT 3 C++ project</td>
<td>85</td>
</tr>
<tr>
<td>10.5 Publish TwinCAT 3 C++ project in version 0.0.0.1</td>
<td>86</td>
</tr>
<tr>
<td>10.6 Implement and publish TwinCAT 3 C++ project version 0.0.0.2</td>
<td>86</td>
</tr>
<tr>
<td>10.7 Create TwinCAT 3 C++ Module instance</td>
<td>87</td>
</tr>
<tr>
<td>10.8 TwinCAT 3 enable C++ debugger</td>
<td>89</td>
</tr>
<tr>
<td>10.9 Create a TwinCAT task and apply it to the module instance</td>
<td>90</td>
</tr>
<tr>
<td>10.10 Activating a TwinCAT 3 project</td>
<td>92</td>
</tr>
<tr>
<td>10.11 TwinCAT 3 C++ Implement project Online Change</td>
<td>94</td>
</tr>
<tr>
<td>11 Debugging</td>
<td>96</td>
</tr>
<tr>
<td>11.1 Details of Conditional Breakpoints</td>
<td>99</td>
</tr>
<tr>
<td>11.2 Visual Studio tools</td>
<td>101</td>
</tr>
<tr>
<td>12 Wizards</td>
<td>104</td>
</tr>
<tr>
<td>12.1 TwinCAT C++ Project Wizard</td>
<td>104</td>
</tr>
<tr>
<td>12.2 TwinCAT Module Class Wizard</td>
<td>105</td>
</tr>
<tr>
<td>12.3 TwinCAT Module Class Editor (TMC)</td>
<td>108</td>
</tr>
<tr>
<td>12.3.1 Overview</td>
<td>110</td>
</tr>
<tr>
<td>12.3.2 Basic Information</td>
<td>111</td>
</tr>
<tr>
<td>12.3.3 Data Types</td>
<td>112</td>
</tr>
<tr>
<td>12.3.4 Modules</td>
<td>129</td>
</tr>
<tr>
<td>12.4 TwinCAT Module Instance Configurator</td>
<td>151</td>
</tr>
<tr>
<td>12.4.1 Object</td>
<td>152</td>
</tr>
<tr>
<td>12.4.2 Context</td>
<td>153</td>
</tr>
<tr>
<td>12.4.3 Parameter (Init)</td>
<td>153</td>
</tr>
<tr>
<td>12.4.4 Data Area</td>
<td>154</td>
</tr>
<tr>
<td>12.4.5 Interfaces</td>
<td>154</td>
</tr>
<tr>
<td>12.4.6 Interface Pointer</td>
<td>154</td>
</tr>
<tr>
<td>12.4.7 Data Pointer</td>
<td>155</td>
</tr>
<tr>
<td>12.5 Customer-specific project templates</td>
<td>155</td>
</tr>
<tr>
<td>12.5.1 Overview</td>
<td>155</td>
</tr>
<tr>
<td>12.5.2 Files involved</td>
<td>156</td>
</tr>
<tr>
<td>12.5.3 Transformations</td>
<td>157</td>
</tr>
<tr>
<td>12.5.4 Notes on handling</td>
<td>158</td>
</tr>
<tr>
<td>13 Programming Reference</td>
<td>161</td>
</tr>
<tr>
<td>13.1 TwinCAT C++ Project properties</td>
<td>162</td>
</tr>
<tr>
<td>13.1.1 Tc SDK</td>
<td>164</td>
</tr>
<tr>
<td>13.1.2 Tc Extract Version</td>
<td>165</td>
</tr>
<tr>
<td>13.1.3 Tc Publish</td>
<td>165</td>
</tr>
<tr>
<td>13.1.4 Tc Sign</td>
<td>166</td>
</tr>
<tr>
<td>13.2 File Description</td>
<td>167</td>
</tr>
<tr>
<td>13.2.1 Compilation procedure</td>
<td>169</td>
</tr>
<tr>
<td>13.3 Online Change</td>
<td>169</td>
</tr>
<tr>
<td>13.4 Limitations</td>
<td>171</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>-----</td>
</tr>
<tr>
<td>13.5 Memory Allocation</td>
<td>172</td>
</tr>
<tr>
<td>13.6 Interfaces</td>
<td>173</td>
</tr>
<tr>
<td>13.6.1 Return values</td>
<td>174</td>
</tr>
<tr>
<td>13.6.2 Interface ITcCyclic</td>
<td>175</td>
</tr>
<tr>
<td>13.6.3 Interface ITcCyclicCaller</td>
<td>176</td>
</tr>
<tr>
<td>13.6.4 Interface ITcFileAccess</td>
<td>178</td>
</tr>
<tr>
<td>13.6.5 Interface ITcFileAccessAsync</td>
<td>186</td>
</tr>
<tr>
<td>13.6.6 Interface ITcIoCyclic</td>
<td>187</td>
</tr>
<tr>
<td>13.6.7 Interface ITcIoCyclicCaller</td>
<td>189</td>
</tr>
<tr>
<td>13.6.8 ITComOnlineCyclic interface</td>
<td>191</td>
</tr>
<tr>
<td>13.6.9 Interface ITComObject</td>
<td>192</td>
</tr>
<tr>
<td>13.6.10 ITComObject interface (C++ convenience)</td>
<td>197</td>
</tr>
<tr>
<td>13.6.11 Interface ITcPostCyclic</td>
<td>198</td>
</tr>
<tr>
<td>13.6.12 Interface ITcPostCyclicCaller</td>
<td>199</td>
</tr>
<tr>
<td>13.6.13 Interface ITcRTTask</td>
<td>201</td>
</tr>
<tr>
<td>13.6.14 Interface ITcTask</td>
<td>202</td>
</tr>
<tr>
<td>13.6.15 Interface ITcTaskNotification</td>
<td>206</td>
</tr>
<tr>
<td>13.6.16 Interface ITcUnknown</td>
<td>207</td>
</tr>
<tr>
<td>13.7 Runtime Library (RtIR0.h)</td>
<td>209</td>
</tr>
<tr>
<td>13.8 ADS Communication</td>
<td>210</td>
</tr>
<tr>
<td>13.8.1 AdsReadWriteDeviceInfo</td>
<td>210</td>
</tr>
<tr>
<td>13.8.2 AdsRead</td>
<td>212</td>
</tr>
<tr>
<td>13.8.3 AdsWrite</td>
<td>214</td>
</tr>
<tr>
<td>13.8.4 AdsReadWrite</td>
<td>216</td>
</tr>
<tr>
<td>13.8.5 AdsReadState</td>
<td>218</td>
</tr>
<tr>
<td>13.8.6 AdsWriteControl</td>
<td>220</td>
</tr>
<tr>
<td>13.8.7 AdsAddDeviceNotification</td>
<td>222</td>
</tr>
<tr>
<td>13.8.8 AdsDelDeviceNotification</td>
<td>224</td>
</tr>
<tr>
<td>13.8.9 AdsDeviceNotification</td>
<td>226</td>
</tr>
<tr>
<td>13.9 Mathematical Functions</td>
<td>227</td>
</tr>
<tr>
<td>13.10 Time Functions</td>
<td>229</td>
</tr>
<tr>
<td>13.11 STL / Containers</td>
<td>229</td>
</tr>
<tr>
<td>13.12 Error messages - understanding</td>
<td>229</td>
</tr>
<tr>
<td>13.13 Module messages for the Engineering (logging / tracing)</td>
<td>230</td>
</tr>
<tr>
<td>14 How to...?</td>
<td>234</td>
</tr>
<tr>
<td>14.1 Using the Automation Interface</td>
<td>234</td>
</tr>
<tr>
<td>14.2 Windows 10 as target system up to TwinCAT 3.1 Build 4022.2</td>
<td>234</td>
</tr>
<tr>
<td>14.3 Publishing of modules</td>
<td>234</td>
</tr>
<tr>
<td>14.4 Publishing modules on the command line</td>
<td>235</td>
</tr>
<tr>
<td>14.5 Clone</td>
<td>235</td>
</tr>
<tr>
<td>14.6 Access Variables via ADS</td>
<td>236</td>
</tr>
<tr>
<td>14.7 TcCallAfterOutputUpdate for C++ modules</td>
<td>236</td>
</tr>
<tr>
<td>14.8 Order determination of the execution in a task</td>
<td>236</td>
</tr>
<tr>
<td>14.9 Setting version/vendor information</td>
<td>237</td>
</tr>
<tr>
<td>14.10 Renaming TwinCAT C++ projects</td>
<td>238</td>
</tr>
<tr>
<td>14.11 Delete Module</td>
<td>240</td>
</tr>
</tbody>
</table>
# Table of contents

## 15 Troubleshooting

- **15.1** Build - "Cannot open include file ntddk.h"... 253
- **15.2** Build - "The target ... does not exist in the project"... 253
- **15.3** Debug - "Unable to attach"... 254
- **15.4** Activation – "invalid object id" (1821/0x71d)...
  255
- **15.5** Error message - VS2010 and LNK1123/COFF...
  255
- **15.6** Using C++ classes in TwinCAT C++ module... 255
- **15.7** Using afxres.h... 255

## 16 C++-samples

- **16.1** Overview... 257
- **16.2** Sample01: Cyclic module with IO... 259
- **16.3** Sample02: Cyclic C++ logic, which uses IO from the IO Task... 260
- **16.4** Sample03: C++ as ADS server...
  260
  - **16.4.1** Sample03: TC3 ADS Server written in C++... 261
  - **16.4.2** Sample03: ADS client UI in C#...
  265
- **16.5** Sample05: C++ CoE access via ADS... 269
- **16.6** Sample06: UI-C#-ADS client uploading the symbolic from module... 270
- **16.7** Sample07: Receiving ADS Notifications... 275
- **16.8** Sample08: provision of ADS-RPC... 276
- **16.9** Sample10: module communication: Using data pointer...
  279
- **16.10** Sample11: module communication: PLC module invokes method of C-module...
  280
  - **16.10.1** TwinCAT 3 C++ module providing methods...
  281
  - **16.10.2** Calling methods offered by another module via PLC...
  296
- **16.11** Sample11a: Module communication: C module calls a method of another C module... 308
- **16.12** Sample12: module communication: Using IO mapping... 308
- **16.13** Sample13: Module communication: C-module calls PLC methods... 309
- **16.14** Sample19: Synchronous File Access... 312
- **16.15** Sample20: FileIO-Write... 313
- **16.16** Sample20a: FileIO-Cyclic Read / Write...
  313
- **16.17** Sample22: Automation Device Driver (ADD): Access DPRAM... 314
- **16.18** Sample23: Structured Exception Handling (SEH)...
  316
- **16.19** Sample25: Static Library...
  318
- **16.20** Sample26: Order of execution in a task...
  319
- **16.21** Sample30: Timing Measurement...
  321
- **16.22** Sample31: Functionblock TON in TwinCAT3 C++... 322
- **16.23** Sample35: Access Ethernet...
  323
- **16.24** Sample37: Archive data...
  324
- **16.25** TcCOM samples...
  325
## Table of contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>16.25.1</td>
<td>TcCOM_Sample01_PlcToPlc</td>
<td>325</td>
</tr>
<tr>
<td>16.25.2</td>
<td>TcCOM_Sample02_PlcToCpp</td>
<td>335</td>
</tr>
<tr>
<td>16.25.3</td>
<td>TcCOM_Sample03_PlcCreatesCpp</td>
<td>339</td>
</tr>
<tr>
<td>17</td>
<td>Appendix</td>
<td>344</td>
</tr>
<tr>
<td>17.1</td>
<td>ADS Return Codes</td>
<td>344</td>
</tr>
<tr>
<td>17.2</td>
<td>Retain data</td>
<td>349</td>
</tr>
<tr>
<td>17.3</td>
<td>Creating and handling C++ projects and modules</td>
<td>352</td>
</tr>
<tr>
<td>17.4</td>
<td>Creating and handling TcCOM modules</td>
<td>355</td>
</tr>
</tbody>
</table>
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®, EtherCAT G®, EtherCAT G10®, EtherCAT P®, Safety over EtherCAT®, TwinSAFE®, XFC®, und XTS® and XPlanar®, 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:


with corresponding applications or registrations in various other countries.

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!

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>⚠️ DANGER</td>
<td>Serious risk of injury! Failure to follow the safety instructions associated with this symbol directly endangers the life and health of persons.</td>
</tr>
<tr>
<td>⚠️ WARNING</td>
<td>Risk of injury! Failure to follow the safety instructions associated with this symbol endangers the life and health of persons.</td>
</tr>
<tr>
<td>⚠️ CAUTION</td>
<td>Personal injuries! Failure to follow the safety instructions associated with this symbol can lead to injuries to persons.</td>
</tr>
<tr>
<td></td>
<td>Damage to the environment or devices Failure to follow the instructions associated with this symbol can lead to damage to the environment or equipment.</td>
</tr>
</tbody>
</table>

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 [20] and Preparation [22]. Limitations are documented here [171].

- **Quick start [62]**
  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 [38]**
  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 [104]**
  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 [161]**
  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 …? [234]** Chapter contains useful hints while working with TwinCAT C++.

- **Samples [257]**
  Some Interfaces and their usage is best described by working code, which is provided as download including source code and solution.
3 Introduction

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

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

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

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

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

Modularization

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

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

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

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

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

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

The best of both worlds

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

The benefits of centralized control technology:

- low overall costs
- available
Introduction

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

are decisive arguments.

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

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

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

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

**TwinCAT 3 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.
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 [45]" 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:

  ![Visual C++ option selected](image)

  - 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:

  ![Desktop development with C++ option selected](image)

- Microsoft Windows Driver Kit (WDK)
  To implement and debug C++ modules, Microsoft Windows Driver Kit (WDK) must be installed:
  Installing "Windows Driver Kit" (WDK) [22].

- 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 [1], which requires a certificate [1] 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. You only have to carry out these steps once:

• Install Microsoft Windows Driver Kit (WDK) [22] and
• Configure the TwinCAT Basis [25] as well as the configuration and platform [26] toolbar.
• On x64 PCs, sign modules so that they can be run; see Documentation for the setup of a test signing [26].
• If the operating system of the target system requires enhanced validation for drivers, for example, through a SecureBoot, carry out a corresponding signing [32] with Microsoft.

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

**NOTE**
WDK installation from 4024.0 optional
The WDK installation described here is optional as of TwinCAT 3.1 Build 4024.0.

Overview
The Microsoft Windows Driver Kit (WDK) enables the development of Windows kernel drivers. TwinCAT C++ drivers are based on this WDK.

• **TwinCAT 3.1 4022:**
  The implementation of TwinCAT 3 C++ modules requires parts of the Windows Driver Kit 7 (WDK 7) up to TwinCAT 3.1 4022.

• **TwinCAT 3.1 4024:**
  From Build 4024, the WDK is no longer required, as TwinCAT provides the required components. If the environment variable WINDDK7 (as described below) exists, the WDK is used, otherwise not. The use of the WDK can also be configured project-related via the Programming Reference [161].

Installation of WDK 7 for TwinCAT 3.1. 4022 and earlier
The installation is only necessary for the TwinCAT 3 engineering environment in order to be able to create and edit C++ modules. The WDK installation is not required on the target system (XAR).

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 of the ISO image that has been downloaded or burnt on CD (on Windows 7 PCs, start the installation with Run As Administrator...).
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, but for now it must be done manually: Navigate to **Start**-> **Control Panel** -> **System** and select **Advanced system settings**.

8. Select the **Advanced** tab and then click on **Environment Variables**...
9. 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.

10. 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

**Efficient engineering through TwinCAT XAE base toolbar**

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

1. Open the **View** menu and select **Toolbars\TwinCAT XAE Base**
5.3 Prepare Visual Studio - Configuration and Platform toolbar

Add the toolbar "Solution Configuration and Solution Platform"

With the Configuration and Platform toolbar you can specify the target platform for the creation of your project. This toolbar is automatically added to the Visual Studio menu by the TwinCAT 3 setup. If you wish to add it manually, however, do the following:

1. Open the View menu and select Toolbars\Customize.
2. Navigate to the Commands tab.
3. Activate the Toolbar option field and then select the Standard toolbar from the list.
4. Click on Add Command...
5. Select the Build category, select the Solution Configurations command and then click on OK.
6. Repeat the last step for the Solution Platforms command.
7. Click on Close.

The two commands now appear below the menu bar.

5.4 Driver signing

TwinCAT C++ modules must be signed with a certificate so that they can be executed.

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

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

Engineering requires no signing

Only the execution requires certificates - the engineering does not.
There are two ways to load modules, different certificates are used for signing:

- Operating system: The C++ modules are loaded as normal kernel drivers and must therefore also have a signature.
  - With TwinCAT 3.1. 4022 or earlier, only this procedure is available.
  - Windows 7 (Embedded) x86 (32bit) does not require signing.
- TwinCAT: The C++ modules are loaded by the TwinCAT runtime system and must be signed with a TwinCAT user certificate.
  - With TwinCAT 3.1. 4024 and higher, this method is also available.
  - This procedure is required to perform new functions such as versioned C++ projects [50] and thus also the C++ Online Change [169].

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

Organizational separation of development and production software

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

1. A certificate which is not countersigned, thus the test mode is needed for the development process. This certificate can also be issued individually by each developer.
2. Only the software that has passed the corresponding final tests is signed by a countersigned certificate. This software can thus also be installed on machines and delivered.

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

5.4.1 Operating system

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

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

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

To use such a certificate in TwinCAT 3, configure the step after compiling your x64 build target as documented in “Creating a test certificate for test mode [28].”

Test certificates

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

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

- Creating a test certificate for test mode [28]
- Delete (test) certificates [30]

Further references:

MSDN, test certificates (Windows driver),

MSDN, MakeCert test certificates (Windows driver),
5.4.1.1 Test signing

Overview

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

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

Note the procedure when creating test certificates

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

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

- **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 **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 command in the Visual Studio 2010 / 2012 prompt with administrator rights (see note above):
   
   ```
   makecert -r -pe -ss PrivateCertStore -n CN=MyTestSigningCert MyTestSigningCert.cer
   ```
   
   *(If you do not have access rights to the PrivateCertStore, you can use a different location. This must also be used in the PostBuild event, as described here.)*

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

   *Check the result with mmc (Use File->Add/Remove Snap-in->Certificates):*

   ![Certificate Manager](image)

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

   **Note** From TwinCAT 3.1 4024.0, the configuration of the certificate to be used is carried out under Tc Sign in the project properties.

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

4. Execute the following using the "Run as administrator" option:
   ```
   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.

5. You may have to switch off "SecureBoot" for this, which can be done in the bios.

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

Compilation of the x64 driver generates the following output:

References:
MSDN, test certificates (Windows driver)
MSDN, MakeCert test certificates (Windows driver),

Also see about this
- TwinCAT C++ Project properties [162]
- Tc Sign [166]
5.4.1.2 Delete test certificate

This article is about how to delete a test certificate.

Overview

A certificate can be deleted with the Microsoft Management Console:

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

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

⇒ The certificates are listed in the node under **PrivateCertStore/Certificates**.
3. Select the certificate to be deleted.

5.4.1.3 Customer Certificates

Configuration Certificate

From TwinCAT 3.1 4024.0, the configuration of the certificate to be used is carried out under **Tc Sign** in the project properties.

If the TwinCAT C++ class wizard is used, the project is prepared for x64 targets using the test certificate procedure described above.

This test signing system can be used for the entire engineering and test process.

If you want to create an infrastructure and sign the kernel drivers with official "Microsoft trusted" certificates, the post-build events of the project properties provide the entry point.
The customer can simply replace the value of the environment variable `TWINCATTESTCERTIFICATE` or determine another certificate to be used. The customer can also change the whole signing process with the signing tool. In this case a CrossSigning is necessary for the Windows drivers. The parameter for the signtool is `/ac`. The certificate provider provides the necessary information. Microsoft provides an overview here.

### 5.4.1.4 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, the development process can be simplified on corresponding test systems by deactivating SecureBoot.

### 5.4.2 TwinCAT

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

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

For signing a TMX file, a TwinCAT user certificate is required [33], which is configured accordingly in the project for signing.

The TMX file must then be loaded by the TwinCAT Loader [54], for which corresponding settings are made in the project during the construction process.
Note: In contrast to the operating system signature, TwinCAT signing is intended for both 32bit and 64bit systems. Thus, the test mode is assumed for a test signature also on 32-bit systems.

Test signing

Just as with the option of loading drivers through the operating system, TwinCAT also provides the test mode of the operating system for loading.

As soon as the TwinCAT user certificate has been countersigned by Beckhoff [34], the test mode can be dispensed with accordingly.

5.4.2.1 Test signing

The test signature for TwinCAT can be carried out with the same TwinCAT user certificate as for the actual delivery (see Driver signing [27]).

1. For test operation, e.g. during software development, the creation of a TwinCAT user certificate, as described here, is sufficient. Make sure that you select the purpose "Sign TwinCAT C++ executable (*.tmx)". For this the Crypto version 2 is required, a message appears.

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

3. Execute the following using the "Run as administrator" option:

   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. You may have to switch off "SecureBoot" for this, which can be done in the bios.

5. During the initial activation (Activate Configuration) with a TwinCAT user certificate, the target system will determine that the certificate is not trusted and the activation process will be aborted:

A local user with administrator rights can trust the certificate through simple running via the created REG file.

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

6. Following this process you can use the TwinCAT user certificate for signing with the test mode of the operating system.

This is configured in the project properties [166].

Use the TcSignTool [58] to avoid storing the password of the TwinCAT user certificate in the project, where it would also end up in version management, for example.

If you have this TwinCAT user certificate countersigned by Beckhoff, you can also use it for delivery without TestMode.

5.4.2.2 Signing for delivery

If you create TwinCAT driver software for delivery, you should use a countersigned TwinCAT user certificate.
With a countersigned TwinCAT user certificate, the test mode is no longer required.
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:

- ITcCyclic is implemented by modules, which are called cyclically ("CycleUpdate"). The module can register via the ITcCyclicCaller interface of a TwinCAT task to obtain cyclic calls.
- The ITcADI interface can be used to access data areas of a module.
- ITcWatchSource is implemented by default; it facilitates ADS 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.
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 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 ITCtRTime 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.

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

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 [139] 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 [308]**

**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.

**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 [131]), 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 [❯ 280]
Sample11a: Module communication: C-module cites a method in the C-module [❯ 308]

Further samples exist for the communication with the PLC [❯ 325].

**ADS**

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

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

The following samples show the realization:

Sample03: C++ as ADS server [❯ 260]
Sample06: UI-C#-ADS client uploads the symbols from the module [❯ 270]
Sample07: reception of ADS notifications [❯ 275]
Sample08: provision of ADS-RPC [❯ 276]
7 Modules - Handling

TcCOM modules are implemented and loaded after a build.

This section describes the handling of modules when they are exchanged between systems.

A distinction must be made between the two C++ project types:

- **C++ projects** that create a sys file to load them by the operating system.
- **Versioned C++ projects** that show a tmx file to load them with the TwinCAT Loader (from TwinCAT 3.1 Build 4024).

**Also see about this**
- Export to TwinCAT 3.1 4022.xx
- Import up to TwinCAT 3.1 4022.xx

7.1 Versioned C++ Projects

**From TwinCAT 3.1 Build 4024.0**

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

Versioned TwinCAT C++ projects result in an architecture-dependent TMX file during building and are loaded via the TwinCAT Loader. They must be signed by a TwinCAT user certificate.

If a C++ project was created using the template "Versioned C++ Project", the binary files are stored by a publish in the TwinCAT repository under C:\TwinCAT\3.x\Repository at a vendor- and version-specific location.

From here required modules are transferred to the target system under C:\TwinCAT\3.1\Boot\Repository, if they are needed.

This can be either at the time of activation (Activate Configuration) or at the time of the Online Change.

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

7.2 Non-versioned C++ projects

TwinCAT C++ drivers (.sys files) are loaded via the Windows operating system. These are kernel-mode drivers which are subject to the normal requirements of the operating system with regard to loading.

If a C++ project has been created using the TwinCAT Driver Project template, the binary files are stored in the TwinCAT folder under C:\TwinCAT\3.x\CustomConfig\Modules by a publish.

From here the driver is transferred to the target system under C:\TwinCAT\3.x\Driver if it is needed.

Additionally, it is possible to create an archive for the transfer of the binary version of this project, which is configured by the project properties.

**Up to TwinCAT 3.1 4022.xx**

Before Release 4024.0, the handling of the export and import functionality was somewhat different, which is documented on the subpages.
7.2.1 Export to TwinCAT 3.1 4022.xx

This article describes how to export a TwinCAT 3 driver that can run on any other TwinCAT PC.

The following steps have to be carried out

1. Implement a TwinCAT 3 C++ project on an engineering PC equipped with a Visual Studio version, see quick start sample [Create a TwinCAT 3 project][62]. Implement the TwinCAT modules as described, compile and test the modules contained in the project before export.

2. Since the result should be able to be used on any machine, TwinCAT generates a 32-bit and a 64-bit version.
   Since x64 drivers must be signed, a certificate must be installed on the machine that exports the module. See x64: Driver signing [26], how to generate and install a certificate.
   (This step can be omitted on an engineering or 32-bit system)

3. To export a TC 3 C++ project, right-click on the module project in the solution tree and select TwinCAT Publish Modules.
   - The project is then compiled (rebuild) - the successful export is displayed in the Build output window.

Please note the successful message at the end:

```
Done building project "TempContr.vcxproj".
Done building project "TempContr.vcxproj" (TcPublishAdditionalFiles target(s)).
Done building project "TempContr.vcxproj".
```

The binary files and the TMC module description are exported to the TempContr folder under C:\TwinCAT\3.x\CustomConfig\Modules.
4. For the import, simply copy the TempContr folder to any other TwinCAT 3 machine.

7.2.2 Import up to TwinCAT 3.1 4022.xx

This article describes how a TC3-C++ driver can be imported and integrated into a PC/IPC controller with TwinCAT 3 XAE (without full version of Visual Studio). The binary TC3-C++ driver was previously implemented and exported on another PC. The following steps have to be carried out

1. Copy the TC3-C++ driver on the second IPC with TwinCAT XAE without the full version of Visual Studio into the destination folder ..\TwinCAT\3.x\CustomConfig\Modules. The TestDriver.zip archive is unpacked in this sample.
The TestDriver (in the subfolders RT and UM) and the corresponding TwinCAT module Class *.tmc file TestDriver.tmc are then available.

2. Start the TwinCAT XAE environment and create a TwinCAT 3 project.
3. Right-click System->TcCOM Objects and select Add New Item....

The new CTestModule module is listed in the dialog box that appears.

4. Create a module instance by selecting the module name and continue with OK.

The instance of the TestModule module now appears under TcCom Objects.

5. Create a new task.
6. Go to the context of the module instance and link the C++ module instance with the previously added Task 1.
7. Activate the configuration.

7.3 Starting Modules

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

- Operating system: The operating system starts the TwinCAT module as a normal driver.
  - On x64bit-PCs the operating system requires a signature, see Driver signing [26].
- **TwinCAT Loader** [54]: The TwinCAT Loader starts the TwinCAT module.
  - The TwinCAT Loader requires a signature [26] with TwinCAT user certificate.
  - This option is mandatory for encrypted modules [56].
  - The TwinCAT Loader is required for the versioned C++ projects [104].

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

![Class Factories Tab](image)

Also see about this

- Driver signing [27]

### 7.4 TwinCAT Loader

**From TwinCAT 3.1 Build 4024.0**

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

TwinCAT 3 has an integrated function for loading modules.

Modules loaded with the TwinCAT Loader

- **must** be signed: Test signing [54]
- **can** be encrypted: Encrypting Modules [56], for which the TwinCAT software Protection must be configured with a user DB.

#### 7.4.1 Test signing

The test signature for TwinCAT can be carried out with the same TwinCAT user certificate as for the actual delivery (see Driver signing [27]).
1. For test operation, e.g. during software development, the creation of a TwinCAT user certificate, as described here, is sufficient. Make sure that you select the purpose "Sign TwinCAT C++ executable (*.tmx)". For this the Crypto version 2 is required, a message appears.

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

3. Execute the following using the "Run as administrator" option:
   ```
   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. You may have to switch off "SecureBoot" for this, which can be done in the bios.

5. During the initial activation (Activate Configuration) with a TwinCAT user certificate, the target system will determine that the certificate is not trusted and the activation process will be aborted:

A local user with administrator rights can trust the certificate through simple running via the created REG file.

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

6. Following this process you can use the TwinCAT user certificate for signing with the test mode of the operating system.

This is configured in the project properties [166].

Use the TcSignTool [58] to avoid storing the password of the TwinCAT user certificate in the project, where it would also end up in version management, for example.

If you have this TwinCAT user certificate countersigned by Beckhoff, you can also use it for delivery without TestMode.

### 7.4.2 Encrypting Modules

TwinCAT C++ modules loaded via the TwinCAT Loader (TMX files) can be encrypted, i.e. a key protects the content of the driver against manipulation and reverse engineering at file level.
No debugging

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

Module encryption is enabled as follows:

- The TwinCAT software protection must be configured.
- A TwinCAT user certificate with Sign UserDB rights is required.

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

2. Select the C++ project and activate encryption there:

3. To start, an encrypted module must be loaded with the TwinCAT Loader (not the operating system).
   - For non-versioned drivers: The drivers are encrypted during transfer to the _deployment directory of the project.
   - For versioned TMX: The drivers are stored unencrypted in XAE and encrypted when they are activated on the target system.
   - If the function is used with versioned C++ projects, the TMX files are stored in the repository [50] as usual.

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

- Operating system: The operating system starts the TwinCAT module as a normal driver.
  - On x64bit-PCs the operating system requires a signature, see Driver signing [26].
- TwinCAT Loader [54]: The TwinCAT Loader starts the TwinCAT module.
  - The TwinCAT Loader requires a signature [26] with TwinCAT user certificate.
  - This option is mandatory for encrypted modules [56].
The TwinCAT Loader is required for the versioned C++ projects [104].

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

Also see about this
[ ] Driver signing [27]

### 7.4.3 Return Codes

Loading a module with the TwinCAT Loader can fail for various reasons. Here is a list of the return codes:

<table>
<thead>
<tr>
<th>Hex</th>
<th>Dec</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xC1</td>
<td>193</td>
<td>File is corrupted; PE file checksum error</td>
</tr>
<tr>
<td>0x241</td>
<td>577</td>
<td>Signature error: TwinCAT user certificate does not match the file hash</td>
</tr>
<tr>
<td>0x4FB</td>
<td>1275</td>
<td>Signature error: File not signed</td>
</tr>
<tr>
<td>0x1772</td>
<td>6002</td>
<td>File is encrypted, but cannot be decoded with known keys.</td>
</tr>
</tbody>
</table>

### 7.4.4 TcSignTool - Storage of the certificate password outside the project

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

The TcSignTool is a command line program located in the path `C:\TwinCAT\3.x\sdk\Bin`.

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

```
tcsigntool grant /f "C:\TwinCAT\3.1\CustomConfig\Certificates\MyCertificate.tccert" /p MyPassword
```

The password is deleted with the following parameters:

```
tcsigntool grant /f "C:\TwinCAT\3.1\CustomConfig\Certificates\MyCertificate.tccert" /r
```

The unencrypted password is stored under `HKEY_CURRENT_USER\SOFTWARE\Beckhoff\TcSignTool\`
8 TwinCAT C++ development

Overview of the development environment

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

1. In the TwinCAT solution, a TwinCAT C++ project can be created by right-clicking on the C++ icon. This project contains the sources (“Untitled Project”) of perhaps several modules [36], 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 [47] 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 / tracking) [230]).

3. The freely configurable toolbar usually contains the toolbar for TwinCAT XAE Base. Activate Configuration, RUN, CONFIG. Choose Target System (in this case <Local>) and several other buttons provide fast access to frequently used functions. The TwinCAT Debugger is the button for establishing a connection to the target system with regard to C++ modules (the PLC uses an independent debugger). Like in other C++ programs, and in contrast to PLC, in TwinCAT C++ a distinction has to be made between "Release" and "Debug". In a build process for "Release", the code is optimized to such an extent that a debugger may no longer reliably reach the breakpoints, and incorrect data may be displayed.

Procedure

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

It provides a general overview of the engineering process for TwinCAT C++ projects with reference to the corresponding detailed documentation. The quick start [61] guide describes the individual common steps.
1. Type declaration and module type:
   The TwinCAT Module Class Editor (TMC) [108] 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). Editing and starting the code generator can take place as often as you like – the code generation pays attention to programmed user code and saves it.

2. Programming
   The familiar Visual Studio C++ programming environment is used for the development and debugging [96] of the user-defined code within the code template.

3. Instantiating modules [36]
   The program describes a class, which is instantiated as objects. The TwinCAT Module Instance Configurator [151] is used for configuring the instance. General configuration elements are: assign task, download symbol information for runtime (TwinCAT Module Instance (TMI) file) or define parameter/interface pointer.

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 to TwinCAT 3.1 4022.xx [51] / Import up to TwinCAT 3.1 4022.xx [52])
   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 Driver signing [26])
   The TwinCAT drivers must be signed for x64 run times, since 64-bit Windows versions require that kernel modules are signed. Therefore, this applies both to the x64 creation and to the publication of modules [234], because these modules contain the x64 binary codes (if not deactivated). The signature process can be user-defined [31].

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 [96] 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!  

9.1 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.

9.2 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++ - Build Project

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


2. Alternatively, create a project by clicking on: File -> New -> Project. All existing project templates are displayed.

3. Select TwinCAT XAE Project and optionally enter a suitable project name.
4. Click 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.

9.3 **Create TwinCAT 3 C++ project**

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

1. Right-click C++ and choose Add New Item... . . .and I'm going to get out of here.
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 [►20]).

The TwinCAT C++ Project Wizard [►104] 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 Sample 25 [►318]).

The TwinCAT Module Assistant [►105] is displayed.
3. 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 TwinCAT Class Wizard dialog box or continue with the Object1 suggestion.
A TwinCAT 3 C++ project with a driver will then be created on the basis of the selected template:

9.4 Implement TwinCAT 3 C++ project

This article describes how the sample project can be changed.

The implementation begins after creating a TwinCAT C++ project and opening `<MyClass>.cpp` (**Module1.cpp** in this sample).
1. The `<MyClass>::CycleUpdate()` method is cyclically called – this is the point where the cyclic logic is to be positioned. The entire cyclic code is inserted at this point. The dropdown menus at the top of the editor can be used for navigation.

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

3. Save the modifications.

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

The new instance "Untitled1_Obj2 (CModule1)" becomes part of the TwinCAT 3 solution: the new node is located precisely under the TwinCAT 3 C++ source "Untitled1 Project".

The module already provides a simple I/O interface with 3 variables in each case:
- Input area: Value, Status, Data
- Output area: Value, Control, Data

The description of these interfaces corresponds in two places:
Quick start

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

- "TwinCAT Module Configuration”.tmc file (in this sample "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 sample the I/O image interface is provided by a C++ module instance, so that no image is necessary at the task for triggering the execution of the C++ module instance.

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).
Quick start

Configuring a TwinCAT 3 C++ module instance that is called from the task
1. Select the C++ module instance in the solution tree.
2. Select the Context tab in the right-hand working area.
3. Select the task for the previously created context in the drop-down task menu. Select the default Task 1 in the sample.

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

9.7 TwinCAT 3 enable C++ debugger

To prevent all dependencies from being loaded for debugging [96], this function is switched off by default and must be activated once before the activation of the configuration.
1. Select C++ Debugger on the C++ node of the Solution tab.
2. Select Enable C++ Debugger.
3. Switch **Enable C++ Debugger** on.

9.8 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 trial 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**.

5. For test operation, e.g. during software development, the creation of a TwinCAT user certificate, as described here, is sufficient. Make sure that you select the purpose "Sign TwinCAT C++ executable (*.tmx)". For this the Crypto version 2 is required, a message appears.

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++ module 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 unfavorable situation.

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

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

   ![TwinCAT Live Watch](image)

   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 Quick start with Online Change

This quick start shows how you can familiarize yourself with the TwinCAT C++ module engineering in a short time. In contrast to the other quick start, this section contains the possibility of Online Change.

Before the quick start, please pay attention to the preparation - just once! [22]

10.1 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++ - Build Project

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


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

#### 10.2 Create TwinCAT 3 C++ project

After creating a TwinCAT 3 project, open the C++ node and carry out the following steps:
1. Right-click **C++** and choose **Add New Item...** and I'm going to get out of here. 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 [► 20]).

2. Select **TwinCAT Versioned C++ Project**, optionally enter a project name and click **OK**. The **“TwinCAT C++ Project Wizard”** is shown and all existing project templates are listed.

3. In this case select **TwinCAT Module Class Online Changeable** and click **OK**. A name is not necessary and also cannot be entered here.
4. Enter a unique name in the **TwinCAT Class Wizard** dialog box or continue with the **Object1** suggestion.
A TwinCAT 3 C++ project with a driver will then be created on the basis of the selected template:

Also see about this
Sample25: Static Library [318]

10.3 TwinCAT 3 C++ Configure project

1. Right-click the project to open the properties.
3. If you have not yet created a TwinCAT user certificate, follow the instructions and observe to select the Sign TwinCAT C++ executeables.
4. Enter the file name of the TwinCAT user certificate and the password.
(Note that this is stored unencrypted in the solution and is therefore also loaded on servers via version control, for example. If necessary use the TcSignTool [58])
10.4 Implement TwinCAT 3 C++ project

This article describes how the sample project can be changed.

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

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

2. In this case a counter is incremented by the value of the Value variable in the input image (m_Inputs). Replace a line in order to increment the counter without dependence on the value of the input image. Replace this line

   ```cpp
   m_counter+=m_Inputs.Value;
   ```

   with this one

   ```cpp
   m_counter++;  
   ```

3. Save the modifications.
4. The version 0.0.0.1 has thus been implemented, as you can see in the TMC Editor.

10.5 Publish TwinCAT 3 C++ project in version 0.0.0.1

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:

   ![Image](image1)

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

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

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

In `<MyClass>.cpp` (in this sample `Module1.cpp`) the implementation can be changed.
1. Replace a line to decrement the counter instead of incrementing it.
   Replace this line
   
   \[ m\_counter++; \]

2. with this one
   
   \[ m\_counter--; \]

3. Save the modifications.

4. Parameterize version 0.0.0.2 in the TMC Editor.

5. Publish this version as well:

\[ \text{There are two versions of a module, which can be exchanged during runtime.} \]

**10.7 Create TwinCAT 3 C++ Module instance**

An instance of the module must be created in order to execute it. Several instances of a module can exist.
After creating a TwinCAT C++ module, open the **C++ 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**....

   ![Solution Explorer](image1)

   ⇒ All existing C++ modules are listed.

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

   ![Insert TcCom Object](image2)

   ⇒ 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 3 variables in each case:

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

The description of these interfaces corresponds in two places:
• "<Classname>Services.h" (in this sample "Untitled1Services.h")

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

10.8 TwinCAT 3 enable C++ debugger

To prevent all dependencies from being loaded for debugging [96], this function is switched off by default and must be activated once before the activation of the configuration.

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

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

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

This configuration step only has to be carried out once. No new task needs to be configured for subsequent creations/new compilations of the C++ module in the same project.
Creating a TwinCAT 3 task

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

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

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

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

![](image)

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

### 10.10 Activating a TwinCAT 3 project

Once a TwinCAT C++ project has been created, compiled and made available, the configuration must be activated:
1. Click on the symbol **Activate Configuration** – all required files for the TwinCAT project are transferred to the target system:

![TwinCAT Project Explorer](image)

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

![Activate Configuration dialog](image)

3. If you have no license on the target system, you will be offered the option to create a 7-day trial license. This can be repeated any number of times.

4. TwinCAT 3 automatically asks whether the mode should be switched to Run mode.

![Restart TwinCAT System in Run Mode](image)

- In the case of **OK**, the TwinCAT 3 project switches to Run mode.
- In the case of **Cancel**, TwinCAT 3 remains in **Config mode**.
5. For test operation, e.g. during software development, the creation of a TwinCAT user certificate, as described here, is sufficient. Make sure that you select the purpose "Sign TwinCAT C++ executable (*.tmx)". For this the Crypto version 2 is required, a message appears.

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

10.11 TwinCAT 3 C++ Implement project Online Change

Running TwinCAT C++ project as described above.

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

3. In the column **Online Version** the currently running version is preselected and marked with the extension (Current).

4. Set a different version.

5. Activate this change by right-clicking and **Apply changed online object versions** on the target.

The version change was made on the target.
11 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 must be enabled.

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

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

2. For all debugging in TwinCAT C++, connect the TwinCAT Engineering with the runtime system (XAR) via the **TwinCAT Debugger** button.

3. **Breakpoints and step-by-step execution**

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

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

---

**WARNING**

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

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

Make sure that the changed behavior of the controlled system does not cause any damage and be sure to note the plant documentation.
On reaching the breakpoint (indicated by an arrow), the execution of the code is stopped.

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

### Conditional breakpoints

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

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

**WARNING**

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

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

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

Select **Condition...** to open the condition window:
Details of the conditions and how they are to be formulated can be found here [99].

**Live Watch**

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

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

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

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

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

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

11.1 Details of Conditional Breakpoints

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

**WARNING**

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

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

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

The option buttons offer two options that are described separately.

**Option: Is true**

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

\[(\text{Maxterm}_1 \&\& \text{Maxterm}_2 \&\& \ldots \&\& \text{Maxterm}_N)\]

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

\[(\text{condition}_1 \| \text{condition}_2 \| \ldots \| \text{condition}_N)\]

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

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

Samples:

\[\text{m\_counter} == 123 \&\& \text{hr} != 0\]
\[\text{m\_counter} == 123 \| \text{m\_counter}_2 == 321 \&\& \text{hr} == 0\]
\[\text{m\_counter} == 123\]

Further comments:

- Monitoring module instances:
  The OID of the object is stored in \text{m\_objId}, so the monitoring of the OID can look like this:\n\[\text{m\_objId} == 0x01010010\]

- Monitoring of tasks:
  A special variable \#taskId is provided to access the OID of the calling task. E.g.\n\[\#\text{taskId} == 0x02010010\]
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

11.2 Visual Studio tools

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

Call stack

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

Autos / Locals and Watch

The corresponding variables and values are displayed in the Autos / Locals window when a breakpoint is reached. Changes are shown in RED.
From here, the values can be applied to the Watch windows by right-clicking:

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>123</td>
</tr>
<tr>
<td>b</td>
<td>108117</td>
</tr>
<tr>
<td>*(res)</td>
<td>108117</td>
</tr>
</tbody>
</table>

**Memory view**

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

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

• The TwinCAT Project Wizard \[\text{104}\] creates a TwinCAT C++ project. For Driver projects, the TwinCAT Class Wizard will be started afterwards.

• The TwinCAT Module Class Wizard \[\text{105}\] 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 \[\text{108}\] (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 \[\text{151}\]

12.1 TwinCAT C++ Project Wizard

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

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

   TwinCAT offers three C++ projects:
   - Driver Project: Projects containing one or more executable modules
   - Versioned C++ Projects: Projects that involve revision control. So you obtain the opportunity to switch
between versions at runtime.
- Static library: Projects with C++ functions that are used by (different) TwinCAT C++ drivers.

2. Select one of the project templates and specify a name and location.
   - The TwinCAT C++ project is created
   - In the case of a driver, the TwinCAT C++ class wizard [105] is started.

### 12.2 TwinCAT Module Class Wizard

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

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


**TwinCAT Modules Class**

Creates a new TwinCAT module class.

This is a template that 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](#280)

**TwinCAT Module Class with ADS port**

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

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

- **ADS Client:**
  - This template provides sample codes to initiate an ADS call by sending an ADS message to an ADS partner.

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

Because ADS can work across networks, the two modules can also run on different machines in the network.

See [Sample03](#260), [ADS Communication](#210)

**TwinCAT Module Class with cyclic caller**

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

Not often used, a module class with cyclic caller and cyclic I/O is preferred.
TwinCAT Module Class with cyclic input/output

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

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

Important:

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

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

See Quick start [61], sample 01 [259].

TwinCAT Module Class with data pointer

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

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

Important:

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

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

The data pointer is configured in the TwinCAT Module Instance Configurator [151].

See sample10 [279]

TwinCAT Module Class for real-time context

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

As described here [45], 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 [339] illustrates the application of such a module.

TwinCAT Module Class with Online Change capability

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

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

The procedure for this Online Change is described here [169].
The module itself otherwise corresponds to a module with cyclic input/output.

12.3 TwinCAT Module Class Editor (TMC)

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

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

The basic idea is:

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

Start the TMC editor

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

Functionalities of the TMC editor:

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

User Help

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

For example, in the event of problems (alignment, invalid standard definitions, ...) within the TMC, the user is guided to the relevant location via red 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.
- Switching the User Help [108] on/off.
- Start the TwinCAT TMC Code Generator:
The editor will store the entered information in the TMC file. The TwinCAT TMC Code Generator converts this TMC description to source code, which is also available in the context menu of the TwinCAT C++ project.

12.3.1 Overview

User interface

- **TMC [111]**: Here you can edit the C++ module vendor's basic information and add an image.
- **Data types [112]**: Data types are added, removed, or re-ordered here.
- **Modules [129]**: Shows the modules of the driver.
- **Implemented Interfaces [131]**: Shows the implemented interfaces of the module.
- **Parameters [132]**: Parameters are added, removed, or re-ordered here.
  - **TraceLevelMax [138]**: Parameter that controls the quantity of logged messages; predefined for (almost) every module.
- **Data Areas [139]**: Data areas are added, removed, or re-ordered here.
- **Data Pointers [146]**: Data pointers are added, removed, or re-ordered here.
- **Interface Pointers [148]**: Interface pointers are added, removed, or re-ordered here.
- **Deployment [149]**: Determines the files that are provided.
12.3.2 Basic Information

Basic information on the TMC file can be found here:

Information about the provider

Name: This is the name of the provider.

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

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

Versioned Classfactory

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

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

Optional features

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

Comment: Optionally, you can enter a comment here.
12.3.3 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 [280]).

**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.

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

12.3.3.1 Overview

User interface
### Data type properties

- **Name**: user-defined name of the data type.
- **GUID**: unique ID of the data type.
- **Specification**: specification of the data type.
- **Size**: size of the data type, if expressly specified.
- **Size X64**: different size of the data type for the x64 platform.

### 12.3.3.2 Add / modify / delete data types

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

This article describes:

- **Step 1**: Create a new data type [113] in the TMC file.
- **Step 2**: Start the TwinCAT TMC Code Generator [116] in order to generate C++ code on the basis of a module description in the TMC file.
- **Using** [129] the data types.

#### Step 1: Generate a new type

1. After starting the TMC Editor, select the **Data Types** node.
2. Extend the list of data types and interfaces by a new data type by clicking on the + button **Add a new data area**.

   A new **data type** is then listed as a new entry:

   ![Image of data type addition](image)

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

   ![Image of data type selection](image)

4. Specify the data type.

   See [here](#) for more precise details.

5. Rename the data type.

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

   ![Image of data type renaming](image)
6. Insert new sub-elements in the structure by clicking on the **Add a new sub item** button.

![Image showing sub-elements](image)

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

![Image showing sub-element properties](image)

8. Give the other sub-elements a new name and select a suitable data type.

9. Save the changes you have made in the TMC file.
Step 2: Start the TwinCAT TMC Code Generator to generate code for the module description.

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

   ![TwinCAT TMC Code Generator](image)

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

   ![data type declaration in Untitled1Services.h](image)

   ⇒ If you add a further data type or a further sub-element, run the TwinCAT TMC Code Generator again.
12.3.3.3 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 [117] in the TMC file.
- **Step 2:** [117]Add methods [117] to the interface in the TMC file.
- **Step 3:** Use the interface [119] by adding it to the "Implemented Interfaces" of the module.
- **Step 4:** Start the TwinCAT TMC [121] Code Generator to generate code for the module description.
- **Optional change of the interface [121].**

**Step 1: generate a new interface**

1. After starting the TMC Editor, select the Data Types node.
2. Click on Add a new interface to extend the list of interfaces by a new interface.
   - A new entry Interface1 is then listed:

   ![Interface1 in TMC Editor](image)

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

   ![Interface details in TMC Editor](image)

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

**Step 2: add methods to the interface**

5. Click on Edit Methods... to get a list of the methods of this interface:
6. Click on the + button to generate a new default method, "Method1".

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

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

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

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

    - In this case "NewState" is the new name – the rest of the settings are not changed.
11. By repeating step 2 "Add methods to interface", all methods are listed – you can re-order the methods with the help of the move up / move down button.

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. Extend the list of implemented interfaces 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**.

![Image of interface selection dialog]

- The new interface "IStateMachine" is part of the module description.
Step 4: Start the TwinCAT TMC Code Generator to generate code for the module description.

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

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

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

Optional change of the interface

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

Editing the properties of data types

- **Name**: DataType1
- **Namespace**: 
- **Guid**: b6c00cde91-3b35-43a3-8b7a-8bf0567e56a5
- **Specification**: Alias

Choose data type
- **Select**: INT
- **Description**: Normal Type

Type Information
- **Namespace**: 
- **Guid**: 18071995-0000-0000-0000-000000000008

Optional data type settings
- **Size (Bits)**: x64 specific
- **C/C++ Name**
  - **default**: 
  - **x64 specific**: 
- **Unit**: 
- **Comment**: 
  - Hide sub items
  - Persistent (avon if unused)

Optional Defaults
- **Value**: 4
  - **Enum**: 
  - **String**: 
- **Min**: 1
- **Max**: 5

Optional properties

Datatype Hideo
General properties

Name: user-defined name of the data type.

<table>
<thead>
<tr>
<th>NOTE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name conflict</td>
</tr>
<tr>
<td>A name collision can occur if the driver is used in combination with a PLC module.</td>
</tr>
<tr>
<td>• Do not use any of the keywords that are reserved for the PLC as names.</td>
</tr>
</tbody>
</table>

Namespace: user-defined namespace of the data type.

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

Sample: an enumeration with a namespace "A":

```c
#include <AutoGeneratedContent id="DataTypes">
#ifdef !_TC_TYPE_41D4A207_3A09_4316_9D89_0DD1881AB8C4_INCLUDED_
#define !_TC_TYPE_41D4A207_3A09_4316_9D89_0DD1881AB8C4_INCLUDED_
enum A_ASampleEnum : SHORT {
    One,
    Two,
    Three
};
#endif // !_TC_TYPE_41D4A207_3A09_4316_9D89_0DD1881AB8C4_INCLUDED_
```

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

```c
#include <AutoGeneratedContent id="DataTypes">
#ifdef !_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 // !_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**: create a user-defined array.
- **Enumeration**: create a user-defined enumeration.
- **Struct**: generate a user-defined structure.
- **Interface**: generate a new interface.

Select data type

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

Description: Define the type as pointer, reference or value by means of the appropriate selection.
- Normal type
- Pointer
- Pointer to pointer
- Pointer to pointer to pointer
- a reference

**Type information**

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

**Optional data type settings**

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

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

**Unit**: a unit of the variable.

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

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

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

**Optional Defaults**

Depending on data type the default could be defined.

**Optional Properties**

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

**Datatype Hides**

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

**12.3.3.5 Specification**

This section describes the Specification of data types.
12.3.3.5.1 Array

Array: create a user-defined array.

Define the dimensions of the array

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

Dimension: Dimension of the array.

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

Elements: Quantity of elements.

Dynamic arrays for parameters and data pointers

Define the dimensions of the array

In the case of parameters [132] and data pointers [146], TwinCAT 3 supports arrays with a dynamic size.

Min: Minimum size of the array.

Max: Maximum size of the array.

Max is unbounded: indicates that there is no upper limit for the array size.
12.3.3.5.2 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.

12.3.3.5.3 Struct

Struct: Creating a user-defined structure.

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

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

Name: Name of the element.
Specification: A struct can contain aliases, arrays or enumerators.

**Type**: Type of the variable.

**Size**: Size and offset of the sub-element.

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

**Unit**: Optional unit.

The details of the configuration page of the sub-element are shown by selecting the data type or double-clicking on the table entry. Similar to Data type properties [123].

### 12.3.3.5.4 Interfaces

**Interfaces**: Creating a user-defined interface.

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

**Method parameters**

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

RPC enable: enablement of remote procedure calls from outside this method.

Include Return Value: enablement of the forwarding of the return value of the method.

The fields correspond to those of the Data type properties [123].

Defining the method parameters

- Name:
- Type: Known from the Data type properties [123].
- Description: Known from the Data type properties [123].
- Default Value: Default value of this parameter; only numbers are allowed.
- RPC direction: As in the case of PLC function blocks, each parameter can either be IN, OUT or INOUT. Over and above that, it can be defined as NONE so that this parameter is ignored in the case of remote procedure calls (RPC).

12.3.4 Modules

Modules: Shows the modules of the driver.
**Class Name:** Name of the module.

**Class ID:** Unique module ID.

**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 [152] 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 [105]
Defining the contexts of the module

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

Optional Properties

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

12.3.4.1 Implemented Interfaces

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

Name: Name of the interface.

Interface ID: Unique ID of the interface.

Disable Code Generation: Enable/disable the code generation.

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

A TcCOM module instance is defined through various parameters.

TwinCAT supports three types of Parameter IDs (PTCID) in the section Configuring the parameter ID [137].

- "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 [153] 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.
### 12.3.4.2.1 Add / modify / delete parameters

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

This article describes:

- **Step 1:** Create a new parameter [133] in the TMC file.
- **Step 2:** Start the TwinCAT TMC Code Generator [134] to generate code for the module description in the TMC file.
- **Step 3:** Observe the transitions of the state machine [135]

#### Step 1: Create new parameters

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

### Symbols and Functions

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Add a new parameter</td>
</tr>
<tr>
<td>-</td>
<td>Deletes the selected type</td>
</tr>
<tr>
<td>↓</td>
<td>Moves the selected element down one position</td>
</tr>
<tr>
<td>↑</td>
<td>Moves the selected element up one position</td>
</tr>
<tr>
<td>8 Byte</td>
<td>Select byte alignment</td>
</tr>
<tr>
<td></td>
<td>Align the selected data type</td>
</tr>
<tr>
<td></td>
<td>Reset data format of the selected data type</td>
</tr>
</tbody>
</table>

**Name**: Name of the interface.

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

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

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

**Context**: Context ID of the parameter.

**Disable Code Generation**: Enable/disable the code generation.
3. Select **Parameter** in the left-hand tree or double-click on the red-marked "Parameter3" or select the node in the tree to obtain details of the new parameter.

4. Configure the parameter as well as the **Data Types**.

5. Give it a more meaningful name – in this sample "bEnable" – and select the data type "BOOL".

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

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

7. Right-click on your project file and select **TwinCAT TMC Code Generator** to receive the parameters in your source code:
You can see the parameter declaration in the header file "Module1.h" of the module.

```c
///<AutoGeneratedContent id="Members">
TcTraceLevel m_TraceLevelMax;
bool m_bEnable;
ModuleInputs m_Inputs;
ModuleOutputs m_Outputs;
ModuleDataArea m_DataArea;
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".

```c
IMPLEMENT_IUnknown(Module1)
IMPLEMENT_IUnknown_SETSTATE_Lock2(Module1)
IMPLEMENT_IUnknown2(Module1)
IMPLEMENT_IUnknown_WATCHSOURCE(Module1)
```

```c
(IConfiguration* tc, TcSysTime ct, IUnknown* unk)
```

```c
BEGIN_SETOBJECTPARA_MAP(Module1)
  SETOBJECTPARA_DATAREGISTRY_MAP()
  SETOBJECTPARA_VALUE(PID_TcTraceLevel, m_TraceLevelMax)
  SETOBJECTPARA_VALUE(PID_Module1Enable, m_bEnable)
  SETOBJECTPARA_ITFRET(PID_Ctx_TaskId, m_spCyclicCaller)
END_SETOBJECTPARA_MAP()
```

```c
BEGIN_GETOBJECTPARA_MAP(Module1)
  GETOBJECTPARA_DATAREGISTRY_MAP()
  GETOBJECTPARA_VALUE(PID_TcTraceLevel, m_TraceLevelMax)
  GETOBJECTPARA_VALUE(PID_Module1Enable, m_bEnable)
  GETOBJECTPARA_ITFRET(PID_Ctx_TaskId, m_spCyclicCaller)
END_GETOBJECTPARA_MAP()
```

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

**Step 3: State machine transitions**

Note the different state transitions of your state machine [45]:

---

[45]: Link to state machine documentation or example.
The parameters are specified during the transition Init->Preop and perhaps Preop->Safeop.

12.3.4.2.2 Parameter properties

Parameter properties: Edit the properties of the parameter.
General properties

Name: Name of the interface.

Specification: Data type of the parameter, see specification.

Select data type

Select: Select data type.

Type information

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

Enter a unique ID Value: Enter a unique ID value, see Parameter [132].

Constant Name: source code name of the parameter ID.

Optional parameter settings

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

Unit: a unit of the variable.

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

Context ID: Context used when accessing parameters by ADS.

Create Symbol: Default setting for ADS icon creation.
Wizards

**Disable Code Generation**: Enable/disable the code generation.

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

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

**Online parameter**: Define as online parameter.

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

**Optional Properties**

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

### 12.3.4.2.3 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)](#) [230]
12.3.4.3  Data Areas

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

Symbol | Function
--- | ---
[+] | Add a new data area
[−] | Delete the selected data area
[Down] | Moves the selected element down one position
[Up] | Moves the selected element up one position
[8 Byte] | 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.

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

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

This article describes:

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

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

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

4. Rename the data area.

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

Editing the name or data type of existing variables
7. Select the sub-node Symbol4 or double-click on the row. The variable properties are shown.
8. Enter a new name, e.g. "bEnableJob" and change the type to BOOL.

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

Note: Remember to run the TMC Code Generator again.

Deleting existing variables

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

2. Run the TMC Code Generator again.

12.3.4.3.2 Data Areas Properties

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

**Number**: Number of the data area.

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

Linkable data areas in the System Manager:
- Input-Source
- Input-Destination
- Output-Source
- Output-Destination
- Retain-Source (for use with NOV-RAM memory, see appendix [349])
- Retain-Destination (for internal use)

Further data areas:
- Standard (visible but not linkable in the System Manager)
- Internal (for internal module symbols, which can be reached via ADS but are not visible in the System Manager)
• MArea (for internal use)
• Not specified (same as standard)

**Name**: Name of the data area.

**Optional parameter settings**

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

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

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

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

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

**Disable Code Generation**: Enable/disable the code generation.

**Optional Defaults**

Depending on data type the default could be defined.

**Optional Properties**

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

**12.3.4.3.3 Symbol Properties**

**Symbols**: Dialog for the editing of the symbols of the data area.
General properties

Name: Name of the symbol.

Specification: Data type of the symbol, see Data type properties [123].

Select data type

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

Description: Define whether the type is the following:
- Normal type
- Pointer
- Pointer to pointer
- Pointer to pointer to pointer
- A reference

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

Optional data type settings

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

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

Comment: Optional comment that is visible, for example, in the instance configurator.
Create Symbol: Default setting for ADS icon creation.

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

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

12.3.4.4 Data Pointers

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

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Add" /></td>
<td>Add a new data pointer</td>
</tr>
<tr>
<td><img src="image" alt="Delete" /></td>
<td>Deletes the selected data pointer</td>
</tr>
<tr>
<td><img src="image" alt="Down" /></td>
<td>Moves the selected element down one position</td>
</tr>
<tr>
<td><img src="image" alt="Up" /></td>
<td>Moves the selected element up one position</td>
</tr>
</tbody>
</table>

**Name:** Name of the data pointer.

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

**Type:** Defines the pointer type.

**Context:** Displays the context ID.

**Disable Code Generation:** Enable/disable the code generation.
12.3.4.1 Data Pointer Properties

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

[Diagram showing 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**

See [here](126).

**Configuring the parameter ID**

**Enter a unique ID Value:** Enter a unique ID value, see Parameter [132].

**Constant Name:** source code name of the parameter ID.

**Optional data pointer settings**

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

**Context ID:** Context ID of the data pointer.
**Disable Code Generation**: Enable/disable the code generation.

### 12.3.4.5 Interface Pointers

**Interface Pointers**: Add, remove or re-order interface pointers.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Add interface pointers</td>
</tr>
<tr>
<td>-</td>
<td>Deletes the selected pointer</td>
</tr>
<tr>
<td>↓</td>
<td>Moves the selected element down one position</td>
</tr>
<tr>
<td>↑</td>
<td>Moves the selected element up one position</td>
</tr>
</tbody>
</table>

**Name**: Name of the interface.

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

**Type**: Type of the interface pointer.

**Context**: Context of the interface.

**Disable Code Generation**: Enable/disable the code generation.

### 12.3.4.5.1 Interface Pointer Properties

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

Name: Name of the interface pointer.

Select the basic interface

Select: Selection of the interface.

Type information

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

Configure the parameter ID

See Parameters [132].

Comment: Optional

Context ID: Context ID of the interface pointer.

Disable Code Generation: Enable/disable the code generation.

12.3.4.6 Deployment

Deployment: Specify storage locations for the provided modules on the target system.
This dialog enables configuration of the source and target file, which are transferred to the target system for the respective platforms.

**Define the files, which should be deployed.**

**Source File:** Path to the source files.

**Destination file:** Path to the binary files.

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

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

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

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

**Add or remove specific file entries**

- **Remove symbol file entries:** Removes the entries for the provision of symbol files (PDB).
- **Remove all entries:** Removes all entries.
- **Reset to default:** Sets the standard entries.
- **Add default file entries:** Adds the entries for the selected platform.
Remove file entries: Removes the entries for the selected platform.

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

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

<table>
<thead>
<tr>
<th>Virtual environment variable</th>
<th>Registry entry (REG_SZ) under key</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%TC_INSTALLPATH%</td>
<td>InstallDir</td>
<td>C:\TwinCAT\3.x \InstallDir</td>
</tr>
<tr>
<td>%TC_CONFIGPATH%</td>
<td>ConfigDir</td>
<td>C:\TwinCAT\3.x \ConfigDir</td>
</tr>
<tr>
<td>%TC_TARGETPATH%</td>
<td>TargetDir</td>
<td>C:\TwinCAT\3.x \TargetDir</td>
</tr>
<tr>
<td>%TC_SYSTEMPATH%</td>
<td>SystemDir</td>
<td>C:\TwinCAT\3.x \SystemDir</td>
</tr>
<tr>
<td>%TC_BOOTPRJPATH%</td>
<td>BootDir</td>
<td>C:\TwinCAT\3.x \BootDir</td>
</tr>
<tr>
<td>%TCRESOURCEPATH%</td>
<td>ResourceDir</td>
<td>C:\TwinCAT\3.x \Target\ResourceDir</td>
</tr>
<tr>
<td>%TC_REPOSITORYPATH%</td>
<td>RepositoryDir</td>
<td>C:\TwinCAT\3.x \RepositoryDir</td>
</tr>
<tr>
<td>%TC_DRIVERPATH%</td>
<td>DriverDir</td>
<td>C:\TwinCAT\3.x \DriverDir</td>
</tr>
<tr>
<td>%TC_DRIVERAUTOINSTALLPATH%</td>
<td>DriverAutoInstallDir</td>
<td>C:\TwinCAT\3.x \Driver\AutoInstallDir</td>
</tr>
<tr>
<td>%TC_SYSSRVEXEPATH%</td>
<td></td>
<td>C:\TwinCAT\3.x \SDK \Bin\TwinCAT UM (x86)</td>
</tr>
<tr>
<td>%CLASSFACTORYNAME%</td>
<td></td>
<td>&lt;Name of the Class Factory&gt;</td>
</tr>
</tbody>
</table>

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

12.4 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 [61]. This chapter describes the configuration of these instances in detail.

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

- **Object Id**: The object ID used for identifying this instance in the TwinCAT system.
- **Object Name**: Name of the object used for displaying the instance in the Solution Explorer tree.
- **Type Name**: Type information (class name) of the instance.
- **GUID**: Module classes GUID.
- **Class Id**: Class ID of the implementation class (GUID and ClassId are usually identical).
- **Class Factory**: Refers to the driver, 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]({#45}) 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.

### 12.4.2 Context

- **Context**: Select the context to be configured (see TMC Editor for Adding different contexts).
  
  **Note**: A data area is assigned to a context

- **Data Areas / Interfaces / Data Pointer and Interface Pointer**: Each instance can be configured to have or not have elements defined in TMC.

- **Result Table**: List of the IDs that need to be configured. At least the context ("Task" column) of the task must be configured accordingly.

### 12.4.3 Parameter (Init)

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 TMCEditor's parameter dialogue as described [here](#132).

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

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.

12.4.5 Interfaces

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

12.4.6 Interface Pointer

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](#132).

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

List of all Data 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 [32].

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

12.5 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.

12.5.1 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.

12.5.2 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), each project type encompassing 3 files:

The .vsdir file provides information that is used when the respective assistant wizard is started. This is essentially a name, a brief description and a file name of the type .vsz containing details for this project type. The general description in 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 .vsdir file provides information that is needed by the wizard. The most important information here is the wizard that is to be started and a list of parameters.
Both wizards have a .xml file as a parameter that describes the transformations of, for example, source files from the template to the specific project. These are located together with the templates for the source code, etc. in the Templates directory.

If a driver is to be created, the TwinCAT C++ Project Wizard starts the TwinCAT Module Class Wizard via the TriggerAddModule parameter.

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

The .ico file merely provides an icon.

### Directory: Templates

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

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

### 12.5.3 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:

```xml
<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 destination 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 names

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 destination file names can be provided by adding the "symbolName" attribute:
   `<TargetFile symbolName="CustomerFileName">[!output SYMBOLNAME].txt</TargetFile>

3. Important symbols are provided by the system itself

<table>
<thead>
<tr>
<th>Symbol Name (Projects)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROJECT_NAME</td>
<td>The project name from the Visual Studio dialog.</td>
</tr>
<tr>
<td>PROJECT_NAME_UPPERCASE</td>
<td>The project name in upper case letters.</td>
</tr>
<tr>
<td>WIN32_WINNT</td>
<td>0x0400</td>
</tr>
<tr>
<td>DRVID</td>
<td>Driver ID in the format: 0x03010000</td>
</tr>
<tr>
<td>PLATFORM_TOOLSET</td>
<td>Toolset version, e.g. v100</td>
</tr>
<tr>
<td>PLATFORM_TOOLSET_ELEMENT</td>
<td>Toolset version as an XML element, e.g. <code>&lt;PlatformToolset&gt;v100&lt;/PlatformToolset&gt;</code></td>
</tr>
<tr>
<td>NEW_GUID_REGISTRY_FORMAT</td>
<td>Creates a new GUID in the format: <code>{48583F97-206A-4C7C-9EF2-D5C8A31F7BDC}</code></td>
</tr>
</tbody>
</table>

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

12.5.4 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
2. Increase the SortPriority.
3. Recommendation: in the directory, create a subdirectory called Class, which is entered in the registry, and a subdirectory called Templates in order to separate the .vsz / .vsdir / .ico files from the templates.
4. Adapt the paths within the files.

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

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

Quick start

A general entry to the wizard 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 the copying template

In C:\TwinCAT\3.x\Components\Base\CppTemplate\Templates

- CustomerModuleCyclicIO
- TcDriverWizard
- TcModuleAdsPort
- TcModuleCyclicCaller
- TcModuleCyclicIO
- TcModuleDataPointer
- TcModuleEmpty
- TcModuleRT
- TcStaticLibrary
2. Rename the .xml file within the folder

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

4. Now reference the copied .vsz file in the .vsdir file and adapt the description.
5. Enter the .xml file created in step 2 in the .vsz file.
6. You can now make changes to the source files in the Template/CustomerModuleCyclicIO/ directory.

   The .xml takes care of replacements when creating a project from this template.

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

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

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

**Alternative creation on the basis of an existing project**

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

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

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

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

TwinCAT C++ project

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

Renaming is not possible at this stage (see Renaming TwinCAT C++ projects).

The encryption of the binary module can be set here, a more detailed description of the requirements can be found here.
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.

13.1 TwinCAT C++ Project properties

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

Different settings can be made in the project properties for a TwinCAT C++ project.
The project properties are opened by right-clicking on the C++ project -> Properties.
TwinCAT pages exist in addition to the Visual Studio C++ dialogs for the settings:
These are described on the subpages.

13.1.1 Tc SDK

Settings for the TwinCAT SDK

Common
- TwinCAT SDK folder: File folder that provides the TwinCAT SDK and shows the value of the environment variable TWINCATSDK.

Intrinsics
- Use intrinsics from platform toolset: the intrinsics should be used.
Windows Driver Kit

If an environment variable WINDDK7 is set, this option prevents the use of the WDK for a specific project.

- **Build with WDK**: Whether the WDK should be used.
- **WinDDK7 folder**: Displays the value of the WINDDK7 environment variable.

### 13.1.2 Tc Extract Version

Version information from the project is provided in a header file and used for the build process. If the `.rc` file contains version information, the header file is generated from it. With versioned C++ projects, the version information is read from the TMC file and the macros from the generated header file are used in the `.rc` file.

**Extract Version**

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

### 13.1.3 Tc Publish

Information about publishing modules.

**General Settings**

- **Platform(s)**: Which platforms should be built in a Publish?
- **Configuration(s)**: Selection whether to build debug / release.
- **Include Debug Information**: For which configurations should the debug symbols (PDBs) be provided in the repository?
- **TMC / TML source file(s)**: TMC / TML files from the project that represent the starting point for the Publish process.
- **Publish as Versioned Library**: Should the publishing take place in the repository?
Installation Settings

- **Archive**: File path for an archive. Extensions .zip (for a ZIP archive) and .exe (for a self-extracting ZIP archive) are allowed. Both contain the content for a repository (versioned C++ projects) or CustomConfig/Modules (non-versioned C++ drivers) on another engineering system.

- **Publish Installation Root Folder**: No installation is performed on the local system with "None". The files are only available under `TWINCATSDK/_products/TcPublish`. An archive can be created to manually transfer these files to another system and install them there. If "Default" is selected, an installation into the repository (versioned C++ projects) or CustomConfig/Modules (non-versioned C++ drivers) is performed on the local system.

Special Settings

- **Additional Files**: Adding additional files to the Publish process, which are stored in the "deploy" subdirectory during installation.

- **Post publish command**: Execute a command after the publish, e.g. to clean up.

### 13.1.4 Tc Sign

**Enable signing**

- **SHA1 signing**: Should an operating system signature, which is necessary for the operating system, be carried out?
- **SHA256 signing**: Should an operating system signature, which is necessary for the operating system, be carried out?
- **TwinCAT signing**: Should a TwinCAT user certificate be used for signing? This is necessary for the TwinCAT Loader [54].

**TwinCAT Certificates**

These parameters are used for all configurations such as debug and release.

- **TwinCAT Certificate Name**: Name of the certificate file (directory: `C:\TwinCAT\3.x\CustomConfig\Certificates`). Alternatively, the environment variable `TcSignTwinCatCertName` can be set to the name of the certificate file.
• **TwinCAT Certificate Password**: Password that protects the TwinCAT user certificate (stored in plain text, leave blank if necessary). The TcSignTool [58] can be used to not store the password of the TwinCAT user certificate in the project, where it would also end up in version management, for example.

• **Verbose Output**: Should extended information be output during the signature?

Windows Certificate (SHA1)

• **Certificate Store Name**: Name of the certificate store in the certificate manager of the operating system.

• **Certificate Name**: Name of the certificate in the certificate store.

• **Certificate ID**: ID of the certificate.

• **Timestamp Server URL**: URL of the timestamp server for use during the signature. This is provided by various CAs.

• **CA Cross Signing Certificate Path**: Path to cross signing certificate. Microsoft provides an overview here.

• **Verbose Output**: Should extended information be output during the signature?

Windows Certificate (SHA256)

• **Certificate Store Name**: Name of the certificate store in the certificate manager of the operating system.

• **Certificate Name**: Name of the certificate in the certificate store.

• **Certificate ID**: ID of the certificate.

• **Timestamp Server URL**: URL of the timestamp server for use during signature, provided by the CA.

• **CA Cross Signing Certificate Path**: Path to cross signing certificate. Microsoft provides an overview here.

• **Verbose Output**: Should extended information be output during the signature?

### 13.2 File Description

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

Here is a list of files related to C++ modules.
<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
<th>Further Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>*.sln</td>
<td>Visual Studio Solution file, hosts TwinCAT and non-TwinCAT projects</td>
<td></td>
</tr>
<tr>
<td>*.tsproj</td>
<td>TwinCAT project, collection of all nested TwinCAT projects, such as TwinCAT C++ or TwinCAT PLC project</td>
<td></td>
</tr>
<tr>
<td>_Config/</td>
<td>Folder contains further configuration files (*.xti) that belong to the TwinCAT project.</td>
<td>See menu Tools</td>
</tr>
<tr>
<td>_Deployment/</td>
<td>Folder for compiled TwinCAT C++ drivers</td>
<td></td>
</tr>
<tr>
<td>*.tmc</td>
<td>TwinCAT Module Class file (XML-based)</td>
<td>See TwinCAT Module Class Editor (TMC) [108]</td>
</tr>
<tr>
<td>*.rc</td>
<td>Resource file</td>
<td>See Setting version/vendor information [237]</td>
</tr>
<tr>
<td><em>.vcxproj.</em></td>
<td>Visual Studio C++ project files</td>
<td></td>
</tr>
<tr>
<td>*ClassFactory.cpp/.h</td>
<td>Class Factory for this TwinCAT driver</td>
<td></td>
</tr>
<tr>
<td>*Ctrl.cpp/.h</td>
<td>Upload and remove drivers for TwinCAT UM platform</td>
<td></td>
</tr>
<tr>
<td>*Driver.cpp/.h</td>
<td>Upload and remove drivers for TwinCAT RT platform</td>
<td></td>
</tr>
<tr>
<td>*Interfaces.cpp/.h</td>
<td>Declaration of the TwinCAT COM interface classes</td>
<td></td>
</tr>
<tr>
<td>*W32.cpp/.def/.idl</td>
<td>One C++/Header file per TwinCAT module in the driver. Insert user code here.</td>
<td></td>
</tr>
<tr>
<td>Resource.h</td>
<td>Required by *.rc file</td>
<td></td>
</tr>
<tr>
<td>TcPch.cpp/.h</td>
<td>Used for creating precompiled headers</td>
<td></td>
</tr>
<tr>
<td>%TC_INSTALLPATH%\Repository&lt;Vendor&gt;&lt;PrjName&gt;&lt;Version&gt;&lt;Platform&gt;*.tmx</td>
<td>Compiled driver that is loaded via the TcLoader. C:\TwinCAT\3.x\Repository\C++ Module Vendor\Untitled1\0.0.0.1\TwinCAT RT *.tmx</td>
<td>See Versioned C++ Projects [50]</td>
</tr>
<tr>
<td>%TC_INSTALLPATH%\CustomConfig\Modules*</td>
<td>Published TwinCAT driver package normally C:\TwinCAT\3.x\CustomConfig Modules*</td>
<td>See Export to TwinCAT 3.1 4022.xx [51]</td>
</tr>
<tr>
<td>%TC_BOOTPRJPATH%\CurrentConfig*</td>
<td>Current configuration setup usually C:\TwinCAT\3.x\Boot</td>
<td></td>
</tr>
<tr>
<td>%TC_DRIVERAUTOINSTALLPATH%*.sys/pdb</td>
<td>Compiled, platform-specific driver that is loaded via the operating system. C:\TwinCAT\3.x\Driver\AutoInstall (system loaded)</td>
<td></td>
</tr>
<tr>
<td>%TC_INSTALLPATH%\Boot\Repository&lt;Vendor&gt;&lt;PrjName&gt;&lt;Version&gt;*.tmx</td>
<td>Compiled platform-specific driver that is loaded via the TcLoader. C:\TwinCAT\3.x\Boot\Repository\C++ Module Vendor\Untitled1\0.0.0.1\Untitled1.tmx</td>
<td></td>
</tr>
<tr>
<td>%TC_BOOTPRJPATH%\TM\OBJECTID.tmi</td>
<td>TwinCAT Module Instance File Describes variables of the driver File name is ObjectID.tmi</td>
<td></td>
</tr>
</tbody>
</table>
### 13.2.1 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:

- Selecting **Build** or **Rebuild** (and thus also **Activate Configuration**), the following steps are performed:
  1. The sources are located in the respective project directory.
  2. The compilations are generated 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. The link process then places the `.sys/.pdb` file also architecture-specific 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` file 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).

### 13.3 Online Change

- **From TwinCAT 3.1 Build 4024.0**
  - The functionality described here is available from TwinCAT 3.1. 4024.0.
TwinCAT 3.1 supports the exchange of C++ modules at runtime, i.e. without interrupting the real-time program.

For this purpose, different versions of a TwinCAT Executable (TMX) are stored on the target system, as already described [here](#)\[50].

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

The procedure is roughly sketched:

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

![Diagram](image)

The Online Change can be carried out through this dialog in engineering.
When dealing with the Online Change, there are therefore some aspects to consider:

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

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

13.4 Limitations

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

- **Win32 API** is not available in kernel mode
- Windows kernel mode API must not be used directly. TwinCAT SDK provides functions, which are supported.
- User mode libraries (DLL) cannot be used. (see Third Party Libraries)
- 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.
- A subset of the C++ runtime library functions (CRT) is supported
- C++ exceptions are not supported.
- Runtime Type Information (RTTI) is not supported.
- Subset of STL is supported (see STL / Containers)
- Support for functions from math.h through TwinCAT implementation (see Mathematical Functions)

**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:
### 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 [207].

### 13.5 Memory Allocation

Generally we recommend reserving memory with the aid of member variables of the module class. This is done automatically for data areas defined in the TMC editor.

It is also possible to allocate and release memory dynamically.

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

This memory allocation can be used in the transitions [45] 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

13.6 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.
### ITcUnknown
ITcUnknown defines the reference count as well as the querying of a reference to a more specific interface.

### ITComObject
The ITComObject interface is implemented by every TwinCAT module.

### ITcCyclic
The interface is implemented by TwinCAT modules that are called once per task cycle.

### ITcCyclicCaller
Interface for logging the ITcCyclic interface of a module onto and off from a TwinCAT task.

### ITcFileAccess
Interface for accessing the file system

### ITcFileAccessAsync
Asynchronous access to file operations.

### ITcPostCyclic
The interface is implemented by TwinCAT modules that are called once per task cycle following the output update.

### ITcPostCyclicCaller
Interface for logging the ITcPostCyclic interface of a module onto and off from a TwinCAT task.

### ITcIoCyclic
This interface is implemented by TwinCAT modules that are called during the input update and output update within a task cycle.

### ITcIoCyclicCaller
Interface for logging the ITcIoCyclic interface of a module onto and off from a TwinCAT task.

### ITcRTTimeTask
Query of extended TwinCAT task information.

### ITcTask
Query of the timestamp and task-specific information of a TwinCAT task.

### ITcTaskNotification
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.\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 [P. 227].
- Subset of STL [P. 229].
- TwinCAT runtime RtlR0.h [P. 209]
- Methods for ADS communication [P. 210]
- Classes / functions with names beginning with "Os" must not be used in a real-time context.

#### 13.6.1 Return values
ITc interfaces methods generally return an HRESULT.

The following return values can be returned in the case of ITc interfaces.
In addition, there is a possibility for ADS Return Codes to be returned as HRESULT. These are also available as macros in the SDK, where they are known, for example, as ADS_E_BUSY for the ADS Error Code ADSERR_DEVICE_BUSY.

### 13.6.2 Interface ITcCyclic

Interface ITcCyclic Interface is implemented by TwinCAT modules which should be called once per task cycle.

**Syntax**

```c
TCOM_DECL_INTERFACE("03000010-0000-0000-e000-000000000064", ITcCyclic)
struct__declspec(novtable) ITcCyclic : public ITcUnknown
```

*Required include: TcIoInterfaces.h*

<table>
<thead>
<tr>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>CycleUpdate</td>
</tr>
</tbody>
</table>

**Comments**

The ITcCyclic interface is implemented by TwinCAT modules. This interface is passed to the ITcCyclicCaller::AddModule() method when a module logs on to a task, usually as the last initialization step in the transition from SafeOP to OP. After login, the CycleUpdate() method of the module instance is called.

### 13.6.2.1 Method ITcCyclic::CyclicUpdate

The CyclicUpdate method is normally called by a TwinCAT Task after the interface has been logged in.

**Syntax**

```c
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
If successful, S_OK ("0") or another positive value will be returned, cf. Return values [174]. Extended messages refer in particular to the column HRESULT in ADS Return Codes.

At present, the return value is ignored by the 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.

13.6.3 Interface ITcCyclicCaller
Interface for logging the ItcCyclic interface of a module onto and off from a TwinCAT task.

Syntax

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

Required include: TcIoInterfaces.h

Methods

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AddModule [176]</td>
<td>Login module that implements the ITcCyclic interface.</td>
</tr>
<tr>
<td>RemoveModule [177]</td>
<td>Log off the previously logged in ITcCyclic interface of a module.</td>
</tr>
</tbody>
</table>

Comments

The ITcCyclicCaller interface is implemented by TwinCAT tasks. A module uses this interface to login its ItcCyclic interface to a task, usually as the last initialization step in the SafeOP to OP transition. After login, the CycleUpdate() method of the module instance is called. The interface is also used to log off the module so that it is no longer called by the task.

13.6.3.1 Method ITcCyclicCaller::AddModule
Reports the ITcCyclic interface of a module to a cyclic caller, e.g. a TwinCAT task.

Syntax

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

Parameter

pEntry: (type: STcCyclicEntry) [in] pointer to a list item that is inserted into the internal list of the cyclic caller; see also description [177].
ipMod: (type: ITcCyclic) [in] interface pointer used by the cyclic caller.
context: (type: ULONG_PTR) [optional] a context value that is transferred to the ITcCyclic::CyclicUpdate() method with each call.
sortOrder: (type: ULONG) [optional] the sorting order can be used for controlling the order of execution if various module instances are executed by the same cyclic caller.

Return value

If successful, S_OK ("0") or another positive value will be returned, cf. Return values [174]. Extended messages refer in particular to the column HRESULT in ADS Return Codes.

The error ADSERR_DEVICE_INVALIDSTATE is returned if the cyclic caller, i.e. the TwinCAT task is not in the OP state.

Description

A TwinCAT module class normally uses a Smart Pointer to refer to the cyclic caller type ITcCyclicCallerPtr. The object ID of the task is stored in this Smart Pointer and a reference to the task can be obtained via the TwinCAT object server. In addition, the Smart Pointer class already contains a list item. Therefore the Smart Pointer can be used as the first parameter for the AddModule method.

The following sample code shows the login of the ITcCyclicCaller interface.

```cpp
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;
        }
    }
}
```

13.6.3.2 Method ITcCyclicCaller:RemoveModule

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

Syntax

```cpp
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 successful, S_OK ("0") or another positive value will be returned, cf. Return values [174]. Extended messages refer in particular to the column HRESULT in ADS Return Codes.

The method returns E_FAIL if the entry is not in the internal list.

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);
} else {
    m_spCyclicCaller = NULL;
}

13.6.4 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

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FileRead [180]</td>
<td>Reads from a file.</td>
</tr>
<tr>
<td>FileFindFirst [184]</td>
<td>Searches for a file, first iteration.</td>
</tr>
<tr>
<td>FileFindByNext [184]</td>
<td>Searches for a file, next iteration.</td>
</tr>
<tr>
<td>FileFindClose [185]</td>
<td>Closes a file search.</td>
</tr>
<tr>
<td>MkDir [185]</td>
<td>Creates a directory.</td>
</tr>
<tr>
<td>RmDir [186]</td>
<td>Deletes a directory.</td>
</tr>
</tbody>
</table>

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 [186] adds a Check() Method, which could be used instead.

Please have a look at Sample20a: FileIO-Cyclic Read / Write [313].

The interface is implemented by module class CID_TcFileAccess.

13.6.4.1 Method ITcFileAccess:FileOpen

Opens a file.
Syntax

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

Parameter

szFileName: (type: PCCH) [in] the name of the file to be opened.

AccessMode: (type: TcFileAccessMode) [in] method of accessing the file; see TcFileAccessServices.h.

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

Return value

If successful, S_OK ("0") or another positive value will be returned, cf. Return values [† 174]. Extended messages refer in particular to the column HRESULT in ADS Return Codes.

A particularly interesting error code is ADSERR_DEVICE_TIMEOUT if the timeout (5 seconds) has elapsed.

Description

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

AccessModes could be used as following:

```c
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;
```

Also see about this

ADS Return Codes [† 344]

13.6.4.2 Method ITcFileAccess:FileClose

Closes a file.

Syntax

virtual HRESULT TCOMAPI FileClose(PTcFileHandle phFile);

Parameter

phFile: (type: TcFileHandle) [out] returned file handle.
Return value

If successful, S_OK ("0") or another positive value will be returned, cf. Return values [174]. Extended
messages refer in particular to the column HRESULT in ADS Return Codes.

A particularly interesting error code is ADSERR_DEVICE_TIMEOUT if the timeout (5 seconds) has elapsed.

Description

The method closes a file defined by the phFile.

Also see about this

ADS Return Codes [344]

13.6.4.3 Method ITcFileAccess::FileRead

Read data from a file.

Syntax

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

Parameter

hFile: (type: TcFileHandle) [in] refers to the previously opened file.
pData: (type: PVOID) [out] storage location of the data to be read.
cbData: (type: PVOID) [in] maximum size of the data to be read (size of the memory behind pData).
pcbRead: (type: PUINT) [out] size of the data that was read.

Return value

If successful, S_OK ("0") or another positive value will be returned, cf. Return values [174]. Extended
messages refer in particular to the column HRESULT in ADS Return Codes.

A particularly interesting error code is ADSERR_DEVICE_TIMEOUT if the timeout (5 seconds) has elapsed.

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.

Also see about this

ADS Return Codes [344]

13.6.4.4 Method ITcFileAccess::FileWrite

Write data to a file.

Syntax

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

Parameter

hFile: (type: TcFileHandle) [in] refers to the previously opened file.
pData: (type: PVOID) [in] storage location of the data to be written.

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

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

**Return value**

If successful, S_OK ("0") or another positive value will be returned, cf. Return values [174]. Extended messages refer in particular to the column HRESULT in ADS Return Codes.

A particularly interesting error code is ADSERR_DEVICE_TIMEOUT if the timeout (5 seconds) has elapsed.

**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.

**Also see about this**

ADS Return Codes [344]

---

### 13.6.4.5 Method ITcFileAccess::FileSeek

Sets position in file.

**Syntax**

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

**Parameter**

- **hFile**: (type: TcFileHandle) [in] refers to the previously opened file.
- **uiPos**: (type: UINT) [in] position at which setting is to take place.

**Return value**

If successful, S_OK ("0") or another positive value will be returned, cf. Return values [174]. Extended messages refer in particular to the column HRESULT in ADS Return Codes.

A particularly interesting error code is ADSERR_DEVICE_TIMEOUT if the timeout (5 seconds) has elapsed.

**Description**

This method sets the position within the file for further actions.

**Also see about this**

ADS Return Codes [344]

---

### 13.6.4.6 Method ITcFileAccess::FileTell

Retrieves position in file.

**Syntax**

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

**Parameter**

- **hFile**: (type: TcFileHandle) [in] refers to the previously opened file.
puiPos: (type: PUINT) [out] storage location of the position to be returned.

**Return value**

If successful, S_OK ("0") or another positive value will be returned, cf. Return values [174]. Extended messages refer in particular to the column HRESULT in ADS Return Codes.

A particularly interesting error code is ADSERR_DEVICE_TIMEOUT if the timeout (5 seconds) has elapsed.

**Description**

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

**Also see about this**

- ADS Return Codes [344]

### 13.6.4.7 Method ITcFileAccess: FileRename

Renames a file.

**Syntax**

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

**Parameter**

szOldName: (type: PCCH) [in] the file name to be changed.

szNewName: (type: PCCH) [in] the new file name.

**Return value**

If successful, S_OK ("0") or another positive value will be returned, cf. Return values [174]. Extended messages refer in particular to the column HRESULT in ADS Return Codes.

A particularly interesting error code is ADSERR_DEVICE_TIMEOUT if the timeout (5 seconds) has elapsed.

**Description**

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

**Also see about this**

- ADS Return Codes [344]

### 13.6.4.8 Method ITcFileAccess: FileDelete

Deletes a file.

**Syntax**

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

**Parameter**

szFileName: (type: PCCH) [in] name of the file to be deleted.
Return value

If successful, S_OK ("0") or another positive value will be returned, cf. Return values [174]. Extended messages refer in particular to the column HRESULT in ADS Return Codes.

A particularly interesting error code is ADSERRDEVICE_TIMEOUT if the timeout (5 seconds) has elapsed.

Description

This method deletes a file from the file system.

Also see about this

ADS Return Codes [344]

13.6.4.9 Method ITcFileAccess:FileGetStatus

Retrieves status of a file.

Syntax

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

Parameter

szFileName: (type: PCCH) [in] the name of the file in question.

pFileStatus: (type: PTcFileStatus) [out] the state of the file, cf. TcFileAccessServices.h.

Return value

If successful, S_OK ("0") or another positive value will be returned, cf. Return values [174]. Extended messages refer in particular to the column HRESULT in ADS Return Codes.

A particularly interesting error code is ADSERRDEVICE_TIMEOUT if the timeout (5 seconds) has elapsed.

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;

Also see about this

ADS Return Codes [344]
13.6.4.10  Method ITcFileAccess::FileFindFirst

Capability to step through files of a directory.

Syntax

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

Parameter

szFileName:  (type: PCCH) [in] directory or path and name of the file sought. The file name can contain placeholders such as asterisk (*) or question mark (?).

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

phFileFind:  (type: PTcFileFindHandle) [out] handle for searching further with FileFindNext.

Return value

If successful, S_OK ("0") or another positive value will be returned, cf. Return values [174]. Extended messages refer in particular to the column HRESULT in ADS Return Codes.

A particularly interesting error code is ADSERR_DEVICE_TIMEOUT if the timeout (5 seconds) has elapsed.

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;
```

Also see about this

ADS Return Codes [344]

13.6.4.11  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

If successful, S_OK ("0") or another positive value will be returned, cf. Return values [174]. Extended messages refer in particular to the column HRESULT in ADS Return Codes.

A particularly interesting error code is ADSERR DEVICE_TIMEOUT if the timeout (5 seconds) has elapsed.

Description

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

```c
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;
```

Also see about this

ADS Return Codes [344]

13.6.4.12 Method ITcFileAccess: FileFindClose

Close finding files of a directory.

Syntax

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

Parameter

hFileFind: (type: PTcFileFindHandle) [in] handle to exit the search.

Return value

If successful, S_OK ("0") or another positive value will be returned, cf. Return values [174]. Extended messages refer in particular to the column HRESULT in ADS Return Codes.

A particularly interesting error code is ADSERR DEVICE_TIMEOUT if the timeout (5 seconds) has elapsed.

Description

This method closes finding of files in a directory.

Also see about this

ADS Return Codes [344]

13.6.4.13 Method ITcFileAccess: MkDir

Create a directory on the filesystem.

Syntax

```c
virtual HRESULT TCOMAPI MkDir (PCCH szDir);
```
Parameter
szDir: (type: PCCH) [in] directory to be created.

Return value
If successful, S_OK ("0") or another positive value will be returned, cf. Return values [174]. Extended messages refer in particular to the column HRESULT in ADS Return Codes.
A particularly interesting error code is ADSERR_DEVICE_TIMEOUT if the timeout (5 seconds) has elapsed.

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

Also see about this
ADS Return Codes [344]

13.6.4.14 Method ITcFileAccess:RmDir
Delete a directory from the filesystem.

Syntax
virtual HRESULT TCOMAPI RmDir(PCCH szDir);

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

Return value
If successful, S_OK ("0") or another positive value will be returned, cf. Return values [174]. Extended messages refer in particular to the column HRESULT in ADS Return Codes.
A particularly interesting error code is ADSERR_DEVICE_TIMEOUT if the timeout (5 seconds) has elapsed.

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

Also see about this
ADS Return Codes [344]

13.6.5 Interface ITcFileAccessAsync
Asynchronous access to file operations. This interface extends ITcFileAccess [178].

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

Required include: TcFileAccessInterfaces.h

Methods

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C [187]check</td>
<td>Query the state of the file operation.</td>
</tr>
</tbody>
</table>
### Interface parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PID_TcFileAccessAsyncSegmentSize</td>
<td>Size of the segments transferred to system service.</td>
</tr>
<tr>
<td>PID_TcFileAccessAsyncTimeoutMs</td>
<td>Sets the timeout in [ms].</td>
</tr>
<tr>
<td>PID_TcFileAccessAsyncNetId(Str)</td>
<td>NetId of the system service to be contacted.</td>
</tr>
</tbody>
</table>

**Comments**

Interface can be obtained from module instance with class ID CID_TcFileAccessAsync.

When using the asynchronous interface, the interface methods inherited from the synchronous variant return ADS_E_PENDING if a query has been successfully submitted but not yet completed. If the call is received while the previous request was still being processed, the error code ADS_E_BUSY is returned.

Description of the module parameters:

- PID_TcFileAccessAsyncAdsProvider: Object ID of a task that provides the ADS interface.
- PID_TcFileAccessAsyncNetId / PID_TcFileAccessAsyncNetIdStr: AmsNetId of the system service used for file access. The "Str" variant takes the AmsNetId as string. Please use one.
- PID_TcFileAccessAsyncTimeoutMs: Timeout for file access.
- PID_TcFileAccessAsyncSegmentSize: The read and write access to file is fragmented with this segment size.

See Sample20a: FileIO-Cyclic Read / Write [313].

#### 13.6.5.1 Method ITcFileAccessAsync::Check()

Query the state of the file operation.

**Syntax**

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

**Parameters**

none

**Return value**

If successful, S_OK ("0") or another positive value will be returned, cf. Return values [174]. Extended messages refer in particular to the column HRESULT in ADS Return Codes.

Particularly interesting error codes are

- ADSERRDEVICE_TIMEOUT if the timeout (5 seconds) has elapsed.
- ADSERRDEVICE_PENDING if the file operation is not completed.

**Description**

This operation checks the state of the previously called file operation.

**Also see about this**

ADS Return Codes [344]

#### 13.6.6 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

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>InputUpdate</code></td>
<td>Is called at the beginning of a task cycle if the interface is logged on to a cyclic I/O caller.</td>
</tr>
<tr>
<td><code>OutputUpdate</code></td>
<td>Is called at the end of a task cycle if the interface is logged on to a cyclic I/O caller.</td>
</tr>
</tbody>
</table>

Comments

ITcIoCyclic can be used to implement a TwinCAT module that acts as a fieldbus driver or I/O filter module. This interface is passed to the ITcIoCyclicCaller::AddIoDriver method when a module logs on to a task, usually as the last initialization step at the transition from SafeOP to OP. After login, the methods `InputUpdate()` and `OutputUpdate()` of the module instance are called once per task cycle.

13.6.6.1 Method ITcIoCyclic:InputUpdate

The `InputUpdate` method is normally called by a TwinCAT task after the interface has been logged in.

Syntax

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

Parameter

- `ipTask`: (type: ITcTask*) refers to the current task context.
- `ipCaller`: (type: ITcUnknown) refers to the calling instance.
- `dwStateIn`: (type: DWORD) future extensions reserved; at present this value is always 0.
- `context`: (type: ULONG_PTR) context contains the value that was transferred to the method ITcCyclicCaller::AddIoDriver().

Return value

If successful, S_OK ("0") or another positive value will be returned, cf. Return values [174]. Extended messages refer in particular to the column HRESULT in ADS Return Codes.

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.

13.6.6.2 Method ITcIoCyclic:OutputUpdate

The `OutputUpdate` method is normally called by a TwinCAT task after the interface has been logged in.
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

If successful, S_OK ("0") or another positive value will be returned, cf. Return values [174]. Extended
messages refer in particular to the column HRESULT in ADS Return Codes.

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.

13.6.7 Interface ITcIoCyclicCaller

Interface for logging the ITcIoCyclic interface of a module onto and off from a TwinCAT task.

Syntax

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

Required include: TcIoInterfaces.h

Methods

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AddIoDriver</td>
<td>Login module that implements the ITcIoCyclic interface.</td>
</tr>
<tr>
<td>RemovelIoDriver</td>
<td>Log off the previously logged in ITcIoCyclic interface of a module.</td>
</tr>
</tbody>
</table>

Comments

The ITcIoCyclicCaller interface is implemented by TwinCAT tasks. A module uses this interface to login its ITcIoCyclic interface to a task, usually as the last initialization step in the SafeOP to OP transition. After login, the CycleUpdate() method of the module instance is called. The interface is also used to log off the module so that it is no longer called by the task.

13.6.7.1 Method ITcIoCyclicCaller::AddIoDriver

Reports the ITcIoCyclic interface of a module to a cyclic I/O caller, e.g. a TwinCAT task.
Syntax

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

Parameter

pEntry: (type: STcIoCyclicEntry) pointer to a list item that is inserted into the internal list of the cyclic I/O caller; see also description [177].

ipDrv: (type: ITcIoCyclic) [in] interface pointer used by the cyclic I/O caller.

context: (type: ULONG_PTR) [optional] a context value that is transferred to the ITcIoCyclic::InputUpdate() and ITcIoCyclic::OutputUpdate methods with each call.

sortOrder: (type: ULONG) [optional] the sorting order can be used for controlling the order of execution if various module instances are executed by the same cyclic caller.

Return value

If successful, S_OK ("0") or another positive value will be returned, cf. Return values [174]. Extended messages refer in particular to the column HRESULT in ADS Return Codes.

Description

A TwinCAT module class normally uses a Smart Pointer to refer to the cyclic I/O caller of type ITcIoCyclicCallerPtr. The object ID of the cyclic I/O caller is stored in this Smart Pointer and a reference can be obtained via the TwinCAT object server. In addition, the Smart Pointer class already contains a list item. Therefore the Smart Pointer can be used as the first parameter for the AddIoDriver method.

The following code sample illustrates the login of the ITcIoCyclicCaller interface.

```c
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;
    }
}
```

13.6.7.2 Method ITcIoCyclicCaller::RemoveIoDriver

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 successful, S_OK ("0") or another positive value will be returned, cf. Return values [174]. Extended messages refer in particular to the column HRESULT in ADS Return Codes.
The method returns E_FAIL if the entry is not in the internal list.

**Description**

Comparable with the AddIoDriver() method, the smart pointer is used for the cyclic I/O caller as a list item if the module instance is to be removed from the cyclic I/O caller.

**Declaration and use of the smart pointer:**

```cpp
ITcIoCyclicCallerInfoPtr m_spIoCyclicCaller;
if ( m_spIoCyclicCaller )
{
    m_spIoCyclicCaller->RemoveIoDriver(m_spIoCyclicCaller);
}
```

**13.6.8 ITComOnlineChange interface**

The ITComOnlineChange interface is used to perform Online Changes of modules.

**Syntax**

```cpp
TCOM_DECL_INTERFACE ("D28A8CD2-5477-4B75-AF0F-998841AF9E44", ITComOnlineChange)
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PrepareOnlineChange</td>
<td>This method is called to prepare the Online Change.</td>
</tr>
<tr>
<td>PerformOnlineChange</td>
<td>This method is called to perform the Online Change.</td>
</tr>
</tbody>
</table>

**Comments**

The implementation of this interface is necessary for a module to be capable of Online Change. Furthermore such a module must be created in a versioned C++ project.

- Here [169] is a general description of the procedure.
- This procedure can be followed for existing modules: Online Change [169].

**Also see about this**

- TwinCAT Module Class Wizard [107]
- TwinCAT C++ Project Wizard [104]

**13.6.8.1 Method ITComOnlineChange::PrepareOnlineChange**

This method is called to prepare the Online Change.

The method is called by TwinCAT to execute the OnlineChange. It runs asynchronously in the background, which must be taken into account when accessing the existing object. The preparation should include all operations that can already be performed.

**Syntax**

```cpp
virtual HRESULT TCOMAPI PrepareOnlineChange(ITComObject* ipOldObj, TmcInstData* pOldInfo) = 0;
```

**Parameter**

- `ipOldObj` (Type: ITComObject*) Reference to the existing object to be exchanged.
- `pOldInfo` (Type: TmcInstData*) reference to information of the existing object.
Return value

If successful, S_OK ("0") or another positive value will be returned, cf. Return values [174]. Extended messages refer in particular to the column HRESULT in ADS Return Codes.

Description

Via ipOldObj, the data of the existing object is made available for transfer so that it can be applied.

For example:

```c
ULONG nData = sizeof(m_Parameter);
PVOID pData = &m_Parameter;
ipOldObj->TcGetObjPara(PID_Module1Parameter, nData, pData);
```

13.6.8.2 Method ITComOnlineChange:PerformOnlineChange

This method is called to perform the Online Change.

The method is called by TwinCAT to execute the OnlineChange. It is called blocking. It should therefore only take a short time.

Syntax

```c
virtual HRESULT TCOMAPI PerformOnlineChange(ITComObject* ipOldObj, TmcInstData* pOldInfo) = 0;
```

Parameter

ipOldObj: (Type: ITComObject*) Reference to the existing object to be exchanged.

pOldInfo: (type: TmcInstData*) reference to information of the existing object.

Return value

If successful, S_OK ("0") or another positive value will be returned, cf. Return values [174]. Extended messages refer in particular to the column HRESULT in ADS Return Codes.

Description

Via ipOldObj, the data of the existing object is made available for transfer so that it can be applied.

For example:

```c
ULONG nData = sizeof(m_Parameter);
PVOID pData = &m_Parameter;
ipOldObj->TcGetObjPara(PID_Module1Parameter, nData, pData);
```

13.6.9 Interface ITComObject

The ITComObject interface is implemented by every TwinCAT module. It makes basic functionalities available.

Syntax

```c
TCOM_DECL_INTERFACE("00000012-0000-0000-e000-000000000064", ITComObject)
struct __declspec(novtable) ITComObject: public ITcUnknown
```
### Methods

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TcGetObjectId(OTCID&amp; objId)</td>
<td>Saves the object ID using the given OTCID reference.</td>
</tr>
<tr>
<td>TcSetObjectId</td>
<td>Sets the object ID of the object to the given OTCID.</td>
</tr>
<tr>
<td>TcGetObjectName</td>
<td>Saves the object names in the buffer with the given length.</td>
</tr>
<tr>
<td>TcSetObjectName</td>
<td>Sets the object name of the object to given CHAR*.</td>
</tr>
<tr>
<td>TcSetObjState</td>
<td>Initializes a transition to a predefined state.</td>
</tr>
<tr>
<td>TcGetObjState</td>
<td>Queries the current state of the object.</td>
</tr>
<tr>
<td>TcGetObjPara</td>
<td>Queries an object parameter identified with its PTCID.</td>
</tr>
<tr>
<td>TcSetObjPara</td>
<td>Sets an object parameter identified with its PTCID.</td>
</tr>
<tr>
<td>TcGetParentObjId</td>
<td>Saves the parent object ID with the help of the given OTCID reference.</td>
</tr>
<tr>
<td>TcSetParentObject</td>
<td>Sets the parent object ID to the given OTCID.</td>
</tr>
</tbody>
</table>

**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.

13.6.9.1 Method ITcComObject:TcGetObjectId(OTCID& objId)

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

**Syntax**

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

**Parameter**

`objId`: (type: OTCID&) reference to OTCID value.

**Return value**

If successful, S_OK ("0") or another positive value will be returned, cf. Return values [174]. Extended messages refer in particular to the column HRESULT in ADS Return Codes.

**Description**

The method stores Object ID using given OTCID reference.

13.6.9.2 Method ITcComObject:TcSetObjectId

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

**Syntax**

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

**Parameters**

`objId`: (type: OTCID) The OTCID, which should be set.
Return value

If successful, S_OK ("0") or another positive value will be returned, cf. Return values [174]. Extended messages refer in particular to the column HRESULT in ADS Return Codes.

At present, the return value is ignored by the TwinCAT tasks.

Description

Indicates the success of the ID change.

13.6.9.3 Method ITcComObject::TcGetObjectName

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

Syntax

```c
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

If successful, S_OK ("0") or another positive value will be returned, cf. Return values [174]. Extended messages refer in particular to the column HRESULT in ADS Return Codes.

Description

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

13.6.9.4 Method ITcComObject::TcSetObjectName

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

Syntax

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

Parameter

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

Return value

If successful, S_OK ("0") or another positive value will be returned, cf. Return values [174]. Extended messages refer in particular to the column HRESULT in ADS Return Codes.

Description

The method TcSetObjectName sets object's's Object Name to the given CHAR*.
13.6.9.5  Method ITcComObject::TcSetObjState

The method TcSetObjState initializes a transition to given state.

Syntax

```
HRESULT TcSetObjState(TCOM_STATE state, ITcomObjectServer* ipSrv, PComInitDataHdr pInitData);
```

Parameter

- **state**: (type: TCOM_STATE) displays the new state.
- **ipSrv**: (type: ITComObjectServer*) ObjServer that handles the object.
- **pInitData**: (type: PComInitDataHdr) points to a list of parameters (optional), see macro IMPLEMENT_ITCOMOBJECT_EVALUATE_INITDATA as an example of how the list can be iterated.

Return value

If successful, S_OK ("0") or another positive value will be returned, cf. Return values [174]. Extended messages refer in particular to the column HRESULT in ADS Return Codes.

Description

The method TcSetObjState initializes a transition to given state.

13.6.9.6  Method ITcComObject::TcGetObjState

The method TcGetObjState retrieves the current state of the object.

Syntax

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

Parameter

- **pState**: (type: TCOM_STATE*) pointer to the state.

Return value

If successful, S_OK ("0") or another positive value will be returned, cf. Return values [174]. Extended messages refer in particular to the column HRESULT in ADS Return Codes.

Description

The TcGetObjState method queries the current state of the object.

13.6.9.7  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)
```
Parameter

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, NULL forwarded.

Return value

If successful, S_OK ("0") or another positive value will be returned, cf. Return values [174]. Extended messages refer in particular to the column HRESULT in ADS Return Codes.

Description

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

13.6.9.8 Method ITcComObject::TcSetObjPara

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

Syntax

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

Parameter

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, NULL forwarded.

Return value

If successful, S_OK ("0") or another positive value will be returned, cf. Return values [174]. Extended messages refer in particular to the column HRESULT in ADS Return Codes.

Description

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

13.6.9.9 Method ITcComObject::TcGetParentObjId

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

Syntax

TEGRULT TcGetParentObjId(OTCID& objId)

Parameter

objId: (type: OTCID&) reference to OTCID value.
Return value

If successful, S_OK ("0") or another positive value will be returned, cf. Return values [174]. Extended messages refer in particular to the column HRESULT in ADS Return Codes.

Description

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

13.6.9.10 Method ITcComObject::TcSetParentObjId

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

Syntax

HRESULT TcSetParentObjId( OTCID objId )

Parameter

objId: (type: OTCID) reference to OTCID value.

Return value

If successful, S_OK ("0") or another positive value will be returned, cf. Return values [174]. Extended messages refer in particular to the column HRESULT in ADS Return Codes.

At present, the return value is ignored by the TwinCAT tasks.

Description

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

13.6.10 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

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OTCID TcGetObjectId</td>
<td>Queries the object ID.</td>
</tr>
<tr>
<td>TcTryToReleaseOpState</td>
<td>Releases resources; must be implemented.</td>
</tr>
</tbody>
</table>

Comments

Further methods exist, which are not itemized here.

This functionality is provided as standard by the module wizards.
13.6.10.1  TcGetObjectId method

The method queries the object ID.

**Syntax**

```plaintext
OTCID TcGetObjectId(void)
```

**Parameters**

**Return Value**

OTCID: Returns the OTCID of the object.

**Description**

The method TcGetObjectId retrieves the Object ID of the object.

13.6.10.2  TcTryToReleaseOpState method

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

**Syntax**

```plaintext
BOOL TcTryToReleaseOpState(void)
```

**Parameters**

**Return value**

If successful, S_OK ("0") or another positive value will be returned, cf. Return values [174]. Extended messages refer in particular to the column HRESULT in ADS Return Codes.

**Description**

The method TcTryToReleaseOpState releases resources, e.g. data pointers, in order to prepare for exiting the OP state. Must be implemented in order to cancel possible mutual dependencies of module instances. See Sample 10 [279].

13.6.11  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**

```plaintext
TCOM_DECL_INTERFACE("03000025-0000-0000-e000-000000000064", ITcPostCyclic)
struct__declspec(novtable) ITcPostCyclic : public ITcUnknown
```

**Required include:** TcIoInterfaces.h
Methods

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PostCycleUpdate [199]</td>
<td>Is called once per task cycle after the output update if the interface has been logged on to a cyclic caller.</td>
</tr>
</tbody>
</table>

Comments

The ITcPostCyclic interface is implemented by TwinCAT modules. This interface is passed to the ITcCyclicCaller::AddPostModule() method when a module logs itself on to a task, usually as the last initialization step at the transition from SafeOP to OP. After login, the PostCycleUpdate() method of the module instance is called.

13.6.11.1 Method ITcPostCyclic:PostCyclicUpdate

The PostCyclicUpdate method normally called by a TwinCAT Task after the output update, after the interface has been logged in.

Syntax

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

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

If successful, S_OK ("0") or another positive value will be returned, cf. Return values [174]. Extended messages refer in particular to the column HRESULT in ADS Return Codes.

At present, the return value is ignored by the 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.

13.6.12 Interface ITcPostCyclicCaller

Interface for logging the ITcPostCyclic interface of a module onto and off from 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

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AddPostModule [200]</td>
<td>Login module that implements the ITcPostCyclic interface.</td>
</tr>
<tr>
<td>RemovePostModule [201]</td>
<td>Log off the previously logged in ITcPostCyclic interface of a module.</td>
</tr>
</tbody>
</table>

#### Comments

The ITcPostCyclicCaller interface is implemented by TwinCAT tasks. A module uses this interface to login its ITcPostCyclic interface to a task, usually as the last initialization step in the SafeOP to OP transition. After login, the PostCycleUpdate() method of the module instance is called. The interface is also used to log off the module so that it is no longer called by the task.

### 13.6.12.1 Method ITcPostCyclicCaller: AddPostModule

Reports the ITcPostCyclic interface of a module to a cyclic caller, e.g. a TwinCAT task.

#### Syntax

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

#### Parameter

- **pEntry**: (type: STcPostCyclicEntry) [in] pointer to a list item that is inserted into the internal list of the cyclic caller; see also [description](#).
- **ipMod**: (type: ITcPostCyclic) [in] interface pointer used by the cyclic caller.
- **context**: (type: ULONG_PTR) [optional] a context value that is transferred to the ITcPostCyclic::PostCycleUpdate() method with each call.
- **sortOrder**: (type: ULONG) [optional] the sorting order can be used for controlling the order of execution if various module instances are executed by the same cyclic caller.

#### Return value

If successful, S_OK ("0") or another positive value will be returned, cf. [Return values](#). Extended messages refer in particular to the column HRESULT in [ADS Return Codes](#).

The error ADSERR_DEVICE_INVALIDSTATE is returned if the cyclic caller, i.e. the TwinCAT task is not in the OP state.

#### Description

A TwinCAT module class uses a Smart Pointer to refer to the cyclic caller of type ITcPostCyclicCallerPtr. The object ID of the task is stored in this Smart Pointer and a reference to the task can be obtained via the TwinCAT object server. In addition, the Smart Pointer class already contains a list item. Therefore the Smart Pointer can be used as the first parameter for the AddPostModule method.

The following code sample illustrates the login of the ITcPostCyclicCaller interface.

```cpp
RESULT hr = S_OK;
if ( m_spPostCyclicCaller.HasOID() ) {
    if ( SUCCEEDED_DBG(hr = m_spSrv->TcQuerySmartObjectInterface(m_spPostCyclicCaller)) ) {
        if ( FAILED(hr =
```
13.6.12.2 Method ITcPostCyclicCaller::RemovePostModule

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

Syntax

```cpp
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 successful, S_OK ("0") or another positive value will be returned, cf. Return values [174]. Extended messages refer in particular to the column HRESULT in ADS Return Codes.

The method returns E_FAIL if the entry is not in the internal list.

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:

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

13.6.13 Interface ITcRTimeTask

Retrieve extended TwinCAT task Information.

Syntax

```cpp
TCOM_DECL_INTERFACE("02000003-0000-0000-e000-000000000064", ITcRTimeTask)
```

struct __declspec(novtable) ITcRTimeTask : public ITcTask

Required include: TcRtInterfaces.h

Methods

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetCpuAccount</td>
<td>Query of the CPU account of a TwinCAT Task.</td>
</tr>
</tbody>
</table>
Remarks
Retrieving and using TwinCAT task information could be done by this interface.
Please have a look at Sample30: Timing Measurement [321]

13.6.13.1 Method ITcRTimeTask::GetCpuAccount()
Query of 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
If successful, S_OK ("0") or another positive value will be returned, cf. Return values [174]. Extended
messages refer in particular to the column HRESULT in ADS Return Codes.
E_POINTER if the parameter pAccount = NULL.

Description
The GetCpuAccount() method can be used to query the current computing time used for the task.

Code snippet showing the use of GetCpuAccount(), e.g. within an ITcCyclic::CycleUpdate() method:

```c++
// 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;
}
```

13.6.14 Interface ITcTask
Query of the timestamp and task-specific information of a TwinCAT task.

Syntax
TCOM_DECL_INTERFACE("02000000-0000-0000-e000-000000000064", ITcTask)
struct __declspec(novtable) ITcTask : public ITcUnknown

Required include: TcRtInterfaces.h
### Methods

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetCycleCounter</td>
<td>Query number of task cycles since task start.</td>
</tr>
<tr>
<td>GetCycleTime</td>
<td>Query of task cycle time in nanoseconds, i.e. time between “begin of task” and next “begin of task”.</td>
</tr>
<tr>
<td>GetPriority</td>
<td>Querying the task priority.</td>
</tr>
<tr>
<td>GetCurrentSysTime</td>
<td>Querying the time when the task cycle starts at intervals of 100 nanoseconds since January 1, 1601 (UTC).</td>
</tr>
<tr>
<td>GetCurrentDcTime</td>
<td>Querying the distributed clock time when the task cycle starts in nanoseconds since January 1, 2000.</td>
</tr>
<tr>
<td>GetCurPentiumTime</td>
<td>Querying the time when the method is called at intervals of 100 nanoseconds since January 1, 1601 (UTC).</td>
</tr>
</tbody>
</table>

**Comments**

With the ITcTask interface the time can be measured in real-time context.

**Syntax**

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

**Parameter**

`pPriority`: (Type: PULONG) [out] Priority value of the task is stored in this parameter.
Return value

If successful, S_OK ("0") or another positive value will be returned, cf. Return values [174]. Extended messages refer in particular to the column HRESULT in ADS Return Codes.

E_POINTER if the parameter pPriority = NULL.

Description

Sample30: Timing Measurement [321] shows usage of this method.

13.6.14.2 Method ITcTask:GetCurrentSysTime

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

Syntax

```cpp
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

If successful, S_OK ("0") or another positive value will be returned, cf. Return values [174]. Extended messages refer in particular to the column HRESULT in ADS Return Codes.

E_POINTER if the parameter pSysTime = NULL.

Description

Sample30: Timing Measurement [321] shows usage of this method.

13.6.14.3 Method ITcTask:GetCurrentDcTime

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

Syntax

```cpp
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

If successful, S_OK ("0") or another positive value will be returned, cf. Return values [174]. Extended messages refer in particular to the column HRESULT in ADS Return Codes.

E_POINTER if the parameter pDcTime = NULL.

Description

Sample30: Timing Measurement [321] shows usage of this method.
13.6.14.4 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;
```

Parameter

pCurTime: (Type: PLONGLONG) [out] This parameter stores the current time (UTC) in 100 nanosecond intervals since January 1, 1601.

Return value

If successful, S_OK ("0") or another positive value will be returned, cf. Return values [174]. Extended messages refer in particular to the column HRESULT in ADS Return Codes.

E_POINTER if the parameter pCurTime = NULL.

Description

Sample30: Timing Measurement [321] shows usage of this method.

13.6.14.5 Method ITcTask:GetCycleCounter

Retrieve number of task cycles since task start.

Syntax

```
virtual HRESULT TCOMAPI GetCycleCounter(PULONG pCnt)=0;
```

Parameter

pCnt: (type: PULONG) [out] the number of task cycles since the task was started is stored in this parameter.

Return value

If successful, S_OK ("0") or another positive value will be returned, cf. Return values [174]. Extended messages refer in particular to the column HRESULT in ADS Return Codes.

E_POINTER if the parameter pCnt = NULL.

Description

Sample30: Timing Measurement [321] shows usage of this method.

13.6.14.6 Method ITcTask:GetCycleTime

Query of task cycle time in nanoseconds, i.e. time between "begin of task" and next "begin of task".

Syntax

```
virtual HRESULT TCOMAPI GetCycleTime(PULONG pCycleTimeNS)=0;
```
Parameters

pCycleTimeNS: (type: PULONG) [out] the configured task cycle time in nanoseconds is stored in this parameter.

Return value

If successful, S_OK ("0") or another positive value will be returned, cf. Return values [174]. Extended messages refer in particular to the column HRESULT in ADS Return Codes.
E_POINTER if the parameter pCnt = NULL.

Description

Sample30: Timing Measurement [321] shows usage of this method.

13.6.15 Interface ITcTaskNotification

Executes a callback if the cycle time was exceeded during the previous cycle. This interface provides comparable functions such as PLC PlcTaskSystemInfo->CycleTimeExceeded.

Syntax

TCOM_DECL_INTERFACE("9CDE7C78-32A0-4375-827E-924B31021FCD", ITcTaskNotification) struct __declspec(novtable) ITcTaskNotification: public ITcUnknown

Required include: TcRtInterfaces.h

Methods

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NotifyCycleTimeExceeded</td>
<td>Called if the cycle time was exceeded.</td>
</tr>
</tbody>
</table>

Comments

Note that the callback does not take place during the calculations, but at the end of the cycle. Therefore, this method does not offer any mechanism for immediately stopping the calculations.

13.6.15.1 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

If successful, S_OK ("0") or another positive value will be returned, cf. Return values [174]. Extended messages refer in particular to the column HRESULT in ADS Return Codes.

Description

Gets called if cycle time was exceeded beforehand. So not immediately on exceeded time, but afterwards.
13.6.16 Interface ITcUnknown

ITcUnknown defines the reference counting as well as querying a reference to a more specific interface.

**Syntax**

```c
TCOM_DECL_INTERFACE("00000001-0000-0000-e000-000000000064", ITcUnknown)
```

Declared in: TcInterfaces.h

Required include: -

### Methods

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TcAddRef [207]</td>
<td>Increments the reference counter.</td>
</tr>
<tr>
<td>TcQueryInterface [207]</td>
<td>Query of the reference to an implemented interface via the IID.</td>
</tr>
<tr>
<td>TcRelease [208]</td>
<td>Decrements the reference counter.</td>
</tr>
</tbody>
</table>

**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.

### 13.6.16.1 Method ITcUnknown:TcAddRef

This method increments the reference counter.

**Syntax**

```c
ULONG TcAddRef( )
```

**Return Value**

Resulting reference count value.

**Description**

Increments the reference counter and returns the new value.

### 13.6.16.2 Method ITcUnknown:TcQueryInterface

Query of an interface pointer with regard to an interface that is given by interface ID (IID).

**Syntax**

```c
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

If successful, S_OK ("0") or another positive value will be returned, cf. Return values [174]. Extended messages refer in particular to the column HRESULT in ADS Return Codes.

If the demanded interface is not available, the method returns ADSERR_DEVICE_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:

```cpp
HRESULT GetTraceLevel(ITcUnknown* ip, TcTraceLevel& tl)
{
    HRESULT hr = S_OK;
    if (ip != NULL)
    {
        ITCOMObjectPtr spObj;
        hr = ip->TcQueryInterface(spObj.GetIID(), &spObj);
        if (SUCCEEDED(hr))
        {
            hr = spObj->TcGetObjPara(PID_TcTraceLevel, &tl, sizeof(tl));
        }
        return hr;
    }
}
```

The interface id associated with the smart pointer can be used as 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:

```cpp
HRESULT GetTraceLevel(ITcUnknown* ip, TcTraceLevel& tl)
{
    HRESULT hr = S_OK;
    ITCOMObjectPtr spObj = ip;
    if (spObj != NULL)
    {
        spObj->TcGetObjParam(PID_TcTraceLevel, &tl);
    } else
    {
        hr = ADS_E_NOINTERFACE;
    }
    return hr;
}
```

When assigning interface pointer ip to smart pointer spObj method TcQueryInterface is implicitly called with IID_ITCComObject on the instance ip refers to. This results in shorter code, however it loses the original return code of TcQueryInterface.

### 13.6.16.3 Method ITcUnknown: TcRelease

This method decrements the reference counter.

**Syntax**

```cpp
ULONG TcRelease( )
```

**Return Value**

Resulting reference count value.
Description
Decrement the reference counter and returns the new value.
If reference counter gets zero, object deletes itself.

13.7 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

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs</td>
<td>Calculates the absolute value.</td>
</tr>
<tr>
<td>atof</td>
<td>Converts a string (char *buf) into a double.</td>
</tr>
<tr>
<td>BitScanForward</td>
<td>Searches for a set bit (1) from LSB to MSB.</td>
</tr>
<tr>
<td>BitScanReverse</td>
<td>Searches for a set bit (1) from MSB to LSB.</td>
</tr>
<tr>
<td>labs</td>
<td>Calculates the absolute value.</td>
</tr>
<tr>
<td>memcmp</td>
<td>Compares two buffers.</td>
</tr>
<tr>
<td>memcpy</td>
<td>Copies one buffer into another.</td>
</tr>
<tr>
<td>memcpy_byte</td>
<td>Copies one buffer into another (bytewise).</td>
</tr>
<tr>
<td>memset</td>
<td>Sets the bytes of a buffer to a value.</td>
</tr>
<tr>
<td>qsort</td>
<td>QuickSort for sorting a list.</td>
</tr>
<tr>
<td>snprintf</td>
<td>Writes formatted data into a character string.</td>
</tr>
<tr>
<td>sprintf</td>
<td>Writes formatted data into a character string.</td>
</tr>
<tr>
<td>sscanf</td>
<td>Reads data from a character string after specification of a format.</td>
</tr>
<tr>
<td>strcat</td>
<td>Appends one character string to another.</td>
</tr>
<tr>
<td>strchr</td>
<td>Searches for a character in a character string.</td>
</tr>
<tr>
<td>strcmp</td>
<td>Compares two character strings.</td>
</tr>
<tr>
<td>strcpy</td>
<td>Copies a character string.</td>
</tr>
<tr>
<td>strlen</td>
<td>Determines the length of a character string.</td>
</tr>
<tr>
<td>strncat</td>
<td>Appends one character string to another.</td>
</tr>
<tr>
<td>strncmp</td>
<td>Compares two character strings.</td>
</tr>
<tr>
<td>strncpy</td>
<td>Copies a character string.</td>
</tr>
<tr>
<td>strstr</td>
<td>Searches for a character string within a character string.</td>
</tr>
<tr>
<td>strtol</td>
<td>Converts a character string into an integer.</td>
</tr>
<tr>
<td>strtoul</td>
<td>Converts a character string into an unsigned integer.</td>
</tr>
<tr>
<td>swscanf</td>
<td>Reads data from a character string after specification of a format.</td>
</tr>
<tr>
<td>tolower</td>
<td>Converts a letter into a lower-case letter.</td>
</tr>
<tr>
<td>toupper</td>
<td>Converts a letter into an upper-case letter.</td>
</tr>
<tr>
<td>vsnprintf</td>
<td>Writes formatted data into a character string ('\0' scheduling).</td>
</tr>
<tr>
<td>vsprintf</td>
<td>Writes formatted data into a character string.</td>
</tr>
</tbody>
</table>

Comments
All functions are based on the C++ runtime library.
13.8 ADS Communication

ADS based on client-server principle. An ADS query calls the corresponding indication methods on the server side. The ADS response calls the corresponding confirmation method on the client side.

In this section both the outgoing and incoming ADS communication is described for TwinCAT 3 C++ modules.

### ADS instruction set

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AdsReadDeviceInfo</td>
<td>The general device information can be read with this command.</td>
</tr>
<tr>
<td>AdsRead</td>
<td>ADS read command for retrieving data from an ADS device.</td>
</tr>
<tr>
<td>AdsWrite</td>
<td>ADS write command for transferring data to an ADS device.</td>
</tr>
<tr>
<td>AdsReadState</td>
<td>ADS command to query the state of an ADS device.</td>
</tr>
<tr>
<td>AdsWriteControl</td>
<td>ADS control command to change the state of an ADS device.</td>
</tr>
<tr>
<td>AdsAddDeviceNotification</td>
<td>Observe variable. The client is informed in case of an event.</td>
</tr>
<tr>
<td>AdsDelDeviceNotification</td>
<td>Removes the variable that was previously linked.</td>
</tr>
<tr>
<td>AdsDeviceNotification</td>
<td>Used to transfer the device notification event.</td>
</tr>
<tr>
<td>AdsReadWrite</td>
<td>ADS read/write command. Data is transmitted to an ADS device (write) and its response data read with one call.</td>
</tr>
</tbody>
</table>

The ADS Return Codes apply to the entire ADS communication.

As an introduction, look at Sample07: Receiving ADS Notifications.

13.8.1 AdsReadDeviceInfo

13.8.1.1 AdsReadDeviceInfoReq

The method AdsDeviceInfoReq enables the transfer of an ADS DeviceInfo command for reading the identification and version number of an ADS server.

AdsReadDeviceInfoCon is called on receipt of the response.

**Syntax**

```c
int AdsReadDeviceInfoReq(AmsAddr rAddr, ULONG invokeId);
```
Parameters

\texttt{rAddr}: (type: AmsAddr&) [in] structure with NetId and port number of the ADS server.

\texttt{invokeId}: (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 \texttt{AdsStatuscodes [344]}.

13.8.1.2 AdsDeviceInfoInd

The method AdsDeviceInfoInd indicates an ADS DeviceInfo command for reading the identification and version number of an ADS server. The \texttt{AdsDeviceInfoRes[211]} must be called afterwards.

Syntax

\begin{verbatim}
void AdsDeviceInfoInd( AmsAddr& rAddr, ULONG invokeId );
\end{verbatim}

Parameter

\texttt{rAddr}: (type: AmsAddr&) [in] structure with NetId and port number of the ADS server.

\texttt{invokeId}: (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.

Return Value

\texttt{void}

13.8.1.3 AdsDeviceInfoRes

The method AdsDeviceInfoRes sends an ADS Read Device Info. \texttt{AdsDeviceInfoCon[212]} forms the counterpart and is subsequently called.

Syntax

\begin{verbatim}
int AdsDeviceInfoRes( AmsAddr& rAddr, ULONG invokeId, ULONG nResult, CHAR name[ADS_FIXEDNAMESIZE], AdsVersion version );
\end{verbatim}

Parameter

\texttt{rAddr}: (type: AmsAddr&) [in] structure with NetId and port number of the responding ADS server.

\texttt{invokeId}: (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.

\texttt{nResult}: (type: ULONG) [in] contains the result of the ADS write command; see \texttt{AdsStatuscodes [344]}.

\texttt{name}: (type: char[ADS_FIXEDNAMESIZE]) [in] contains the name of the device.

\texttt{version}: (type: AdsVersion) [in] structure of build (int), revision (byte) and version (byte) of the device.
Return value
Type: int

Error code - see AdsStatuscodes [344].

### 13.8.1.4 AdsReadDeviceInfoCon

The method AdsReadDeviceInfoCon permits to receive an ADS read device info confirmation. The receiving module has to provide this method. The AdsReadDeviceInfoReq [210] is the counterpart and need to be called beforehand.

**Syntax**

```c
void AdsReadDeviceInfoCon( 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.
- **invokeId**: (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 [344].
- **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**

void

### 13.8.2 AdsRead

#### 13.8.2.1 AdsReadReq

The method AdsReadReq enables the sending of an ADS read command for the data transmission from an ADS device. AdsReadCon [214] is called on receipt of the response.

**Syntax**

```c
int AdsReadReq( AmsAddr& rAddr, ULONG invokeId, ULONG indexGroup, ULONG indexOffset, ULONG cbLength );
```

**Parameter**

- **rAddr**: (type: AmsAddr&) [in] structure with NetId and port number of the ADS server.
- **invokeId**: (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.
- **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 to be read (pData).

Return value
Type: int
Error code - see AdsStatuscodes [344].

13.8.2.2 AdsReadInd

The method AdsReadInd permits to receive an ADS read request. The AdsReadRes [213] needs to be called for sending the result.

Syntax

```c
void AdsReadInd( AmsAddr& rAddr, ULONG invokeId, ULONG indexGroup, ULONG indexOffset, ULONG cbLength );
```

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.

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 to be read (pData).

Return value
Type: int
ADS Return Code - see AdsStatuscodes [344].

13.8.2.3 AdsReadRes

The method AdsReadRes enables the sending of an ADS read response. AdsReadCon [214] forms the counterpart and is subsequently called.

Syntax

```c
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 [344].
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 [344].

13.8.2.4 AdsReadCon

The method AdsReadCon enables the reception of an ADS read confirmation. The receiving module must provide this method. The counterpart AdsReadReq [212] must have been called beforehand.

Syntax

void AdsReadCon( 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 [344].

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

void

13.8.3 AdsWrite

13.8.3.1 AdsWriteReq

The method AdsWriteReq enables the sending of an ADS write command for transferring data to an ADS device. AdsWriteCon [216] is called on receipt of the response.

Syntax

int AdsWriteReq( AmsAddr& rAddr, ULONG invokeId, ULONG indexGroup, ULONG indexOffset, ULONG cbLength, PVOID pData );

Parameter

rAddr: (type: AmsAddr&) [in] structure with NetId and port number of the 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.
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 to be written (pData).

pData: (Type: PVOID) [in] pointer to the data buffer in which the written data are located.

Return value
Type: int
Error code - see AdsStatuscodes [344].

13.8.3.2 AdsWriteInd

The method AdsWriteInd indicates an ADS write command, for the transfer of data to an ADS device. The AdsWriteRes [215] has to be called for confirming the operation.

Syntax

```c
void AdsWriteInd( AmsAddr& rAddr, ULONG invokeId, ULONG indexGroup, ULONG indexOffset, ULONG cbLength, PVOID pData );
```

Parameter

rAddr: (type: AmsAddr&) [in] structure with NetId and port number of the 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.

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 to be written (pData).

pData: (type: PVOID) [in] pointer to the data buffer in which the written data are located.

Return value
void
Error code - see AdsStatuscodes [344].

13.8.3.3 AdsWriteRes

The method AdsWriteRes sends an ADS write response. AdsWriteCon [216] forms the counterpart and is subsequently called.

Syntax

```c
int AdsWriteRes( AmsAddr& rAddr, ULONG invokeId, ULONG nResult );
```

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 [344].

**Return Value**

Type: int

ADS Return Code - see AdsStatuscodes [344].

### 13.8.3.4 AdsWriteCon

The method AdsWriteCon enables the reception of an ADS write confirmation. The receiving module must provide this method. AdsWriteReq [214] forms the counterpart and must have been called beforehand.

**Syntax**

```c
void AdsWriteCon( AmsAddr& rAddr, ULONG invokeId, ULONG nResult );
```

**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 [344].

**Return Value**

void

### 13.8.4 AdsReadWrite

#### 13.8.4.1 AdsReadWriteReq

The method AdsReadWriteReq permits to send an ADS readwrite command, for the transfer of data to and from an ADS device. The AdsReadWriteCon [218] will be called on arrival of the answer.

**Syntax**

```c
int AdsReadWriteReq( AmsAddr& rAddr, ULONG invokeId, ULONG indexGroup, ULONG indexOffset, ULONG cbReadLength, ULONG cbWriteLength, PVOID pData );
```

**Parameter**

- **rAddr**: (type: AmsAddr&) [in] structure with NetId and port number of the 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.
- **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 to be read (pData).

cbWriteLength: (type: ULONG) [in] contains the length in bytes of the data to be written (pData).

pData: (type: PVOID) [in] pointer to the data buffer in which the written data are located.

Return value
Type: int
Error code, see AdsStatuscodes [344].

### 13.8.4.2 AdsReadWriteInd

The method AdsReadWriteInd indicates an ADS readwrite command, for the transfer of data to and from an ADS device. The AdsReadWriteRes [219] needs to be called for sending the result.

**Syntax**

```c
void AdsReadWriteInd( AmsAddr& rAddr, ULONG invokeId, ULONG indexGroup,
ULONG indexOffset, ULONG cbReadLength, ULONG cbWriteLength, PVOID pData );
```

**Parameter**

- `rAddr`: (type: AmsAddr&) [in] structure with NetId and port number of the 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.
- `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 to be read (pData).
- `cbWriteLength`: (type: ULONG) [in] contains the length in bytes of the data to be written (pData).
- `pData`: (type: PVOID) [in] pointer to the data buffer in which the written data are located.

**Return Value**

```c
void
```

### 13.8.4.3 AdsReadWriteRes

The method AdsReadWriteRes permits to receive an ADS read write confirmation. The AdsReadWriteCon [218] is the counterpart and will be called afterwards.

**Syntax**

```c
int AdsReadWriteRes( 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 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 [344].

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 [344].

13.8.4.4 AdsReadWriteCon

The method AdsReadWriteCon enables the reception of an ADS read/write confirmation. The receiving module must provide this method.

AdsReadWriteReq [216] forms the counterpart and must be called beforehand.

Syntax

void AdsReadWriteCon( 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 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 [344].

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

void

13.8.5 AdsReadState

13.8.5.1 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 [220] 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 Invokeld is specified from the source device and serves to identify the commands.

Return value

Type: int

Error code - see AdsStatuscodes [344].

### 13.8.5.2 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 [219] needs to be called for sending the result.

**Syntax**

```c
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 Invokeld is specified from the source device and serves to identify the commands.

**Return Value**

void

### 13.8.5.3 AdsReadStateRes

The method AdsWriteRes enables the sending of an ADS status read response. AdsReadStateCon [220] forms the counterpart and is subsequently called.

**Syntax**

```c
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 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 [344].

**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 [344].

### 13.8.5.4 AdsReadStateCon

The method AdsWriteCon enables the reception of an ADS state read confirmation. The receiving module must provide this method. AdsReadStateReq [218] forms the counterpart and must be called beforehand.

**Syntax**

```c
void AdsReadStateCon( 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 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 [344].
- **adsState**: (type: USHORT) [in] contains the ADS state of the device.
- **deviceState**: (type: USHORT) [in] contains the device status of the device.

**Return Value**

void

### 13.8.6 AdsWriteControl

#### 13.8.6.1 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 [222] will be called on arrival of the answer.

**Syntax**

```c
int AdsWriteControlReq( AmsAddr& rAddr, ULONG invokeId, USHORT adsState, USHORT deviceState, ULONG cbLength, PVOID pData );
```

**Parameter**

- **rAddr**: (type: AmsAddr&) [in] structure with NetId and port number of the ADS server.
- **invokeId**: (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.
- **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 to the data buffer in which the written data are located.

**Return value**

Type: int

Error code - see AdsStatuscodes [344].

### 13.8.6.2 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 [221] has to be called for confirming the operation.

**Syntax**

```c
void AdsWriteControlInd( AmsAddr& rAddr, ULONG invokeId, USHORT adsState, USHORT deviceState, 
ULONG cbLength, PVOID pDeviceData );
```

**Parameter**

- **rAddr**: (type: AmsAddr&) [in] structure with NetId and port number of the ADS server.
- **invokeId**: (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.
- **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 to the data buffer in which the written data are located.

**Return Value**

void

### 13.8.6.3 AdsWriteControlRes

The method AdsWriteControlRes permits to send an ADS write control response. The AdsWriteControlCon [222] is the counterpart and will be called afterwards.

**Syntax**

```c
int AdsWriteControlRes( AmsAddr& rAddr, ULONG invokeId, ULONG nResult );
```

**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 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 [344].
Return value
Type: int
ADS Return Code - see AdsStatuscodes [344].

13.8.6.4 AdsWriteControlCon

The method AdsWriteCon enables the reception of an ADS write control confirmation. The receiving module
must provide this method. AdsWriteControlReq [220] forms the counterpart and must be called beforehand.

Syntax
void AdsWriteControlCon( AmsAddr& rAddr, ULONG invokeId, ULONG nResult );

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 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 [344].

Return Value
void

13.8.7 AdsAddDeviceNotification

13.8.7.1 AdsAddDeviceNotificationReq

The method AdsAddDeviceNotificationReq permits to send an ADS add device notification command, for
adding a device notification to an ADS device. The AdsAddDeviceNotificationCon [224] will be called on
arrival of the answer.

Syntax
int AdsAddDeviceNotificationReq( AmsAddr& rAddr, ULONG invokeId, ULONG indexGroup, ULONG
indexOffset,
    AdsNotificationAttrib noteAttrib);
Return value
Type: int

Error code - see AdsStatuscodes [344].

13.8.7.2 AdsAddDeviceNotificationInd

The method AdsAddDeviceNotificationInd should enable sending AdsDeviceNotification [226]. The AdsAddDeviceNotificationRes [223] has to be called for confirming the operation.

Syntax

```c
void AdsAddDeviceNotificationInd( AmsAddr& rAddr, ULONG invokeId, ULONG indexGroup, ULONG indexOffset, AdsNotificationAttrib noteAttrib );
```

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 Invokeld is specified by the source device and is used for the identification of 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 the specification of the notification parameters (cbLength, TransMode, MaxDelay).

Return Value

void

13.8.7.3 AdsAddDeviceNotificationRes

The method AdsAddDeviceNotificationRes permits to send an ADS add device notification response. The AdsAddDeviceNotificationCon [224] is the counterpart and will be called afterwards.

Syntax

```c
void AdsAddDeviceNotificationCon( AmsAddr& rAddr, ULONG invokeId, ULONG nResult, ULONG handle );
```

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 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
- **Handle**: (type: ULONG) [in] handle to generated device notification.

Return Value

void
13.8.7.4 AdsAddDeviceNotificationCon

The method AdsAddDeviceNotificationCon confirms an ADS device addition notification request. AdsAddDeviceNotificationReq \(\rightarrow 222\) forms the counterpart and must be called beforehand.

Syntax

```c
void AdsAddDeviceNotificationCon( AmsAddr& rAddr, ULONG invokeId, ULONG nResult, ULONG handle );
```

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 InvokedId 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.
- **Handle**: (type: ULONG) [in] handle to generated device notification.

Return Value

- **void**

13.8.8 AdsDelDeviceNotification

13.8.8.1 AdsDelDeviceNotificationReq

The method AdsDelDeviceNotificationReq permits to send an ADS delete device notification command, for removing a device notification from an ADS device. The AdsDelDeviceNotificationCon \(\rightarrow 225\) will be called on arrival of the answer.

Syntax

```c
int AdsDelDeviceNotificationReq( AmsAddr& rAddr, ULONG invokeId, ULONG hNotification );
```

Parameter

- **rAddr**: (type: AmsAddr&) [in] structure with NetId and port number of the ADS server.
- **invokeId**: (type: ULONG) [in] handle of the command that is sent. The InvokedId is specified by the source device and is used for the identification of the commands.
- **hNotification**: (type: ULONG) [in] contains the handle of the notification to be removed.

Return value

- **Type**: int
- **Error code - see AdsStatuscodes \(\rightarrow 344\).**

13.8.8.2 AdsDelDeviceNotificationInd

The method AdsAddDeviceNotificationCon permits to receive an ADS delete device notification confirmation. The receiving module has to provide this method. The AdsDelDeviceNotificationRes \(\rightarrow 225\) has to be called for confirming the operation.
Syntax

```c
void AdsDelDeviceNotificationCon( AmsAddr& rAddr, ULONG invokeId, ULONG nResult );
```

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 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 [344].

Return Value

*void*

13.8.8.3 **AdsDelDeviceNotificationRes**

The method AdsAddDeviceNotificationRes permits to receive an ADS delete device notification. The AdsDelDeviceNotificationCon [225] is the counterpart and will be called afterwards.

Syntax

```c
int AdsDelDeviceNotificationRes( AmsAddr& rAddr, ULONG invokeId, ULONG nResult );
```

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 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 command; see AdsStatuscodes [344].

Return value

*Int*

Returns the result of the ADS command, see AdsStatuscodes [344].

13.8.8.4 **AdsDelDeviceNotificationCon**

The method AdsAddDeviceNotificationCon enables the reception of an ADS device deletion notification confirmation. The receiving module must provide this method. AdsDelDeviceNotificationReq [224] forms the counterpart and must be called beforehand.

Syntax

```c
void AdsDelDeviceNotificationCon( AmsAddr& rAddr, ULONG invokeId, ULONG nResult );
```

Parameter

**rAddr**: (type: AmsAddr&) [in] structure with NetId and port number of the responding ADS server.
invokeld: (type: ULONG) [in] handle of the transmitted command; the Invokeld is specified by the source device and serves to identify the commands.

nResult: (type: ULONG) [in] contains the result of the ADS write command; see AdsStatuscodes [344].

Return Value

void

13.8.9 AdsDeviceNotification

13.8.9.1 AdsDeviceNotificationReq

The method AdsAddDeviceNotificationReq permits to send an ADS device notification, to inform an ADS device. The AdsDeviceNotificationInd [226] 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 [344].

13.8.9.2 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 [227] must be called by AdsDeviceNotificationReq [226] to check the transfer.

Syntax

void AdsDeviceNotificationInd( AmsAddr& rAddr, ULONG invokeId, ULONG cbLength, AdsNotificationStream* pNotifications );

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.

cbLength: (type: ULONG) [in] contains the length of pNotifications.
pNotifications: (type: AdsNotificationStream*) [in] pointer to the notifications. This array consists of AdsStampHeader with notification handle and data via AdsNotificationSample.

Return Value

void

13.8.9.3 AdsDeviceNotificationCon

The sender can use the method AdsAddDeviceNotificationCon to check the transfer of an ADS device notification. AdsDeviceNotificationReq [226] must be called first.

Syntax

```c
void AdsDeviceNotificationCon( AmsAddr& rAddr, ULONG invokeId, ULONG nResult );
```

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 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

13.9 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
  TwinCAT 3.1 4018 provides an fpu87.h with the same methods. This continues to exist and redirects to TcMath.h.
### Methods provided

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sqr_</td>
<td>Calculates the square.</td>
</tr>
<tr>
<td>sqrt_</td>
<td>Calculates the square root.</td>
</tr>
<tr>
<td>sin_</td>
<td>Calculates the sine.</td>
</tr>
<tr>
<td>cos_</td>
<td>Calculates the cosine.</td>
</tr>
<tr>
<td>tan_</td>
<td>Calculates the tangent.</td>
</tr>
<tr>
<td>atan_</td>
<td>Calculates the angle whose tangent is the specified value.</td>
</tr>
<tr>
<td>atan2_</td>
<td>Calculates the angle whose tangent is the quotient of two specified values.</td>
</tr>
<tr>
<td>asin_</td>
<td>Calculates the angle whose sine is the specified value.</td>
</tr>
<tr>
<td>acos_</td>
<td>Calculates the angle whose cosine is the specified value.</td>
</tr>
<tr>
<td>exp_</td>
<td>Calculates e to the specified power.</td>
</tr>
<tr>
<td>log_</td>
<td>Calculates the logarithm of a specified value.</td>
</tr>
<tr>
<td>log10_</td>
<td>Calculates the base 10 logarithm of a specified value.</td>
</tr>
<tr>
<td>fabs_</td>
<td>Calculates the absolute value.</td>
</tr>
<tr>
<td>fmod_</td>
<td>Calculates the remainder.</td>
</tr>
<tr>
<td>ceil_</td>
<td>Calculates the smallest integer that is greater than or equal to the specified number.</td>
</tr>
<tr>
<td>floor_</td>
<td>Calculates the largest integer that is smaller than or equal to the specified number.</td>
</tr>
<tr>
<td>pow_</td>
<td>Calculates a specified number to the specified power.</td>
</tr>
<tr>
<td>sincos_</td>
<td>Calculates the sine and cosine of x.</td>
</tr>
<tr>
<td>fmodabs_</td>
<td>Calculates the absolute value that meets the Euclidean definition of the mod operation.</td>
</tr>
<tr>
<td>round_</td>
<td>Calculates a value and rounds to the nearest integer.</td>
</tr>
<tr>
<td>round_digits_</td>
<td>Calculates a rounded value with a specified number of decimal places.</td>
</tr>
<tr>
<td>cubic_</td>
<td>Calculates the cubic value.</td>
</tr>
<tr>
<td>ldexp_</td>
<td>Calculates a real number (double) from mantissa and exponent.</td>
</tr>
<tr>
<td>ldexpf_</td>
<td>Calculates a real number (float) from mantissa and exponent.</td>
</tr>
<tr>
<td>sinh_</td>
<td>Calculates the hyperbolic sine of the specified angle.</td>
</tr>
<tr>
<td>cosh_</td>
<td>Calculates the hyperbolic cosine of the specified angle.</td>
</tr>
<tr>
<td>tanh_</td>
<td>Calculates the hyperbolic tangent of the specified angle.</td>
</tr>
<tr>
<td>finite_</td>
<td>Determines whether the specified value is finite.</td>
</tr>
<tr>
<td>isnan_</td>
<td>Determines whether the specified value is not a number (NAN).</td>
</tr>
<tr>
<td>rands_</td>
<td>Calculates a pseudo random number between 0 and 32767. The parameter holdrand is set randomly and changed with every call.</td>
</tr>
</tbody>
</table>

### 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.
13.10  Time Functions

TwinCAT provides functions for time conversion, they are declared in TcTimeConversion.h, which is part of TwinCAT SDK.

Methods provided

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TcDayOfWeek</td>
<td>Determines the day of the week (Sunday is 0).</td>
</tr>
<tr>
<td>TcIsLeapYear</td>
<td>Determines whether the given year is a leap year.</td>
</tr>
<tr>
<td>TcDaysInYear</td>
<td>Determines the number of days in a given year.</td>
</tr>
<tr>
<td>TcDaysInMonth</td>
<td>Determines the number of days in a given month.</td>
</tr>
<tr>
<td>TcSystemTimeToFileTime</td>
<td>Converts the given system time into a file time.</td>
</tr>
<tr>
<td>TcFileTimeToSystemTime</td>
<td>Converts the given file time into a system time.</td>
</tr>
<tr>
<td>TcSystemTimeToFileTime</td>
<td>Converts the given system time into a file time</td>
</tr>
<tr>
<td></td>
<td>(ULONGLONG format).</td>
</tr>
<tr>
<td>TcFileTimeToSystemTime</td>
<td>Converts the given file time (ULONGLONG format)</td>
</tr>
<tr>
<td></td>
<td>into a system time.</td>
</tr>
</tbody>
</table>

13.11  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\algorithm for a specific list of supported algorithms.

Restrictions

- Class templates do not exist for all data types.
- Some header files should not be used directly.

More detailed documentation on memory management, which uses STL, can be found here [172].

13.12  Error messages - understanding

In TwinCAT you receive very detailed information about errors that occur.

For instance, this error message means:
The error occurred during the transition from SAFE OP to OP.

The affected object is "Untitled1_Obj1" (CModule1).

The error code 1821 / 0x71d indicates that the object ID is invalid.

Therefore, you should examine the method "SetObjStateSP()", which is responsible for this transition, in more detail. In the case of the generated standard code, you can see that the addition of the module takes place there.

The reason for this error is that no task has been assigned to this module, so the module cannot have a task in which it is executed.

13.13 Module messages for the Engineering (logging / tracing)

Overview

TwinCAT 3 C++ offers the option of sending messages from a C++ module to the TwinCAT 3 Engineering as tracing or logging.
Syntax

The syntax for recording messages is as follows:

```cpp
m_Trace.Log(TLEVEL, FNMACRO "A message", ...);
```

With these properties:

- **TLEVEL** categorizes a message into one of five levels.
  The recording of the higher level always includes the recording of the lower levels: i.e. a message classified on level "tlWarning" will occur with level "tlAlways", "tlError" and "tlWarning" - it will NOT record the "tlInfo" and "tlVerbose" messages.

<table>
<thead>
<tr>
<th>Level 0</th>
<th>tlAlways</th>
</tr>
</thead>
<tbody>
<tr>
<td>Level 1</td>
<td>tlError</td>
</tr>
<tr>
<td>Level 2</td>
<td>tlWarning</td>
</tr>
<tr>
<td>Level 3</td>
<td>tlInfo</td>
</tr>
<tr>
<td>Level 4</td>
<td>tlVerbose</td>
</tr>
</tbody>
</table>

- **FNMACRO** can be used to place the function name before the message to be printed
  - FENTERA: Used when entering a function; prints the function name followed by ">>>".
  - FNAMEA: Used within a function; prints the function name.
  - FLEAVEA: Used when exiting a function; prints the function name followed by "<<<".
- The %q format specifier is used to output pointers and other variables of platform-specific size.
Sample

```cpp
HRESULT CModule1::CycleUpdate(ITcTask* ipTask, ITcUnknown* ipCaller, ULONG_PTR context)
{
    HRESULT hr = S_OK;

    // Sample to showcase trace logs
    ULONGLONG cnt = 0;
    if (SUCCEEDED(ipTask->GetCycleCounter(&cnt)))
    {
        if (cnt%500 == 0)
            m_Trace.Log(tlAlways, FENTERA "Level tlAlways: cycle= %llu", cnt);
        if (cnt%510 == 0)
            m_Trace.Log(tlError, FENTERA "Level tlError: cycle=%llu", cnt);
        if (cnt%520 == 0)
            m_Trace.Log(tlWarning, FENTERA "Level tlWarning: cycle=%lld", cnt);
        if (cnt%530 == 0)
            m_Trace.Log(tlInfo, FENTERA "Level tlInfo: cycle=%llu", cnt);
        if (cnt%540 == 0)
            m_Trace.Log(tlVerbose, FENTERA "Level tlVerbose: cycle=%llu", cnt);
    }

    // TODO: Replace the sample with your cyclic code
    m_counter++;
    m_Outputs.Value = m_counter;

    return hr;
}
```

Use tracking level

The tracking level can be preconfigured at the level of the module instance.

1. Navigate to the instance of the module in the solution tree.
2. Select the Parameters (Init) tab on the right.
3. Make sure that you activate Show Hidden Parameters.
4. Select the tracking level.
5. To test everything, select the highest level tlVerbose.

Alternatively, you can change the tracking level at runtime by going to the instance, selecting a level at Value for TraceLevelMax parameters, right-clicking in front of the first column, and selecting Online Write.
Filter message categories

Visual Studio Error List allows you to filter entries by category. The three categories Errors, Warnings and Messages can be enabled or disabled independently by simply switching the keys.

In this screenshot only warnings are enabled - errors and messages are disabled:

In this screenshot, only messages are enabled - errors and warnings, on the other hand, are disabled for the display:
14 How to...?

This is a collection of frequently asked questions about common paradigms of coding as well as handling of TwinCAT C++ modules.

14.1 Using the Automation Interface

The Automation Interface can be used for C++ projects.

This includes creating projects [104] and using the wizard for creating module classes [105]. In addition, the project properties can be set and the TMC Code Generator and the publishing of modules called. The corresponding documentation [352] is part of the Automation Interface.

Irrespective of the programming language, access to and creation and handling of TcCOM modules [355] may be relevant.

From there, common System Manager tasks such as linking of variables can be executed.

14.2 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 [149]. In later versions this is done implicitly when the target system uses Windows 10 as operating system.

14.3 Publishing of modules

The section Export to TwinCAT 3.1 4022.xx [51] describes how TwinCAT modules are published so that they can be transferred to any TwinCAT system and imported [52].

The engineering system (XAE) does not necessarily have to be on the same platform type as the executing system. To this end, TwinCAT creates all versions of the module in the course of the publication.

Certain use cases require an adaptation of the publication of the modules:

- When working in a pure 32-bit (x86) environment, the x64 settings can be skipped so that no certificates are required.
- The User Mode (UM) settings can be skipped if they are not used.

Migration

If a build is not contained in the published module, the corresponding platform cannot be used as an executing system.

Add the corresponding target platforms in the provision [149] of the TMC Editor or remove them.

The following provision configuration, for example, only makes the TwinCAT RT (x86) build available:
14.4 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`).

14.5 Clone

The runtime data can be transferred from one machine to another by means of a file copy if both 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".

14.4 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`).

14.5 Clone

The runtime data can be transferred from one machine to another by means of a file copy if both 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".
How to...?

1. Create (or enable) 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 transfer MYDRIVER.pdb as well.
5. If drivers are new on a machine:
   TwinCAT must carry out a registration once. To do this, switch TwinCAT to RUN mode 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

You can now start the target machine.

14.6 Access Variables via ADS

Variables of C++ modules can be reached via ADS if the variables are marked in the TMC Editor as "Create Symbol":

The name of the variable for access by ADS is derived from the name of the instance.

For the TraceLevelMax parameter it could be:

Untitled1_Obj1 (CModule1).TraceLevelMax

14.7 TcCallAfterOutputUpdate for C++ modules

Comparable with the PLC attribute TcCallAfterOutputUpdate, C++ modules can be called following the output update.

The ITcPostCyclic [198] interface is used in the same way as the ITcCyclic [175] interface.

14.8 Order determination of the execution in a task

Different module instances can be assigned to a task, so the user needs a mechanism to determine the order of execution in the task.

It is configured under Sort Order in the context [153] of the TwinCAT Module Instance Configurator [151].
See Sample26: Order of execution in a task [319], how this is to be implemented.

### 14.9 Setting version/vendor information

Windows offers a mechanism to query vendor and version resources that are defined in the course of a .rc file for the compilation time.

These are accessible, for example, via the Details tab of each properties file.

TwinCAT offers this behavior via the familiar Windows mechanisms of .rc files, which are generated in the course of creating the TwinCAT C++ project.
Edit the .rc file in the Source Files folder with the resource editor in order to define these properties:

14.10 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 using 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 .vcxproj.user file exists, check the contents; this is where user settings are stored. Also rename this file if necessary.

6. Open the TwinCAT Solution. Re-link the renamed project to the C++ node using Add existing Item: navigate to the renamed subdirectory and select the .vcxproj file there.

7. Rename the ClassFactory, services and interfaces as well as header/source code files to the new project name. In addition, rename the TMC file and the corresponding files in the project folders "TwinCAT RT Files" and "TwinCAT UM Files". This renaming should also be mapped in the version management system; if the version management system is not integrated in Visual Studio, this step must also be carried out in the version management system. Replace all occurrences in the source code (case-sensitive): "OLDPROJECT" becomes "NEWPROJECT" and "OldProject" becomes "NewProject". Use the Find and Replace dialog in Visual Studio for this; note that the "NewProject Project" in the Solution Explorer has to be selected.
**Incorrect source code**
The simple renaming of all occurrences of the character string may result in incorrect source code, for example if the project name is used within a method name.

- If such occurrences are possible, carry out the renaming individually (*Replace* instead of *Replace All*).

How to build the project:
1. A) If instances from the project should exist, update them. To do this, right-click on the instance, select **TTMI/TMC File**->**Reload TMI/TMC File**... and select the renamed new TMC file.

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.

### 14.11 Delete Module

A TwinCAT C++ module can be deleted from a C++ project with the help of the TMC Editor.

1. Right-click on the module (in this case CModule2)
2. Select **Delete**.
3. Confirm the deletion via TMC.

4. Note that the .cpp and .h files are retained – delete them manually if necessary. Delete other components concerned (e.g. header files, structures). See Compiler error messages for more information.

14.12 Add revision control and Online Change subsequently

An existing project with C++ TcCOM modules can be subsequently extended by the Online Change property.

This is a manual process.

The changeover takes place in several steps:

- Convert C++ project to a versioned C++ project [241].
- Make C++ module class Online Change capable [244].

These steps are described on the subpages. However, it may be useful to create a new project yourself and transfer the corresponding differences. Thus, the respective context is visible.

14.12.1 C++ Project -> Revision control

These instructions describe how to add subsequently a revision control to a TwinCAT C++ project.

The code to be changed is stored in bold in the source code.

✓ C++ project. For the sample "Untitled1" is used as C++ project name.
   In addition, an empty, new project with a versioned C++ project is to be created. This serves as a copy template.

1. Open the file Untitled1.vcxproj in an editor.
2. Make the following addition:

   <PropertyGroup Label="Globals">
   <ProjectGuid>{…}</ProjectGuid>
   <RootNamespace>Untitled1</RootNamespace>
   <Keyword>Win32Proj</Keyword>
   <AutomaticRetargetPlatformVersion>true</AutomaticRetargetPlatformVersion>
   </PropertyGroup>

   <PropertyGroup Label="TcGeneral">
   <TcGeneralUseTmx>true</TcGeneralUseTmx>
   </PropertyGroup>
   <Import Project="$(VCTargetsPath)\Microsoft.Cpp.Default.props" />

3. Transfer the Untitled1.rc and Untitled1W32.rc files from the new project to the project to be migrated, overwriting the Untitled1.rc file.
4. Open the project.
5. Under TwinCAT UM Files -> Context menu / Add Existing Items select the file Untitled1W32.rc and add it to the project.
6. This file only needs to be built for TwinCAT UM platforms, so it should otherwise be excluded for the build process. This is done via right-click and properties:

7. Open the TMC Editor.
8. Set the name of the class factory and the version to "0.0.0.1".

10. Change in header `Modul1.h` DECLARE_IPERSIST_LIB():

```cpp
public:
    DECLARE_IUNKNOWN()
    DECLARE_IPERSIST_LIB()
    DECLARE_ITCOMOBJECT_LOCKOP()
```

11. Add `Modul1.cpp` to the source code:

```cpp
#pragma hdrstop
#include "Module1.h"
#include "Untitled1Version.h"

#ifdef _DEBUG
```

12. Add `Modul1.cpp` to the source code:

```cpp
END_INTERFACE_MAP()

IMPLEMENT_IPERSIST_LIB(CModule1, VID_Untitled1, CID_Untitled1CMODULE1)
IMPLEMENT_ITCOMOBJECT(CModule1)
IMPLEMENT_ITCOMOBJECT_SETSTATE_LOCKOP2(CModule1)
```

13. Call the code generator.

14. Change in the file `Untitled1Classfactory.cpp`:

```cpp
CUntitled1ClassFactory::CUntitled1ClassFactory() : CObjClassFactory() {
    TcDbgUnitSetImageName(TCDBG_UNIT_IMAGE_NAME_TMX(SRVNAME_Untitled1));
#ifdef TCDBG_UNIT_VERSION
    TcDbgUnitSetVersion(TCDBG_UNIT_VERSION(Untitled1));
#endif //defined(TCDBG_UNIT_VERSION)
}
```
15. Change the signing in the project properties in the tab **Tc Sign**. To do this, switch **SHA1 signing** off and **TwinCAT signing** on; provide TwinCAT user certificate and password at the same time.

16. Trigger the **Rebuild** of the project.

    The result is a C++ project that supports revision control.

If a module is to be made Online Change capable, this can be achieved by the following instructions.

### 14.12.2 C++ Module -> OnlineChange

These instructions describe how to subsequently make a module OnlineChange-capable in a versioned TwinCAT C++ project.

- **Versioned C++ project with C++ module, which is not yet Online Change capable.** For this sample, a module "Module1" is assumed. In addition, an empty, new project can be created with an Online Change-capable module, from which the changes can be more easily adopted.

1. Open the project and the TMC Editor.
2. Set the **Auto generate on save** option for the module so that the ClassID is changed automatically.
3. Under **Implemented Interfaces**, delete the interfaces ITcADI and ITcWatchsource and add ITComOnlineChange.

4. Delete the CyclicCaller under **Interface Pointer**.
5. Under **Parameters** some predefined parameters have to be added. Press + under **Parameters** and select **Predefined** to select the predefined ParameterIDs:

You create the following parameters with the given names, in each case without code generation:

- "PID_LibraryID" with the name "LibraryID"
- "PID_ModuleClsId" with the name "ModuleClsId"
- "PID_Ctx_TaskSortOrders" with the name "SortOrders"
- "PID_Ctx_TaskOids" with the name "Contexts"
- "IOFFS_TcIoDataAreaSize" with the name "DataAreas"

The result in the overview:
6. Start the code generation.

7. Some changes must be made in the header of the module "Module1.h". First, delete the declarations of the interfaces that are no longer required and the corresponding maps in two places.

```cpp
class CModule1 : public ITComObject,
    public ITcADI,
    public ITcWatchSource
    {\
    public:
        DECLARE_IUNKNOWN()
        DECLARE_IPERSIST(CID_Untitled1CModule1)
        DECLARE_ITCOMOBJECT_LOCKOP()
        DECLARE_ITCADI()
        DECLARE_ITCWATCHSOURCE()
        DECLARE_OBJPARAWATCH_MAP()
        DECLARE_OBJDATAAREA_MAP()
    }
```

8. Create a new member variable in the header:

```cpp
ITcADIPtr m_spADI;
// TODO: Custom variable
```

9. Some changes need to be made to the source code of the module "Module1.cpp". First, delete the implementations of the interfaces that are no longer required in two places.

```cpp
BEGIN_INTERFACE_MAP(CModule1)
    INTERFACE_ENTRY_ITCOMOBJECT()
    INTERFACE_ENTRY(IID_ITcADI, ITcADI)
    INTERFACE_ENTRY(IID_ITcWatchSource, ITcWatchSource)
    INTERFACE_ENTRY(IID_ITcCyclic, ITcCyclic)
END_INTERFACE_MAP()
```

10. Delete the implementation of the maps belonging to the deleted interfaces:

```cpp
BEGIN_SETOBJPARA_MAP(CModule1)
    SETOBJPARA_DATAAREA_MAP()
    SETOBJPARA_VALUE(PID_TcTraceLevel, m_TraceLevelMax)
    SETOBJPARA_VALUE(PID_Module1Parameter, m_Parameter)
    SETOBJPARA_ITFPTR(PID_Ctx_TaskOid, m_spCyclicCaller)
END_SETOBJPARA_MAP()
```

```cpp
BEGIN_GETOBJPARA_MAP(CModule1)
    GETOBJPARA_DATAAREA_MAP()
    GETOBJPARA_VALUE(PID_TcTraceLevel, m_TraceLevelMax)
    GETOBJPARA_VALUE(PID_Module1Parameter, m_Parameter)
    GETOBJPARA_ITFPTR(PID_Ctx_TaskOid, m_spCyclicCaller)
END_GETOBJPARA_MAP()
```

```cpp
BEGIN_OBJPARAWATCH_MAP(CModule1)
    OBJPARAWATCH_DATAAREA_MAP()
END_OBJPARAWATCH_MAP()
```
11. The \texttt{m\_spAPI} pointer must be obtained in the transition P->S in the state machine:

```c++
HRESULT CModule1::SetObjStatePS(PTComInitDataHdr pInitData)
{
    m_Trace.Log(tlVerbose, FENTERA);
    HRESULT hr = S_OK;
    IMPLEMENT_ITCOMOBJECT_EVALUATE_INITDATA(pInitData);
    // query TcCOM object server for ITcADI interface with own object id, 
    // which retrieves a reference to the TMC module instance handler 
    m_spADI.SetOID(m_objId);
    hr = FAILED(hr) ? hr : m_spSrv->TcQuerySmartObjectInterface(m_spADI);
    // TODO: Add initialization code
}
```

12. Add the following in the state machine in the transition S->O, the calls to AddModuleToCaller or RemoveModuleFromCaller are omitted.

```c++
HRESULT hr = S_OK;
// Retrieve pointer to data areas via ITcADI interface from TMC module handler
///<AutoGeneratedContent id="DataAreaPointerInitialization">
///</AutoGeneratedContent>
// TODO: Add any additional initialization
if ( FAILED(hr) )
{
    SetObjStateOS();
}
```

13. Add the following in the state machine in the transition O->S, the call RemoveModuleFromCaller is omitted:

```c++
HRESULT hr = S_OK;
// Release pointer to data areas via ITcADI interface from TMC module handler
///<AutoGeneratedContent id="DataAreaPointerRelease">
///</AutoGeneratedContent>
// TODO: Add any additional deinitialization
```

14. The \texttt{AddModuleToCaller} and \texttt{RemoveModuleFromCaller} methods are not required and can be deleted.

15. Change the accesses to the DataAreas:

```c++
HRESULT CModule1::CycleUpdate(ITcTask* ipTask, ITcUnknown* ipCaller, ULONG_PTR context)
{
    HRESULT hr = S_OK;
    // TODO: Replace the sample with your cyclic code
    m_Counter+=m_pInputs->Value;
    m_pOutputs->Value=m_Counter;
    return hr;
}
```

16. Implement the \texttt{ITcOnlineChange}. The functions are created by the previous code generation, but must not return NOTIMPL.

```c++
HRESULT CModule1::PrepareOnlineChange(ITComObject* ipOldObj, TmcInstData* pOldInfo)
{
    HRESULT hr = S_OK;
    ULONG nData = sizeof(m_Parameter);
    PVOID pData = &m_Parameter;
    ipOldObj->TcGetObjPara(PID_Module1Parameter, nData, pData);
    return hr;
}
```
HRESULT CModule1::PerformOnlineChange(ITComObject* ipOldObj, TmcInstData* pOldInfo)
{
    HRESULT hr = S_OK;
    ULONG nData = sizeof(m_Counter);
    PVOID pData = &m_Counter;
    ipOldObj->TcGetObjPara(PID_Module1Counter, nData, pData);
    return hr;
}

17. Start the TMC Code Generator again to generate the code for the initialization of the data area pointers.

### 14.13 Initialization of TMC-member variables

All member variables of a TcCOM module must be initialized. The TMC Code Generator supports this with:

```cpp
///<AutoGeneratedContent id="MemberInitialization">
```

The TMC Code Generator replaces this with:

```cpp
///<AutoGeneratedContent id="MemberInitialization">
m_TraceLevelMax = tlAlways;
memset(&m_Parameter, 0, sizeof(m_Parameter));
memset(&m_Inputs, 0, sizeof(m_Inputs));
memset(&m_Outputs, 0, sizeof(m_Outputs));
///</AutoGeneratedContent>
```

The projects generated with the TwinCAT C++ Wizard prior to TwinCAT 3.1 Build 4018 do not use this property, but can easily be adapted by inserting this line in the corresponding code (e.g. Constructor):

```cpp
///<AutoGeneratedContent id="MemberInitialization">
```

### 14.14 Using PLC strings as method parameters

To transfer a character string from PLC to C++ as a method parameter, use a pointer with length information when declaring the method in TMC:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nStr</td>
<td>UDINT</td>
<td></td>
</tr>
<tr>
<td>pStr</td>
<td>SINT</td>
<td>Is Pointer</td>
</tr>
</tbody>
</table>

Such a method can be called by means of implementing a method within the wrapper function block:

```cpp
METHOD SetString : HRESULT
VAR_INPUT
    sSent : STRING(80);
END_VAR

IF (ipStateMachine <> 0) THEN
    SetString := ipStateMachine.SetString(SIZEOF(sSent), ADR(sSent));
END_IF
```

The reason is the different handling of method parameters in the two worlds:

- PLC: Uses the call by value for STRING(nn) data types.
- TwinCAT C++ (TMC): Uses the call by reference.
14.15 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 samples illustrate the use of ADS in C++ modules:

Sample03: C++ as ADS server [260]
Sample07: Receiving ADS Notifications [275]
Sample08: provision of ADS-RPC [276]

14.16 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:
15 Troubleshooting

This is a list of pitfalls and glitches within the handling of TwinCAT C++ modules.

15.1 Build - "Cannot open include file ntddk.h"

When creating a TwinCAT C++ project, this error message indicates a WinDDK problem on your engineering computer.

If this error message appears, carry out a check according to the WinDDK installation manual [22]:

• Make sure that WinDDK is installed.
• The environment variable WinDDK7 and its configured value must exist; see the description in the above-mentioned document. The value must correspond to the path where WinDDK is installed, including the first subfolder. Restart the computer after changing this value.

15.2 Build - "The target ... does not exist in the project"

In particular when transferring a TwinCAT solution from one machine to another, Visual Studio may display error messages to the effect that not all targets (such as Build, Rebuild, Clean) exist in the project.

Check the configuration of the "platform toolset" of the C++ project. It may need to be reconfigured if solutions migrate from one Visual Studio version to another:
15.3 Debug - "Unable to attach"

If this error message appears when starting the debugger in order to debug a TwinCAT C++ project, then a configuration step is missing:

In this case, navigate to **System -> Real-Time**, select the **C++ Debugger** tab and activate the option **Enable C++ Debugger**.
15.4 Activation – “invalid object id” (1821/0x71d)

If the ADS Return Code 1821 / 0x71d is reported during the course of the start, check the context of the module instance as described in Quick start [p. 90].

15.5 Error message - VS2010 and LNK1123/COFF

During the 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 being used, but without Service Pack 1, which is required [p. 20] for TwinCAT C++ modules.

Download the installation program for the service pack from Microsoft.

15.6 Using C++ classes in TwinCAT C++ module

When adding (non-TwinCAT) C++ classes using the Visual Studio context menu Add->Class..., the compiler/linker reports:

```
Error 4 error C1010: unexpected end of file while looking for precompiled header. Did you forget to add '#include ""' to your source?
```

Insert the following lines at the start of your generated class file:

```c
#include "TcPch.h"
#pragma hdrstop
```

15.7 Using afxres.h

In some templates afxres.h is included, which in some systems is not provided.
This header file can be replaced by winres.h.
16 C++-samples

16.1 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.
<table>
<thead>
<tr>
<th>Number</th>
<th>Title</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>Sample01: cyclic with IO module [259]</td>
<td>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 &quot;Value&quot; in the data area. The data area can be assigned to the physical IO or another logical input or another module instance.</td>
</tr>
<tr>
<td>02</td>
<td>Sample02: cyclic with IO task [260]</td>
<td>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.</td>
</tr>
<tr>
<td>03</td>
<td>Sample03: ADS Server Client [260]</td>
<td>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.</td>
</tr>
<tr>
<td>05</td>
<td>Sample05: CoE access via ADS [269]</td>
<td>Shows how CoE registers of EtherCAT devices can be accessed over ADS.</td>
</tr>
<tr>
<td>06</td>
<td>Sample06: ADS C# client uploads ADS symbols [270]</td>
<td>Shows how symbols in an ADS server can be accessed via the ADS interface. C# ADS client connects to a module implemented in PLC/C++/Matlab. Upload the available symbol information and read/write subscription for process values.</td>
</tr>
<tr>
<td>07</td>
<td>Sample07: Receiving ADS Notifications [275]</td>
<td>Describes the implementation of a TC3 C++ module that receives ADS notifications regarding data changes on other modules.</td>
</tr>
<tr>
<td>08</td>
<td>Sample08: provision of ADS-RPC [276]</td>
<td>Describes the implementation of methods that can be called by ADS via the task.</td>
</tr>
<tr>
<td>10</td>
<td>Sample10: Module communication: Use of data pointers [279]</td>
<td>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.</td>
</tr>
<tr>
<td>11</td>
<td>Sample11: Module communication: PLC module calls a method of a C-module [280]</td>
<td>This sample contains two parts • A C++ module which functions as a state machine that provides an interface with methods for starting/stopping and also for setting/maintaining the state machine. • Second PLC module for interacting with the first module by calling methods from the C++ module.</td>
</tr>
<tr>
<td>11a</td>
<td>Sample11a: Module communication: C-module cites a method in the C-module [308]</td>
<td>This sample contains two classes in one driver (can also be done between two drivers) • One module that provides a calculation method. Access is protected through a Critical section. • A second module that acts as the caller in order to use the methods in the other module.</td>
</tr>
<tr>
<td>12</td>
<td>Sample12: Module communication: IO mapping used [308]</td>
<td>• 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.</td>
</tr>
</tbody>
</table>
### Sample13: Module communication: C++ module calls PLC methods [309]

- Describes how a TwinCAT C++ module calls a PLC function block using TcCOM interface methods.

### Sample19: Synchronous File Access [312]

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.

### Sample20: FileIO-Write [313]

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.

### Sample20a: FileIO-Cyclic Read / Write [313]

A more extensive sample than S20 and S19. It describes the cyclic read and/or write access to files from a TC3 C++ module.

### Sample22: Automation Device Driver (ADD): Access DPRAM [314]

Describes how the TwinCAT Automation Device Driver (ADD) is to be written for access to the DPRAM.

### Sample25: Static Library [318]

Describes how to use the TC3 C++ static library contained in another TC3 C++ module.

### Sample26: Order of execution in a task [319]

Describes the determination of the task execution order, if a task is assigned to more than one module.

### Sample27: Using the JobTask

Describes the use of a JobTask by means of four variants.

### Sample30: Timing Measurement [321]

Describes the measurement of the TC3 C++ cycle or execution time.

### Sample31: Functionblock TON in TwinCAT3 C++ [322]

Describes the implementation of a behavior in C++, which is comparable to a TON function block of PLC / 61131.

### Sample37: Archive data [324]

Describes the loading and saving of the state of an object during the initialization and de-initialization.

### TcCOM

TcCOM samples [325]

Several samples are provided to illustrate the module communication between PLC and C++.

### 16.2 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 sample describes the quick start for the purpose of creating a C++ module that increments a counter on each cycle and assigns the counter to the logical output Value in the data area. The data area can be assigned to the physical IO or another logical input or another module instance. The sample is described step by step here [61] in the short instructions.

16.3 Sample02: Cyclic C++ logic, which uses IO from the IO Task

This article describes the implementation of a TC3 C++ module that uses 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 TcIoTaskImageAccessDrv with a module instance TcIoTaskImageAccessDrv_Obj1
- A "Task1" with an image, 10 input variables (Var1..Var10) and 10 output variables (Var11..Var20).
- They are linked: The instance is called by the task and uses the image of Task1.

The C++ code accesses the values via a data image, which is initialized during the transition from SAFEOP to OP (SO).

In the cyclically executed method CycleUpdate the value of each input variable is checked by calling the helper method CheckValue. If it is less than 0, the corresponding output variable is set to 1, if it is greater than 0, it is set to 2, if it is 0, the output is set to 3.

After activation of the configuration you can access the variables via the Solution Explorer and set them. Double-click on the Task1 image of system for an overview.

16.4 Sample03: C++ as ADS server

This article describes:
• The creation of a TC3 C++ module that acts as an ADS server. The server provides an ADS interface for starting / stopping / resetting a counter variable in the C++ module. The counter is available as a module output and can be assigned to an output terminal (analog or a number of digital IOs).

How the TC3 ADS server function written in C++ is to be implemented. [261]

• The creation of a C# ADS client to interact with the C++ ADS server. The client provides a UI for connection locally or via a network to an ADS server with the ADS interface to be counted. The UI enables the starting / stopping / reading / overwriting and resetting of the counter.

Sample code: ADS Client UI written in C# [265].

Understanding the sample

Options for the automatic determination of an ADS port are used in the sample. The disadvantage of this is that the client has to be configured at each start in order to access the correct ADS port.

Additionally, the ADS port can be hard-coded in the module as shown below. Disadvantage here: The C++ module cannot be instanced more than once as it is not possible to share an ADS port.

16.4.1 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 a C++ module that acts as an ADS server. The server grants access to a counter that can be started, stopped, and read.

The header file of the module defines the counter variable m_bCount, and the corresponding .cpp file initializes the value in the constructor and implements the logic in the CycleUpdate method.

The AdsReadWriteInd method in the .cpp file analyzes the incoming messages and returns the return values. A define in the header file is added for a further added message type.

Details such as the definition of the ADS message types are described in the following cookbook, where you can compile the sample manually.

Cookbook

This is a step by step description about the creation of the C++ module.

1. Create a new TwinCAT 3 project solution

Follow the steps for creating a new TwinCAT 3 project [62].

2. Create a C++ project with ADS port

Follow the steps for the creation of a new TwinCAT 3 C++ project [63].

Select TwinCAT Module Class with ADS port in the Class templates dialog.

3. Add the sample logic to the project

1. Open the header file <MyClass>.h (in this sample Module1.h) and add the counter m_bCount to the protected area as a new member variable:

```cpp
class CModule1
    : public ITComObject
    , public ITcCyclic
    , ...
{
    public:
        DECLARE_IUNKNOWN()
    ....
    protected:
        DECLARE_ITCOMOBJECT_SETSTATE();
        //</AutoGeneratedContent id="Members">
        ITcCyclicCallerInfoPtr m_spCyclicCaller;
        ....
        //</AutoGeneratedContent>
        ULONG m_ReadByOidAndPid;
        BOOL m_bCount;
};
```

2. Open the class file <MyClass>.cpp (in this sample Module1.cpp) and initialize the new values in the constructor:

```cpp
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:

<table>
<thead>
<tr>
<th>indexGroup</th>
<th>indexOffset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x01</td>
<td>0x01</td>
<td>m_bCount = TRUE, counter is incremented.</td>
</tr>
<tr>
<td>0x01</td>
<td>0x02</td>
<td>Counter value is transferred to ADS client.</td>
</tr>
<tr>
<td>0x02</td>
<td>0x01</td>
<td>m_bCount = FALSE, counter is no longer incremented.</td>
</tr>
<tr>
<td>0x02</td>
<td>0x02</td>
<td>Reset counter.</td>
</tr>
<tr>
<td>0x03</td>
<td>0x01</td>
<td>Overwrite counter with value transferred by ADS client.</td>
</tr>
</tbody>
</table>

These parameters are defined in modules1Adsk.h – change the source code to add a new command for IG_RESET.

```c
#include "TcDef.h"
enum Module1IndexGroups : ULONG
{
    Module1IndexGroup1 = 0x00000001,
    Module1IndexGroup2 = 0x00000002, // add command
    IG_OVERWRITE = 0x00000003 // and new command
};
enum Module1IndexOffsets : ULONG
{
    Module1IndexOffset1 = 0x00000001,
    Module1IndexOffset2 = 0x00000002
};
```

Change the source code in your <MyClass>::AdsReadWriteInd() method (in this case in Module1.cpp).

```c
switch(indexGroup)
{
    case Module1IndexGroup1:
        switch(indexOffset)
        {
            case Module1IndexOffset1:
                // TODO: add custom code here
                m_bCount = TRUE; // received IG=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:
```
3.b. Add sample logic to the cyclic part

The method `<MyClass>::CycleUpdate()` is cyclically called - this is the place to modify the logic.

```cpp
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;
}
```

In this case the counter `mCounter` is incremented if the boolean variable `m_bCount` is true.

Insert this If-Case to your cyclic method

In this case the counter is incremented if the boolean variable is true.

4. Execute server sample

1. Run the TwinCAT TMC Code Generator [61] in order to provide the inputs/outputs for the module.
2. Save the project.
3. Compile [61] the project.
4. Create a module instance.
5. Create a cyclic task and configure the C++ module for the execution in this context.
6. Scan the hardware IO and assign the symbol Value of outputs to certain output terminals (this is optional).
7. Activate [73] the TwinCAT project.
   => The sample is ready for operation.
5. Determine the ADS port of the module instance

Generally the ADS port may be

- pre-numbered, so that the same port is always used for this module instance.
- kept customizable, in order to offer several module instances the option to have their own ADS port assigned on startup of the TwinCAT system.

In this sample the default setting (keep flexible) is selected. First of all you have to determine the ADS port that was assigned to the module that has just been activated.

1. Navigate to the module instance.
2. Select the Parameter Online tab.
   - 0x8235 or decimal 33333 is assigned to the ADS port (this may be different in your sample). If more and more instances are created, each instance is allocated its own unique AdsPort.
   - The counter is still at "0" because the ADS message to start the incrementation has not been sent.

The server part is completed - continue with ADS client sends the ADS messages [p. 265].

Also see about this
- Create TwinCAT 3 C++ Module instance [p. 68]
- Create a TwinCAT task and apply it to the module instance [p. 70]

16.4.2 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!
- Unpack the downloaded ZIP file.
- Open the sln file contained in it with Visual Studio.
3. Create the sample on your local machine (right-click on the project and click on Build).
4. Start the program with a right-click on Project, Debug->Start new instance.

**Description**

This client performs two tasks:

- Testing the ADS-server, which was described before.
- Providing sample code for implementing a ADS-client

**Using the client**

**Selecting a communication partner**

Enter both ADS parameters in order to specify your ADS communication partner:

- **NetID:**
  127.0.0.1.1.1 (for ADS 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** [261], 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.

![ADS Client UI](image)

ADS messages are sent to the ADS server with the help of the **Start / Read / Stop / Overwrite / Reset** buttons.
The specific **indexGroup / indexOffset** commands were already **designed** in the ADS interface of the ADS server [261].
The result of clicking on the command buttons can also be seen in the module instance in the Parameters (online) tab.

C# program

Here is the "Core" code of the ADS client – download for the GUI or ZIP file above.

```csharp
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);
```
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);
}
}

16.5 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 AdsAdress:

4. The module parameters are read out and displayed by the sample code during the course of the initialization:

16.6 Sample06: UI-C#-ADS client uploading the symbolic from module

This article describes the implementation of an ADS client to
- connect to an ADS server that provides a process image (data area); the connection can be established locally or remotely via a network,
• upload symbol information,
• read / write data synchronously,
• subscribe to symbols, in order to obtain values on change as callback.

Download

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 sln file contained in it with Visual Studio.
3. Create the sample on your local machine (right-click on the project and click on "Build").
4. Start the program with a right-click on Project, Debug->Start new instance.

The client sample should be used with example 03 "C++ as ADS server".

Open Sample 03 [p 261] before you begin with this client-side sample!

Description

The possibilities of the ADS are described on the basis of this sample.

The details of the implementation are described in Form1.cs, which is included in the download. The connection via ADS with the target system is established in the btnLoad_Click method, which is called on clicking on the Load Symbols button. From there you can explore the different GUI functions.

Background information:

For this ADS client it is irrelevant whether the ADS server is based on TwinCAT 2 or TwinCAT 3. It also doesn't matter if the server is a C++ module, a PLC module or an IO task without any logic.

The ADS client UI

On starting of the sample the user interface (UI) is displayed.
Selecting a communication partner

After starting the client, enter the two ADS parameters, 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).

---

**Do not confuse the ADS server port with the regular ADS port.**

Do not confuse the ADS server port (which was explicitly implemented in sample 03 for providing your own message handler) with the regular ADS port for the purpose of access to symbols (this is provided automatically, without the need for user intervention):

The regular ADS port is required to access symbols. You can find the AdsPort for the IO task of your instance or the module instance yourself (since the module is executed in the context of the IO task).
Navigate to IO task Task1 and note the value of the port: 350.

Since the C++ module instance is executed in the context of Task1, the ADS port is also 350.

Enabled symbols for access available via ADS

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 C/S column.

In this sample all symbols are marked and therefore available for ADS access.

After making the changes, click on Activate configuration.
Load symbols

Once the NetID and the ADS port have been set up, click on the Load Symbols button to make a connection with the target system and load the symbols.

All available symbols are then visible. You can then:

- Write a new value:
  select a symbol in the left-hand tree, e.g. Counter.
  enter a new value in the edit box Value on the right-hand side 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.
16.7 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 2 modules for this purpose.
• A C++ module, which registers for querying ADS notifications of a variable.
• For a simple understanding: A PLC program that provides a variable MAIN.PlcVar. If its value changes, an ADS notification is sent to the C++ module.
• The C++ module utilizes the message recording options. For a better understanding of the code, simply start the sample and note the output / error log when you change the value Main.PlcVar of the PLC module.

The address is prepared during the module transition PREOP->SAFEOP (SetObjStatePS). The CycleUpdate method contains a simple state machine, which sends the required ADS command. Corresponding methods show the receipts.

The inherited and overloaded method AdsDeviceNotificationInd is called when a notification is received. During shutdown, ADS messages are sent during transition for the purpose of logoff (SetObjStateOS), and the module waits for receipts of confirmation until a timeout occurs.

Start of the module development
Creating a TwinCAT 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 [› 210]

16.8 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 [108].

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.

If the type information of the methods is to be available on the target system, the TMI file of the module can be transferred to the target system.

The TwinCAT OPC-UA server offers the option to call these methods also by OPC-UA – the TMI files are required on the target system for this.

C++ 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:

```
Press key to call all methods

Calling foo.bar.IRpcTest#CallIn
Send: 0

Calling foo.bar.IRpcTest#CallIn#HResult
Value given: 1
ReturnCode: 0

Calling foo.bar.IRpcTest2#AddWHRResult
Value given A: 1
Value given B: 2
Value got (A+B): 3

Calling foo.bar.IRpcTest2#AddWHRResult
Value given A: 1
Value given B: 2
ReturnCode: 0
Value got (A+B): 3
```

### 16.9 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 setting the input Bit1, the output Switch1 will be set accordingly.
• When setting the input Bit2, the output Switch2 will be set accordingly.
• When the input ValueIn is set, the output ValueOut is incremented twice in each cycle.

All modules are configured such that they have the same task context, which is necessary since access via pointers offers no synchronization mechanism. The order of execution corresponds to the order specified on the context configuration tab. This value is passed on as parameter SortOrder and stored in the smart pointer of the cyclic caller (m_spCyclicCaller), which also contains the object ID of the cyclic caller.

Understanding the sample

The module ModuleDataInOut has input and output variables. They are linked with the corresponding variables of the data provider.

The module ModuleDataProvider provides an input and output data array and implements the ITcIoCylic interface. The method InputUpdate copies data from the input variables to the DataIn symbol of the standard data area Data, and the method OutputUpdate copies data from the DataOut symbol to the output variables.

The modules ModuleDataAccessA and ModuleDataAccessB contain pointers to data areas of the data provider via links. These pointers are initialized during the transition from SAFEOP to OP. ModuleDataAccessA cyclically sets BitOut1 according to Bit1. ModuleDataAccessB accordingly, with BitOut2 / Bit2. Both increment ValueOut through multiplication of the internal counter with the value ValueIn.

It is important that all modules are executed in the same context, since there is no synchronization mechanism via data pointers. The order of execution is defined by the Sort Order on the Context tab of the respective module. This is provided as parameter SortOrder in the SmartPointer (m_SpCyclicCaller), which also includes the ObjectID.

16.10 Sample11: module communication: PLC module invokes method of C-module

This article describes the implementation:

• of a C++ module [281] 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 [296] 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.

16.10.1 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 [280] - i.e. affecting the C++ code from the PLC.
- Calling the function from the C++ logic [308] - i.e. interaction between two C++ modules.

This article describes:

- Step 1: create a new TwinCAT 3 project [281].
- Step 2: create a new TwinCAT 3 C++ driver [282].
- Step 3: create a new TwinCAT 3 interface.
- Step 4: add methods to the interface [285].
- Step 5: add a new interface to the module.
- Step 6: start the TwinCAT TMC Code Generator to generate code for the module class description [290].
- Step 7: implement the member variables and the constructor.
- Step 8: implement methods.
- Step 9: implement cyclic update.
- Step 10: compile code [293]
- Step 11: create an instance of the C++ module.
- Step 12: done. Check the results.

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**

<table>
<thead>
<tr>
<th>NOTE</th>
</tr>
</thead>
</table>
| **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. |
Start the TMC editor by double-clicking on StateMachineDrv.tmc.

Select Data Types within the TMC editor.

Add a new interface by clicking on Add a new interface.

A new entry IInterface1 is then listed.

Open IInterface1 by double-clicking in order to change the properties of the interface.

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

The interface has been created.

Step 4: add methods to the interface

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

Click on the + button to create a new default method, Method1.
2. Replace the default name Method1 by a more meaningful name, in this sample "Start".

3. Add a second method and name it "Stop".

4. Add a third method and name it "SetState".
5. Subsequently, you can add parameters by clicking on **Add a new parameter** or edit parameters of the `SetState` method.

   ![Parameter Setup](image)

   The new parameter, Parameter1, is generated by default as **Normal Type INT**.

6. Click on the name "Parameter1" and change the name in the edit box to "State".

7. After `Start`, `Stop` and `SetState` have been defined, define a further method.

8. Rename it "GetState".

9. Add a parameter and name it "pState" (which is conceived to become a pointer later on).

10. Change Normal Type to **Is Pointer**.

   ![Parameter Setup](image)
You then obtain a list of all methods. You can change the order of the methods with the buttons.

### Step 5: add a new interface to the module

1. Select the module that is to be extended by the new interface - in this case select the destination `Modules->CStateMachineModule`.

2. Extend the list of implemented interfaces by a new interface with **Add a new interface to the module** by clicking on the + button.
3. All available interfaces are listed - select the new interface IStateMachine and end with OK.

⇒ The new interface IStateMachine is part of the module description.
Step 6: Start the TwinCAT TMC Code Generator

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

The module StateMachineModule.cpp now contains the new interfaces:
- CModule1: Start()
- CModule1: Stop()
- CModule1: SetState(SHORT State)
- CModule1: GetState(SHORT* pState).

Step 7: 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:
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:
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.

Also see about this
- TwinCAT 3 C++ module providing methods [p. 284]
- TwinCAT 3 C++ module providing methods [p. 288]
- TwinCAT 3 C++ module providing methods [p. 290]
- TwinCAT 3 C++ module providing methods [p. 291]
- TwinCAT 3 C++ module providing methods [p. 292]
- TwinCAT 3 C++ module providing methods [p. 294]
16.10.2 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 the available interfaces.
- Step 2: create a new PLC project.
- Step 3: add a new FB state machine (which acts as the proxy that calls the C++ module methods).
- Step 4: clean up the function block interface.
- Step 5: add the FB methods "FB_init" and "exit".
- Step 6: implement the FB methods.
- Step 7: call the FB state machine in the PLC.
- Step 8: compile the PLC code.
- Step 9: link the PLC FB with the C++ instance.
- Step 10: observe the execution of both modules, PLC and C++.

Step 1: check available interfaces

Option 1:
1. Navigate to the C++ module instance.
2. Select the Interfaces tab.
   ⇒ The IStateMachine interface is in the list with its specific IID (Interface ID).

Option 2:
1. Navigate to System.
2. Select the Interfaces tab.
The **IStateMachine** interface is in the list with its specific IID (Interface ID).

The lower section shows the stored code in different programming languages.

**Step 2: creating a new PLC project**

A standard PLC project called "PLC-calling state machine" is created.

1. Right-click on the PLC node.
2. Select **Standard PLC Project**.
3. Adapt the name.

   ![Image showing the creation of a new PLC project](image)

   The project has been successfully created.

**Step 3: add a function block (FB) (which serves as the proxy for calling the C++ module methods)**

1. Right-click on **POUs**.

   ![Image showing the addition of a function block](image)
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**.

![Add POU dialog](image)

- 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*...
2. Add the methods FB_exit and FB_init - both with Structured Text (ST) as the implementation language. They are available as predefined name.

3. Exit the dialog in each case by clicking on **Open**.

   In the end, all required methods are available:

   ![Add Method window]

   **Step 6: implement FB methods**

   Now all methods have to be filled with code.
NOTE

Missing attributes lead to unexpected behavior
Attribute statements in brackets represent code to be added.

More precise information on the attributes is given in the PLC documentation.

1. Implement the variable declarations of the FB_Statemachine. The FB itself does not require cyclically executable code.

```
FB_StateMachine* FB_Init;
```

2. Implement the variable declarations and the code area of the method FB_exit.

```
METHOD FB_exit : BOOL
VAR_INPUT
  bInCopyCode : BOOL: // if TRUE, the exit method is called
END_VAR

IF NOT bInCopyCode THEN // no online change
  FW_safeRelease (ADR(ipStateMachine));
END_IF
```

3. Implement the variable declarations and the code area of the method FB_init.

```
METHOD FB_init : BOOL
VAR_INPUT
  bInitRetains : BOOL: // if TRUE, the retain variables are initialized (warm start / cold start)
  bInCopyCode : BOOL: // if TRUE, the instance afterwards gets moved into the copy code (online change)
END_VAR

IF NOT bInCopyCode THEN // no online change
  IF ipStateMachine = 0 THEN
    hrInit := FW_GetInstance (cId := cIdInstance,
                                iid := RC_GSMAB_TID_LIST.IID_IStateMachine,
                                plink := ADR(ipStateMachine));
  END_IF
END_IF
```
4. Implement the variable declaration and the code area of the method `GetState` (the generated pragmas can be deleted as they are not required for a proxy FB).

```c++
FB_StateMachine.GetState* FB_StateMachine.GetState* FB_StateMachine.GetState*
1 METHOD GetState : HRESULT
2 VAR_INPUT
3 pState : POINTER TO INT;
4 END_VAR
5

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).

```c++
FB_StateMachine.SetState* FB_StateMachine.GetState* FB_StateMachine.GetState*
1 METHOD SetState : HRESULT
2 VAR_INPUT
3 State : INT;
4 END_VAR
5

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).

```c++
1 METHOD Start : HRESULT

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).

```c++
FB_StateMachine.Stop FB_StateMachine.Start FB_StateMachine.SetState
1 METHOD Stop : HRESULT

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 nCounter is also set to "0", so that everything starts again from the beginning.

```
PROGRAM MAIN
VAR
  nCounter     : INT;
  ncurrentState : INT;
  fbStateMachine : FB_StateMachine;
END_VAR

nCounter := nCounter + 1;

IF nCounter = 500 THEN
  fbStateMachine.SetState(1);
  fbStateMachine.GetState(ADR(ncurrentState));
  fbStateMachine.Start();
END_IF

IF nCounter = 1000 THEN
  fbStateMachine.SetState(2);
  fbStateMachine.GetState(ADR(ncurrentState));
END_IF

IF nCounter = 1500 THEN
  fbStateMachine.Stop();
  nCounter := 0;
END_IF
```
Step 8: compile PLC code

1. Right-click on the PLC project and click on Build.

The compilation result shows “1 succeeded - 0 failed”.

Step 9: link PLC FB with C++ instance

The benefits of all previous steps now become apparent:

The PLC FB FB_StateMachine can be configured with regard to linking with every instance of the C++ module StateMachine. This is a very flexible and powerful method of connecting PLC and C++ modules on the machine with each other.

1. Navigate to the instance of the PLC module in the left-hand tree and select the Symbol Initialization tab on the right-hand side.

⇒ All instances of FB_StateMachine are listed; in this sample we have only defined one FB instance in POU MAIN.
2. Select the drop-down field **Value** and then the C++ module instance that is to be linked to the FB instance.

![Image of TwinCAT configuration](image)

⇒ 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 [72] and follow the steps in the quick start [61] in order to start the debugging (right-hand side of the following illustration).

![Image of TwinCAT debugging](image)

**Also see about this**

- Calling methods offered by another module via PLC [296]
- Calling methods offered by another module via PLC [297]
- Calling methods offered by another module via PLC [300]
- Calling methods offered by another module via PLC [302]
16.11 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 DataWriteCounterModul parameters of CModuleDataWrite and DataReadCounterModulo (CModuleDataRead) enable the moment to be determined, at which the module instances initiate the access.

CriticalSections are described in the SDK in TcRtInterfaces.h and are therefore intended for the real-time context.

16.12 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 (linked) to the input "value" of the Back instance.

The Back instance copies the input Value to its output Value, which can be monitored by the user (see the following quick start steps to start debugging: Debugging a TwinCAT 3 C++ project [75])

Ultimately, the user can define the input value of the Front instance and observe the output value of Back.

16.13 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 [325], 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 [325]. 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.

In the actual code of the module in CycleUpdate() the interface pointer is set using the object ID transmitted from the PLC. It is important that this happens in the CycleUpdate() and thus in real-time context, since the PLC must first provide the function block. When this has taken place once, the methods can be called.

In addition, as can be seen above, the interface pointer is cleared when the program shuts down. This happens in the SetObjStateOS method.

4. Now build the C++ project.
5. Create an instance of the module.
6. Connect the input of the C++ module to the output of the PLC.

The project can be started. When the PLC is running, the OID is made known through the mapping to the C++ instance. Once this has occurred, the method can be called.

16.14 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 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 the Error List tab in the TwinCAT 3 output window. The amount of information can be set by changing the variable TraceLevelMax in the instance TcFileTestDrv_obj1 in Parameter (Init) tab.

16.15 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.

16.16 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

The sample describes how to access files for reading and/or writing using the CycleUpdate method, i.e. in a cyclic manner.
This sample contains the following projects and module instances.

- A static library (TcAsyncFileIo) offers the file access. The code for the file access can be shared, therefore this code is located in a static library that is used by the driver projects.
- A driver (TcAsyncBufferReadingDrv) provides two instances:
  - ReadingModule: uses the static library to read the AsyncTest.txt file.
  - WriteDetectModule: Detects write operations and initiates read operations.
- A driver (TcAsyncBufferWritingDrv) provides one instance:
  - WriteModule: uses the static library to write to the AsyncTest.txt file.

On starting the sample, the writing module begins to write data to the file located in the boot project path (normally C:\TwinCAT\3.x\Boot\AsyncTest.txt). The input variable bDisableWriting can be used to prevent writing.

The objects are connected to one another: once the writing is complete, the WritingModule triggers the DetectModule of TcAsyncBufferReadingDrv. As a result of this, the ReadingModule initiates a reading procedure.

Observe the nBytesWritten / nBytesRead output variables of the WritingModule / ReadingModule. Over and above that, protocol messages are generated at verbose level. As before, these can be configured with the help of the TraceLevelMax parameter of the module.

- A driver (TcAsyncFileFindDrv) provides one instance
  - FileFindModule: list the files of a directory with the help of the static library.

Initiate the action with the help of the input variable bExecute. The FilePath parameter contains the directory whose files are to be listed (default value: c: \TwinCAT\3.1\Boot\*).

Observe the sequence tracking (verbose protocol level) with regard to the list of files found.

**Understanding the sample**

The project TcAsyncFileIO contains various classes in a static library. This library is used by driver projects for reading and writing.

Each class is intended for a file access operation such as Open / Read / Write / List / Close / .... Since execution takes place in a cyclic real-time context, each operation has a status, and the class encapsulates this state machine.

As an entry point for understanding the file access, begin with the classes TcFsmFileReader and TcFsmFileWriter.

If too many history tracking messages occur, which hamper understanding of the sample, you can disable modules!

See also

- Sample S19 [312]
- Sample S20 [313]
- Sample S25 [318]
- Interface ITcFileAccess [178] / Interface ITcFileAccessAsync [186]


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.

**Note** Read the configurations details below prior to the activation.

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 ![Activate](image).

⇒ The sample is ready for operation.

**Description**

This sample is intended to switch the Link Detect bit of the network adapter (i.e. of an CX5010) cyclically on and off.

The C++ module is connected to the NOV/DP-RAM device via the PciDeviceAdi interface pointer of the C++ module.

**Configuration**

To make the sample work, the hardware addresses must be configured to match your own hardware. Check the PCI configuration:

![PCI Configuration](image)

To check whether the communication with NOV/DPRAM is set up correctly, use the DPRAM (Online) view:
16.18 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
   A NULL-pointer access takes place once again here, but this time a TcFinally{} block is executed in every case.
4. A customer-specific structured exception
   By means of TcRaiseException() an exception is generated that is intercepted and processed by the
   FilterException() method. Since this is an exception defined in the module, the FilterException() method additionally outputs a further (specific) message.

5. Exception with Continue block
   Here too, a NULL-pointer access takes place once again with TcExcept{}; however, this time the exception is forwarded after handling in the FilterException() method so that the further TcExcept{} also handles the exception.

16.19 Sample25: Static Library

This article describes the implementation and use of a module of a static TC3 C++ library.

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 DriverUsingStaticLib project uses the static content of the StaticLib project.

StaticLib:
On the one hand, StaticLib offers the function ComputeSomething in the StaticFunction.h/.cpp.
On the other hand an interface ISampleInterface is defined (see TMCEditor) and implemented in the MultiplicationClass.

DriverUsingStaticLib:
In the CycleUpdate method of the ModuleUsingStaticLib, both the class and the function of StaticLib is used.

Understanding the sample
Follow the steps below to create and use a static library.

- Manual recompilation
  Note that Visual Studio does not automatically recompile the static library during driver development. Do that manually.

  ✔ During development of a C++ project use the TwinCAT Static Library Project template for creating a static library.
  ✔ For the following steps use the Edit dialog of VisualStudio, so that afterwards %(AdditionalIncludeDirectories) or %(AdditionalDependencies) is used.
1. In the driver add the directory of the static library to the compiler under Additional Include Directories.

   ![Additional Include Directories](image1)

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:

   ![Project Properties](image2)

### 16.20 Sample26: Order of execution in a task

This article describes the determination of the task execution order if more than one module is assigned to a task.

**Download**

Here you can access the source code for this sample.

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 the SortOrder module, which is instanced twice. The sort order determines the execution order, which can be configured via the TwinCAT Module Instance Configurator [151].
For example, the CycleUpdate method tracks the object name and ID along with the sort order of this module. On the console window you can see the order of execution:

```
05.09.2014 13:01:14 343 ms | TCOM Server (10): CSortOrderModule::CycleUpdate() I am 'SortOrder1' (0x0100010) w/ SortOrder 150
05.09.2014 13:01:14 343 ms | TCOM Server (10): CSortOrderModule::CycleUpdate() I am 'SortOrder2' (0x0100020) w/ SortOrder 170
```

In the sample, one instance is configured with Sort Order 150 and one with 170, while both instances are assigned to a task.

**Understanding the sample**

- A TcCOM C++ module with cyclic IO.
  1. The module requires a context-based parameter Sort order of task, which will automatically select PID_Ctx_TaskSortOrder as name.
     Note that the parameter must be an alias (specification) of data type UDINT:

```
typedef UDINT 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.

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.

```cpp
////////////////////////////////////////////////////////////////////////////
// Set parameters of CSortOrderModule
BEGIN_SETOBJPARAMA_MAP(CSortOrderModule)
  SETOBJPARAMA_ITF PTR(PID_CTX_TaskOid, m_spCyclicCaller)
  SETOBJPARAMA_ITF PTR(PID_TcTraceLevel, m_TraceLevelMax)
END_SETOBJPARAMA_MAP()
```
5. In this sample the object name, ID and sort order are tracked cyclically:

```cpp
// 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 [151]:

---

**16.21 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 since January 1, 1601 (UTC)
- Querying the distributed clock time when the task cycle starts in nanoseconds since January 1, 2000
- Querying the time when the method is called at intervals of 100 nanoseconds since January 1, 1601 (UTC)

See also
ITcTask interface [16.22]

16.22 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 this module is comparable with a module that was created with the Cyclic IO wizard. m_input.Value is added to m_Output.Value. As opposed to the Cyclic IO module, this module only adds m_input.Value to m_Output.Value if the defined time interval (1000 ms) has elapsed.

This is achieved with the help of a CTON class, which is comparable with the TON function block of PLC / 61131.

Understanding the sample
The C++ class CTON (TON.h/.cpp) provides the behavior of a TON function block of PLC / 61131. The Update() method is comparable with the rump of the function block, which has to be called regularly.

The Update() method contains two "in" parameters:
- IN1: Starts the timer switch on a rising edge and resets the timer switch on a falling edge.
- PT: Describes the time to wait before Q is set.

And two "out" parameters:
- Q: TRUE if PT exhibited a rising edge seconds after IN.
- ET: Designates the elapsed time.

Beyond that, ITcTask must be provided to query the time base.
16.23 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 an instance of the TcEthernetSample module, which sends and received ARP packets for the purpose of determining the remote hardware address (MAC).

The CycleUpdate method implements a rudimentary state machine for sending ARP packets and waiting for a response with a timeout.

The sample uses two Ethernet components from TwinCAT:

1. An ITcEthernetAdapter (instance name in the sample is m_spEthernetAdapter) represents an RT Ethernet adapter. It enables access to the adapter parameters such as hardware MAC address, link speed and link errors. It can be used to send Ethernet frames and enables a module instance to register itself as an ItcIoEthProtocol via the registerProtocol method.
2. The ITcIOoEthProtocol is extended by the sampling module, which ensures that a notification takes place via the ITcEthernetAdapter in case of Ethernet events.

Configuration

The downloaded TwinCAT project must be configured for execution in a network environment. Please carry out the following steps:

✓ This sample demands the use of the TwinCAT driver by the Ethernet card.
1. Start TcRteInstall.exe either from the XAE via the menu TwinCAT->Show Realtime Ethernet compatible devices… or from the hard disk on the XAR systems.

2. You may have to install and activate the driver with the help of the buttons.

3. TwinCAT must know which Ethernet card is to be used. Open the project in XAE and click on Select I/O / Devices / Device 1 (RT-Ethernet Adapter).

4. Click on the Adapter tab and select the adapter with Search.

5. TcEthernetSample_Obj1 must be configured. Open the instance window and set the following values:
   - Parameter (Init): SenderIpAddress (IP of the network adapter configured in step 2)
   - Parameter (Init): TargetIpAddress (IP of target host)
   - Interface pointer: EthernetAdapter must point to I/O / Devices / Device 1 (RT-Ethernet Adapter).

16.24 Sample37: Archive data

The sample TcCOM object archive describes restoration and saving of an object state during initialization and deinitialization.

- **TwinCAT supports retain data**
  
  TwinCAT also supports retain data, in order to utilize the NOVRAM of a device to make data persistent.

**Download**

- **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 sample TcCOM object archive describes restoration and saving of an object state during initialization and deinitialization. The state of the sample class CmoduleArchive corresponds to the value of the counter CModuleArchive::m_counter.
During the transition from PREOP to SAFEOP, i.e. when calling the method 
CModuleArchive::SetObjStatePS(), the object archive server (ITComObjArchiveServer) is used to create an 
object archive for reading, which is accessed via the interface ITComArchiveOp. This interface provides 
overloads from operator>>() in order to read them in the archive.

During the transition from SAFEOP to PREOP, i.e. when calling the method 
CModuleArchive::SetObjStateSP(), the TCOM object archive server is used to create an object archive for 
writing, which is accessed via the interface ITComArchiveOp. This interface provides overloads from 
operator<<() in order to write them in the archive.

16.25   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 [325] 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 [335] 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 target system, a C++ development environment is not necessary on the target system or on the 
development computer.

The TcCOM_Sample03 sample [339] 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.

16.25.1   TcCOM_Sample01_PlcToPlc

This sample describes a TcCOM communication between two PLCs.

Functionalties provided by a function block in the first PLC (also called "provider" in the sample), are called 
from the second PLC (also called "caller" in the sample). To this end it is not necessary for the function block 
or its program code to be copied. Instead the program works directly with the object instance in the first PLC.

Both PLCs must be in a TwinCAT runtime. In this connection a function block offers its methods system-wide 
via a globally defined interface and represents itself a TcCOM object. As is the case with every TcCOM 
object, such a function block is also listed at runtime in the TcCOM Objects node.
The procedure is explained in the following sub-chapters:

- Creating an FB in the first PLC that provides its functionality globally [326]
- Creating an FB in the second PLC that, as a simple proxy, also offers this functionality there [331]
- Execution of the sample project [333]

Downloading the sample: https://infosys.beckhoff.com/content/1033/TC3_C/Resources/zip/2343046667.zip

Race Conditions in the case of Multi-Tasking (Multi-Threading) use

The function block that provides its functionality globally is instantiated in the first PLC. It can be used there like any function block. In addition, if it is used from a different PLC (or, for example, from a C++ module), make sure that the methods offered are thread-safe, as the various calls could take place simultaneously from different task contexts or mutually interrupt one another, depending on the system configuration. In this case the methods must not access member variables of the function block or global variables of the first PLC. If this should be absolutely necessary, prevent simultaneous access. Observe the function TestAndSet() from the Tc2_System library.

System requirements

<table>
<thead>
<tr>
<th>TwinCAT version</th>
<th>Hardware</th>
<th>Libraries to be integrated</th>
</tr>
</thead>
<tbody>
<tr>
<td>TwinCAT 3.1, Build 4020</td>
<td>x86, x64, ARM</td>
<td>Tc3_Module</td>
</tr>
</tbody>
</table>

16.25.1.1 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.

6. Now implement the I_Calculation interface in the FB_Calculation function block and append the c++_compatible attribute.
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.

```cpp
1  METHOD FB_reinit : BOOL
2  VAR_INPUT
3  END_VAR

1  SUPER.FB_reinit();
```

10. Implement the `TcQueryInterface()` method of the Interface ITcUnknown. 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()`.

```cpp
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
10  VAR_INPUT
11    iid : REFERENCE TO IID;
12    pipItf : POINTER TO PVOID;
13  END_VAR

14  VAR
15    ipCalc : I_Calculation;
16  END_VAR

17  IF GuidEqual(ADR(iid), ADR(TC_GLOBAL_IID_LIST.IID_I_Calculation)) THEN
18    ipCalc := THIS; // cast to interface pointer
19    pipItf^ := ITCUNKNOWN_TO_PVOID(ipCalc);
20    TcAddRef();
21    TcQueryInterface := S_OK;
22  ELSE
23    TcQueryInterface := SUPER.TcQueryInterface(iid, pipItf);
24  END_IF
```

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.
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.

16.25.1.2 Creating an FB which likewise offers this functionality there as a simple proxy in the second PLC,

1. Create a PLC and append a new function block there.
   - This proxy function block should provide the functionality which was programmed in the first PLC. It does this via an interface pointer of the type of the global interface I_Calculation.

2. In the declaration part of the function block declare as an output an interface pointer to the global interface which later provides the functionality outward.
3. In addition create the object ID and the interface ID as local member variables. While the interface ID is already available via a global list, the object ID is assigned via a link in the process image.

4. Implement the PLC proxy function block. First add the GetInterfacePointer() method to the function block. The interface pointer is fetched to the specified interface of the specified TcCOM object with the help of the FW_ObjMgr_GetObjectInstance() function. This will only be executed if the object ID is valid and the interface pointer has not already been allocated. The object itself increments a reference counter.
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.GetInterfacePointer
```

6. Instantiate the Proxy function block FB_CalculationProxy in the application and call its method GetInterfacePointer() to get a valid interface pointer.

```
IF NOT bInCopyCode THEN // if not online change
    FW_SafeRelease(ADR(ip));
END_IF
```

This completes the implementation of the Proxy function block.

6. Instantiate the Proxy function block FB_CalculationProxy in the application and call its method GetInterfacePointer() to get a valid interface pointer.

An instance of the proxy block is declared in the application to call the methods provided via the interface. The calls themselves take all place over the interface pointer defined as output of the function block. As is typical for pointers a prior null check must be made. Then the methods can be called directly, also via Intellisense.

```
main
```

The sample is ready for testing.

Order irrelevant

The sequence in which the two PLCs start later is irrelevant in this implementation.

16.25.1.3 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.

```
IF fbCalc.ip = 0 THEN
    hrCalc := fbCalc.GetInterfacePointer();
END_IF
```

```
IF fbCalc.ip <> 0 THEN
    hrCalc := fbCalc.ip.Addition(a,b,nSum);
    hrCalc := fbCalc.ip.Subtraction(a,b,nDiff);
END_IF
```

In the online view of the PLC application “Provider” the generated object ID of the C++ object can be seen in the PLC function block FB_Calculation. The project node “TcCOM Objects” keeps the generated object with its object ID and the selected name in its list.
In the online view of the PLC application "Caller" the Proxy function block has been allocated the same object ID via the process image. The interface pointer has a valid value and the methods are executed.
16.25.2 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**
2. **Creating an FB in the PLC, which as a simple wrapper offers the functionality of the C++ object**
3. **Execution of the sample project**

Downloading the sample: https://infosys.beckhoff.com/content/1033/TC3_C/Resources/zip/2343048971.zip

**System requirements**

<table>
<thead>
<tr>
<th>TwinCAT version</th>
<th>Hardware</th>
<th>Libraries to be Integrated</th>
</tr>
</thead>
<tbody>
<tr>
<td>TwinCAT 3.1, Build 4020</td>
<td>x86, x64</td>
<td>Tc3_Module</td>
</tr>
</tbody>
</table>

**16.25.2.1 Instantiating a TwinCAT++ class as a TwinCAT TcCOM Object**

The TwinCAT C++ driver must be available on the target 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) are 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 target 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**.

**Creation of the C++ driver**

In the documentation for TwinCAT C++ there is a detailed explanation on how C++ drivers for TwinCAT are created.

To create the above-mentioned driver archive, **Publish TwinCAT Modules** is selected from the C++ project context as the last step in the creation of a driver.
16.25.2.2 Creating an FB in the PLC that, as a simple proxy, offers the functionality of the C++ object

1. Create a PLC and append a new function block there.
This Proxy function block should provide the functionality that was programmed in C++. It is able to do this via an interface pointer that was defined from the C++ class and is known in the PLC due to the TMC description file.

2. In the declaration part of the function block declare as an output an interface pointer to the interface which later provides the functionality outward.

3. Create the object ID and the interface ID as local member variables.
While the interface ID is already available via a global list, the object ID is allocated via the TwinCAT symbol initialization. The TcInitSymbol attribute ensures that the variable appears in a list for external symbol initialization. The object ID of the created C++ object should be allocated.

```
FUNCTION_BLOCK FB_IncrementProxy

VAR_OUTPUT
  ip : IIncrement;
END_VAR

VAR
  {attribute 'TcInitSymbol'}
  {attribute 'displaymode'='hex'}
  nObjId : OCID;  // Instance configured to be retrieved
  iid : IID := TC_GLOBAL_IID_LIST.IID_IIncrement;
  hrInit := HRESULT;
END_VAR
```
4. Here, assign an existing object ID to the symbol name of the variable by drop-down. This value is assigned when the PLC is downloaded so it can be defined prior to the PLC run-time. New symbol initializations or changes are accordingly entered with a new download of the PLC.

As an alternative, the passing of the object ID could also be implemented by means of process image linking as implemented in the first sample (TcCOM_Sample01_PlcToPlc [325]).

5. Implement the PLC Proxy function block.
   First the FB_init constructor method is added to the function block. For the case that it is no longer an OnlineChange but rather the initialization of the function block, the interface pointer to the specified
interface of the specified TcCOM object is obtained with the help of the function FW_ObjMgr_GetObjectInstance(). In this connection the object itself increments a reference counter.

6. It is imperative to release the used reference again. To this end call the FW_SafeRelease() function in the FB_exit destructor of the function block.

7. Declare an instance of the Proxy function block to call the methods provided via the interface in the application.
   The calls themselves take all place over the interface pointer defined as output of the function block. As is typical for pointers a prior null check must be made. Then the methods can be called directly, also via Intellisense.

6. This completes the implementation of the Proxy function block.

7. The sample is ready for testing.

16.25.2.3 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.

16.25.3 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 [► 340]
2. Creating an FB in the PLC that creates the C++ object and offers its functionality [► 341]
3. Execution of the sample project [► 343]

Downloading the sample: https://infosys.beckhoff.com/content/1033/TC3_C/Resources/zip/2343051531.zip

System requirements

<table>
<thead>
<tr>
<th>TwinCAT version</th>
<th>Hardware</th>
<th>Libraries to be integrated</th>
</tr>
</thead>
<tbody>
<tr>
<td>TwinCAT 3.1, Build 4020</td>
<td>x86, x64</td>
<td>Tc3_Module</td>
</tr>
</tbody>
</table>
16.25.3.1 Provision of a TwinCAT C++ driver and its classes

The TwinCAT C++ driver must be available on the target 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) are available as a driver archive. This archive (IncrementerCpp.zip) is unpacked in the following folder:

C:\TwinCAT\3.1\CustomConfig\Modules\IncrementerCpp\n
The TwinCAT Deployment copies the file(s) later in the following folder upon the activation of a configuration in the target system:

C:\TwinCAT\3.1\Driver\AutoInstall\n
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 target system when TwinCAT starts up. In addition this selection provides for the described deployment.

![Solution Explorer](image)

Creation of the C++ driver

In the documentation for TwinCAT C++ [11] 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 TcCOM initialization data and without TcCOM parameters.
16.25.3.2 Creating an FB in the PLC that creates the C++ object and offers its functionality

1. Create a PLC and append a new function block there.
   This Proxy function block should provide the functionality that was programmed in C++. It manages this via an interface pointer that was defined by C++ and is known in the PLC due to the TMC description file.

2. In the declaration part of the function block declare as an output an interface pointer to the interface (IIncrement) which later provides the functionality outward.

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.

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.

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.

6. Declare an instance of the Proxy function block to call the methods provided via the interface in the application. The calls themselves take all place over the interface pointer defined as output of the function block. As is typical for pointers a prior null check must be made. Then the methods can be called directly, also via Intellisense.

The sample is ready for testing.
16.25.3.3 Execution of the sample project

1. Select the target 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.
Appendix

17 Appendix

- The ADS Return Codes [344] are important across TwinCAT 3, particularly if ADS communication [210] itself is implemented.
- The Retain data [349] (in NOVRAM memory) can be used in a similar way from the PLC and also C++.
- In addition to the TcCOM Module [36] 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 [352]
  - Creating and handling TcCOM modules [355]

17.1 ADS Return Codes

Grouping of error codes: 0x000 [344], ..., 0x500 [345], ..., 0x700 [346], ..., 0x1000 [348], ...

Global error codes
<table>
<thead>
<tr>
<th>Hex</th>
<th>Dec</th>
<th>HRESULT</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>0</td>
<td>0x9811 0000</td>
<td>ERR_NOERROR</td>
<td>No error.</td>
</tr>
<tr>
<td>0x1</td>
<td>1</td>
<td>0x9811 0001</td>
<td>ERR_INTERNAL</td>
<td>Internal error.</td>
</tr>
<tr>
<td>0x2</td>
<td>2</td>
<td>0x9811 0002</td>
<td>ERR_NORTIME</td>
<td>No real-time.</td>
</tr>
<tr>
<td>0x3</td>
<td>3</td>
<td>0x9811 0003</td>
<td>ERR_ALLOCLOCKEDMEM</td>
<td>Allocation locked – memory error.</td>
</tr>
<tr>
<td>0x4</td>
<td>4</td>
<td>0x9811 0004</td>
<td>ERR_INSERTMAILBOX</td>
<td>Mailbox full – the ADS message could not be sent. Reducing the number of</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ADS messages per cycle will help.</td>
</tr>
<tr>
<td>0x5</td>
<td>5</td>
<td>0x9811 0005</td>
<td>ERR_WRONGRECEIVEHMSG</td>
<td>Wrong HMSG.</td>
</tr>
<tr>
<td>0x6</td>
<td>6</td>
<td>0x9811 0006</td>
<td>ERR_TARGETPORTNOTFOUND</td>
<td>Target port not found – ADS server is not started or is not reachable.</td>
</tr>
<tr>
<td>0x7</td>
<td>7</td>
<td>0x9811 0007</td>
<td>ERR_TARGETMACHINENOTFOUND</td>
<td>Target computer not found – AMS route was not found.</td>
</tr>
<tr>
<td>0x8</td>
<td>8</td>
<td>0x9811 0008</td>
<td>ERR_UNKNOWNCMDID</td>
<td>Unknown command ID.</td>
</tr>
<tr>
<td>0x9</td>
<td>9</td>
<td>0x9811 0009</td>
<td>ERR_BADTASKID</td>
<td>Invalid task ID.</td>
</tr>
<tr>
<td>0xA</td>
<td>10</td>
<td>0x9811 000A</td>
<td>ERR_NOIO</td>
<td>No IO.</td>
</tr>
<tr>
<td>0xB</td>
<td>11</td>
<td>0x9811 000B</td>
<td>ERR_UNKNOWNAMSCMD</td>
<td>Unknown AMS command.</td>
</tr>
<tr>
<td>0xC</td>
<td>12</td>
<td>0x9811 000C</td>
<td>ERR_WIN32ERROR</td>
<td>Win32 error.</td>
</tr>
<tr>
<td>0xD</td>
<td>13</td>
<td>0x9811 000D</td>
<td>ERR_PORTNOTCONNECTED</td>
<td>Port not connected.</td>
</tr>
<tr>
<td>0xE</td>
<td>14</td>
<td>0x9811 000E</td>
<td>ERR_INVALIDAMSLENGTH</td>
<td>Invalid AMS length.</td>
</tr>
<tr>
<td>0xF</td>
<td>15</td>
<td>0x9811 000F</td>
<td>ERR_INVALIDAMSNETID</td>
<td>Invalid AMS Net ID.</td>
</tr>
<tr>
<td>0x10</td>
<td>16</td>
<td>0x9811 0010</td>
<td>ERR_LOWINSTLEVEL</td>
<td>Low installation level – TwinCAT 2 license error.</td>
</tr>
<tr>
<td>0x11</td>
<td>17</td>
<td>0x9811 0011</td>
<td>ERR_NODEBUGINTAVAILABLE</td>
<td>No debugging available.</td>
</tr>
<tr>
<td>0x12</td>
<td>18</td>
<td>0x9811 0012</td>
<td>ERR_PORTDISABLED</td>
<td>Port disabled – TwinCAT system service not started.</td>
</tr>
<tr>
<td>0x13</td>
<td>19</td>
<td>0x9811 0013</td>
<td>ERR_PORTALREADYCONNECTED</td>
<td>Port already connected.</td>
</tr>
<tr>
<td>0x14</td>
<td>20</td>
<td>0x9811 0014</td>
<td>ERR_AMSSYNC_W32ERROR</td>
<td>AMS Sync Win32 error.</td>
</tr>
<tr>
<td>0x15</td>
<td>21</td>
<td>0x9811 0015</td>
<td>ERR_AMSSYNC_TIMEOUT</td>
<td>AMS Sync Timeout.</td>
</tr>
<tr>
<td>0x16</td>
<td>22</td>
<td>0x9811 0016</td>
<td>ERR_AMSSYNC_AMSERROR</td>
<td>AMS Sync error.</td>
</tr>
<tr>
<td>0x17</td>
<td>23</td>
<td>0x9811 0017</td>
<td>ERR_AMSSYNC_NOINDEXINMAP</td>
<td>No index map for AMS Sync available.</td>
</tr>
<tr>
<td>0x18</td>
<td>24</td>
<td>0x9811 0018</td>
<td>ERR_INVALIDAMSPORT</td>
<td>Invalid AMS port.</td>
</tr>
<tr>
<td>0x19</td>
<td>25</td>
<td>0x9811 0019</td>
<td>ERR_NOMEMORY</td>
<td>No memory.</td>
</tr>
<tr>
<td>0x1A</td>
<td>26</td>
<td>0x9811 001A</td>
<td>ERR_TCPSEND</td>
<td>TCP send error.</td>
</tr>
<tr>
<td>0x1B</td>
<td>27</td>
<td>0x9811 001B</td>
<td>ERR_HOSTUNREACHABLE</td>
<td>Host unreachable.</td>
</tr>
<tr>
<td>0x1C</td>
<td>28</td>
<td>0x9811 001C</td>
<td>ERR_INVALIDAMSFRAGMENT</td>
<td>Invalid AMS fragment.</td>
</tr>
<tr>
<td>0x1D</td>
<td>29</td>
<td>0x9811 001D</td>
<td>ERR_TLSSEND</td>
<td>TLS send error – secure ADS connection failed.</td>
</tr>
<tr>
<td>0x1E</td>
<td>30</td>
<td>0x9811 001E</td>
<td>ERR_ACCESSDENIED</td>
<td>Access denied – secure ADS access denied.</td>
</tr>
</tbody>
</table>

Router error codes
<table>
<thead>
<tr>
<th>Hex</th>
<th>Dec</th>
<th>HRESULT</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x500</td>
<td>1280</td>
<td>0x9811 0500</td>
<td>ROUTERERR_NOLOCKEDMEMORY</td>
<td>Locked memory cannot be allocated.</td>
</tr>
<tr>
<td>0x501</td>
<td>1281</td>
<td>0x9811 0501</td>
<td>ROUTERERR_RESIZEMEMORY</td>
<td>The router memory size could not be changed.</td>
</tr>
<tr>
<td>0x502</td>
<td>1282</td>
<td>0x9811 0502</td>
<td>ROUTERERR_MAILBOXFULL</td>
<td>The mailbox has reached the maximum number of possible messages.</td>
</tr>
<tr>
<td>0x503</td>
<td>1283</td>
<td>0x9811 0503</td>
<td>ROUTERERR_DEBUGBOXFULL</td>
<td>The Debug mailbox has reached the maximum number of possible messages.</td>
</tr>
<tr>
<td>0x504</td>
<td>1284</td>
<td>0x9811 0504</td>
<td>ROUTERERR_UNKNOWNPORTTYPE</td>
<td>The port type is unknown.</td>
</tr>
<tr>
<td>0x505</td>
<td>1285</td>
<td>0x9811 0505</td>
<td>ROUTERERR_NOTINITIALIZED</td>
<td>The router is not initialized.</td>
</tr>
<tr>
<td>0x506</td>
<td>1286</td>
<td>0x9811 0506</td>
<td>ROUTERERR_PORTALREADYINUSE</td>
<td>The port number is already assigned.</td>
</tr>
<tr>
<td>0x507</td>
<td>1287</td>
<td>0x9811 0507</td>
<td>ROUTERERR_NOTREGISTERED</td>
<td>The port is not registered.</td>
</tr>
<tr>
<td>0x508</td>
<td>1288</td>
<td>0x9811 0508</td>
<td>ROUTERERR_NOMOREQUEUES</td>
<td>The maximum number of ports has been reached.</td>
</tr>
<tr>
<td>0x509</td>
<td>1289</td>
<td>0x9811 0509</td>
<td>ROUTERERR_INVALIDPORT</td>
<td>The port is invalid.</td>
</tr>
<tr>
<td>0x50A</td>
<td>1290</td>
<td>0x9811 050A</td>
<td>ROUTERERR_NOTACTIVATED</td>
<td>The router is not active.</td>
</tr>
<tr>
<td>0x50B</td>
<td>1291</td>
<td>0x9811 050B</td>
<td>ROUTERERR_FRAGMENTBOXFULL</td>
<td>The mailbox has reached the maximum number for fragmented messages.</td>
</tr>
<tr>
<td>0x50C</td>
<td>1292</td>
<td>0x9811 050C</td>
<td>ROUTERERR_FRAGMENTTIMEOUT</td>
<td>A fragment timeout has occurred.</td>
</tr>
<tr>
<td>0x50D</td>
<td>1293</td>
<td>0x9811 050D</td>
<td>ROUTERERR_TOBEREMOVED</td>
<td>The port is removed.</td>
</tr>
</tbody>
</table>

**General ADS error codes**
<table>
<thead>
<tr>
<th>Hex</th>
<th>Dec</th>
<th>HRESULT</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x700</td>
<td>1792</td>
<td>0x9811 0700</td>
<td>ADSSERR_DEVICE_ERROR</td>
<td>General device error.</td>
</tr>
<tr>
<td>0x701</td>
<td>1793</td>
<td>0x9811 0701</td>
<td>ADSSERR_DEVICE_SRVN OTSUPP</td>
<td>Service is not supported by the server.</td>
</tr>
<tr>
<td>0x702</td>
<td>1794</td>
<td>0x9811 0702</td>
<td>ADSSERR_DEVICE_INVALIDGRP</td>
<td>Invalid group.</td>
</tr>
<tr>
<td>0x703</td>
<td>1795</td>
<td>0x9811 0703</td>
<td>ADSSERR_DEVICE_INVALIDOFFSET</td>
<td>Invalid offset.</td>
</tr>
<tr>
<td>0x704</td>
<td>1796</td>
<td>0x9811 0704</td>
<td>ADSSERR_DEVICE_INVALIDACCESS</td>
<td>Reading or writing not permitted.</td>
</tr>
<tr>
<td>0x705</td>
<td>1797</td>
<td>0x9811 0705</td>
<td>ADSSERR_DEVICE_INVALIDSIZE</td>
<td>Parameter size not correct.</td>
</tr>
<tr>
<td>0x706</td>
<td>1798</td>
<td>0x9811 0706</td>
<td>ADSSERR_DEVICE_INVALIDDATA</td>
<td>Invalid parameter values.</td>
</tr>
<tr>
<td>0x707</td>
<td>1799</td>
<td>0x9811 0707</td>
<td>ADSSERR_DEVICE_NOTREADY</td>
<td>Device is not ready to operate.</td>
</tr>
<tr>
<td>0x708</td>
<td>1800</td>
<td>0x9811 0708</td>
<td>ADSSERR_DEVICE_BUSY</td>
<td>Device is busy.</td>
</tr>
<tr>
<td>0x709</td>
<td>1801</td>
<td>0x9811 0709</td>
<td>ADSSERR_DEVICE_INVALIDCONTEXT</td>
<td>Invalid operating system context. This can result from use of ADS function blocks in different tasks. It may be possible to resolve this through multitasking synchronization in the PLC.</td>
</tr>
<tr>
<td>0x70A</td>
<td>1802</td>
<td>0x9811 070A</td>
<td>ADSSERR_DEVICE_NOMEMORY</td>
<td>Insufficient memory.</td>
</tr>
<tr>
<td>0x70B</td>
<td>1803</td>
<td>0x9811 070B</td>
<td>ADSSERR_DEVICE_INVALIDP ARM</td>
<td>Invalid parameter values.</td>
</tr>
<tr>
<td>0x70C</td>
<td>1804</td>
<td>0x9811 070C</td>
<td>ADSSERR_DEVICE_NOTFOUND</td>
<td>Not found (files, ...).</td>
</tr>
<tr>
<td>0x70D</td>
<td>1805</td>
<td>0x9811 070D</td>
<td>ADSSERR_DEVICE_SYNTAX</td>
<td>Syntax error in file or command.</td>
</tr>
<tr>
<td>0x70E</td>
<td>1806</td>
<td>0x9811 070E</td>
<td>ADSSERR_DEVICE_INCOMPATIBLE</td>
<td>Objects do not match.</td>
</tr>
<tr>
<td>0x70F</td>
<td>1807</td>
<td>0x9811 070F</td>
<td>ADSSERR_DEVICE_EXISTS</td>
<td>Object already exists.</td>
</tr>
<tr>
<td>0x710</td>
<td>1808</td>
<td>0x9811 0710</td>
<td>ADSSERR_DEVICE_SYMBOLNOTFOUND</td>
<td>Symbol not found.</td>
</tr>
<tr>
<td>0x711</td>
<td>1809</td>
<td>0x9811 0711</td>
<td>ADSSERR_DEVICE_SYMBOLVERSIONINVALID</td>
<td>Invalid symbol version. This can occur due to an online change. Create a new handle.</td>
</tr>
<tr>
<td>0x712</td>
<td>1810</td>
<td>0x9811 0712</td>
<td>ADSSERR_DEVICE_INV ALIDSTATE</td>
<td>Device in invalid state.</td>
</tr>
<tr>
<td>0x713</td>
<td>1811</td>
<td>0x9811 0713</td>
<td>ADSSERR_DEVICE_TRANS MODENOTSUPP</td>
<td>AdsTransMode not supported.</td>
</tr>
<tr>
<td>0x714</td>
<td>1812</td>
<td>0x9811 0714</td>
<td>ADSSERR_DEVICE_NOTIFYHINDINVALID</td>
<td>Notification handle is invalid.</td>
</tr>
<tr>
<td>0x715</td>
<td>1813</td>
<td>0x9811 0715</td>
<td>ADSSERR_DEVICE_CLIENTUNKNOWN</td>
<td>Notification client not registered.</td>
</tr>
<tr>
<td>0x716</td>
<td>1814</td>
<td>0x9811 0716</td>
<td>ADSSERR_DEVICE_NOPMOREHDLS</td>
<td>No further notification handle.</td>
</tr>
<tr>
<td>0x717</td>
<td>1815</td>
<td>0x9811 0717</td>
<td>ADSSERR_DEVICE_INVALIDWATCHSIZE</td>
<td>Notification size too large.</td>
</tr>
<tr>
<td>0x718</td>
<td>1816</td>
<td>0x9811 0718</td>
<td>ADSSERR_DEVICE_NOTINIT</td>
<td>Device not initialized.</td>
</tr>
<tr>
<td>0x719</td>
<td>1817</td>
<td>0x9811 0719</td>
<td>ADSSERR_DEVICE_TIMEOUT</td>
<td>Device has a timeout.</td>
</tr>
<tr>
<td>0x71A</td>
<td>1818</td>
<td>0x9811 071A</td>
<td>ADSSERR_DEVICE_NOINTERFACE</td>
<td>Interface query failed.</td>
</tr>
<tr>
<td>0x71B</td>
<td>1819</td>
<td>0x9811 071B</td>
<td>ADSSERR_DEVICE_INVALIDINTERFACE</td>
<td>Wrong interface requested.</td>
</tr>
<tr>
<td>0x71C</td>
<td>1820</td>
<td>0x9811 071C</td>
<td>ADSSERR_DEVICE_INVALIDCLSID</td>
<td>Class ID is invalid.</td>
</tr>
<tr>
<td>0x71D</td>
<td>1821</td>
<td>0x9811 071D</td>
<td>ADSSERR_DEVICE_INVALIDOBJID</td>
<td>Object ID is invalid.</td>
</tr>
<tr>
<td>0x71E</td>
<td>1822</td>
<td>0x9811 071E</td>
<td>ADSSERR_DEVICE_PENDING</td>
<td>Request pending.</td>
</tr>
<tr>
<td>0x71F</td>
<td>1823</td>
<td>0x9811 071F</td>
<td>ADSSERR_DEVICE_ABORTED</td>
<td>Request is aborted.</td>
</tr>
<tr>
<td>0x720</td>
<td>1824</td>
<td>0x9811 0720</td>
<td>ADSSERR_DEVICE_WARNING</td>
<td>Signal warning.</td>
</tr>
<tr>
<td>0x721</td>
<td>1825</td>
<td>0x9811 0721</td>
<td>ADSSERR_DEVICE_INVALIDARRAYIDX</td>
<td>Invalid array index.</td>
</tr>
<tr>
<td>0x722</td>
<td>1826</td>
<td>0x9811 0722</td>
<td>ADSSERR_DEVICE_SYMBOLNOTACTIVE</td>
<td>Symbol not active.</td>
</tr>
<tr>
<td>0x723</td>
<td>1827</td>
<td>0x9811 0723</td>
<td>ADSSERR_DEVICE_ACCESSDENIED</td>
<td>Access denied.</td>
</tr>
<tr>
<td>0x724</td>
<td>1828</td>
<td>0x9811 0724</td>
<td>ADSSERR_DEVICE_LICENSEN OTFOUN D</td>
<td>Missing license.</td>
</tr>
<tr>
<td>0x725</td>
<td>1829</td>
<td>0x9811 0725</td>
<td>ADSSERR_DEVICE_LICENSEEXPIRED</td>
<td>License expired.</td>
</tr>
<tr>
<td>0x726</td>
<td>1830</td>
<td>0x9811 0726</td>
<td>ADSSERR_DEVICE_LICENSEEXCEDE D</td>
<td>License exceeded.</td>
</tr>
<tr>
<td>0x727</td>
<td>1831</td>
<td>0x9811 0727</td>
<td>ADSSERR_DEVICE_LICENSEINVALID</td>
<td>Invalid license.</td>
</tr>
<tr>
<td>0x728</td>
<td>1832</td>
<td>0x9811 0728</td>
<td>ADSSERR_DEVICE_LICENSESYSTEMID</td>
<td>System ID license invalid.</td>
</tr>
<tr>
<td>0x729</td>
<td>1833</td>
<td>0x9811 0729</td>
<td>ADSSERR_DEVICE_LICENSEN OTCLEAR</td>
<td>License not cleared.</td>
</tr>
<tr>
<td>0x72A</td>
<td>1834</td>
<td>0x9811 072A</td>
<td>ADSSERR_DEVICE_LICENSEFUTUREISSUE</td>
<td>License problem: Time in the future.</td>
</tr>
<tr>
<td>0x72B</td>
<td>1835</td>
<td>0x9811 072B</td>
<td>ADSSERR_DEVICE_LICENSEEXPIRED</td>
<td>License expired.</td>
</tr>
<tr>
<td>0x72C</td>
<td>1836</td>
<td>0x9811 072C</td>
<td>ADSSERR_DEVICE_EXCEPTION</td>
<td>Exception at system startup.</td>
</tr>
<tr>
<td>0x72D</td>
<td>1837</td>
<td>0x9811 072D</td>
<td>ADSSERR_DEVICE_LICENSEDEPLETED</td>
<td>License file read twice.</td>
</tr>
<tr>
<td>0x72E</td>
<td>1838</td>
<td>0x9811 072E</td>
<td>ADSSERR_DEVICE_SIGNATUREINVALID</td>
<td>Invalid signature.</td>
</tr>
<tr>
<td>0x72F</td>
<td>1839</td>
<td>0x9811 072F</td>
<td>ADSSERR_DEVICE_LICENSEINVALID</td>
<td>Invalid certificate.</td>
</tr>
<tr>
<td>0x730</td>
<td>1840</td>
<td>0x9811 0730</td>
<td>ADSSERR_DEVICE_LICENSEE MQN</td>
<td>Public key not known from OEM.</td>
</tr>
<tr>
<td>0x731</td>
<td>1841</td>
<td>0x9811 0731</td>
<td>ADSSERR_DEVICE_LICENSERESTRICTED</td>
<td>License not valid for this system ID.</td>
</tr>
<tr>
<td>0x732</td>
<td>1842</td>
<td>0x9811 0732</td>
<td>ADSSERR_DEVICE_LICENSEDEEMENDED</td>
<td>Demo license prohibited.</td>
</tr>
<tr>
<td>0x733</td>
<td>1843</td>
<td>0x9811 0733</td>
<td>ADSSERR_DEVICE_INVALIDFNCID</td>
<td>Invalid function ID.</td>
</tr>
<tr>
<td>0x734</td>
<td>1844</td>
<td>0x9811 0734</td>
<td>ADSSERR_DEVICE_OUTOF RANGE</td>
<td>Outside the valid range.</td>
</tr>
<tr>
<td>0x735</td>
<td>1845</td>
<td>0x9811 0735</td>
<td>ADSSERR_DEVICE_INVALIDALIGNMENT</td>
<td>Invalid alignment.</td>
</tr>
</tbody>
</table>
## TC3 C++

<table>
<thead>
<tr>
<th>Hex</th>
<th>Dec</th>
<th>HRESULT</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x736</td>
<td>1846</td>
<td>0x9811 0736</td>
<td>ADSERR_DEVICE_LICENSEPLATFORM</td>
<td>Invalid platform level.</td>
</tr>
<tr>
<td>0x737</td>
<td>1847</td>
<td>0x9811 0737</td>
<td>ADSERR_DEVICE_FORWARD_PL</td>
<td>Context – forward to passive level.</td>
</tr>
<tr>
<td>0x738</td>
<td>1848</td>
<td>0x9811 0738</td>
<td>ADSERR_DEVICE_FORWARD_DL</td>
<td>Context – forward to dispatch level.</td>
</tr>
<tr>
<td>0x739</td>
<td>1849</td>
<td>0x9811 0739</td>
<td>ADSERR_DEVICE_FORWARD_RT</td>
<td>Context – forward to real-time.</td>
</tr>
<tr>
<td>0x740</td>
<td>1856</td>
<td>0x9811 0740</td>
<td>ADSERR_CLIENT_ERROR</td>
<td>Client error.</td>
</tr>
<tr>
<td>0x741</td>
<td>1857</td>
<td>0x9811 0741</td>
<td>ADSERR_CLIENT_INVALIDPARN</td>
<td>Service contains an invalid parameter.</td>
</tr>
<tr>
<td>0x742</td>
<td>1858</td>
<td>0x9811 0742</td>
<td>ADSERR_CLIENT_LISTEMPTY</td>
<td>Polling list is empty.</td>
</tr>
<tr>
<td>0x743</td>
<td>1859</td>
<td>0x9811 0743</td>
<td>ADSERR_CLIENT_VARUSED</td>
<td>Var connection already in use.</td>
</tr>
<tr>
<td>0x744</td>
<td>1860</td>
<td>0x9811 0744</td>
<td>ADSERR_CLIENT_DUPLINVOKEID</td>
<td>The called ID is already in use.</td>
</tr>
<tr>
<td>0x745</td>
<td>1861</td>
<td>0x9811 0745</td>
<td>ADSERR_CLIENT_SYNCTIMEOUT</td>
<td>Timeout has occurred – the remote terminal is not responding in the specified ADS timeout. The route setting of the remote terminal may be configured incorrectly.</td>
</tr>
<tr>
<td>0x746</td>
<td>1862</td>
<td>0x9811 0746</td>
<td>ADSERR_CLIENT_W32ERROR</td>
<td>Error in Win32 subsystem.</td>
</tr>
<tr>
<td>0x747</td>
<td>1863</td>
<td>0x9811 0747</td>
<td>ADSERR_CLIENT_TIMEOUTINVALID</td>
<td>Invalid client timeout value.</td>
</tr>
<tr>
<td>0x748</td>
<td>1864</td>
<td>0x9811 0748</td>
<td>ADSERR_CLIENT_PORTNOTOPEN</td>
<td>Port not open.</td>
</tr>
<tr>
<td>0x749</td>
<td>1865</td>
<td>0x9811 0749</td>
<td>ADSERR_CLIENT_NOAMSADDR</td>
<td>No AMS address.</td>
</tr>
<tr>
<td>0x750</td>
<td>1872</td>
<td>0x9811 0750</td>
<td>ADSERR_CLIENT_SYNCHINTERAL</td>
<td>Internal error in Ads sync.</td>
</tr>
<tr>
<td>0x751</td>
<td>1873</td>
<td>0x9811 0751</td>
<td>ADSERR_CLIENT_ADDHASH</td>
<td>Hash table overflow.</td>
</tr>
<tr>
<td>0x752</td>
<td>1874</td>
<td>0x9811 0752</td>
<td>ADSERR_CLIENT_REMOVEHASH</td>
<td>Key not found in the table.</td>
</tr>
<tr>
<td>0x753</td>
<td>1875</td>
<td>0x9811 0753</td>
<td>ADSERR_CLIENT_NOMORESYM</td>
<td>No symbols in the cache.</td>
</tr>
<tr>
<td>0x754</td>
<td>1876</td>
<td>0x9811 0754</td>
<td>ADSERR_CLIENT_SYNCREINVALID</td>
<td>Invalid response received.</td>
</tr>
<tr>
<td>0x755</td>
<td>1877</td>
<td>0x9811 0755</td>
<td>ADSERR_CLIENT_SYNCPORTLOCKED</td>
<td>Sync Port is locked.</td>
</tr>
</tbody>
</table>

## RTime error codes

<table>
<thead>
<tr>
<th>Hex</th>
<th>Dec</th>
<th>HRESULT</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1000</td>
<td>4096</td>
<td>0x9811 1000</td>
<td>RTERR_INTERNAL</td>
<td>Internal error in the real-time system.</td>
</tr>
<tr>
<td>0x1001</td>
<td>4097</td>
<td>0x9811 1001</td>
<td>RTERR_BADTIMERPERSIODS</td>
<td>Timer value is not valid.</td>
</tr>
<tr>
<td>0x1002</td>
<td>4098</td>
<td>0x9811 1002</td>
<td>RTERR_INVALIDTASKPTR</td>
<td>Task pointer has the invalid value 0 (zero).</td>
</tr>
<tr>
<td>0x1003</td>
<td>4099</td>
<td>0x9811 1003</td>
<td>RTERR_INVALIDSTACKPTR</td>
<td>Task stack pointer has the invalid value 0.</td>
</tr>
<tr>
<td>0x1004</td>
<td>4100</td>
<td>0x9811 1004</td>
<td>RTERR_PRIOEXISTS</td>
<td>The request task priority is already assigned.</td>
</tr>
<tr>
<td>0x1005</td>
<td>4101</td>
<td>0x9811 1005</td>
<td>RTERR_NOMORETCB</td>
<td>No free TCB (Task Control Block) available. The maximum number of TCBs is 64.</td>
</tr>
<tr>
<td>0x1006</td>
<td>4102</td>
<td>0x9811 1006</td>
<td>RTERR_NOMORESEMAS</td>
<td>No free semaphores available. The maximum number of semaphores is 64.</td>
</tr>
<tr>
<td>0x1007</td>
<td>4103</td>
<td>0x9811 1007</td>
<td>RTERR_NOMOREQUEUES</td>
<td>No free space available in the queue. The maximum number of positions in the queue is 64.</td>
</tr>
<tr>
<td>0x100D</td>
<td>4109</td>
<td>0x9811 100D</td>
<td>RTERR_EXTRIQALREADYDEF</td>
<td>An external synchronization interrupt is already applied.</td>
</tr>
<tr>
<td>0x100E</td>
<td>4110</td>
<td>0x9811 100E</td>
<td>RTERR_EXTRIONOTDEF</td>
<td>No external sync interrupt applied.</td>
</tr>
<tr>
<td>0x100F</td>
<td>4111</td>
<td>0x9811 100F</td>
<td>RTERR_EXTRIQINSTALLFAILED</td>
<td>Application of the external synchronization interrupt failed</td>
</tr>
<tr>
<td>0x1010</td>
<td>4112</td>
<td>0x9811 1010</td>
<td>RTERR__IQLNOTLESSOREQUAL</td>
<td>Call of a service function in the wrong context</td>
</tr>
<tr>
<td>0x1017</td>
<td>4119</td>
<td>0x9811 1017</td>
<td>RTERR_VMXNOTSUPPORTED</td>
<td>Intel VT-x extension is not supported.</td>
</tr>
<tr>
<td>0x1018</td>
<td>4120</td>
<td>0x9811 1018</td>
<td>RTERR_VMXDISABLED</td>
<td>Intel VT-x extension is not enabled in the BIOS.</td>
</tr>
<tr>
<td>0x1019</td>
<td>4121</td>
<td>0x9811 1019</td>
<td>RTERR_VMXCONTROLMESSAGING</td>
<td>Missing function in Intel VT-x extension.</td>
</tr>
<tr>
<td>0x101A</td>
<td>4122</td>
<td>0x9811 101A</td>
<td>RTERR_VMXENABLEFAILS</td>
<td>Activation of Intel VT-x fails.</td>
</tr>
</tbody>
</table>

## TCP Winsock error codes
<table>
<thead>
<tr>
<th>Hex</th>
<th>Dec</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x274C</td>
<td>10060</td>
<td>WSAETIMEDOUT</td>
<td>A connection timeout has occurred. Connection error because the remote</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>terminal did not respond properly after a certain period of time, or the</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>connected host did not respond and the connection could therefore not be</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>maintained.</td>
</tr>
<tr>
<td>0x274D</td>
<td>10061</td>
<td>WSAECONNREFUSED</td>
<td>Connection refused. No connection could be established because the target</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>computer explicitly rejected it. This error usually results from an attempt</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>to connect to a service that is inactive on the external host, that is, a</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>service for which no server application is running.</td>
</tr>
<tr>
<td>0x2751</td>
<td>10065</td>
<td>WSAEHOSTUNREACH</td>
<td>No route to host. A socket operation referred to an unavailable host.</td>
</tr>
</tbody>
</table>

More Winsock error codes: Win32 error codes

17.2 Retain data

This section describes the option to make data available even after an ordered or spontaneous system restart. The NOV-RAM of a device is used for this purpose. The EL6080 cannot be used for these retain data, because the corresponding data must first be transferred, which leads to corresponding runtimes.

The following section describes the retain handler, which stores data and makes them available again, and the application of the different TwinCAT 3 programming languages.

Configuring a retain device

1. The retain data are stored and made available by a retain handler, which is part of the NOV-DP-RAM device in the IO section of the TwinCAT solution. Create a NOV-RAM DP Device in the IO area of the

Solution.

2. Create one or more Retain Handler below this device.

Storage location: NOVRAM
3. Configure the NOV-DP RAM device. In the **Generic NOV-DP-RAM Device** tab, use **Search**... to define the area to be used.

4. An additional retain directory for the symbols is created in the TwinCAT boot directory.

**Using the retain handler with a PLC project**

In a PLC project the variables are either created in a VAR RETAIN section or identified with the attribute TcRetain.

```plaintext
PROGRAM MAIN
VAR RETAIN
  l: UINT;
  k: UINT;
END_VAR
VAR
  {attribute 'TcRetain':='1'}
  m: UINT;
  x: UINT;
END_VAR
```

Corresponding symbols are created after a “Build”.

The assignment to the retain handler of the NOV-DP-RAM device is done in column **Retain Hdl**.

If self-defined data types (DUTs) are used as retain, the data types must be available in the TwinCAT type system. You can either use the option **Convert to Global Type** or you can create structures directly as **STRUCT RETAIN**. However, the Retain Handler then handles all occurrences of the structure. Retain data cannot be used for POUs (function blocks) as a whole. However, individual elements of a POU can be used.
Using the retain handler with a C++ module

In a C++ module a data area of type Retain Source is created, which contains the corresponding symbols.

At the instances of the C++ module, a retain handler of the NOV-DP-RAM device to be used for this data area is defined in column Retain Hdl.

Conclusions

When a retain handler is selected as target in the respective project, the symbols under retain handler and a mapping are created automatically after a "Build".
17.3 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 [104].

The customer could define own templates, which is documented at the corresponding sub-section if C++ Section / Wizards [155].

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#):

```csharp
ITcSmTreeItem cpp = systemManager.LookupTreeItem("TIXC");
ITcSmTreeItem cppProject = cpp.CreateChild("NewCppProject", 0, ",", pathToTemplateFile);
```

Code snippet (Powershell):

```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#):**

```csharp
ITcSmTreeItem cppModule = cppProject.CreateChild("NewModule", 1, ",", pathToTemplateFile);
```

**Code snippet (Powershell):**

```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:

![MessageBox Image]

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 (tczip file).

**Code snippet (C#):**

```csharp
ITcSmTreeItem cpp = systemManager.LookupTreeItem("TIXC");
ITcSmTreeItem newProject = cpp.CreateChild("NameOfProject", 1, ",", pathToProjectOrTczipFile);
```

**Code snippet (Powershell):**

```powershell
$cpp = $systemManager.LookupTreeItem("TIXC")
$newProject = $cpp.CreateChild("NameOfProject", 1, ",", $pathToProjectOrTczipFile)
```

Please note that C++ projects can’t be renamed, thus the original project name needs to be specified. (cmp. Renaming TwinCAT C++ projects [238])

Creating module instances

TcCOM Modules could be created at the System -> TcCOM Modules node. Please see documentation there [355].
Appendix
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++ [} 51].
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#):

354

Version: 1.9

TC3 C++


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.

17.4 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#):**

```csharp
ITcSmTreeItem tcComObjects = systemManager.LookupTreeItem("TIRC^TcCOM Objects");
```

**Code Snippet (Powershell):**

```powershell
$tcComObjects = $systemManager.LookupTreeItem("TIRC^TcCOM Objects")
```

The code above assumes that there is already a systemManager objects present in your AI code.
Adding existing TcCOM modules

To add existing TcCOM modules to your TwinCAT configuration, these modules need to be detectable by TwinCAT. This can be achieved by either of the following ways:

- Copying TcCOM modules to folder `%TWINCAT3.XDIR%\CustomConfig\Modules`
- Editing `%TWINCAT3.XDIR%\Config\Io\TcModuleFolders.xml` to add a path to a folder of your choice and place the modules within that folder

Both ways will be sufficient to make the TcCOM modules detectable by TwinCAT.

A TcCOM module is being identified by its GUID or name:

- This GUID can be used to add a TcCOM module to a TwinCAT configuration via the `ITcSmTrieItem::CreateChild()` method. The GUID can be determined in TwinCAT XAE via the properties page of a TcCOM module.

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#):**
```csharp
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):**
```powershell
$tcModuleTable = @"
$tcModuleTable.Add("TempContr", ":8f5fdcff-ee4b-4ee5-80b1-25eb23bd1b45")
$tempController = $tcComObjects.CreateChild("Test", 0, ",", $tcModuleTable["TempContr"])
```

Please note that the vInfo parameter of the method `ITcSmTreeItem::CreateChild()` contains the GUID of the TcCOM module which is used to identify the module in the list of all registered TcCOM modules in that system.

- This name can be used to add a TcCOM module to a TwinCAT configuration via the `ITcSmTreeItem::CreateChild()` method. The name can be determined in TwinCAT XAE via the TMC Editor.
This can be done by the following way:

**Code Snippet (C#):**

```csharp
ITcSmTreeItem tempController = tcComObjects.CreateChild("Test", 1, ",", "NewModule");
```

**Code Snippet (Powershell):**

```powershell
$tempController = $tcComObjects.CreateChild("Test", 0, ",", "NewModule")
```

### Iterating through added TcCOM modules

To iterate through all added TcCOM module instances, you may use the ITcModuleManager2 interface. The following code snippet demonstrates how to use this interface.

**Code Snippet (C#):**

```csharp
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):**

```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#):**

```csharp
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#):**

```csharp
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())
XmlNode targetDoc = new XmlNode();
XmlNode moduleInstanceElement = targetDoc.CreateElement("TreeItem");
XmlNode moduleElement = targetDoc.CreateElement("Module");
XmlNode parametersElement = (XmlNode) targetDoc.ImportNode(sourceParameters, true);
moduleElement.AppendChild(parametersElement);
moduleInstanceElement.AppendChild(moduleElement);
targetDoc.AppendChild(moduleInstanceElement);

// 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.XmlElement] $treeItemElement = $targetDoc.CreateElement("TreeItem")
[System.XML.XmlElement] $moduleInstanceElement = $targetDoc.CreateElement("TcModuleInstance")
[System.XML.XmlElement] $moduleElement = $targetDoc.CreateElement("Module")
$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();
XmlNode sourceDataAreas = tempControllerDoc.SelectSingleNode("TreeItem/TcModuleInstance/Module/
DataAreas");

// Second step: Build target XML (for later ConsumeXml())
XmlNode targetDoc = new XmlNode();
XmlNode treeItem = targetDoc.CreateElement("TreeItem");
XmlNode moduleInstance = targetDoc.CreateElement("TcModuleInstance");
XmlNode module = targetDoc.CreateElement("Module");
XmlNode dataAreas = (XmlNode) 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["CreateSymbol"];  

// 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
$targetDoc = New-Object System.XML.XmlDocument
$treeItem = $targetDoc.CreateElement("TreeItem")
$moduleInstance = $targetDoc.CreateElement("TcModuleInstance")
$module = $targetDoc.CreateElement("Module")
$dataAreas = $targetDoc.ImportNode($sourceDataAreas, $true)
$module.AppendChild($dataAreas)
$moduleInstance.AppendChild($module)
$treeItem.AppendChild($moduleInstance)
$targetDoc.AppendChild($treeItem)
$destModule = $targetDoc.TreeItem.TcModuleInstance.Module
$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
TaskObjectld. Both are equivalent to the corresponding parameters in TwinCAT XAE:

Please note that the TaskObjectld is shown in hex in TwinCAT XAE.
Code Snippet (C#):

```csharp
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#):**

```csharp
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().