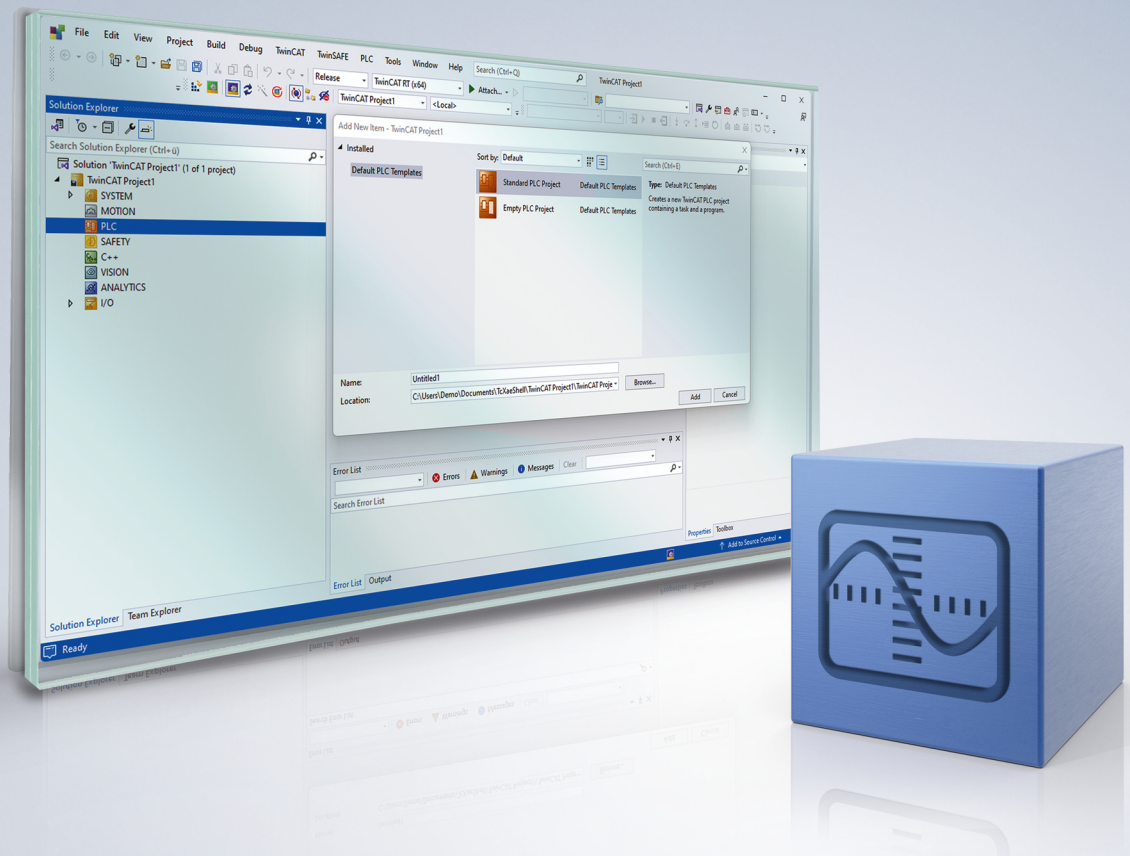
BECKHOFF New Automation Technology

Manual | EN

TF3710

TwinCAT 3 | Interface for LabVIEW™



2026-03-11 | 1.9.2

Table of contents

1 Foreword 5
1.1 Notes on the documentation 5
1.2 For your safety 6
1.3 Notes on information security 7
1.4 Documentation issue status 8
2 Installation 9
3 Licensing 14
4 Quick start 17
5 Technical introduction 21
5.1 TwinCAT ADS 21
5.2 Communication modes 23
5.2.1 One-time reading 26
5.2.2 One-time writing 28
5.2.3 Reading data continuously 29
5.2.4 Writing data continuously 34
5.3 Type Resolving 35
5.4 Heartbeat Connection Monitoring 37
6 LabVIEW™ VIs 40
6.1 ADS DAQ 45
6.2 ADS FlexDAQ 54
6.3 ADS Write Assistant 60
6.4 Symbol Interface 65
6.5 Init 66
6.6 ADS-Read 67
6.7 ADS-Write 72
6.8 TypeResolver 74
6.9 Release 74
6.10 Utilities 75
6.10.1 Notification 78
6.10.2 LVBuffer 79
6.10.3 CoE 81
6.10.4 Heartbeat 84
6.10.5 Project Automation 86
6.10.6 TwinCAT 3 EventLogger Interface 89
6.11 Low-Level 92
6.11.1 Init 92
6.11.2 Read 94
6.11.3 Write 96
6.11.4 TypeResolver 98
6.11.5 SumUp 104
6.12 With TypeResolving 106
6.13 ADS Remote Procedure Call (RPC) 108
7 Samples 111

7.1 Basic examples ... 111
7.2 Application example ... 121

8 Appendix ... 123
8.1 Overview of error codes ... 123
8.2 ADS Return Codes ... 124
8.3 Data types ... 127
8.4 Runtime Return Codes ... 129
8.5 Support Return Codes ... 133

9 Support and Service ... 134

1 Foreword

1.1 Notes on the documentation

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

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

Disclaimer

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

Trademarks

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



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

Copyright

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

Third-party trademarks

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

1.2 For your safety

Safety regulations

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

Exclusion of liability

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

Personnel qualification

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

Signal words

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

Personal injury warnings

⚠
Hazard with high risk of death or serious injury.

⚠
Hazard with medium risk of death or serious injury.

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

Warning of damage to property or environment

The environment, equipment, or data may be damaged.

Information on handling the product

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

1.3 Notes on information security

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

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

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

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

1.4 Documentation issue status

Version	Changes
1.9.x	**New:** • LabVIEW™ VIs > Utilities > TwinCAT 3 Eventlogger Interface [▶ 89] • LabVIEW™ VIs > ADS RPC [▶ 108]
1.8.1	Optimized VI icons
1.8.x	**New:** • VIs that can be used to create a Heartbeat [▶ 84] connection monitoring between LabVIEW™ and TwinCAT. • Project Automation [▶ 86]: VIs that can be used to create, configure, and activate a TwinCAT 3 project.
1.7.x	• Supports LabVIEW™ 2024 • Settings for displaying timestamps in UTC time format [▶ 128] • Further information on the CoE interface has been added, see Utilities > CoE [▶ 81].
1.6.x	• Adjustments in the ADS FlexDAQ [▶ 54] area • **New**: Description of the VIs Add Read/Write symbol [▶ 92]
1.5.x	TwinCAT 3.1 Build 4026 • Brief information on installation [▶ 9]
	New: • CoE Read and CoE Write [▶ 81] ◦ VIs for reading and writing CoE objects from LabVIEW™ • Sample CoE Reading or Writing [▶ 118]

2 Installation

System requirements

The minimum requirements for using the TF3710 TwinCAT 3 Interface for LabVIEW™ product are described below.

Please observe the following distinction: the TF3710 license is required on the TwinCAT runtime environment. The components of the TF3710 TwinCAT 3 Interface for LabVIEW™, on the other hand, are required on the system with the LabVIEW™ installation.

Engineering

In your engineering environment you need a TwinCAT 3.1 ADS installation, in addition to LabVIEW™. There is no need to install TwinCAT 3.1 ADS if you have a TwinCAT 3 Engineering environment or a TwinCAT Runtime installed on your engineering system.

- TwinCAT 3.1 ADS
- LabVIEW™ 2017, 2018, 2019, 2020, 2021, 2022, 2023, 2024, 2025
  - Runtime, Base, Full, Community, Professional Edition
  - 32-bit, 64-bit

Runtime

For the TwinCAT runtime you need the following components:

- TwinCAT 3.1 XAR build 3.1.4024.12 or higher
  If a lower version is required, please contact Beckhoff Support.
- Operating systems: Windows operating system, TwinCAT/BSD, Beckhoff RT Linux® (only from TwinCAT 3 Build 4026)
- One license for TF3710 TwinCAT 3 Interface for LabVIEW™

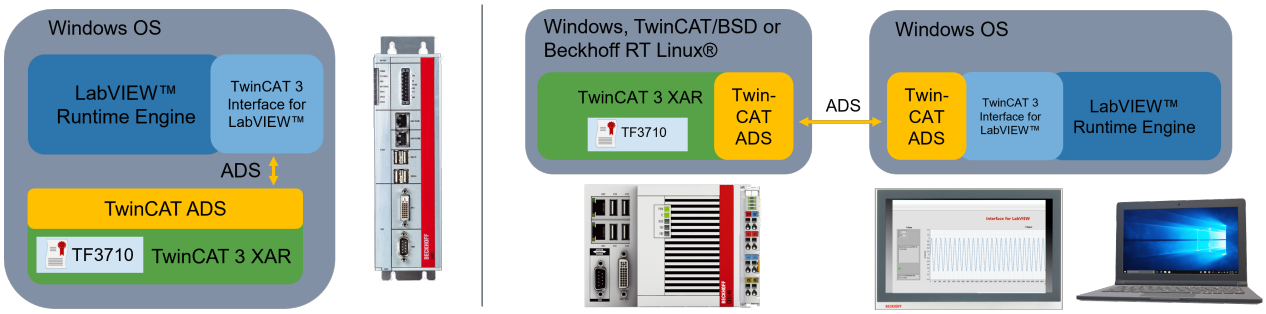


For testing purposes, a 7-day trial version can be activated repeatedly.

For the LabVIEW™ runtime the following components are required:

- LabVIEW™ Runtime Engine 2017, 2018, 2019, 2020, 2021, 2022, 2023, 2024, 2025
- Windows NT-based operating system

It should be noted that the TwinCAT and LabVIEW™ runtimes can be installed on the same system or on separate systems connected via a network.



TwinCAT Package Manager: Installation (TwinCAT 3.1 Build 4026)

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

Install the following workload to be able to use the product:

TwinCAT Package Manager UI: TF3710 | TwinCAT 3 Interface for LabVIEW™

TwinCAT Package Manager CLI:
```
tcpkg install TF3710.InterfaceForLabVIEW.XAE
```

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

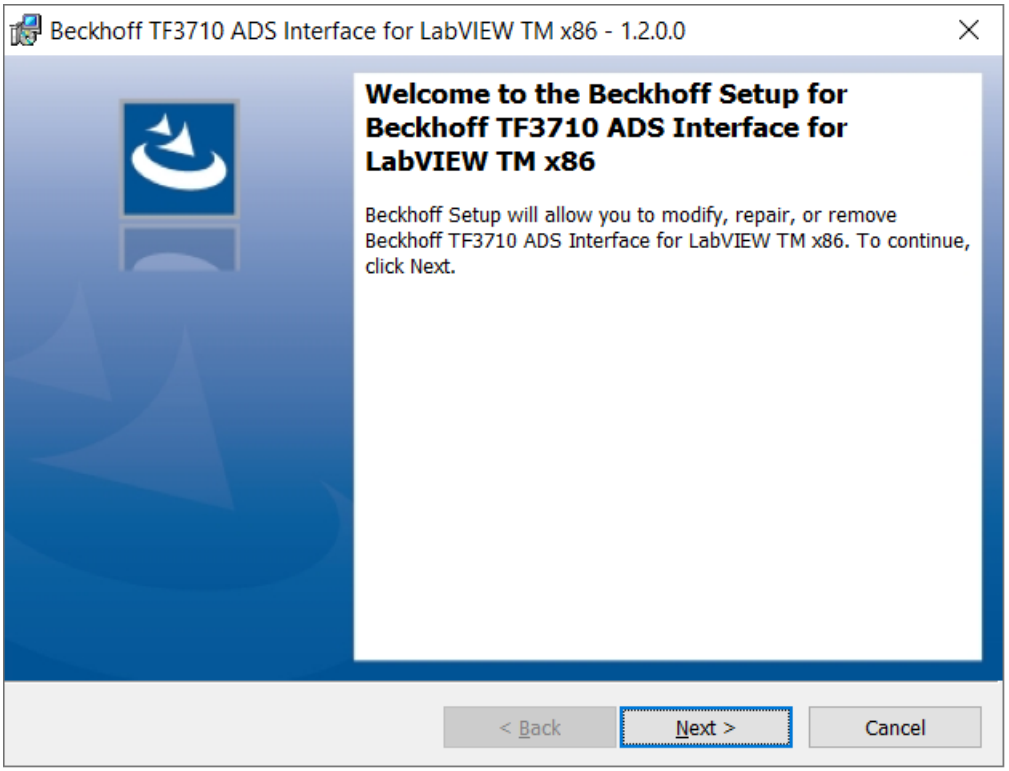
The following section describes the installation of the TwinCAT 3 Function TwinCAT 3 Interface for LabVIEW™ for Windows-based operating systems.

Different setups for 32bit and 64bit LabVIEW™

Two setups can be found on the download page of the Beckhoff website. The setup labeled "x86" integrates the interface for LabVIEW™ into installed 32bit LabVIEW™ environments. The setup labeled "x64" is suitable for 64bit LabVIEW™ environments.

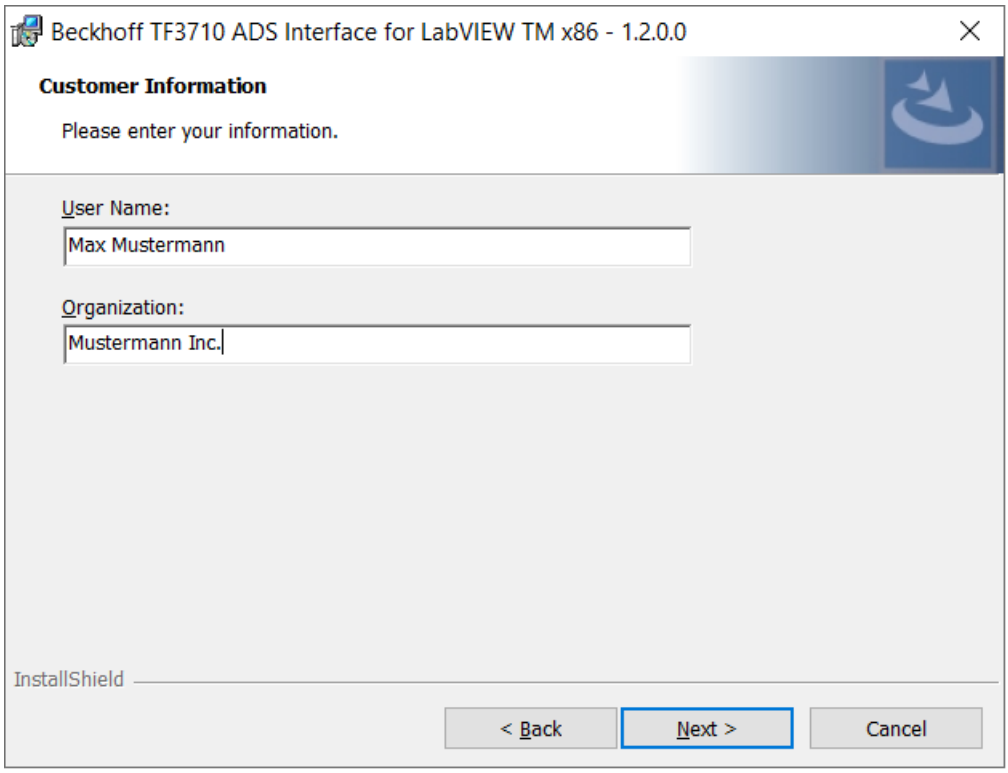
✓ The TwinCAT 3 function setup file was downloaded from the Beckhoff website.

1. Run the setup file as administrator. To do this, select the **Run As Admin** command in the context menu of the file.
   ⇨ The installation dialog opens.

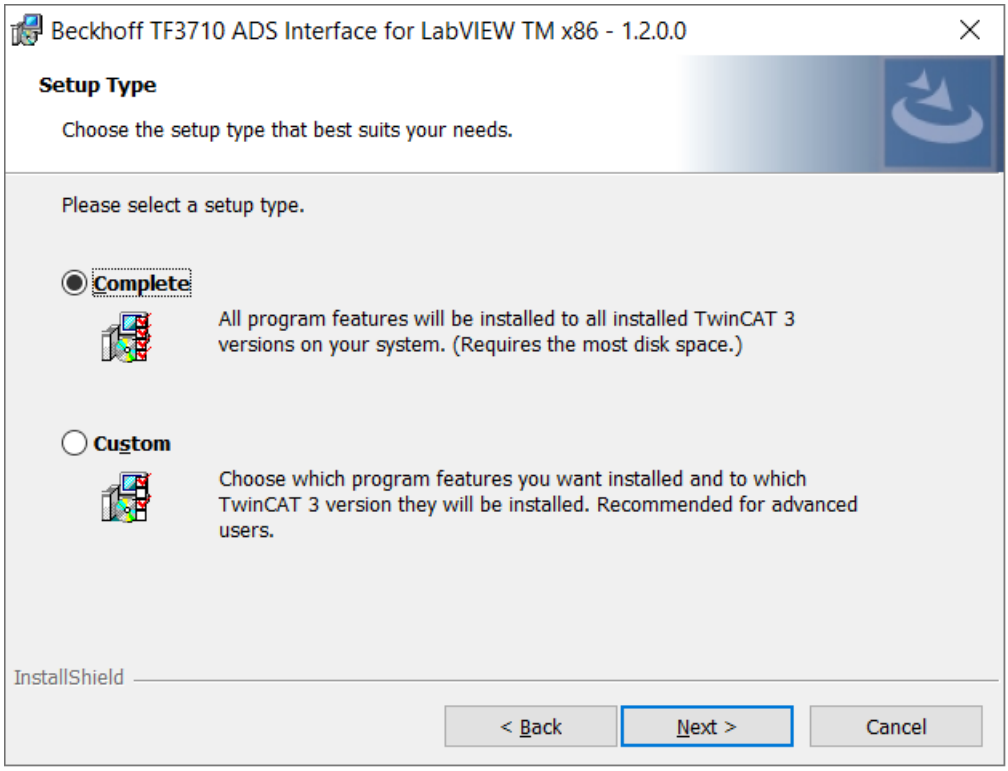


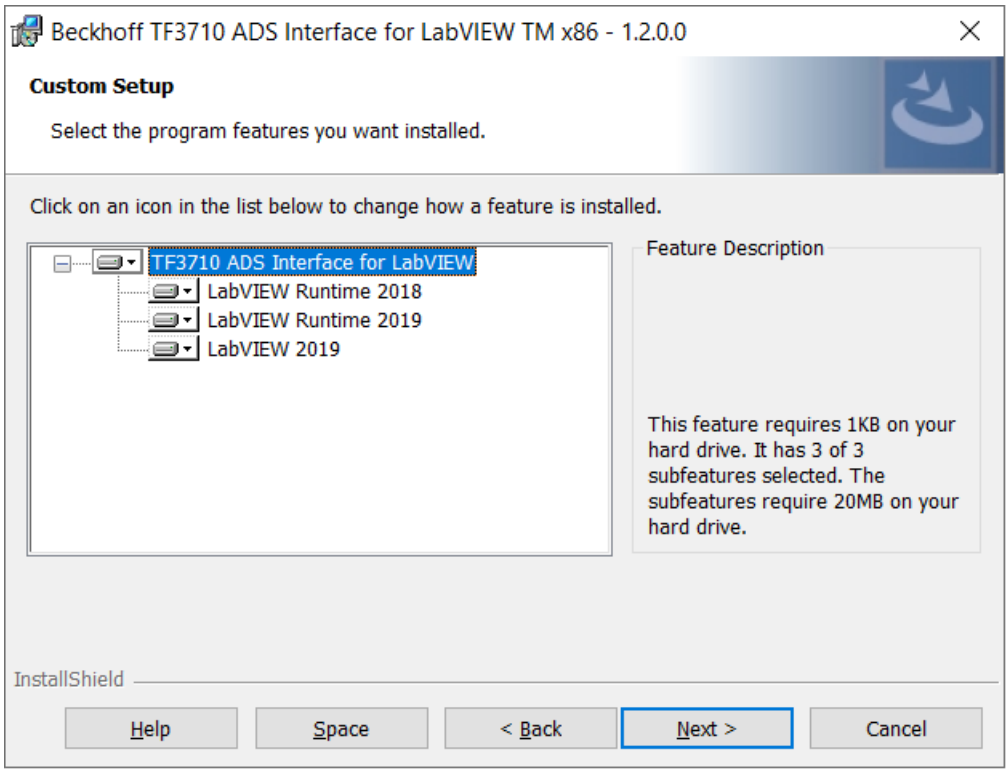
2. Accept the end user licensing agreement and click **Next**.

3. Enter your user data.

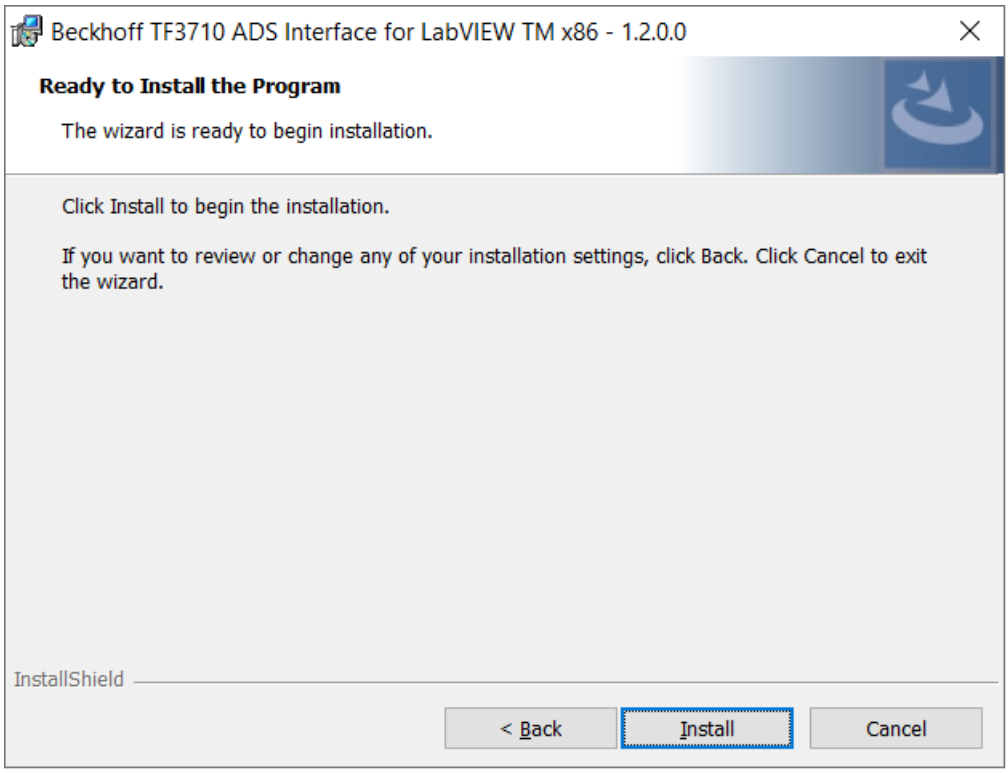


4. If you want to install the full version of the TwinCAT 3 function, select **Complete** as installation type. This installs the product for each LabVIEW™ found on the system. Select **Custom** if you only want to install the product for individual LabVIEW™ environments.



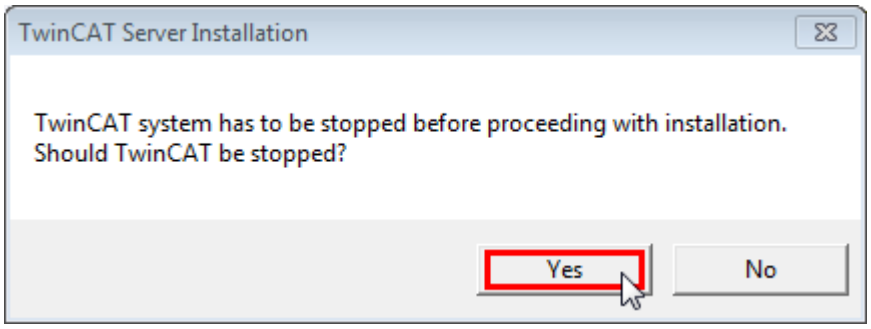


5. Click **Next**, then **Install** to start the installation.



⇨ A dialog box informs you that the TwinCAT system must be stopped to proceed with the installation.

6. Confirm the dialog with **Yes.**



7. Click **Finish** to exit the setup.

⇨ The TwinCAT 3 function has been successfully installed and can be licensed (see Licensing).

Mass compile

During installation, the TwinCAT 3 Interfaces for LabVIEW™ library is added for the mass compilation of VIs. A LabVIEW™ VI is started for this purpose.

3 Licensing

The TwinCAT 3 function can be activated as a full version or as a 7-day test version. Both license types can be activated via the TwinCAT 3 development environment (XAE).

Licensing the full version of a TwinCAT 3 Function

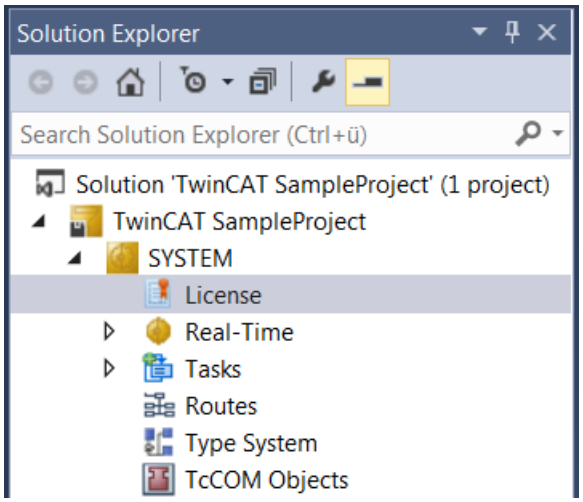
A description of the procedure to license a full version can be found in the Beckhoff Information System in the documentation "TwinCAT 3 Licensing".

Licensing the 7-day test version of a TwinCAT 3 Function

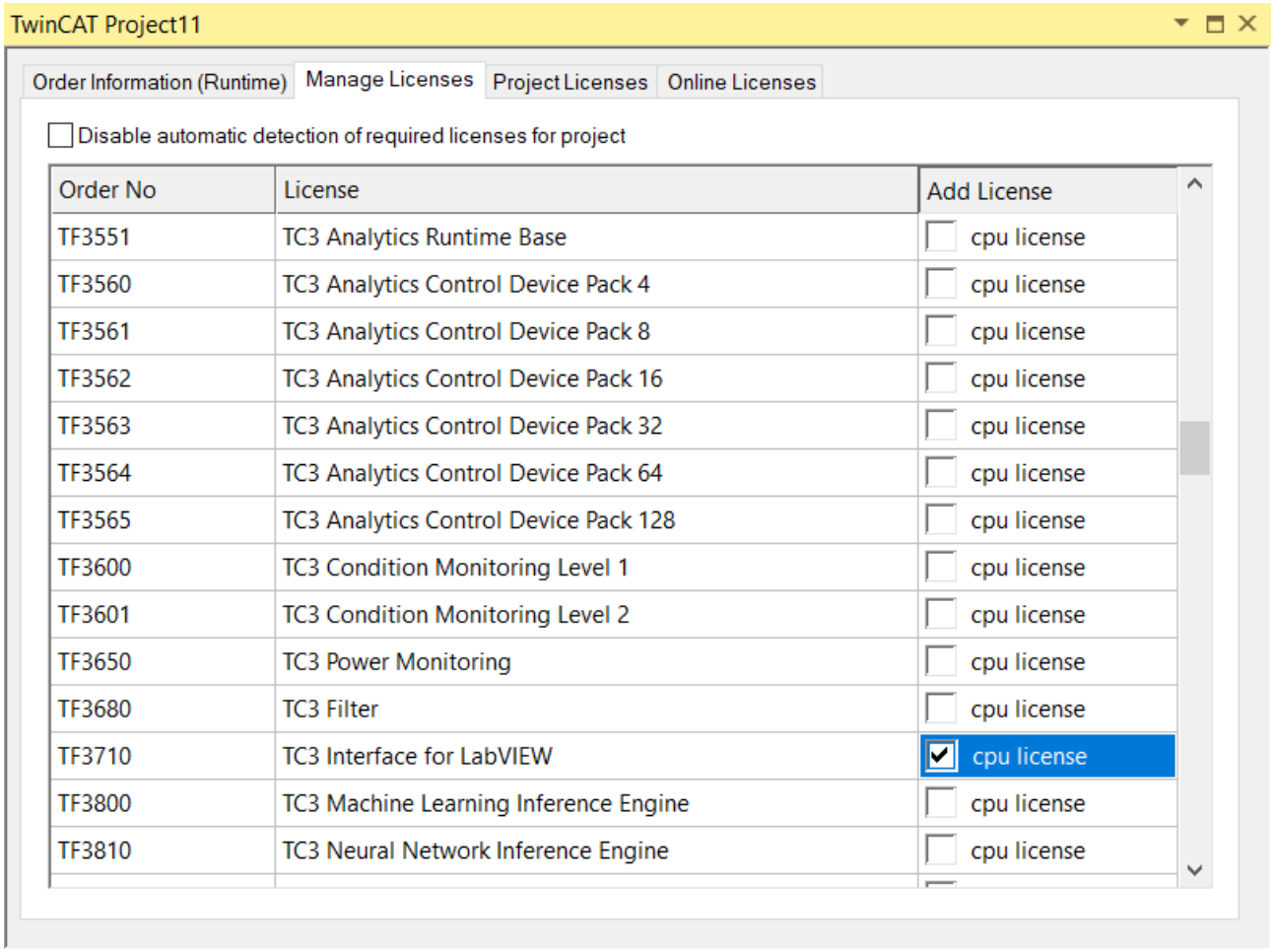


A 7-day test version cannot be enabled for a TwinCAT 3 license dongle.

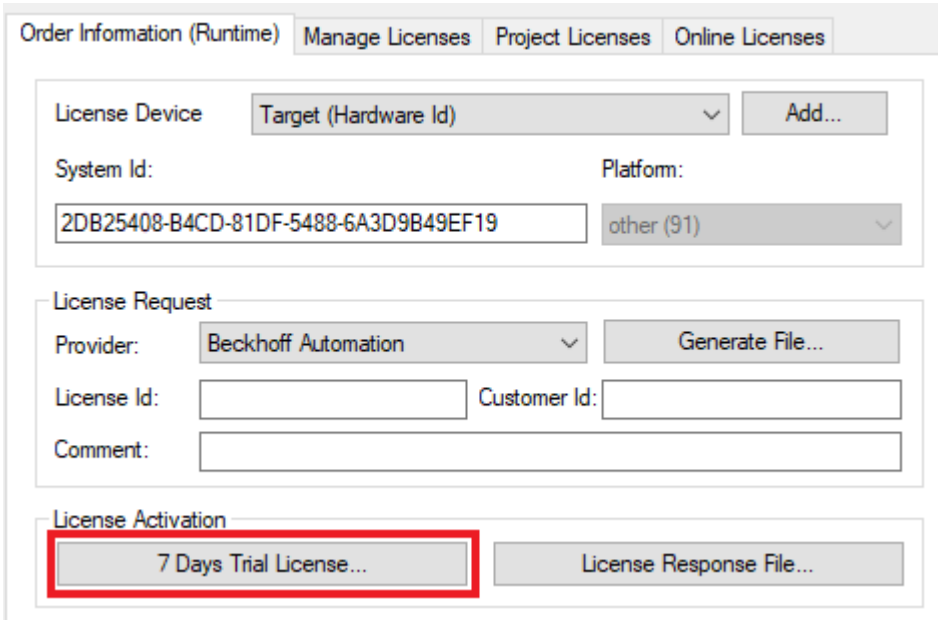
1. Start the TwinCAT 3 development environment (XAE).
2. Open an existing TwinCAT 3 project or create a new project.
3. If you want to activate the license for a remote device, set the desired target system. To do this, select the target system from the **Choose Target System** drop-down list in the toolbar.
   ⇨ The licensing settings always refer to the selected target system. When the project is activated on the target system, the corresponding TwinCAT 3 licenses are automatically copied to this system.
4. In the **Solution Explorer**, double-click **License** in the **SYSTEM subtree.**



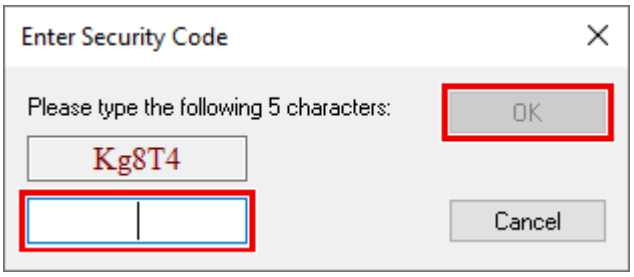
   ⇨ The TwinCAT 3 license manager opens.
5. Open the **Manage Licenses** tab. In the **Add License** column, check the check box for the license you want to add to your project (e.g. "TF3710 TwinCAT 3 Interface for LabVIEW").



6. Open the **Order Information (Runtime)** tab.
   ⇨ In the tabular overview of licenses, the previously selected license is displayed with the status “missing”**.**
7. Click **7-Day Trial License...** to activate the 7-day trial license.



⇨ A dialog box opens, prompting you to enter the security code displayed in the dialog.



8. Enter the code exactly as it is displayed and confirm the entry.
9. Confirm the subsequent dialog, which indicates the successful activation.
   ⇨ In the tabular overview of licenses, the license status now indicates the expiry date of the license.
10. Restart the TwinCAT system.

⇨ The 7-day trial version is enabled.

Quick-Guide: license query fails

Different error messages may occur during the license query in LabVIEW™, e.g.: *License state issue*, *No License found, License expired, License invalid, License invalid System id*.

Check the following:

- If a license query failed previously, restart LabVIEW™ so that the ADS client in Interface for LabVIEW™ is completely unloaded.
- Use the TwinCAT XAE to check the license status on the target system and adjust it if necessary.

4 Quick start

The following describes how to set up the connection between LabVIEW™ and TwinCAT using an exemplary measuring task. Thereby data is generated in TwinCAT, which is read by LabVIEW™.

Activating a TwinCAT project

✓ Download this https://infosys.beckhoff.com/content/1033/TF3710_TC3_Interface_for_LabVIEW/Resources/10083995531.zip. There is a tnzip file in the ZIP archive.

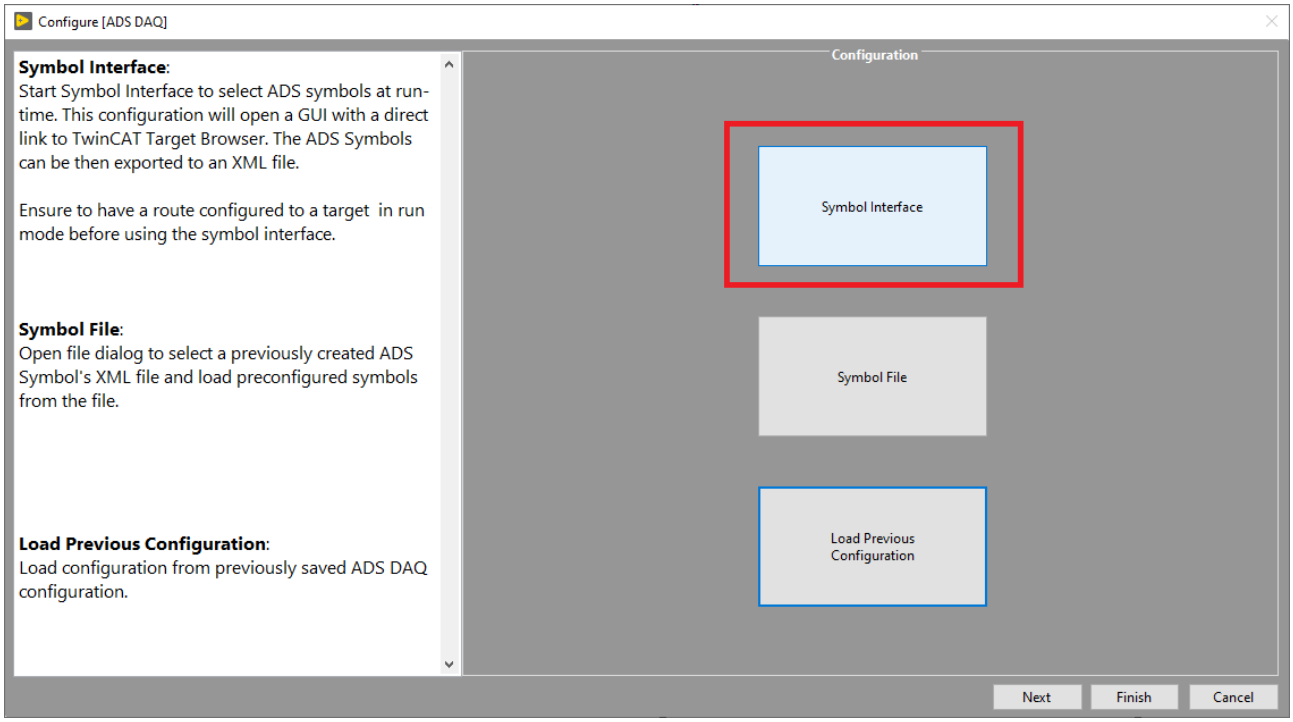
1. Open TwinCAT XAE and select
   File > Open > Open Solution From Archive... to load the tnzip.
2. If you do not already have a valid TF3710 license on the target system, go to **System > License > Manage Licenses** and select the checkbox for the TF3710 license.
3. Activate the project, e.g. on your local PC or on a remote target system and start the PLC.

⇨ The TwinCAT project is now running on your target system.

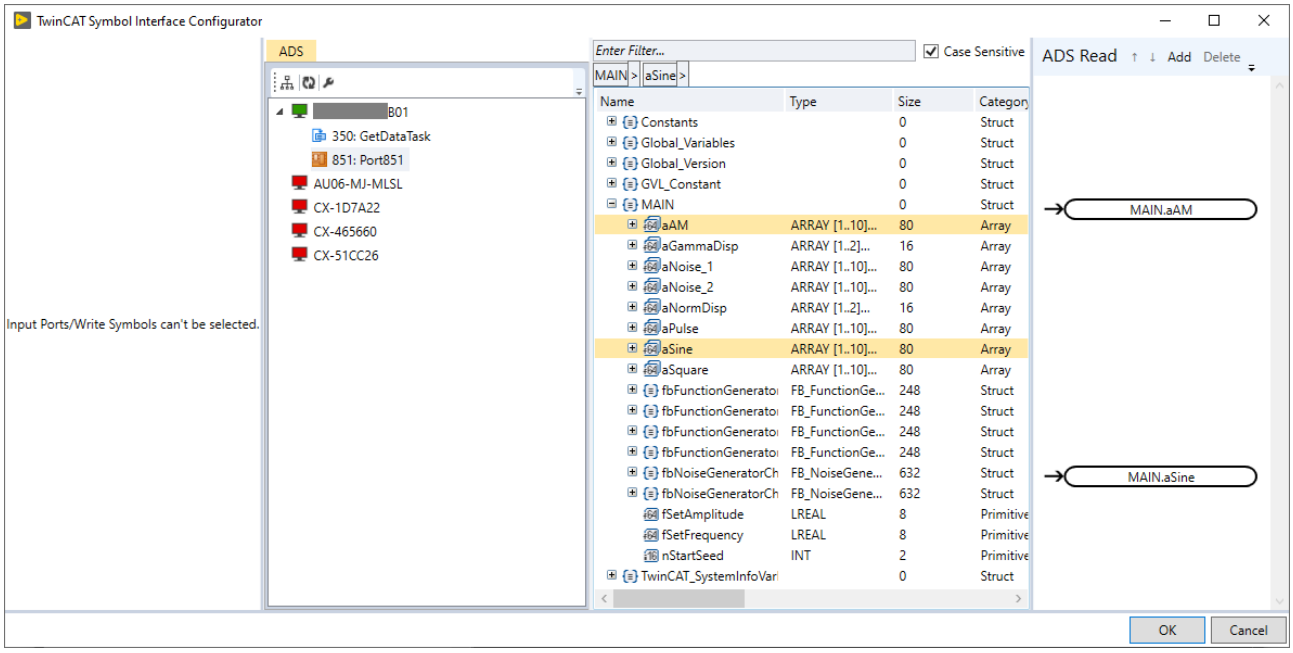
Creating LabVIEW™ project

✓ LabVIEW™ is open.

1. Create a new project and open an empty VI.
2. Save the VI.
3. Place an instance of the ADS DAQ VI on the block diagram. To do this, navigate in the **Functions palette** to
   User Libraries > Beckhoff LabVIEW Interface > ADS DAQ.
   ⇨ A user interface for configuring the connection to TwinCAT opens automatically.
4. Select **Symbol Interface** to create a new configuration.

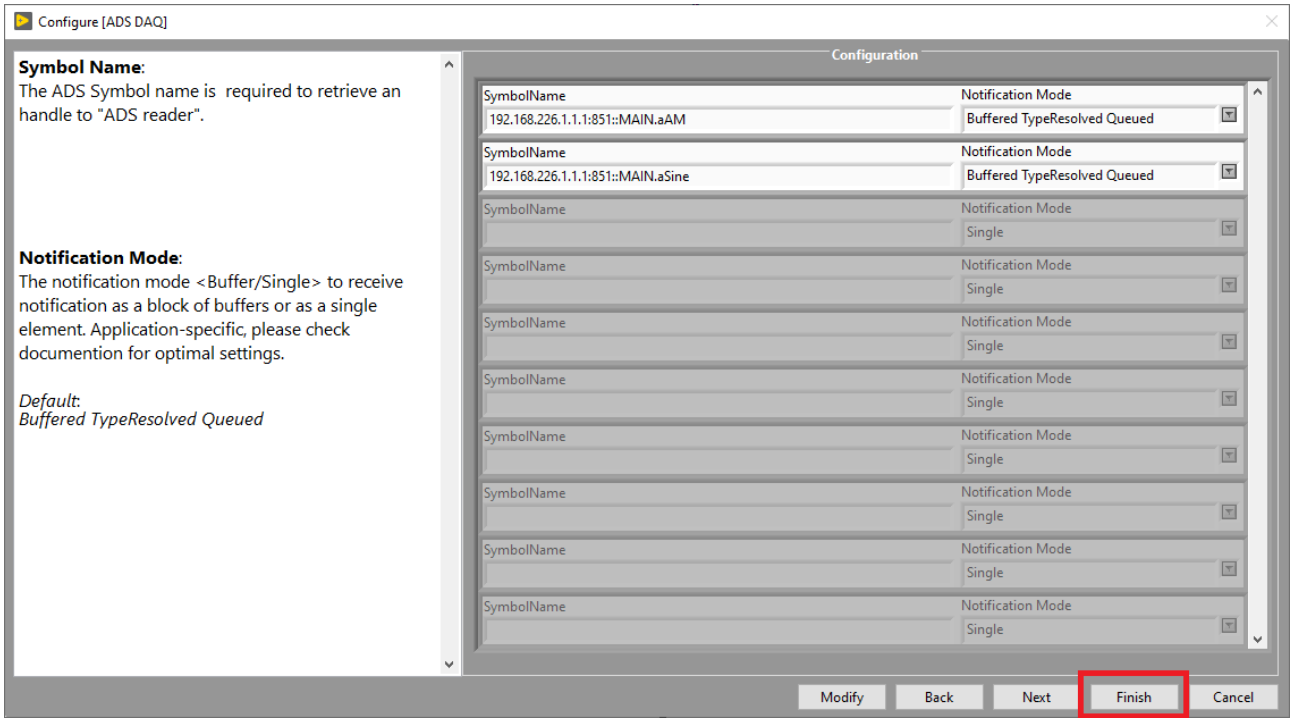


5. Browse with the Target Browser in the middle field into your target on which you have activated the TwinCAT project.
6. Then select the ADS symbols to be read. Navigate to **851: Port851 > MAIN** and select the ADS symbols aAM and aSine. Drag and drop both icons to the right area (**ADS Read** area).

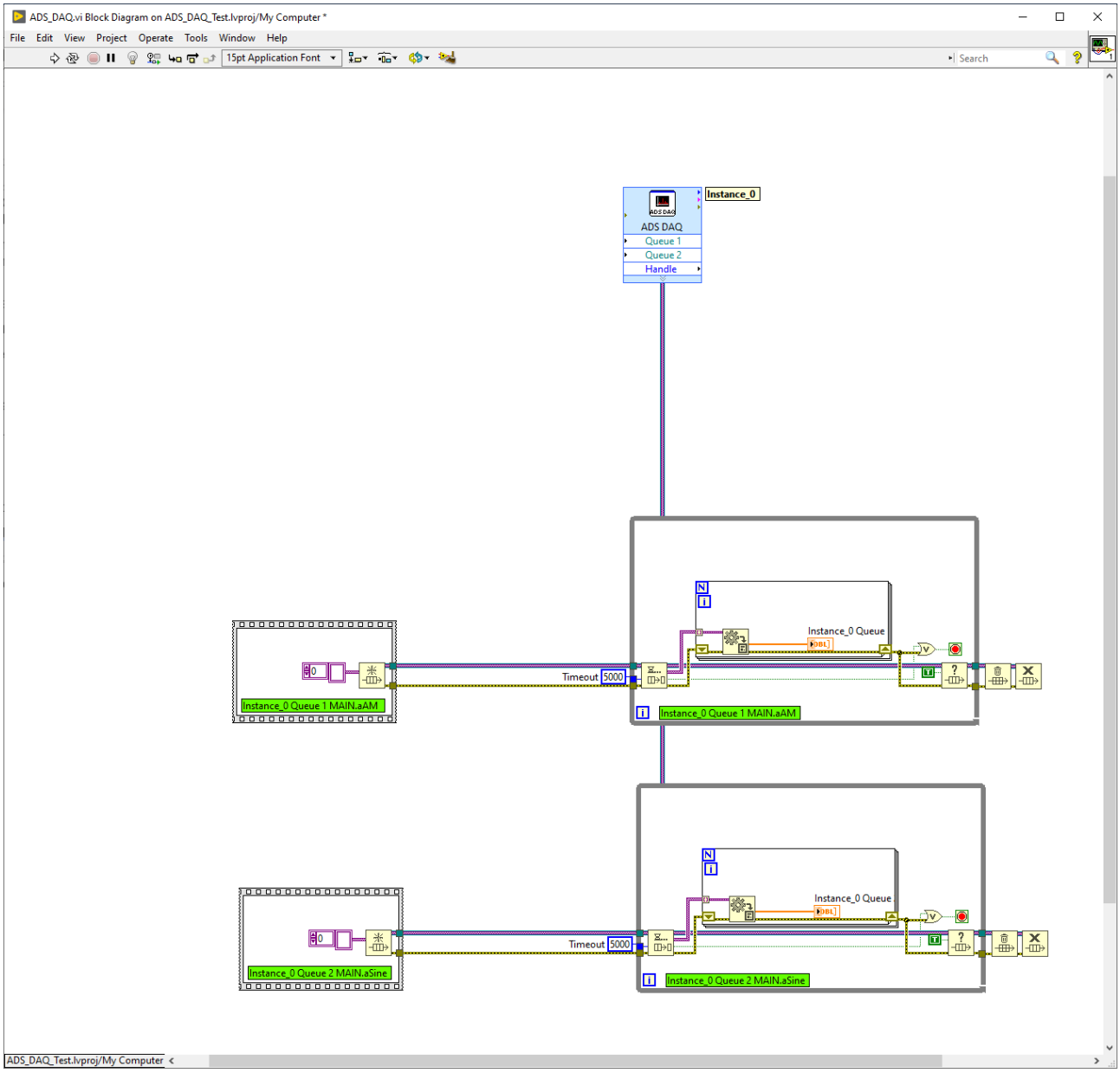


7. Click **OK**.

   ⇨ The Symbol Interface Configurator closes and you are shown further setting options. Don't change the default settings and select **Finish**.



   ⇨ The user interface is closed and code is automatically generated on the block diagram according to the configuration. You can change and extend this individually.

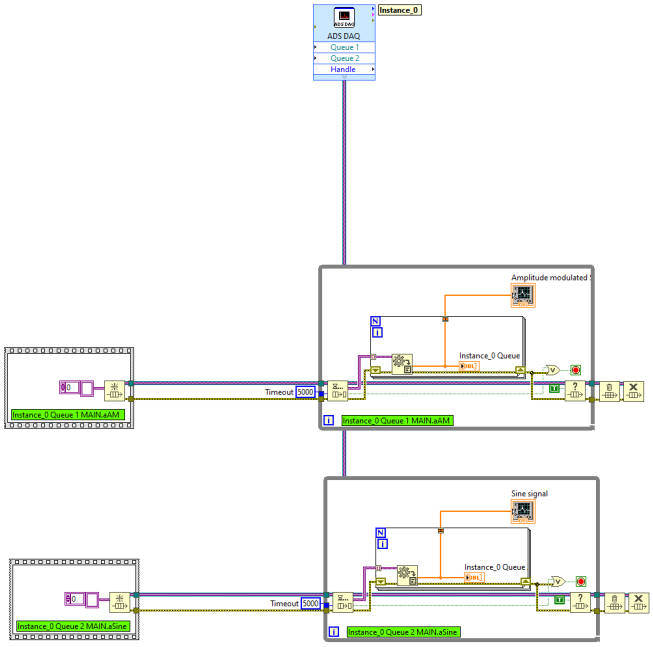


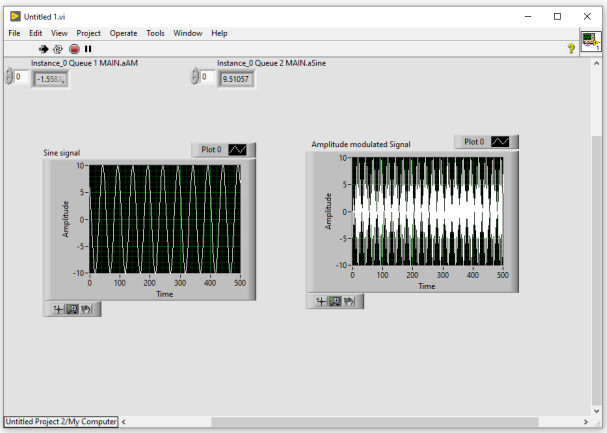
Extending block diagram

✓ Code was generated on the block diagram (see above).

8. Extend the block diagram, e.g. with a waveform graph.
9. Set your VI to Run mode.

⇨ In this case, look at the two time signals on the front panel.


Instance_0
ADS DAQ
Queue 1
Queue 2
Handle
Amplitude modulated
Instance_0 Queue
Timeout
5000
Instance_0 Queue 1 MAIN.aAM
Sine signal
Instance_0 Queue 2 MAIN.aSine


Untitled 1.vi
File Edit View Project Operate Tools Window Help
Instance_0 Queue 1 MAIN.aAM
Instance_0 Queue 2 MAIN.aSine
Sine signal
Plot 0
Amplitude
Time
Amplitude modulated Signal
Untitled Project 2/My Computer

5 Technical introduction

Data exchange between LabVIEW™ and TwinCAT 3 takes place using the Automation Device Specification (ADS) server-client protocol. With the Interface for LabVIEW™ an ADS client is realized in LabVIEW™, which enables a performant data exchange with TwinCAT runtimes. The TwinCAT runtimes implement the ADS server accordingly.

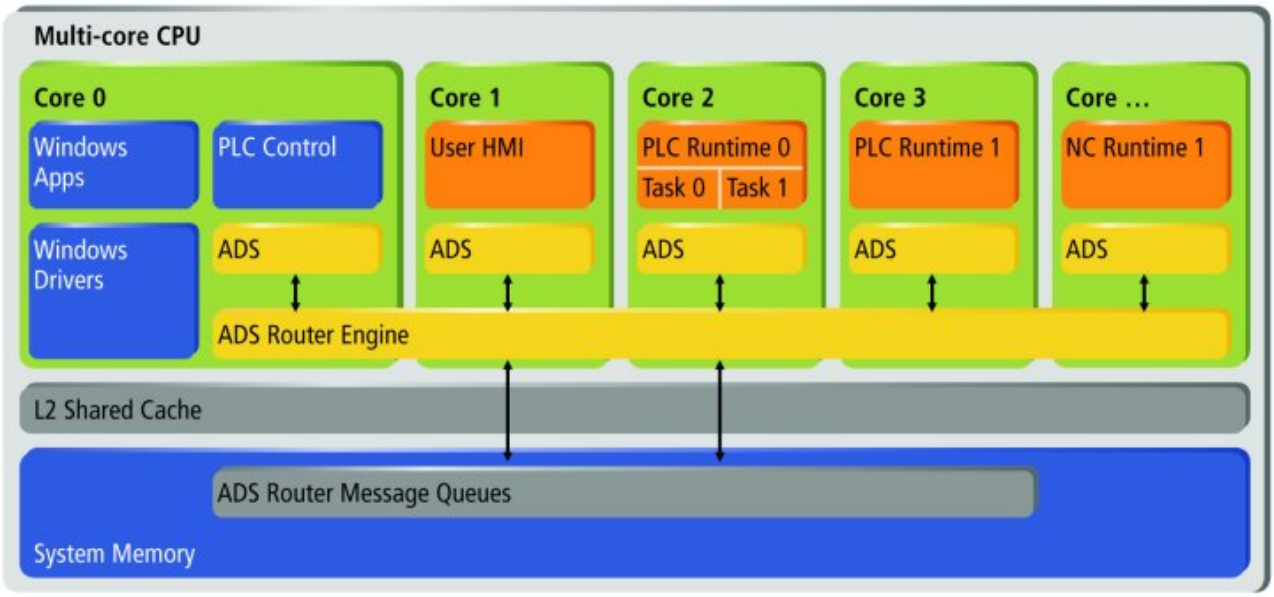
5.1 TwinCAT ADS

Basic structure of ADS devices and ADS symbols

Automation Device Specification (ADS) forms the basis for the LabVIEW™ interface. ADS describes a device- and fieldbus-independent interface and enables communication between ADS devices.

The ADS device concept and the identification of an ADS device are explained below.

The modular system architecture of TwinCAT allows the individual parts of the software (e.g. TwinCAT PLC, TwinCAT NC ...) to be regarded as independent devices: there is a software module for each individual task. The servers in the system are the executing devices that provide certain services. The clients are programs that request the services of the servers. A client initially establishes a connection to the server and requests a service: for example, it requests reading the value of a variable or it requests writing a variable.



The interface for LabVIEW™ provides an ADS client interface, which enables data exchange (read and write) with TwinCAT runtimes. The TwinCAT runtime, or its ADS devices, thus provide their services as ADS servers and can be used from LabVIEW™.

ADS data exchange between ADS devices takes place via the ADS router. As shown in the diagram above, data exchange between ADS devices implemented on the same system takes place via the system memory. If two ADS devices, e.g. LabVIEW™ and the TwinCAT runtime, are on different systems, a route can be created between two ADS routers. When creating the ADS route, the transport type (usually TCP/IP) for communication between the two ADS routers can be defined. Accordingly, an ADS device identifies itself via the AMS NetId of the ADS router and a port number, which then specifies the ADS device on the system. For example, port 851 is the default port for the first PLC instance in the TwinCAT runtime. ADS services of an ADS device are then specified by two parameters, the Index Group and the Index Offset. For example, a PLC variable is accessible for reading or writing under a specific Index Group and Index Offset.

If an application is to be created that is to be used on several target systems and in which LabVIEW™ and TwinCAT run on the same system, the relative AMS NetId can be used. The relative AMS NetId always points to the local system. For this purpose, 0.0.0.0.0.0 is used for the address.

Summary

- AMS NetId: Identifies the ADS router, i.e. the system.
- ADS Routes
- Port: Identifies an ADS device.
- Index Group/Offset: Specifies the ADS system service, e.g. a variable for reading and writing.

In order to make the addressing of variables in a TwinCAT runtime more convenient for the user, TwinCAT creates ADS symbols, which can be searched with the Target Browser, for example. The Target Browser is also integrated into the interface for LabVIEW™ in the VI Symbol Interface [▶ 65] so that ADS symbols can be selected easily and quickly. An ADS symbol for a variable in TwinCAT then contains the information mentioned above: AMS NetId, Port, Index Group and Index Offset and furthermore the Bit Size as well as a symbol name and the data type of the variable.

For more information on ADS, please see the following links:

- AMS NetId and Port: ADS device identification
- Index Group and Index Offset: Specification for ADS devices
- ADS Routes: System Node "Routes" and Add Route
- Connecting Devices with same AMS NetId: AmsNAT
- Using MQTT and Message Broker with ADS: ADS-over-MQTT
- TLS encrypted ADS: Secure ADS
- Target Browser
- Record ADS communication: ADS Monitor

Basic ADS data communication

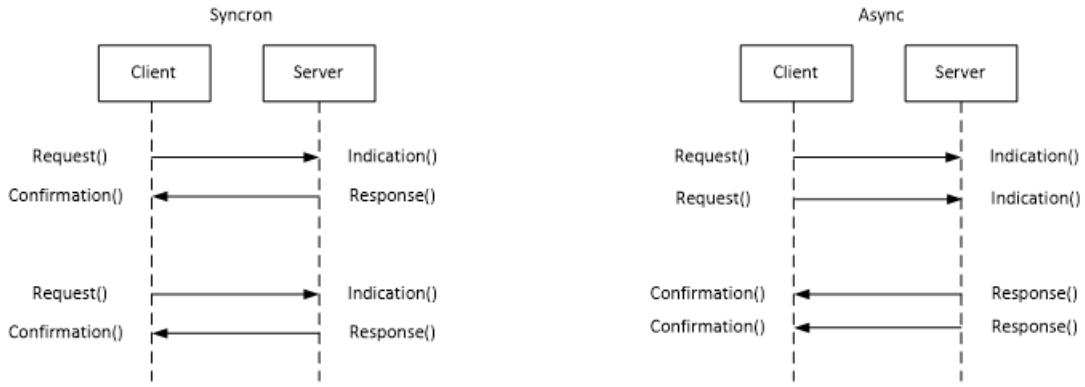
Basically, there are three different possibilities/modes for data communication in ADS.

Synchronous reading or writing

- The ADS client in LabVIEW™ waits for a response from the ADS server before proceeding with code execution.

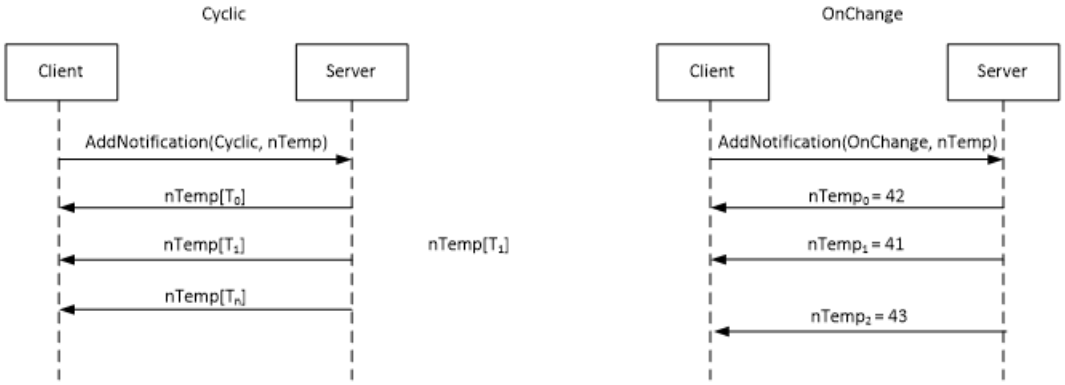
Asynchronous reading or writing

- The ADS client in LabVIEW™ sends a read or write request to the ADS server but does not wait for a response; instead, it continues to execute other parts of the program (for example, requests to other variables).



Event-based communication (read only)

- An ADS notification is only registered once on the server. The notification can be registered "on change" or "cyclic". After a notification has been registered at the server, it sends requested data to the server without any further requests from the client.



The VIs ADS Read [▶ 67] and ADS Write [▶ 72] are polymorphic. The desired communication mode (sync, async or notification) can be selected there. The VIs also provide extensions to these basic modes that involve LabVIEW™-side buffering of data and passing it to the LabVIEW™ process.

5.2 Communication modes

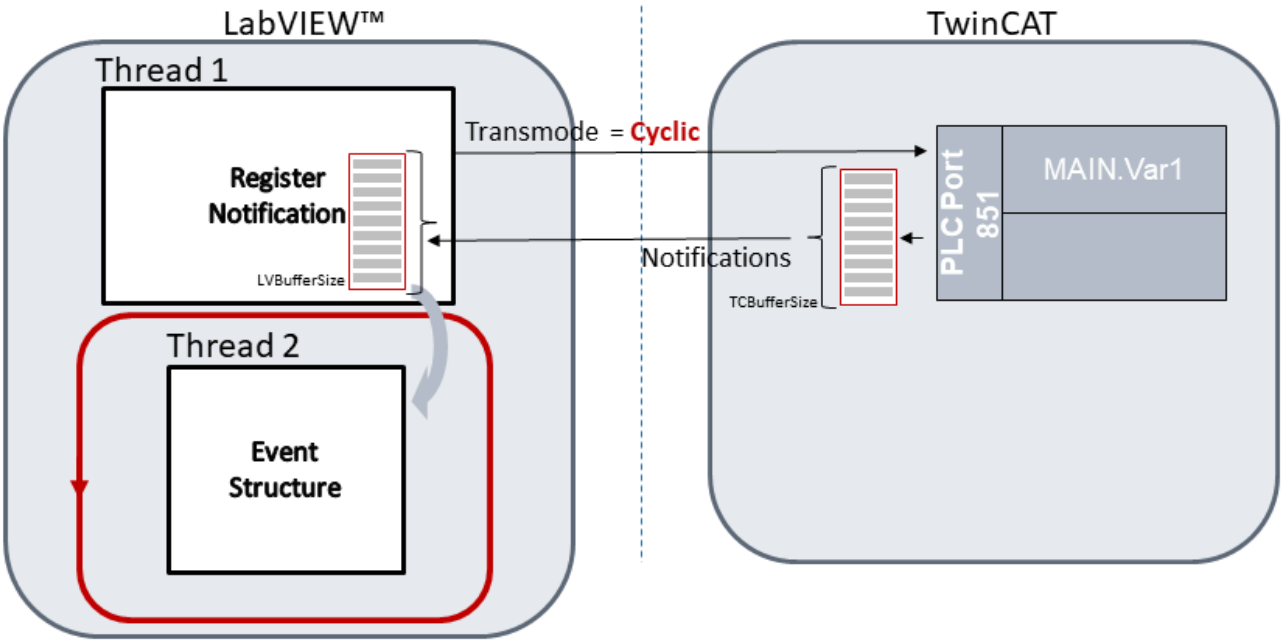
Initial classification

The extensions to the basic functionalities (sync, async and notification) are explained below. Practical recommendations are also given as to which mode is a preferred choice for exemplary applications.

The following table shows all provided modes with an initial classification.

Mode	Read	Write	Short description
Sync Single [▶ 26]	X	X	Read/write a variable in a self-contained one-time action. The client waits during the request to the server.
Async Single [▶ 27]	X	X	Read/write a variable in a self-contained one-time action. The client does not wait during the request to the server.
Noti. Single [▶ 27]	X		Waiting for an "On Change" event in TwinCAT as a one-time action. The notification is unregistered after the event.
Noti. Buffered [▶ 27]	X		One-time reading of a time series with a defined length. The notification is unregistered after receiving the time series.
E-Noti. Single [▶ 31]	X		Continuous reading (cyclic or "on change"). The notification is not automatically unregistered; it runs until it is explicitly unregistered. The notifications received are forwarded to an event structure in LabVIEW™.
E-Noti. Buffered [▶ 33]	X		Buffered continuous reading (cyclic or "on change“). The notification is not automatically unregistered; it runs until it is explicitly unregistered. The notifications received are forwarded to an event structure in LabVIEW™. The buffer enables higher data throughput, but causes latency.
LVB-Noti. Single [▶ 29]	X		Continuous reading (cyclic or "on change"). The notification is not automatically unregistered; it runs until it is explicitly unregistered. On the LabVIEW side, the data is written to a buffer to which the user has direct access.

In the table above some modes are assigned the property "Buffered". The diagram below illustrates the system structure. For "Noti. Buffered" and "E-Noti. Buffered", the addressed buffer refers to a LabVIEW™-side buffer of size `LVBufferSize`. The buffer is filled with data read from TwinCAT and transferred to LabVIEW™ when the buffer size is reached. In addition, there is a TwinCAT-side buffer from the standard ADS interface, which, however, can only be used with ADS notifications of the type *Cyclic*. By default the size of this buffer is `TcBufferSize` = 10 samples, although it can be adjusted.



In the following section, the above example is considered with the following parameters:

TcBufferSize = 10

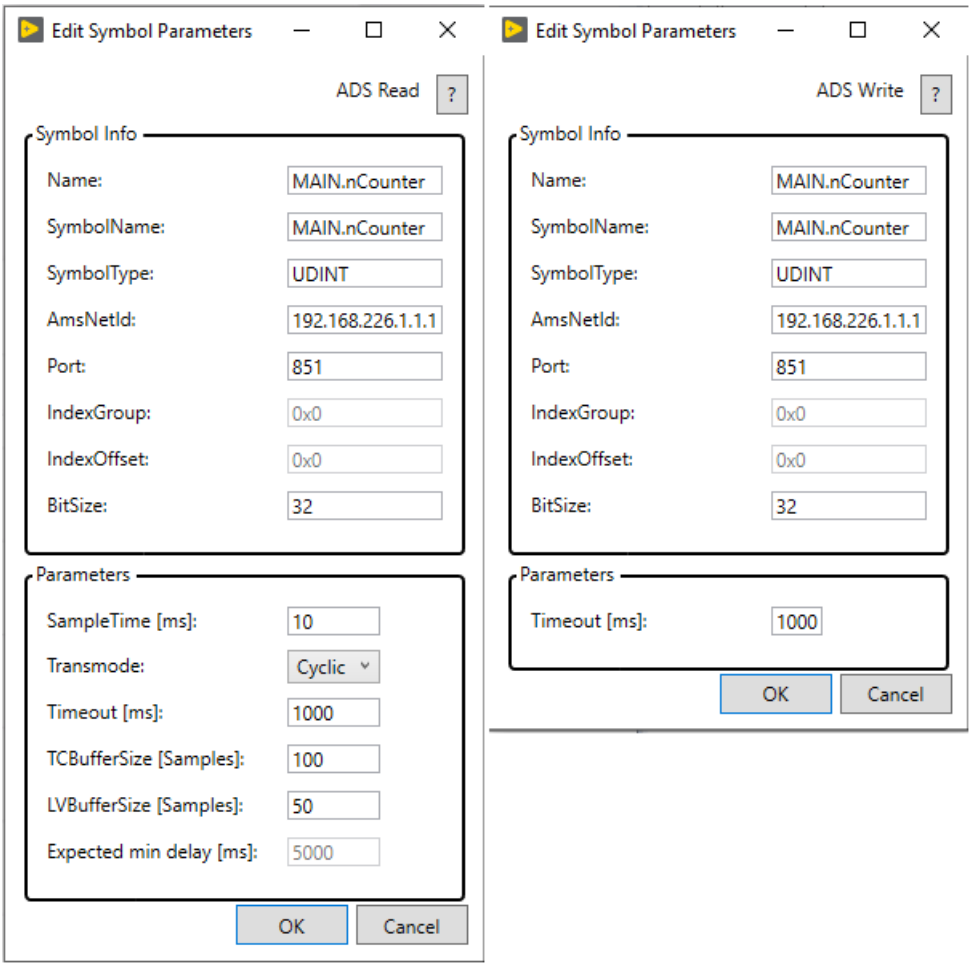
LVBufferSize = 500

ADS notification of the Cyclic type with 1 ms sample time (corresponds here to the cycle time of the PLC).

Every 1 ms a sample is now written into the TwinCAT-side buffer. After 10 ms the buffer is full and is transferred to the LabVIEW™-side buffer. After receiving 50 messages with 10 samples each, i.e. after 500 ms (plus communication time), the LabVIEW™-side buffer is full and then transfers the entire data packet of 500 samples to LabVIEW™. Accordingly, there is a *delay time* of at least 500 ms until the data packet is received in LabVIEW™. The minimum expected delay time is also displayed in the configuration dialog (Edit Symbol Parameters) in the lower part as "Expected min delay".

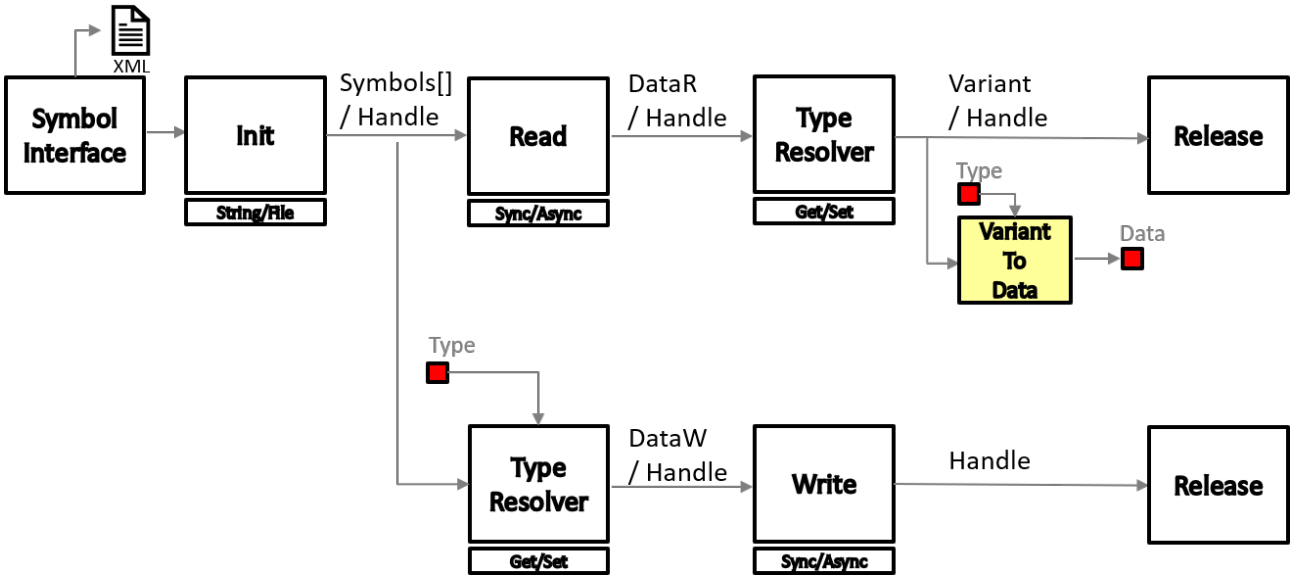
Parameterizing and realizing the communication modes

The parameters mentioned in the above section can be set in the User Interface (UI) of the Symbol Interface [▶ 65].



In addition to the parameterization of the ADS symbols, the basic structure of the LabVIEW™ program must be observed, which is outlined below.

The Symbol Interface [▶ 65] opens a dialog with the Target Browser and the possibility to parameterize each ADS symbol according to the above figure. The output of the Symbol Interface is a string in XML format describing all selected ADS symbols and their parameterization. This string is linked to the Init VI [▶ 66]. Alternatively, the Symbol Interface can export the XML. This XML file can then be passed directly to the Init VI. The Init VI then passes handles of the ADS symbols to a Read VI [▶ 67], which can be set as a polymorphic VI to the options described above. At the output there are data handles, which are linked with a Type Resolver VI [▶ 74]. The output of the Type Resolver VI is a Variant Type, which can be cast to the correct format. When writing, the order of Type Resolver and Write VI [▶ 72] is reversed accordingly.



The above diagram illustrates the structure with Low Level VIs. To simplify programming, several Low Level VIs are combined, depending on the communication mode. See Sync Read [▶ 26] or Async Read [▶ 27], for example.

Practical considerations

There are various options for reading data. In order to simplify the selection in practice, a table is provided below which describes exemplary application scenarios and suggests a communication mode.

Signal type in TwinCAT	Example	Communication mode
Quasi-static parameters	Read the current parameter set of a PID controller or the ID of the current product in the machine. Reading machine settings.	Sync Read [▶ 26], Async Read [▶ 27]
Continuous signal	Stream of one or more sensor signals (temperature, strain, force, …). There is no defined end of the data stream.	Fast response required: E-Noti. Single [▶ 31] (Transmode "cyclic")
	General DAQ application	High data throughput: E-Noti. Buffered [▶ 27] or LVB-Noti. Single [▶ 29]
Event	Waiting for TwinCAT event, e.g. start of a process: bStart changes from FALSE to TRUE. The event appears once. It is then no longer required and can be unregistered.	Noti. Single [▶ 27] (Transmode "on change")
Event	Responding to changes in states, e.g. changing the signals to be read depending on the state of the machine: Viewing the nState variables. The event occurs frequently and this event should be observed throughout.	E-Noti. Single [▶ 31] (Transmode "on change")
Signal of defined length	Reading of exactly 1000 samples of a specific sensor signal.	Noti. Buffered [▶ 27]

5.2.1 One-time reading

This communication mode is ideal, for reading a TwinCAT configuration or a completely filled data buffer in the PLC once, for example. In contrast to continuous reading [▶ 29], one-time reading requires no additional programming in LabVIEW™.

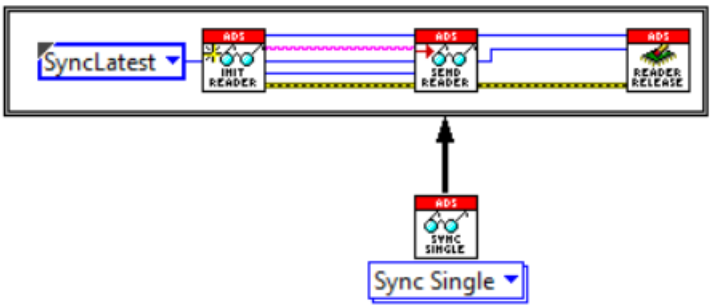
The TF3710 TwinCAT 3 Interface for LabVIEW™ product categorizes one-time reading into four cases:

1. Synchronous reading
2. Asynchronous reading
3. Notification Single
4. Notification Buffered

Synchronous reading

With synchronous reading, after a request has been sent by the client, a response from the ADS server is awaited before the program code is executed further. The reader is released immediately after successful confirmation from the server. This type of reading is therefore suitable for calculating or displaying the data directly after it has been received from TwinCAT.

The polymorphic "**Sync Single**" block is composed of several low-level VIs [▶ 92] and thus combines the creation of a handle, reading and releasing.

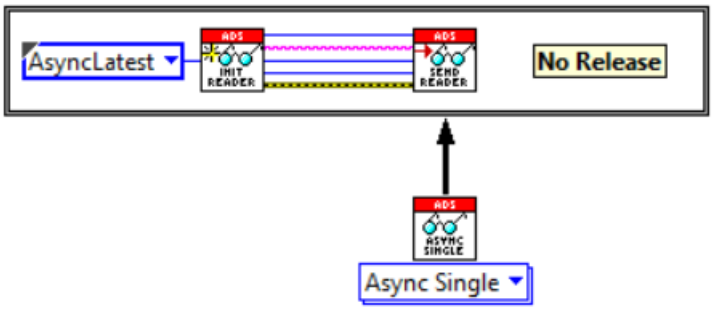


Examples in LabVIEW™: Simple Read with TypeResolving [▶ 111]

Asynchronous reading

With asynchronous reading, the client does not wait for a response from the ADS server. In this mode, there is no guarantee that the reader has already received the data packet or not. As a result, the reader cannot be released.

The polymorphic VI "**Async Single**" therefore initializes the reader and sends the request to TwinCAT. The release is not part of the VI.



Examples in LabVIEW™: Read Async with TypeResolving [▶ 112]

Notification Single

Notification Single is only supported for *Transmode* **"on change"**.

Unlike Notification E-Single [▶ 31], the block "Noti.Single" does not require a user-programmed event structure. The notification is only intended for one-time use, i.e. to catch a one-time event. The notification is registered in the background, received and then removed.

This block returns an array of two values, the last state before the value change and the new state after the value change, e.g. [FALSE, TRUE], if a variable has changed its value from FALSE to TRUE.



Examples in LabVIEW™: Simple Read with TypeResolving [▶ 111]

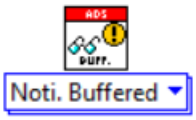
Notification Buffered

Unlike E-Buffered Notification [▶ 33], the block "Noti.Buffered" does not require a user-programmed event structure. The block uses a buffer of size `LVBufferSize` to save the data received from TwinCAT in an intermediate layer. The notification is not removed until the buffer is full. The saved data are then passed to LabVIEW™.

Accordingly, reception of a time series with predefined length (LVBufferSize samples) can easily be realized with this communication mode.

LVBufferSize

The buffer size is determined by the parameter `LVBufferSize` when creating the ADS symbol, see Symbol Interface [▶ 65] VI.



Examples in LabVIEW™: Simple Read with TypeResolving [▶ 111]

5.2.2 One-time writing

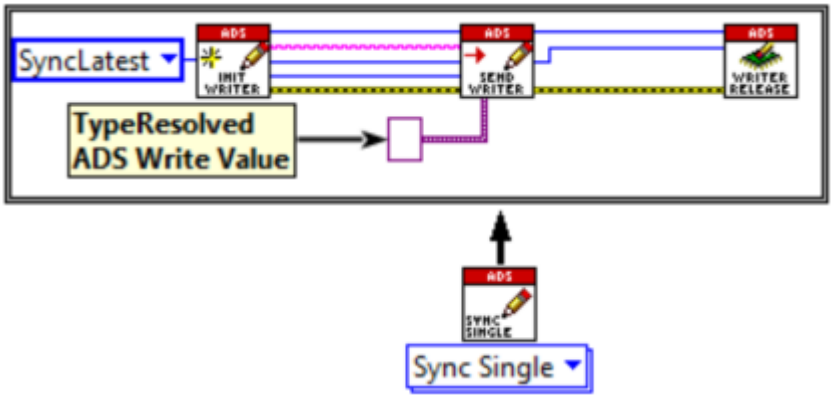
The TwinCAT 3 Interface for LabVIEW™ categorizes one-time writing as follows:

1. Synchronous writing
2. Asynchronous writing

Synchronous writing

During synchronous writing, the system waits for a response from the server (TwinCAT). The writer is released after the data packet has successfully been sent to TwinCAT.

This process is performed in the background by the polymorphic VI "**Sync Single**".

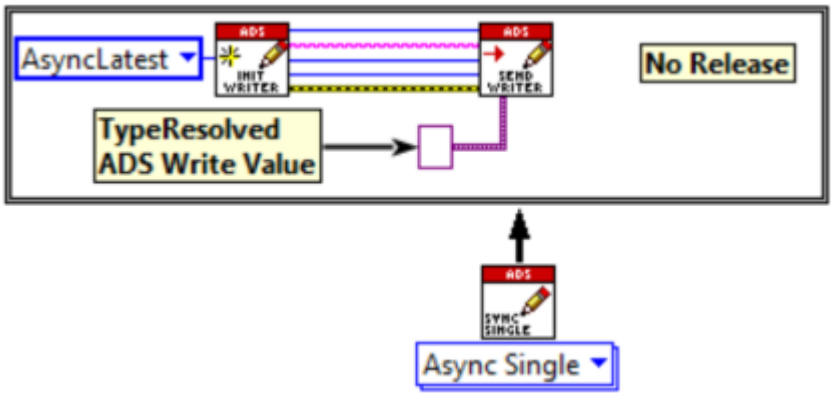


Examples in LabVIEW™: Write Sync with TypeResolving [▶ 113]

Asynchronous writing

When writing asynchronously, the system does not wait for a positive acknowledgement from the server. In this mode, there is no guarantee that the Writer has already sent the data packet or not. As a result, the writer cannot be released.

The polymorphic VI "**Async Single**" therefore initializes the writer and sends only the request to the server. A release is not executed in the VI.



Examples in LabVIEW™: Write Async with TypeResolving [▶ 113]

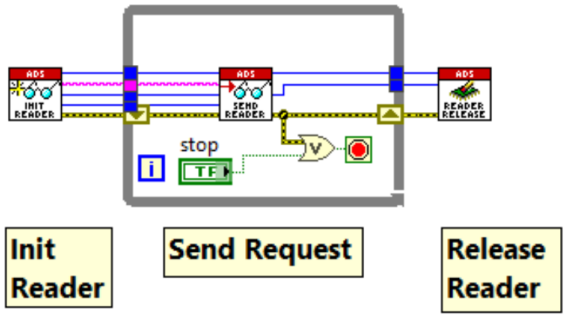
5.2.3 Reading data continuously

When continuously reading a time series from TwinCAT, the client in LabVIEW™ continuously receives data packets from the ADS server. The data packets can be requested cyclically through a polling procedure or event-based as ADS notification. Both options are explained below.

Reading with polling cycle

Simple reading

With polling, the client sends requests to the server at a defined time interval. This can be easily built using the Low-Level [▶ 92] VIs, for example. The ADS reader is initialized only once and requests a new data packet from TwinCAT with each cycle of the loop. With each cycle a new request is sent to the server and a corresponding response is awaited. The ADS reader is released once the termination condition of the loop has been reached. The image below shows the complete procedure in LabVIEW™.

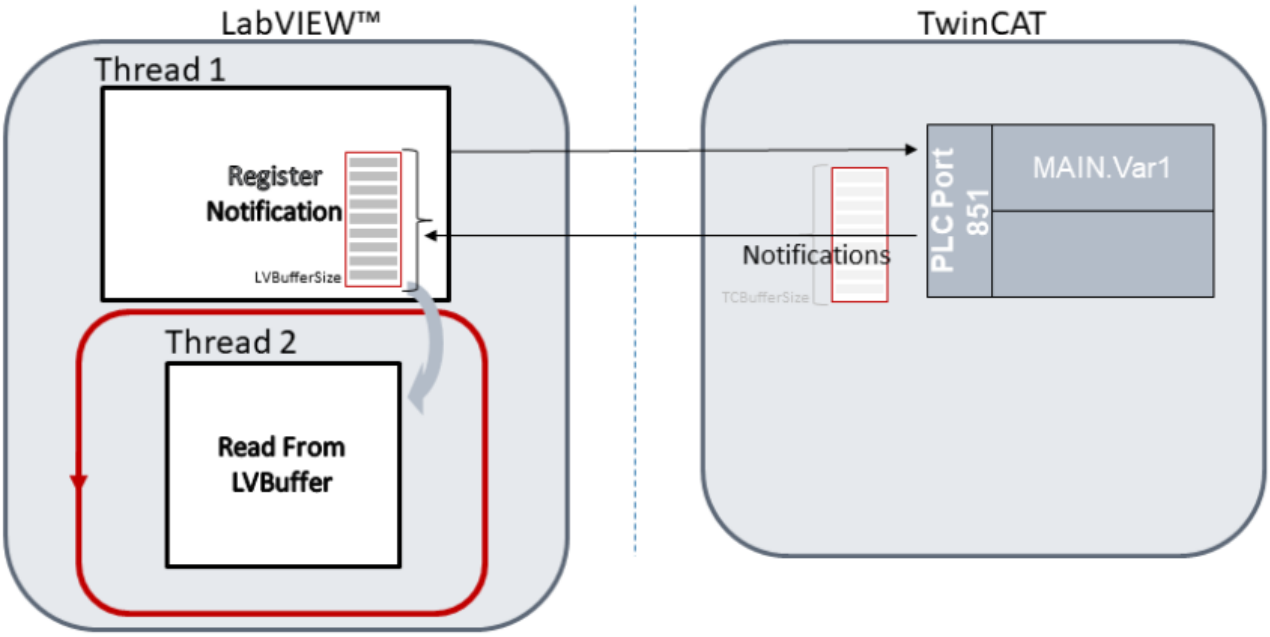


Example in LabVIEW™: Continuos Read Sync Base [▶ 113]

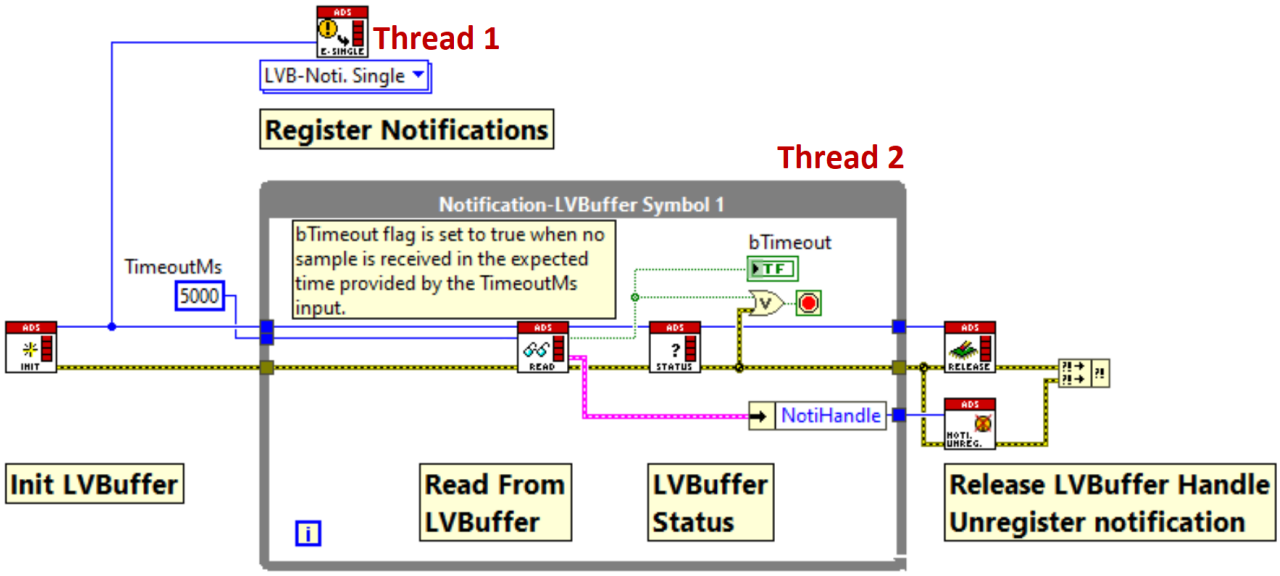
The procedure is error-prone in the sense that it cannot be guaranteed that a cyclically changing value in the PLC is sampled without gaps. For this use case, it is recommended to work with e-notifications, which are described below.

Reading with LVB notification

Compared to Event-driven reading [▶ 31], the ADS notifications can also be read continuously with a specific polling cycle. Reading in this case happens asynchronously, as described in the graphic below. Here the ADS notifications are registered by the Thread1 and written into the LVBuffer. Later the LVBuffer is read with the Thread2.



In contrast to Notification E-Buffered [▶ 33], no LabVIEW™ events are used in this case. Accordingly, the LabVIEW™ user has direct access to the **LVBuffer**. In LabVIEW™ the block diagram looks like this:



Examples in LabVIEW™: Read Notification-LVBuffer Multiple [▶ 115]

The block diagram uses the LVBuffer blocks (see Init LVBuffer Handle [▶ 80]):

- Init LVBuffer Handle
- Read From LVBuffer
- LVBuffer status
- Release LVBuffer Handle

The block diagram uses the ADS notification blocks:

- LVB-Noti. Single Symbol [▶ 71]
- Unregister Notification [▶ 79]

Polling cycle

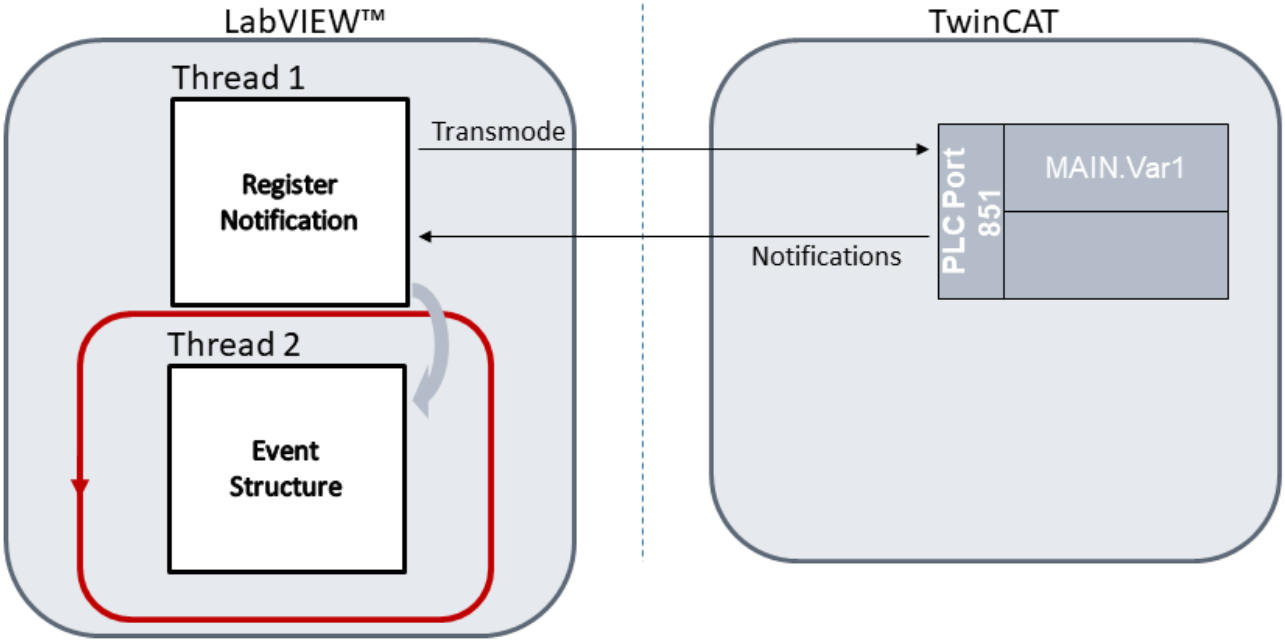
If the polling cycle is slower than the notification cycle time, the LVBuffer may overflow and samples may be lost.

5.2.3.1 Event driven reading

With event-driven reading, LabVIEW™ events and ADS notifications are used together. An ADS notification only has to be registered once on the server. The server data are then received cyclically or on change (see "Transmode" in chapter Communication modes [▶ 23]) from the client. Accordingly, the client only needs to issue one request, and the server then controls whether a new message needs to be sent to the client.

The ADS client in LabVIEW™ forwards the received notifications (data packets) as LabVIEW™ event to an *event structure*. See also the LabVIEW™ documentation on User events. The event structure inserts the received LabVIEW™ event into an internal queue. The events in the queue are processed by LabVIEW™ using the FIFO principle. If the notifications are received faster than it is possible to process the event structure in LabVIEW™, the queue becomes larger. This ensures that no notification is lost. It must be noted that this increases the amount of memory used. Accordingly, it must be taken into account that a point in time must be reached at which the memory requirement (the pending events) can be processed.

We recommend using the LabVIEW™ *Event Inspector Window* (View > Event Inspector Window) to observe the processing of LabVIEW™ events.



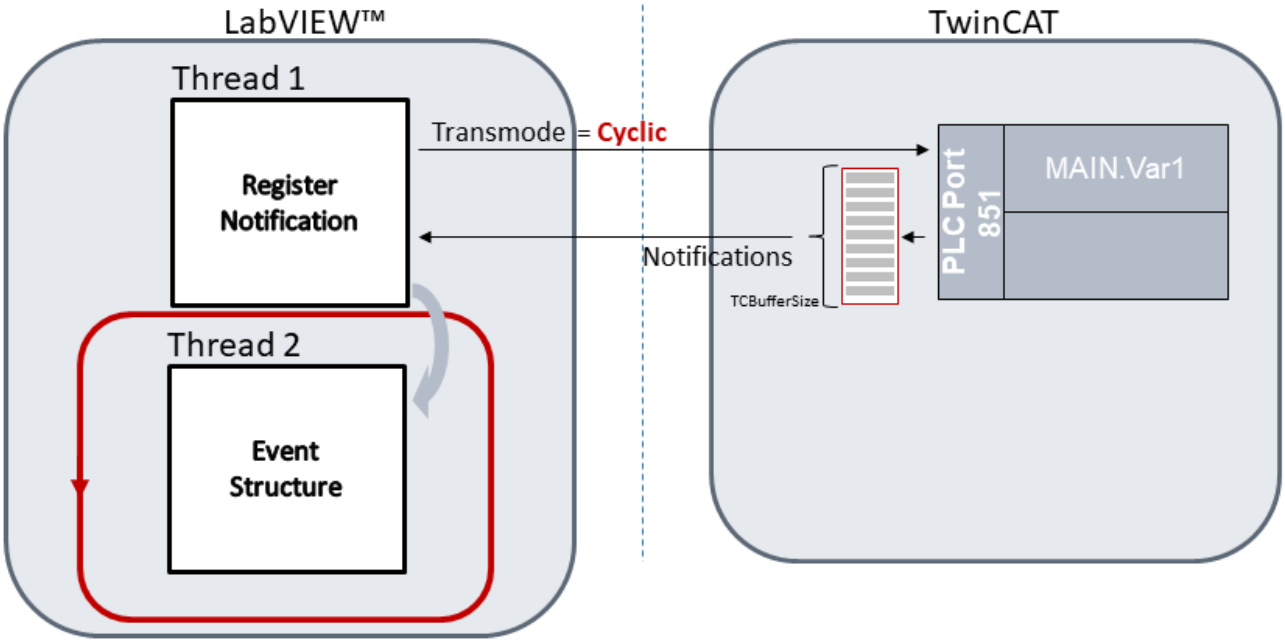
The interface for LabVIEW™ offers two different modes of operation for continuously registering ADS notifications as LabVIEW™ events:

1. E-Notification Single
2. E-Notification Buffered

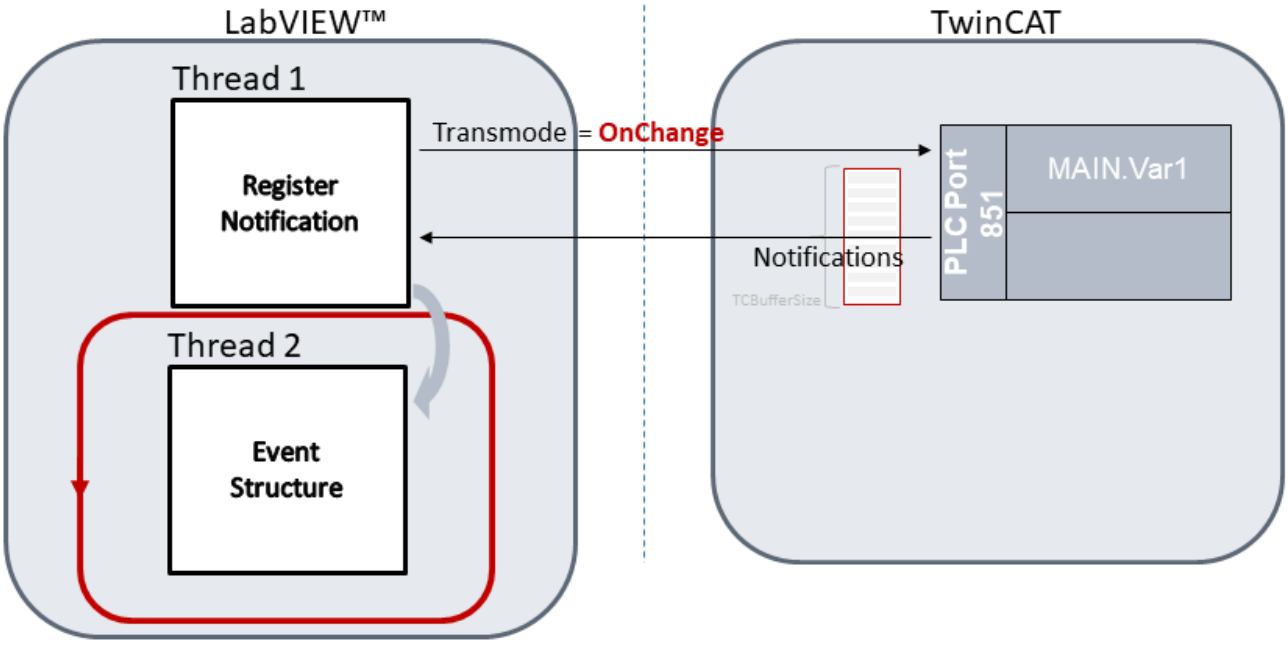
E-Notification Single

The operation mode *E-Notification Single* immediately forwards the received ADS data packet to the LabVIEW™ event structure, i.e. it directly generates a LabVIEW™ event. A data packet can contain several samples. *Transmode* and *TCBufferSize* are key parameters for this:

Transmode "**Cyclic**" uses the TwinCAT-side buffer (of size TCBufferSize). In this case notifications are first written to the TCBuffer and then bundled and sent to the LabVIEW™ client when the buffer is full. This achieves higher data throughput, and there is less load on network thanks to fewer messages with low user data volume.



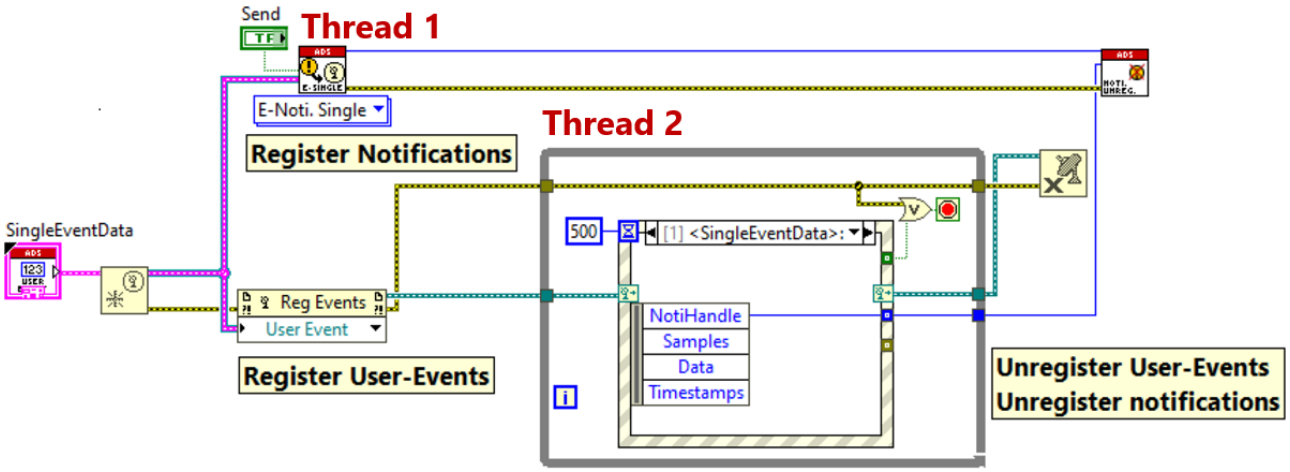
With *Transmode* "**OnChange**" the TCBuffer is **not** used. Here, the individual notifications **are forwarded directly to the client without** buffering. The client receives any change of state with minimal latency and can respond directly.



In LabVIEW™ the block diagram corresponds in principle to the above picture. Here the *Register Notification Block* and the *Event Structure* run in two different threads. The event is registered only once. The system then waits for the notifications in the event structure. Each received notification contains:

- Notification Handle
- Number of samples
- Array of notification data
- Array of ADS timestamps

If no notification is received, the event structure times out.



Examples in LabVIEW™: Read Notification-Event Single [▶ 114], Read Notification-Event Multiple [▶ 115]

E-Notification Buffered

The operation mode *E-Notification Buffered* uses a data buffer on LabVIEW™ side in addition to the above structure. The buffer size is set by the parameter `LVBufferSize` when the ADS symbol is created. The received ADS data packets are first written to the LabVIEW™-side buffer in an intermediate layer and only forwarded to the event structure as a LabVIEW™ event when the buffer has been filled.

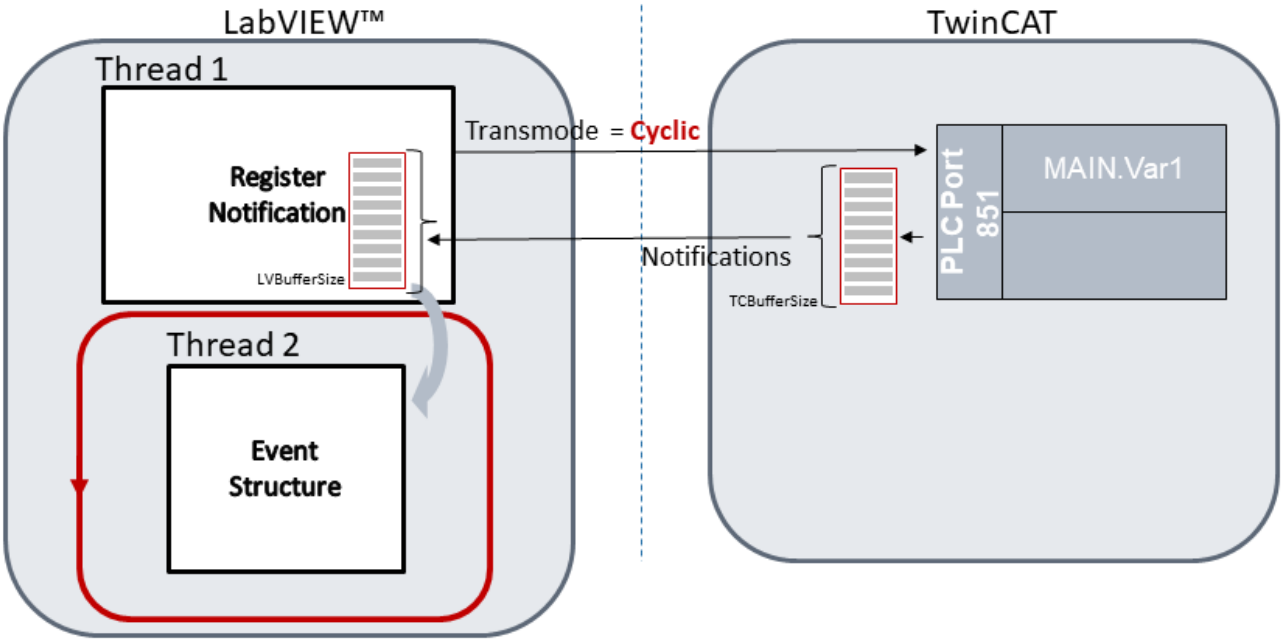
This approach generates LabVIEW™ events less frequently than in the E-Notification Single variant. By submitting larger data packets to LabVIEW™, processing can be made more efficient, resulting in higher overall data throughput.

Buffer size

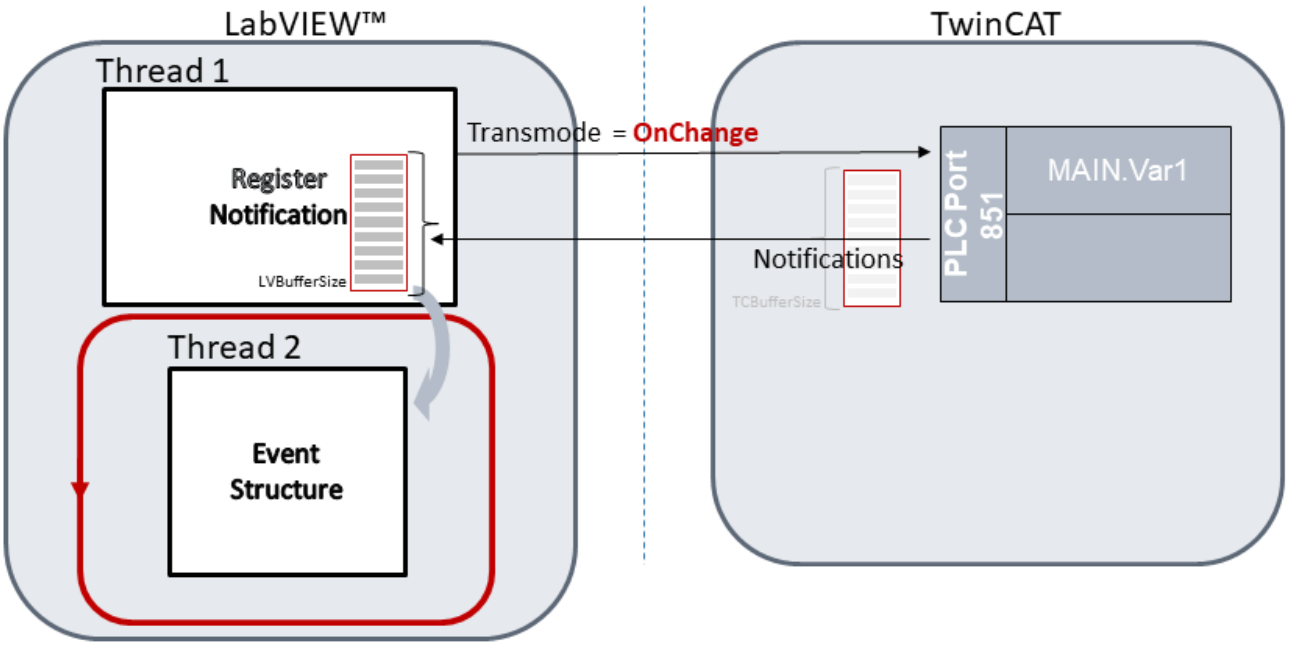
The TwinCAT 3 interface for LabVIEW™ reserves an additional 10% buffer memory when generating LVBuffer.

Transmode and *TCBufferSize* are also the key parameters for this operation mode:

With *Transmode* "**Cyclic**" the TCBuffer and the LVBuffer are used.

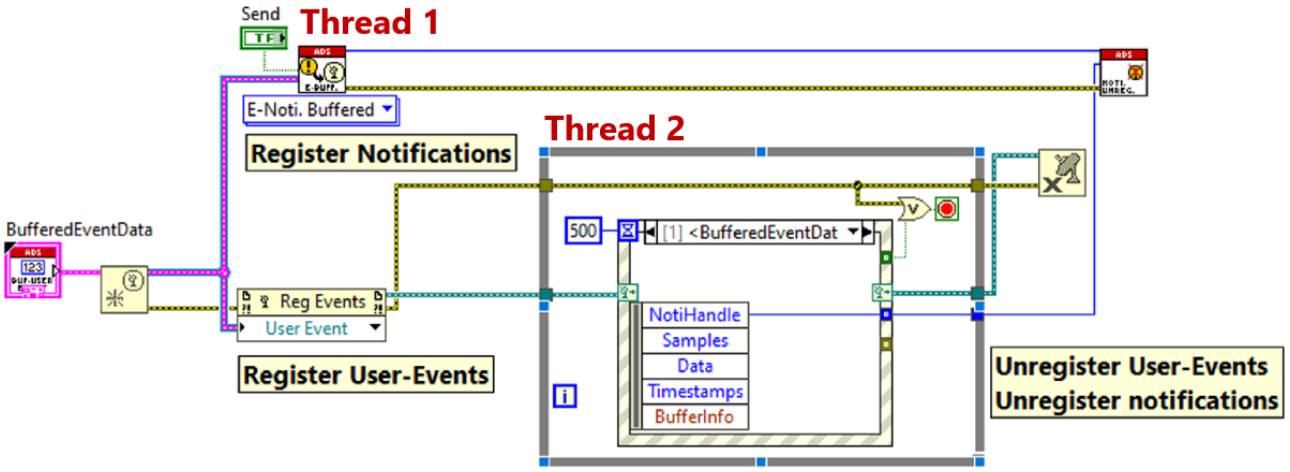


With *Transmode* "**OnChange**" the TCBuffer is **not** used. The individual notifications are transferred directly to LVBuffer without buffering.



In LabVIEW™ the block diagram is similar to the E-Notification Single mode. Each received notification has an additional field called *BufferInfo*. BufferInfo provides additional information about the LVBuffer:

- Previous Overflow Samples: Describes a counter that indicates how often the created buffer was not sufficient. Example: The buffer has been created by the user with 50 samples. As described above, 10% more storage space is created internally, i.e. 55 samples. Several samples can be transferred for each notification thanks to TwinCAT-side buffering. If the buffer in LabVIEW™ is already occupied with 45 samples and another packet with 10 samples arrives, the maximum buffer is not exceeded and the counter does not increment. If, on the other hand, the ADS packet was to contain more than 10 samples, data would be lost because the buffer would not be sufficient. Accordingly, the counter would increment by one.
- Missing Samples: Describes the difference between received and expected samples. Example: The LabVIEW™-side buffer has 50 samples, so 50 samples are expected. However, if only 48 samples are received, the value of Missing Samples is two.
- Buffer Usage: in percent %



Example in LabVIEW™: Read Notification-Event Buffered [▶ 114], Read Notification-Event Multiple [▶ 115]

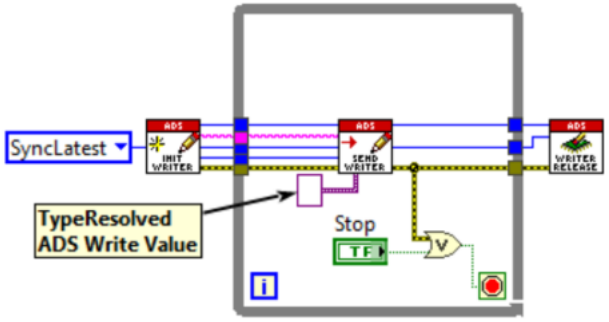
5.2.4 Writing data continuously

During continuous writing LabVIEW™ sends write requests to TwinCAT. Sending of requests is based on a polling cycle.

Writing with polling cycle

Writing with the ADS synchronous block uses so-called continuous polling. Here the low-level [▶ 92] writer blocks are used. The ADS writer is initialized only once. It sends a new *TypeResolved* data packet to the ADS server with each cycle (iteration). With each cycle a new request is sent to the server and an acknowledgement is awaited. The ADS writer is released once the condition for exiting the loop had been reached. The image below shows the complete procedure in LabVIEW™.

This type of writing from LabVIEW™ to TwinCAT can be used to write values directly to an output terminal with short cycle times, for example.



Example in LabVIEW™: Continuos Write Sync Base [▶ 116]

5.3 Type Resolving

The TwinCAT 3 Interface for LabVIEW™ provides **TypeResolver,** to parse the ADS data stream from LabVIEW™ data type to TwinCAT 3 data type or vice versa. A comparison is first made between the LabVIEW™ data type and the TwinCAT 3 data type. If the data types are the same, the ADS data stream is parsed and copied appropriately. The TypeResolver supports the following operation modes:

1. Resolve ADS data stream from TwinCAT 3.
2. Resolve ADS data stream for TwinCAT 3.

Resolve ADS data stream from TwinCAT 3

The block **From TC** initializes the TypeResolver, parses and converts the ADS data read from TwinCAT with ADS Read into the LabVIEW™ data type *Variant*. The block Type Release releases the TypeResolver from memory. *Variant* is converted to the appropriate LabVIEW™ data type via Variant-to-Data VI.

The following graphic shows how the composition of TypeResolver from the Low Level VIs [▶ 98]. In this example we assume a structure in TwinCAT that consists of three elements.

Definition of the structure:

```
TYPE ST_toLabVIEW :
STRUCT
    TCSignedWord  : INT;
    TCUnsignedInt : UINT;
    TCStringArray : ARRAY [1..5] OF STRING;
END_STRUCT
END_TYPE
```

The TwinCAT 3 data type and the LabVIEW™ data type must match for the conversion process. In the appendix you will find a translation table [▶ 127] as an overview. On the right side of the block diagram is a LabVIEW™ container that mirrors the above structure.

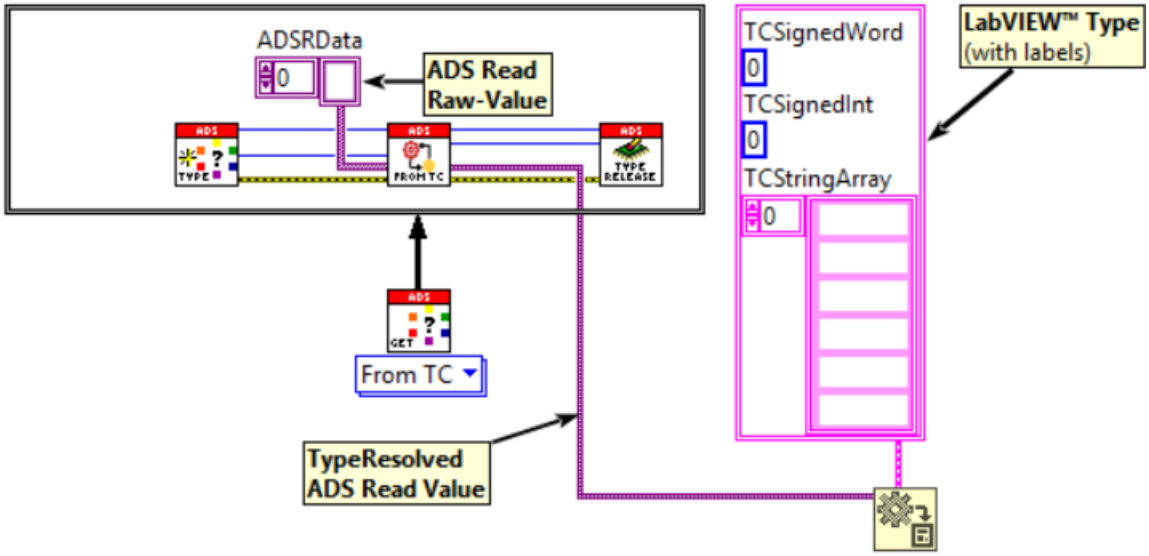
Structure of the LabVIEW™ container (not initialized, merely defined):

- I16
- U16
- TCStringArray is 1..n Elements and of type string (not further specified in LabVIEW™)

Copying data may fail for the following reasons:

- The elements of the LabVIEW™ container are not configured in the correct order.

- The entries of the structure and the container do not have the same data type.



Resolve ADS data stream for TwinCAT 3

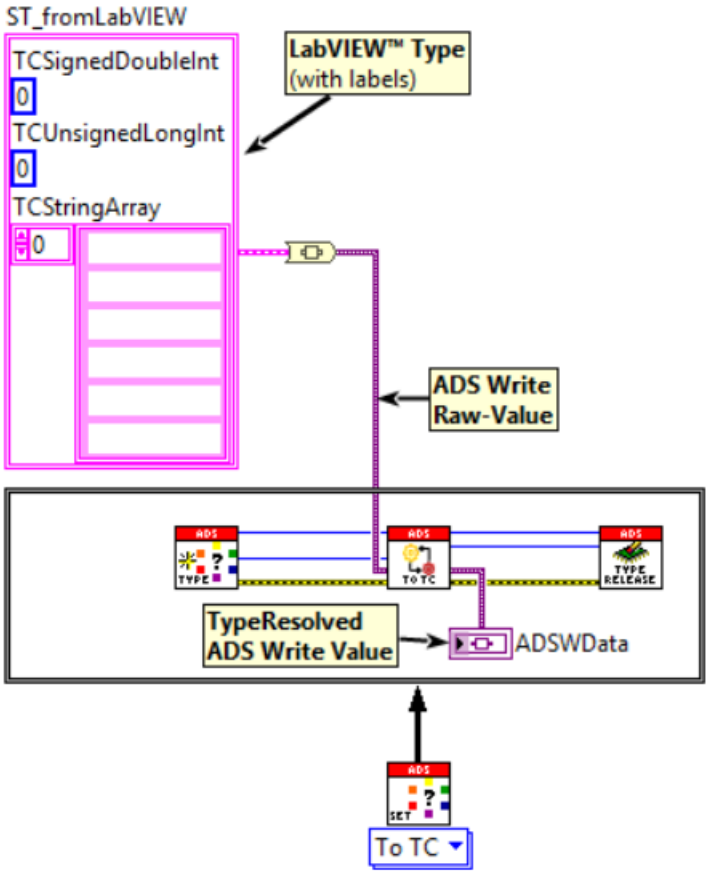
The block *To TC* initializes the TypeResolver, compares the LabVIEW™ data type with the TwinCAT 3 data type and converts the LabVIEW™ data into ADS raw data. The block then releases the TypeResolver from the memory. The converted ADS data can then be passed directly to an ADS Write [▶ 72] block.

The diagram below shows a similar scenario as described earlier for *From TC*. In this case the data are converted from a LabVIEW™ data type to a TwinCAT 3 data type. The conversion may fail for the following reasons:

- The elements of the LabVIEW™ container are not configured in the correct order.
- The entries of the structure and the container do not have the same data type.
- The elements in the structure are not initialized.

Initialization of the LabVIEW™ data type

When converting from LabVIEW™ to TwinCAT 3 data type, the LabVIEW™ data type must be pre-initialized. For a complex data type, such as a structure, the complete data type must be initialized. For an array the individual elements must be initialized.



For the diagram above, for example, the following TwinCAT structure is conceivable in the PLC as a target for writing:

```
TYPE ST_fromLabVIEW :
STRUCT
    TCSignedDoubleInt  : DINT;
    TCUnsignedLongInt : ULINT;
    TCStringArray : ARRAY [1..6] OF STRING(80);
END_STRUCT
END_TYPE
```

The matching initialized LabVIEW™ container is then structured as follows:

- I32
- U64
- String array with 6 elements (80 characters per array element may be used here)

Automatic type generation

Manual creation of matching data types in LabVIEW™ can be time consuming and error-prone. The TwinCAT Interface for LabVIEW™ offers the possibility of automatic type generation. Use the TypeResolver [▶ 98] and the Generate Type [▶ 76] wrapper block for this purpose. See the chapter Examples [▶ 111] for several variations on how to use these VIs effectively: Example Type Resolver [▶ 76].

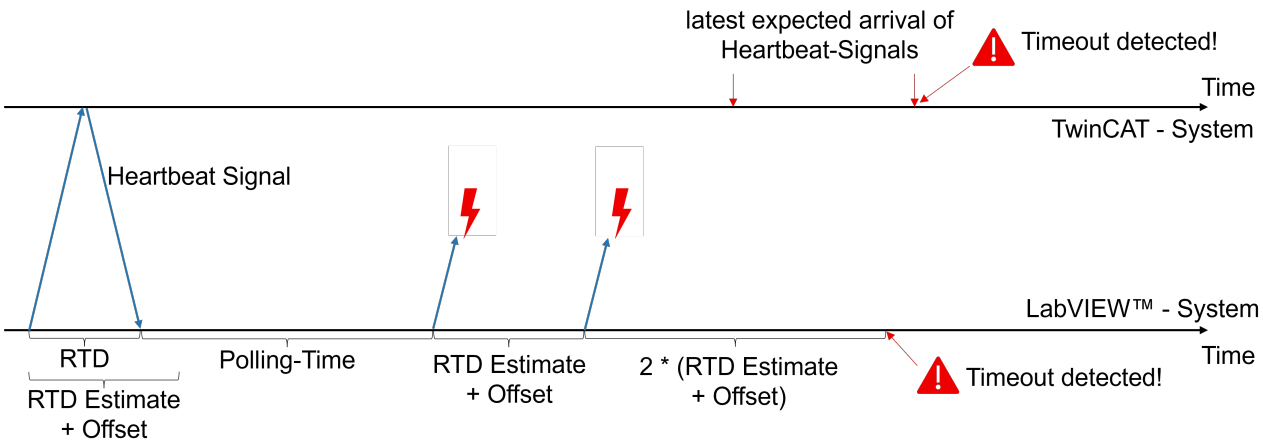
5.4 Heartbeat Connection Monitoring

To monitor the connection between LabVIEW™ and TwinCAT, Heartbeat signals can be exchanged periodically between the two systems. This requires corresponding functions on both sides.

PLC project required

A PLC and a PLC license are required in the TwinCAT project to use Heartbeat Connection Monitoring.

The following shows how the Heartbeat signals are used to detect an interruption in the connection.



Note: This figure serves as an illustration. The time spans drawn are not to scale.

The maximum expected **R**ound**t**rip **D**elay (RTD) is estimated in LabVIEW™ for each Heartbeat signal. This is the time required for the transmitted signal to be received by TwinCAT and sent back again. The estimate is referred to as *RTD Estimate*. If the Heartbeat signal arrives back in LabVIEW™ within the RTD Estimate (as shown in the figure on the left), the connection is OK. If this is not the case, the connection may be interrupted (as shown in the illustration on the right). To check this, another Heartbeat signal is sent out for validation. This increases the RTD estimate. If this Heartbeat signal also fails, an interruption is detected (timeout detected).

The Heartbeat signals are sent periodically with the polling time T. This is used by TwinCAT 3 to determine the time until which the next Heartbeat signal is expected. If the Heartbeat signal arrives in TwinCAT before this time, the connection is OK. If the Heartbeat signal fails twice in succession, an interruption is also detected on the TwinCAT side (timeout detected).

Roundtrip Delay (RTD) Estimation

Estimating the next maximum expected RTD is central to connection monitoring. The estimation is used both in LabVIEW™ and in TwinCAT. The underlying algorithm is identical in both systems. There are two ways of estimation: dynamic or static.

Dynamic RTD Estimation: In dynamic estimation, the next expected RTD (RTD Estimate) is calculated on the basis of recorded RTDs. A combination of different algorithms is used here. An offset parameter (see below) can be used to influence the estimate. The configured offset is added to the result of the dynamic RTD estimation. This allows the risk of a falsely detected interruption and the response time to be individually adapted to the use case. A larger offset leads to a lower probability of an interruption being falsely detected and at the same time increases the response time of the monitoring.

Static RTD estimation: With static estimation, a constant value is set as the maximum RTD. This value is set by the user via the offset parameter.

Maximum delay until an interruption is detected

The maximum delay until an interruption is detected depends on the estimation method for the RTD.

The following approximation applies to the dynamic estimate:

Response time ≈ Polling time + 3*RTD Estimate + Offset

The following applies approximately to the static estimate:

Response time ≈ polling time + 3* offset

Parameterization

The parameters can be used to influence the behavior of the connection monitoring. The following parameters are available for this purpose:

Name	Name
Polling time	Timespan in ms that elapses between two Heartbeat signals.

Name	Name
UseRtdEstimation?	Selection of the method for estimating the RTD (TRUE: dynamic estimation; FALSE: static estimation)
Offset	Dynamic estimation: Additional waiting time in ms Static estimation: Constant RTD estimate in ms

Parameterization only in LabVIEW™

The parameters are configured in LabVIEW™ and automatically adopted by TwinCAT. Parameterization in TwinCAT is not required.

It makes sense to use the parameters to adjust the behavior of the monitoring to your individual use case. The following considerations should provide support here:

Polling time: The longer the polling time, the longer the response time. At the same time, a shorter polling time leads to more Heartbeat signals being sent, which may increase the network load. The response time of the connection monitoring can be easily set via the polling time (please note that polling time != response time).

The polling time has no influence on the risk of a falsely detected interruption.

Offset: The offset can be used to reduce the risk of an incorrectly detected interruption at the expense of the response time.

Creating a function block in TwinCAT

For Heartbeat Connection Monitoring, an instance of the FB_Heartbeat function block must be created for each port in the TwinCAT project for which monitoring is to take place. To do this, the "Tc3_HeartbeatConnectionMonitoring" PLC library must be added in TwinCAT and then an instance of the FB_Heartbeat function block must be created and called.

```
PROGRAM MAIN
VAR
   fbHeartbeat  :  FB_Heartbeat;
END_VAR
fbHeartbeat();
```

The function block has outputs that display the state of the connection.

Name	Meaning
bConnected	Indicates whether there is a connection to the LabVIEW™ system.
bTimeoutDetected	Indicates whether the connection has been interrupted.
bStopped	Indicates whether connection monitoring has been stopped from LabVIEW™.
AmsAddr	Specifies the target system to which the Heartbeat connection exists.

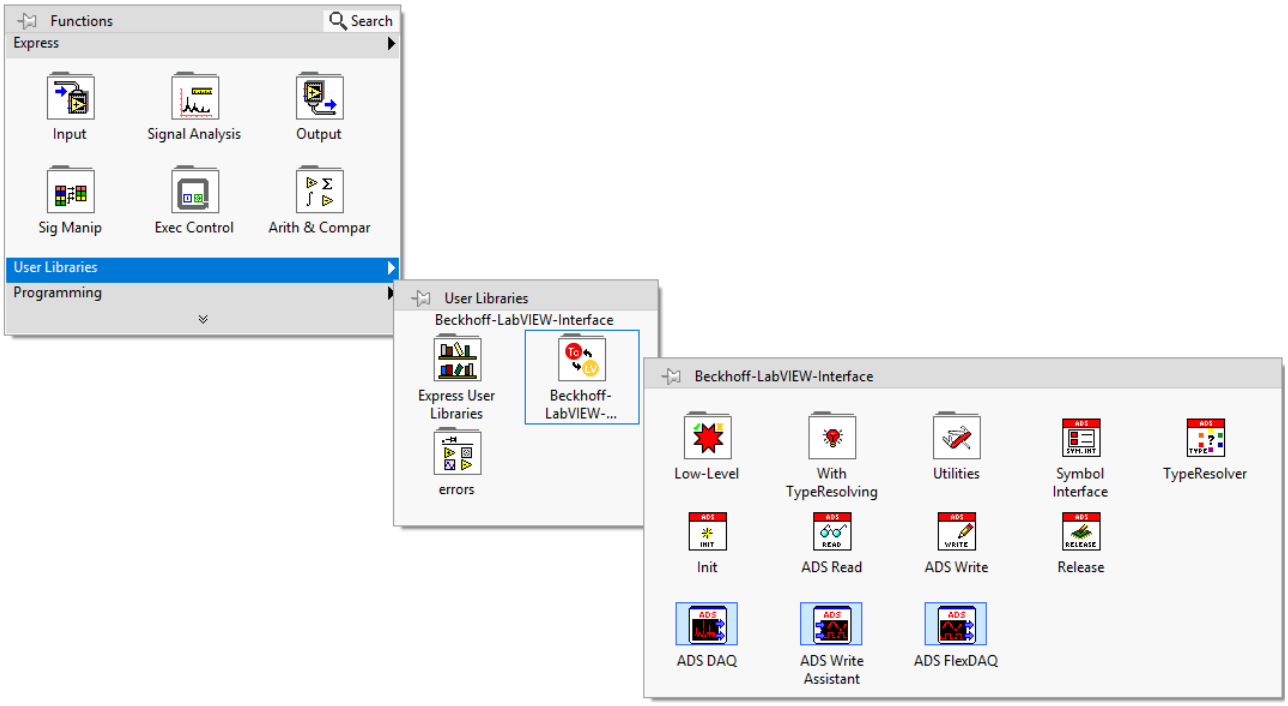
Connection monitoring to multiple target systems

A Heartbeat signal from a LabVIEW™ project can be received for each instance of FB_Heartbeat, but a LabVIEW™ project can realize connection monitoring to several TwinCAT systems.

6 LabVIEW™ VIs

The TwinCAT 3 Interface for LabVIEW™ provides controls and VIs for use in LabVIEW™.

The VIs are located in the block diagram in the *functions palette*: **Functions > User Libraries > Beckhoff-LabVIEW-Interface**.



The main folder contains the basic VIs that can be used to build a program for reading via ADS, writing via ADS, TypeResolving and releasing the ADS client. In addition, the main folder contains the subfolders: Low-Level, With TypeResolving and Utilities.

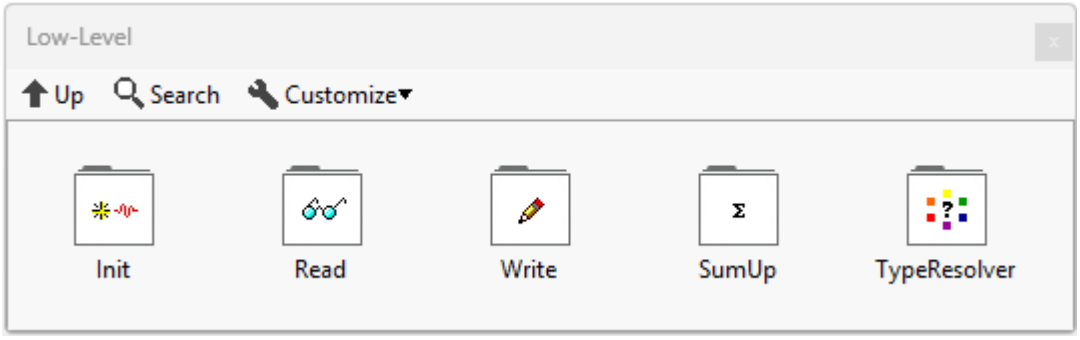
Low-Level

The *Low-Level* subfolder contains Low-Level-VIs. The low-level VIs operate based the same principle as the basic VIs. The low-level VIs involve a little more programming effort but offer higher performance (in terms of data throughput) than the basic VIs and more flexibility for realizing complex programs. Reading with Polling Cycle [▶ 29] is an example that uses low-level VIs for fast reading via ADS. Writing with polling cycle [▶ 35] is a similar example. Not only reading and writing can be accelerated in this way, but also TypeResolving, see Continous Read [▶ 113], for example.

The table describes the subfolders and their contents and function:

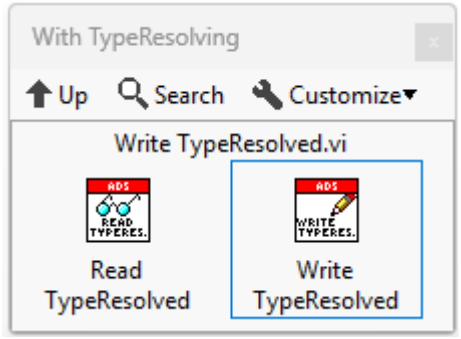
Subfolder	VIs	Function
Init [▶ 92]	Base Init	Initializes the ADS client.
	Get List of ReadWrite Symbols	Creates a list of ADS read and write symbols.
	Get List of Registered Targets	Creates a list of registered ADS target systems.
Read [▶ 94]	Init Reader	Initializes the ADS Reader.
	Send Reader-Request	Sends a request to the ADS server.
	Register Notification	Registers the notification on the ADS server.
	TryReadData	Checks the response from the server and reads the data stream.
	Release Reader	Releases the reader from memory.
Write [▶ 96]	Init Writer	Initializes the ADS Writer.
	Send Writer Request	Sends a request to the ADS server.

Subfolder	VIs	Function
	CheckWriteStatus	Checks for the response from the ADS server to ascertain whether the data packet was received.
	Release Writer	Releases the writer from memory.
SumUp [▶ 104]	Init SumUp	Initializes the ADS-SumUp.
	Add SubCommand	Adds a new command to the SumUp.
	Put Data	Puts new data to the initialized command that is to be written to TwinCAT.
	Get Data	Gets data from a SumUp command that was read from TwinCAT.
	Enable Autosend	Enables automatic sending of SumUp commands.
	Send SumpUp	Sends all SumUp commands to the TwinCAT runtime.
	Release SumUp	Releases the SumUp handle in the memory.
TypeResolver [▶ 98]	Init Type	Initializes the TypeResolver.
	Resolve From TC Type	Converts the TwinCAT 3 data type to a LabVIEW™ data type variant.
	Resolve To TC Type	Converts the LabVIEW™ data type variant into a TwinCAT 3 data type.
	Release Type	Releases the TypeResolver from memory.



With TypeResolving

The *With TypeResolving* subfolder contains two VIs for reading and writing via ADS with integrated TypeResolver block.

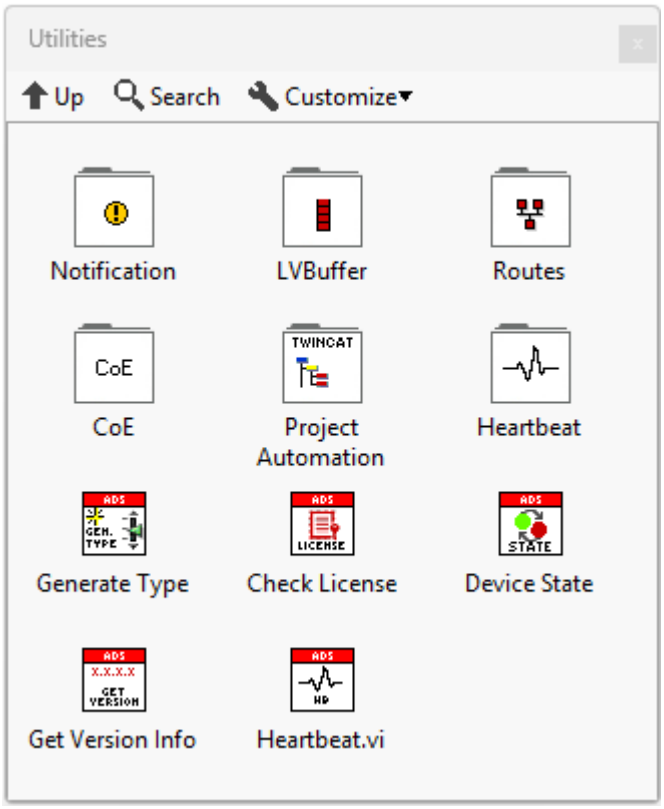


Utilities

The *Utilities* subfolder contains additional VIs for the following purposes:

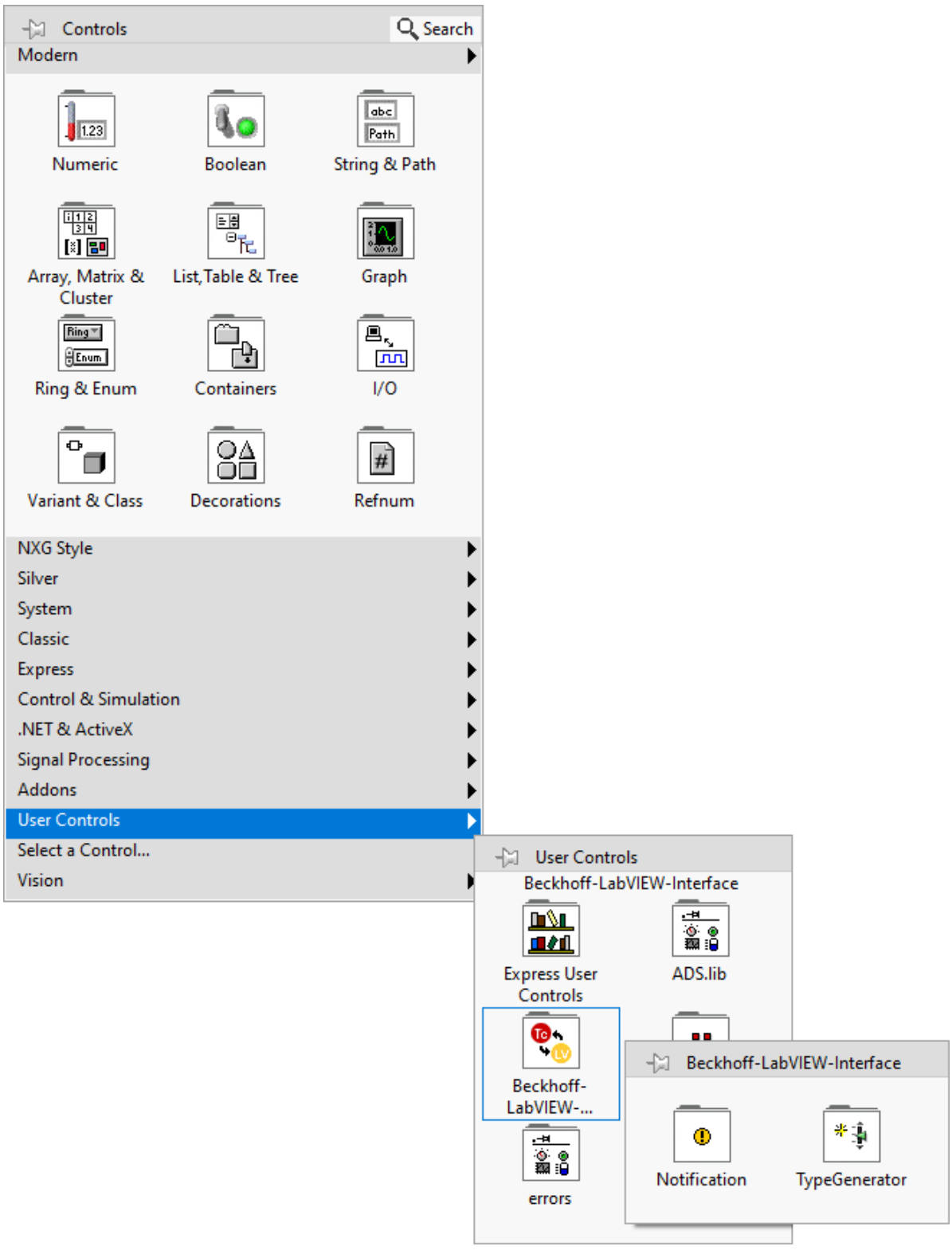
Subfolder	VIs	Function
Notification [▶ 78]	ADS To LabVIEW Timestamp	Converts ADS timestamps to LabVIEW™ timestamps.
	Notification Data To Variant Array	Builds an array of LabVIEW™ *Variant* from the notification data stream.
	Stop Notification	Stops the ADS notifications.

Subfolder	VIs	Function
	Start Notification	Starts the ADS notifications.
	Unregister Notification	Unregisters the notification on the server.
LVBuffer	Init LVBuffer Handle	Initializes a handle for reading an ADS notification using the LabVIEW™-side data buffer.
	Read From LVBuffer	Reading data from the data buffer
	LVBuffer status	Reading the number of samples in the data buffer
	Release LVBuffer Handle	Releases the handle to the data buffer
Routes	Get Routes	Reads all target systems that are connected to the local TwinCAT system.
	Get Route Info	Specifies all subsystems and ports for a target system.
CoE	Read CoE List	Reads a CoE directory of a terminal and lists the available CoE objects.
	Read CoE Description	Reads the description of a CoE object and specifies the sub-elements.
	Read CoE Entry	Reads an entry from a CoE directory and outputs the properties.
	Read CoE Value	Reads the value of a CoE entry.
	Write CoE Value	Writes the value of a CoE entry.
Project Automation	Init Project	Initializes a TwinCAT 3 engineering project. Either an existing project is opened or a new one is created.
	Create Task	Creates a task in a TwinCAT 3 project or changes the task parameters.
	Scan IO	Checks existing inputs and outputs on the target system and adds them to the project tree.
	Create PLC	Creates a PLC within a TwinCAT 3 project.
	Activate Solution	Activating a TwinCAT 3 project.
	Release Project	Releases the handle to the TwinCAT 3 project.
Heartbeat	Init Heartbeat	Creates a Heartbeat instance with a handle.
	Run Heartbeat	Starts connection monitoring.
	Stop Heartbeat	Stops connection monitoring.
	Release Heartbeat	Releases the handle to the Heartbeat instance from memory.
	Get RTD Estimate	Indicates the current estimate of the Round Trip Delay.
	Heartbeat	Initializes, starts and stops connection monitoring in a VI.
	Generate Type	Generates a corresponding constant or control with the LabVIEW™ data types for a symbol using the TypeInfo of the TypeResolver.
	Check License	Checks the license state on a given target system.
	Set Device State, Get Device State	Reads or changes the state of an ADS device.
	Get Version Info	Provides information regarding the product version.



Controls

The controls are located in the front panel in the *controls palette* under: **User Controls > Beckhoff-LabVIEW-Interface**.



The Notification subfolder contains controls that are required when initializing the LabVIEW™ event for ADS notifications.

Furthermore, the subfolder “TypeGenerator” contains TypeGenerator [▶ 99] class objects to convert TwinCAT types into LabVIEW™ types (see Generate LabVIEW™ type [▶ 116]).

The following table describes the function of the controls:

Subfolder	Controls	Function
Notification	Single User-Event Data	Control for initializing a LabVIEW™ event for **Single** ADS notifications.

Subfolder	Controls	Function
	Buffered User-Event Data	Control for initializing a LabVIEW™ event for **Buffered** ADS notifications.
TypeGenerator	CBase	Class object; base class of the TypeGenerator
	CBool	Class object; Boolean class of the TypeGenerator.
	CNumeric	Class object; numeric class of the TypeGenerator.
	CString	Class object; LabVIEW™ string class of the TypeGenerator.
	CArray	Class object; array class of the TypeGenerator.
	CTimestamp	Class object; LabVIEW™ timestamp class of the TypeGenerator.
	CCluster	Class object; LabVIEW™ cluster class of the TypeGenerator.

6.1 ADS DAQ

The ADS DAQ (**D**ata **Acq**uisition) VI is a LabVIEW™ Express VI for easy configuration of measuring tasks with TwinCAT 3, i.e. you can use this VI for read access to TwinCAT runtimes.

The user interface of the ADS DAQ VI guides you step by step through the configuration of your measuring task:

- Selection of the data points to be read (ADS symbols)
- Configuration of the read mode (ADS notification)
- Type generator configuration
- Configuration of start and end condition of the measuring task

The configuration window opens after placing the ADS DAQ instance in the LabVIEW™ block diagram or by double-clicking. The configurations can be made with the help of the selection windows described below. After the configuration is complete, the instance creates all the necessary resources for reading the data.

Save VI before using the ADS DAQ VI

The ADS DAQ VI stores the configuration of an instance in the path of the current project/VI. Therefore, it is necessary to save the project/VI beforehand.

Open ADS DAQ VI in an accelerated way

✓ The library must be precompiled.

1. Open the settings for "Mass Compile" in the LabVIEW™ settings at **Tools > Advanced**.
2. Select the folder of the TwinCAT 3 Interfaces for LabVIEW™ library, e.g. *C:\Program Files\ National Instruments\LabVIEW 2024\user.lib\Beckhoff-LabVIEW-Interface*.
3. Start "Mass Compile".



Selection	Description
[20] Handle	Handle to the ADS client
[25] Loaded Types	An array of LabVIEW™ Enums: • Describes which data types have been generated.
[26] Selection	LabVIEW™ cluster consisting of two elements: • SymbolName: The name of the ADS symbol • Notification Mode: LabVIEW™ Enum ◦ Single TypeResolved Queue: Reads only one sample as notification and adds the sample to the LabVIEW™ queue (only in LabVIEW™ 32-bit).

Selection	Description
	◦ Single TypeResolved Control: Reads only one sample as notification and writes the sample to the LabVIEW™ display element (only in LabVIEW™ 32-bit). ◦ Buffered TypeResolved Queue: Reads a number of samples as described by **LVBufferSize**, and adds the samples to the LabVIEW™ queue. ◦ Buffered TypeResolved Control: Reads a number of samples as described by **LVBufferSize**, and writes the samples to the LabVIEW™ display element.

Number of symbols

Currently, the ADS DAQ block supports reading a maximum of 10 ADS symbols per instance. Use multiple instances or the upgrade ADS FlexDAQ [▶ 54] if you want to read more than 10 ADS symbols.

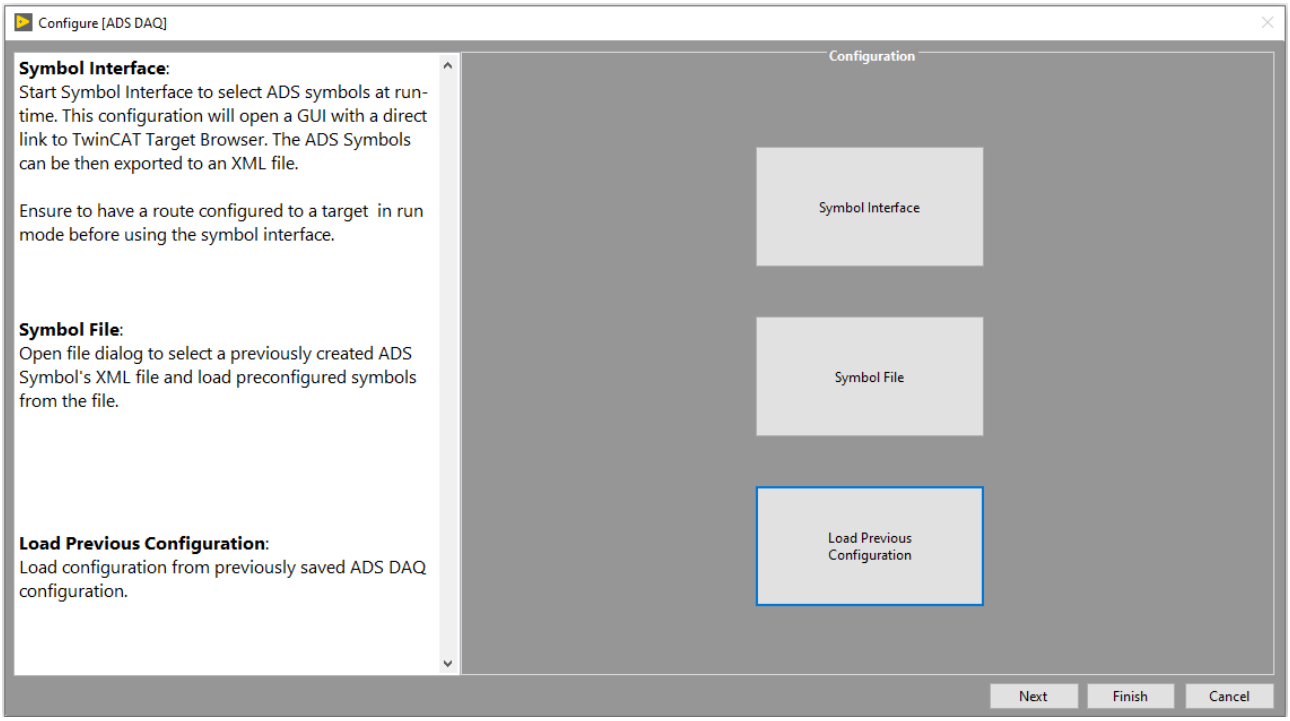
Generation of TwinCAT 3 data types

To support all notification modes, all generated types are converted to arrays. Notifications Buffered as well as Notifications Single are supported in the same way.

Symbol selection window

This window is used to select the ADS symbols to be read with the ADS DAQ instance. The window offers three different ways to select the ADS symbols:

1. Symbol Interface [▶ 65]**:** Opens an additional graphical user interface for browsing ADS symbols.
   - Browse into the connected targets.
   - Select the desired ADS symbols and drag and drop them into the right field.
   - Optionally, in the right field you can export the selected ADS symbols as a list and save them as an XML file.
2. **Symbol File:** Opens a LabVIEW™ file dialog box to read in the ADS symbols using an XML file exported from the Symbol Interface.
   - Reads only the symbols, but does not include all other settings of the ADS DAQ VI from the other configuration views.
3. **Load Previous Configuration:** Loads the last configuration (if existing) with which the ADS DAQ instance has already been configured.
   - The configuration always opens as an empty configuration. If you want to adjust the existing configuration, select **Load Previous Configuration**.



After selecting the ADS symbols, select **Next**.

Click **Finish** in the dialog at any time to save and close the configuration.

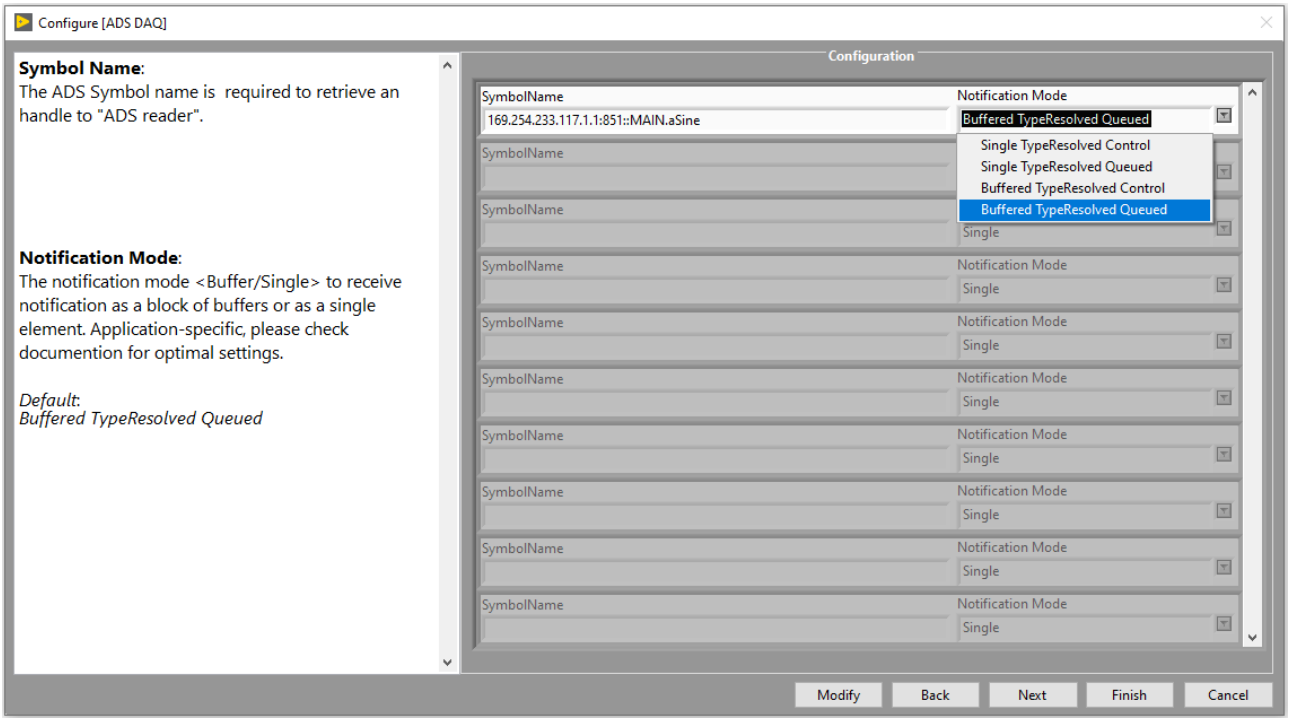
Notification selection window

In this window the notification mode is specified for individual selected ADS symbols. The Notification Mode describes on one side the way of reading (*single/buffered*) and on the other side the display of the read data in LabVIEW™ (*control/queue*). Default setting is *Buffered TypeResolved Queued*.

The following table describes the different properties:

	Single TypeResolved	Buffered TypeResolved
Control (Only recommended if the data do not need to be processed or saved)	The individual notifications are displayed directly in a LabVIEW™ control.	Notifications are first written to an intermediate buffer layer and passed into the LabVIEW™ process when the buffer is filled. The data are displayed in a LabVIEW™ control.
Queued (Recommended if data are to be processed, saved, ...)	The individual notifications are inserted directly into a LabVIEW™ queue.	Notifications are first written to an intermediate buffer layer and passed into the LabVIEW™ process when the buffer is filled. The data are inserted into a LabVIEW™ queue.

For more information on single and buffered mode, see Communication modes [▶ 23] and Event driven reading [▶ 31].

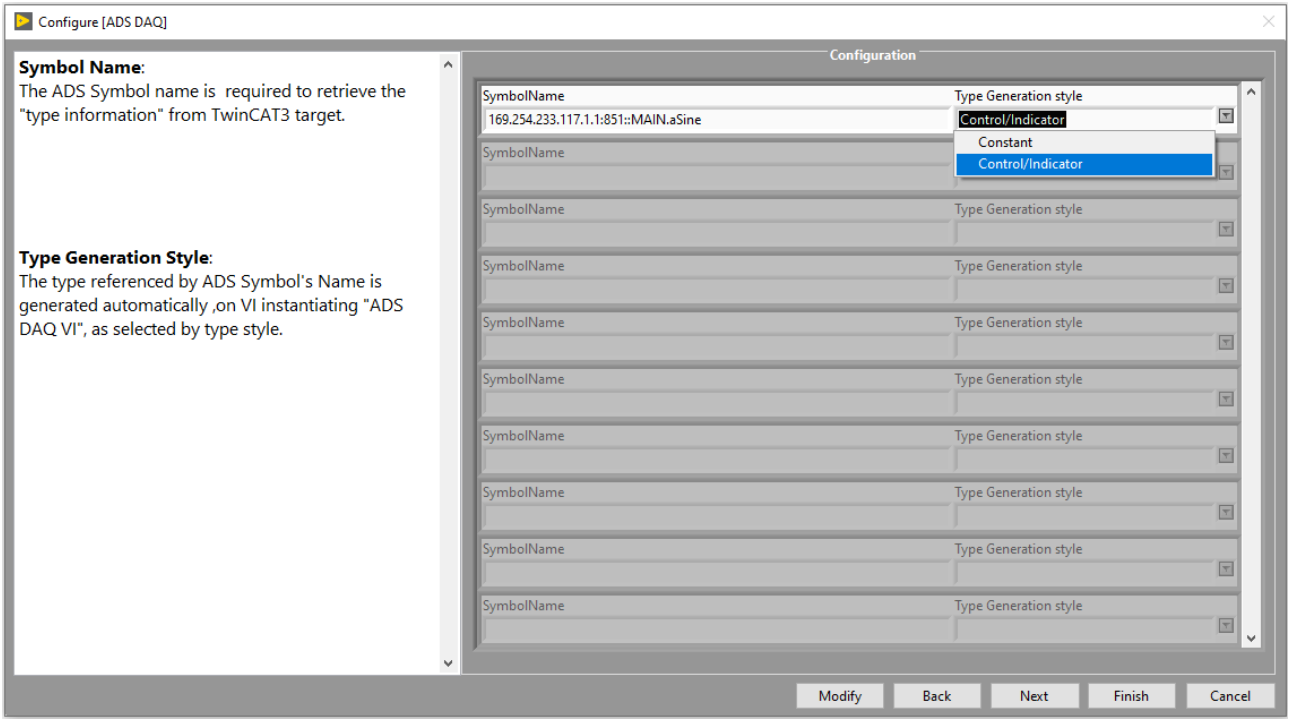


Click **Modify** to open the Symbol Interface and make changes to the ADS symbols to be read.

Click **Next** to go to the next configuration page.

Type generation selection window

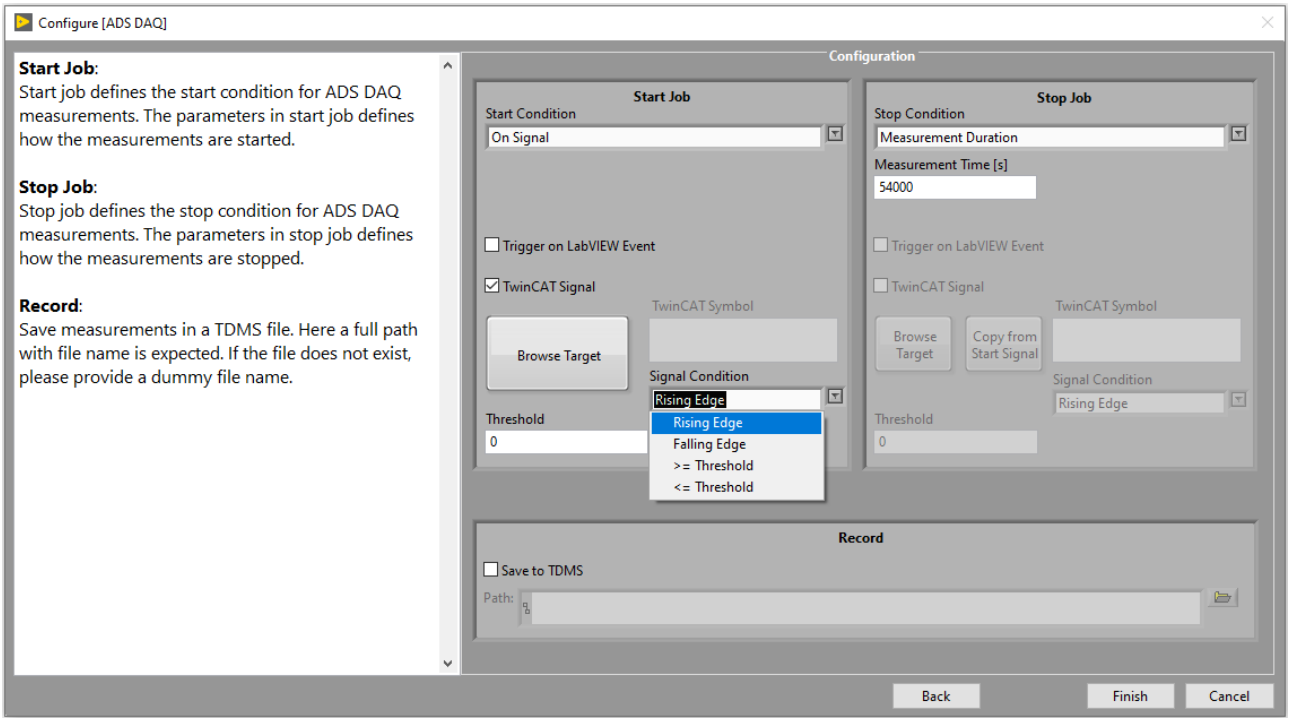
In this window you can select whether the respective data type of the ADS symbol to be read is to be generated as a LabVIEW™ constant or as a control/display element. Both constants and control/display elements are automatically generated in the block diagram of the VI, where the ADS DAQ instance is also located. Default setting is *Control/Indicator*.



Measurement Job configurator

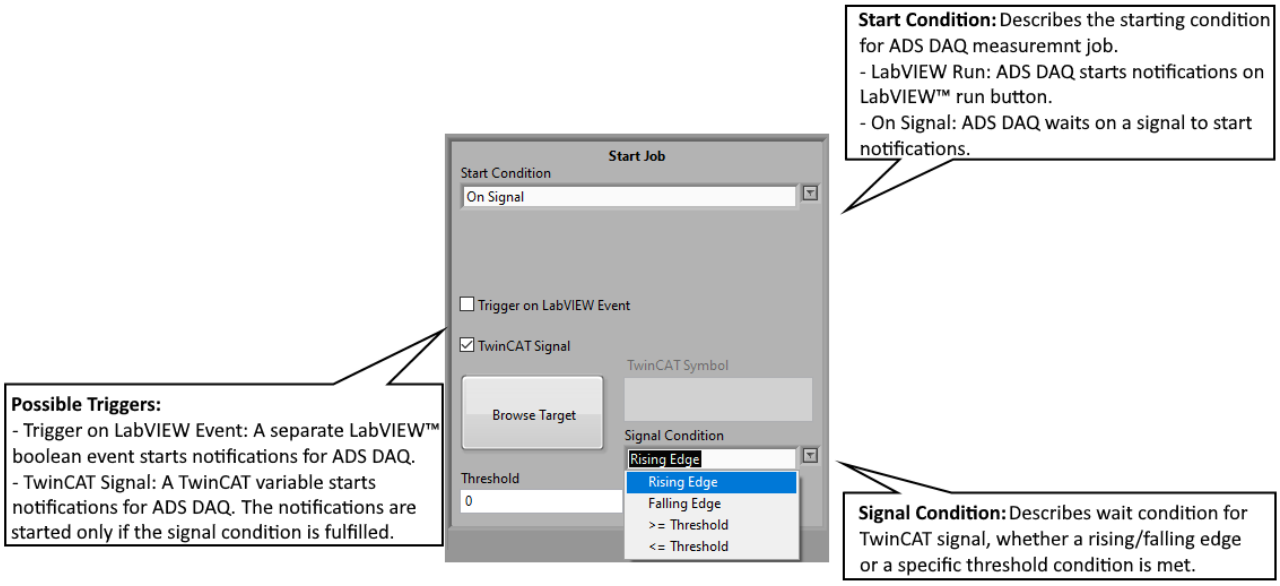
In this window the ADS DAQ instance can be configured with additional start/stop/record conditions.

Default setting is at *Start: LabVIEW run* and at *Stop: LabVIEW Abort, no Record*.



Start Job

The control *Start Job* configures the start of the reading process. The parameter *Start Condition* describes the way of starting.



LabVIEW™ Run: The ADS notifications are both automatically registered and started after starting the VI.

- **On Signal:** The ADS notifications are not started automatically in this case. The ADS DAQ instance waits for a specific trigger from LabVIEW™ or TwinCAT.
  - **Trigger on LabVIEW™ Event:** With this selection the ADS DAQ instance generates a separate event logic to start the ADS notifications from LabVIEW™.

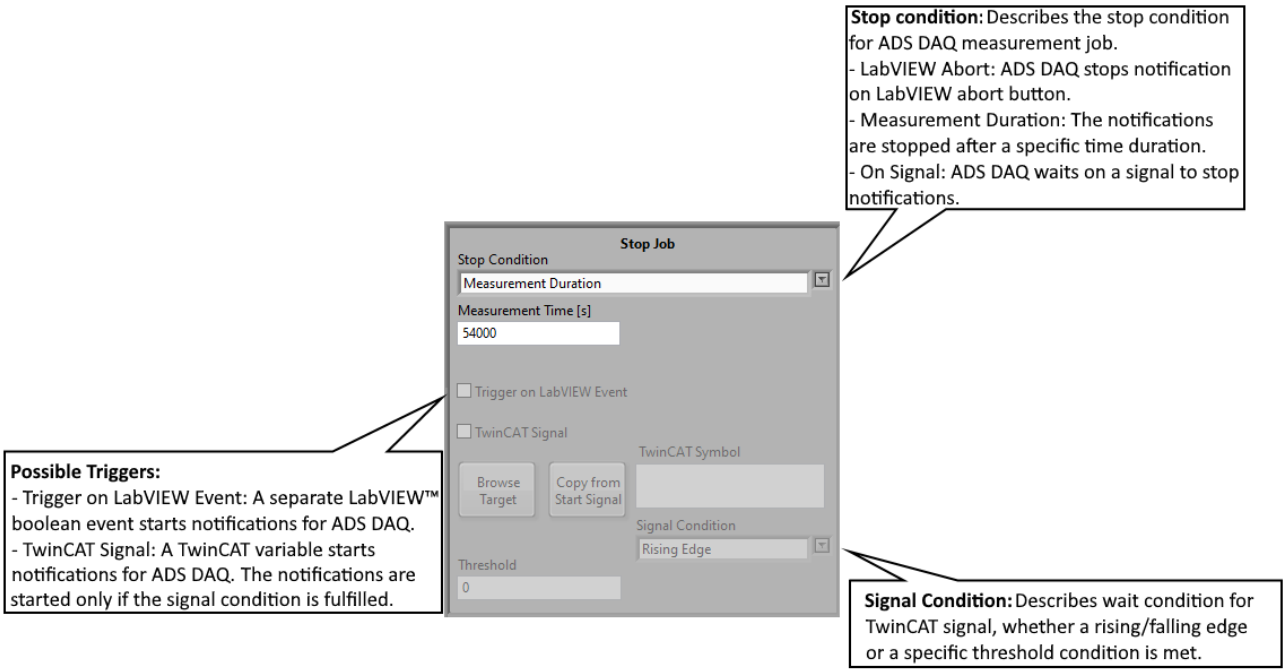
- **TwinCAT Signal:** The DAQ instance is triggered by an (additional) TwinCAT signal and therefore does not start the ADS notifications automatically. The button *Browse Target* opens the Symbol Interface. Drag an ADS symbol to the right onto the "Read" area. All primary data types are allowed. Furthermore, a signal condition has to be defined on which the ADS DAQ VI is to be triggered.

The table below describes the transition (*Last value* ➔ *New value*) of different TwinCAT types with the definition of Rising and Falling Edge used here.

TwinCAT type	Rising Edge		Falling Edge	
	Last value	New value	Last value	New value
Boolean type	0	1	1	0
Numeric integer type (i8, i16, i32, i64, u8, u16, u32, u64)	< Threshold	= Threshold	> Threshold	= Threshold
Numeric rational type (float32, float64) Epsilon: 1.0 e^{-7}	< Threshold	= Threshold	> Threshold	= Threshold

Stop Job

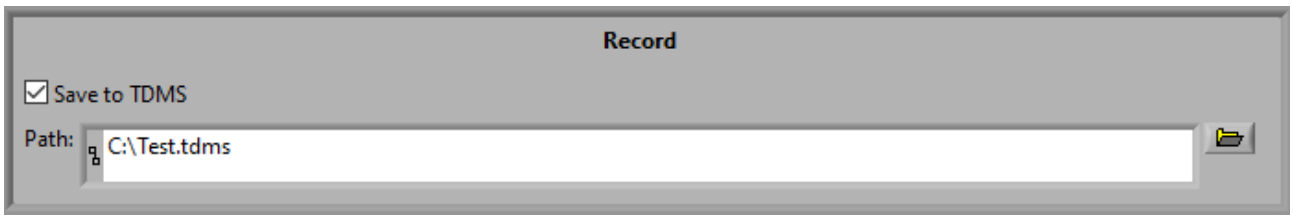
The control *Stop Job* configures the stopping of ADS notifications, i.e. the stopping of the reading process. The parameter *Stop Condition* describes the way of stopping.



- **LabVIEW™ Abort:** The ADS notifications are automatically unregistered as well as stopped after stopping the VI.
- **Measurement Duration:** The ADS notifications stop automatically after the measurement time has elapsed.
- **On Signal:** This option behaves identically to the selection *On Signal* in **Start Job** (see above)**.**
  - Click **Copy from Start Signal** to copy the start condition from Start to Stop Job (applies only to the TwinCAT signal).

Record Job

The recording of ADS DAQ measured data can be configured with the control *Record Job*. After configuration and clicking **Finish**, the ADS DAQ instance generates an additional Block To TDMS to save the received data as a LabVIEW™ TDMS file under the specified file name and path.

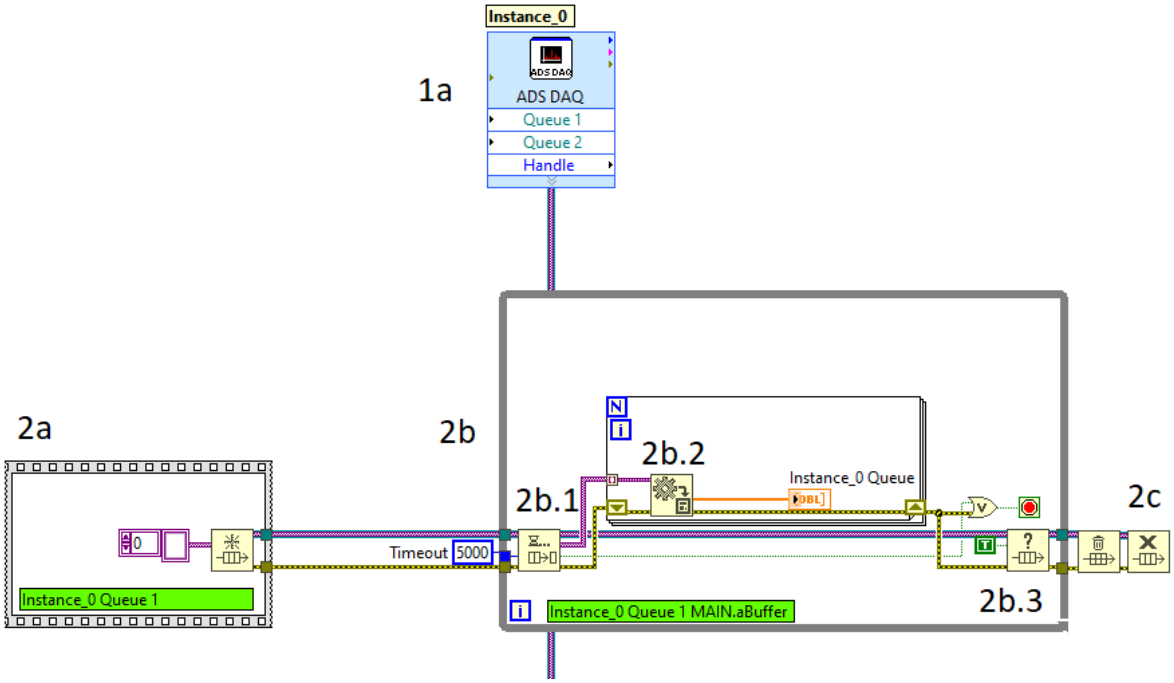


Click **Finish** to save the settings. The automatic code generation for your configuration starts.

Automatically generated code in the block diagram

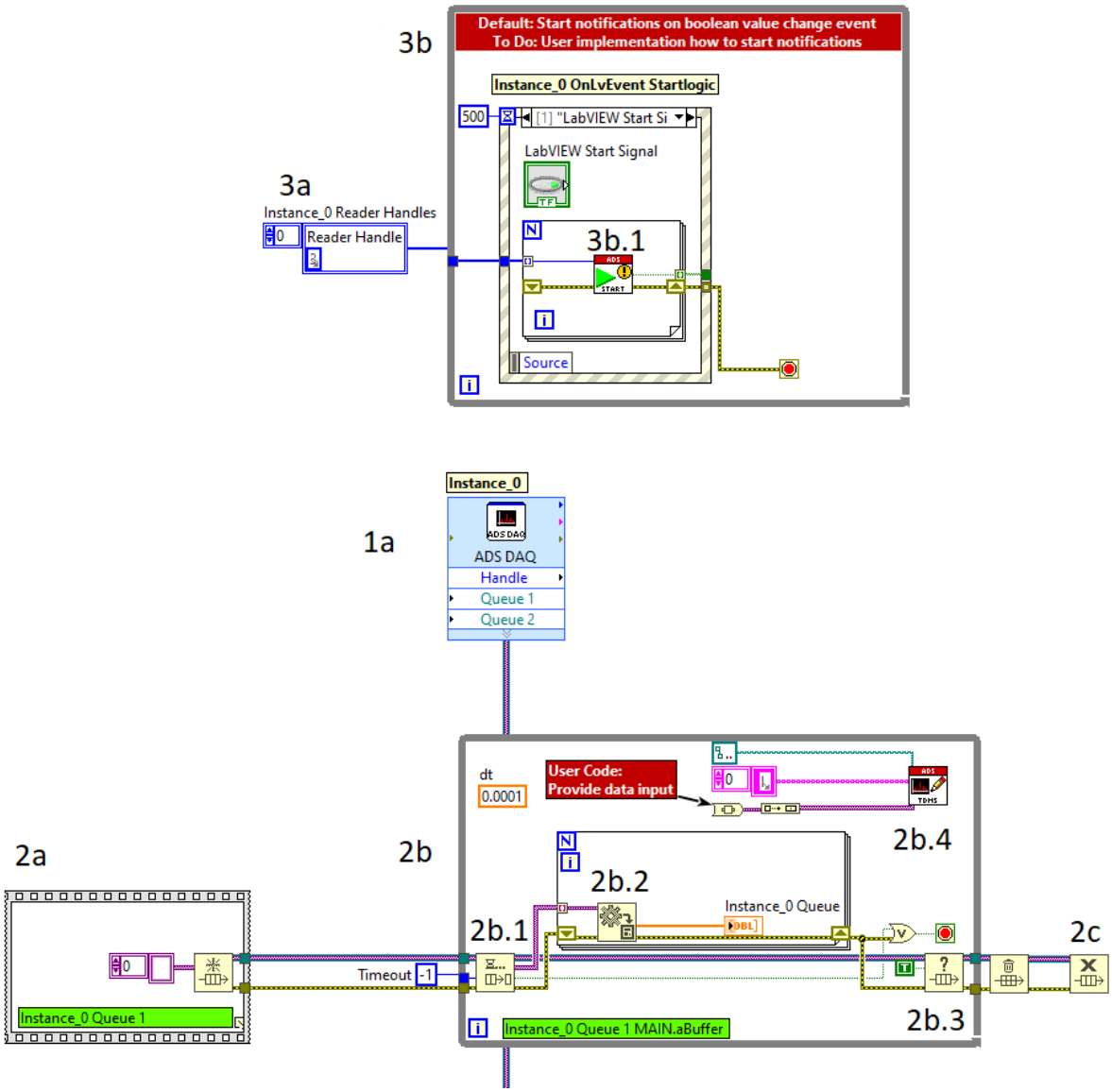
In the following, two variants of the automatically generated code are explained as examples.

In the first example, the ADS DAQ VI is generated with **default settings**, i.e. Buffered Type Resolved Queue, Control/Indicator, Start with LabVIEW™ Run, Stop with LabVIEW™ Stop. An ADS symbol MAIN.aBuffer is read.



- **1a:** A handle from Queue 1 goes from the instance of ADS DAQ VI to the underlying queue blocks. The queue of the ADS DAQ VI already contains Type-Resolved data packets. Each data packet is of size *LvBufferSize* (cf. settings in the Symbol Properties [▶ 23]), as **Buffered Type Resolved Queue** has been configured.
- **2a:** Initialization of the queue with a LabVIEW™ Variant Array as data type.
- **2b:** While loop with start condition "LabVIEW™ Run" and end condition "Timeout or error occurred".
- **2b.1:** Dequeue element: Waits for received data packets with specified timeout of 5 seconds (customizable).
- **2b.2:** For loop: Takes the individual elements of the Variant array and converts the data type to the corresponding LabVIEW™ data type. Here, for example, the user can directly access the converted type and continue working with it.
- **2b.3:** Checks the current state of the queue, e.g. whether the queue is growing steadily. If this is the case, the data will arrive from ADS DAQ VI faster than it can be converted to the LabVIEW™ data type. Optionally, the user can insert a display element or similar here to monitor the queue.
- **2c:** Releases the queue memory. After that the queue is released from the LV memory. Optionally, after these steps, the user can release the ADS client handle from memory.

In the second example, the start condition is changed from LabVIEW™-Run to Trigger on LabVIEW™-Event and the *recording* to a TDMS file is enabled.

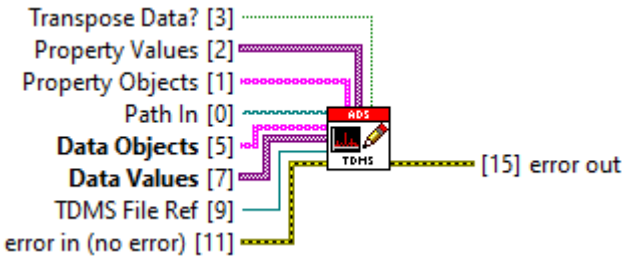


- The blocks already described in the first example remain identical in their function. Only the timeout was set to -1 in this case (wait infinitely), because the start of the measurement is triggered by LabVIEW™.
- **2b.4:** The TDMS block is automatically generated based on the setting that a *recording* should take place. The user must manually link the data to be saved to the To Variant block. If the received data correspond to a LabVIEW™ signal, they can be converted to a LabVIEW™ waveform before the To Variant. The user can use the automatically generated constant *dt* which corresponds to the time distance between two data points (sampling period duration).
- **3a:** The ADS reader handles are regenerated with each new configuration of the ADS DAQ VI. The obsolete reader handles are automatically released from memory.
- **3b:** Logic is created that generates a trigger-event when a Boolean value change occurs. This is used to start the ADS notifications.
- **3b.1:** Starts the ADS Notification for the registered ADS symbols.

Notes on the "To TDMS" in the block diagram

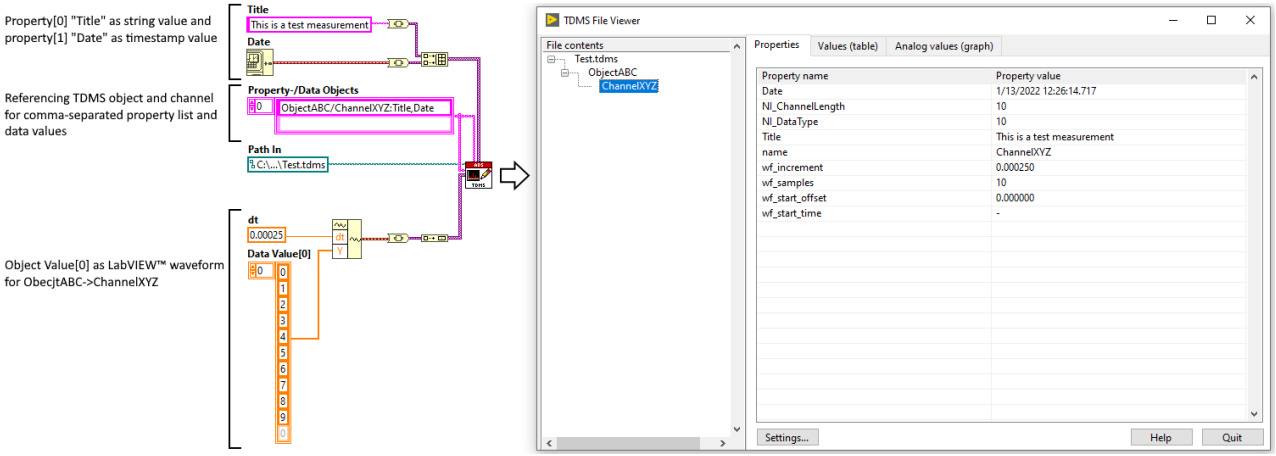
The *To TDMS* block allows you to save ADS DAQ instance data to a LabVIEW™ TDMS file (the NI TDMS file format). This block is automatically generated by the ADS DAQ instance, but only if *Queued* has been selected as the notification mode, cf. Notification selection window [▶ 47].

The TDMS file format provides TDMS objects and TDMS channels to hierarchically arrange data or properties (name, date, ...) among TDMS objects. The inputs **Property-/Data Objects** can be used to reference or classify both properties and data. The inputs **Property-/Data Values** help to enter measured values under TDMS objects and channels.



Selection	Description
[0] Path In	The path to the TDMS file
[1] Property Objects	An array of a LabVIEW™ string: • References a specific object (channel) for properties with the format described below: 1. Object and channels are separated with slash "/". 2. The list of properties is entered by a comma-separated string (","). 3. Items 1 and 2 are separated by a colon (":"). **Sample:** Property Object[0]=ObjectABC/ChannelXYZ:Name,Author,Date TDMS Object=ObjectABC TDMS Channel=ChannelXYZ Property[0]=name (string) Property[1]=author (string) Property[2]=date (timestamp)
[2] Property Values	An array of LabVIEW™ variants: • Describes the values of properties.
[3] Transpose Data?	*Flag* to transpose the entered 2D of 3D arrays.
[5] Data Objects	An array of a LabVIEW™ string: • References a specific object (channel) for data. The following format is used for referencing: 1. Object and channels are separated with slash ("/"). **Sample:** Data Object[0]=ObjectABC/ChannelXYZ
[7] Data Values	An array of LabVIEW™ variants with the following supported LabVIEW™ types: • LabVIEW™ waveform • 1D, 2D, 3D array (2D, 3D array are internally converted to 1D): ◦ Boolean type ◦ Integer numeric types (i8 ... i64,u8 ... u64) ◦ Non-integer numeric types (float32, float64) ◦ LabVIEW™ string

The following graphic shows an example of the *To TDMS* block in use.



6.2 ADS FlexDAQ

The ADS FlexDAQ (**Flex**ible **D**ata **Acq**uisition) VI is a further development of the ADS DAQ [▶ 45] VI. As with the ADS DAQ, this is a LabVIEW™ Express VI that simplifies the configuration of measuring tasks with TwinCAT 3 and provides read access to the TwinCAT runtime.

The ADS FlexDAQ VI also offers the following additional options:

- Any number of ADS symbols can be read per instance.
- A unique Loop-ID can be assigned to the symbols. Symbols with the same Loop-ID are read in the same while loop.
- Each symbol can be assigned a corresponding TDMS flag to store the data to be read in a TDMS file.

The user interface guides you step by step through the configuration of your measuring task:

- Selection of the data points to be read (ADS symbols)
- Configuration of the read mode (ADS notification)
- Configuration of the Loop-IDs
- Configuring the storage of measurement data in a TDMS file
- Configuration of start and end condition of the measuring task

The configuration window opens after placing the ADS FlexDAQ instance in the LabVIEW™ block diagram or by double-clicking. The configurations can be made with the help of the selection windows described below. After the configuration is complete, the instance creates all the necessary resources for reading the data.

Save VI before using the ADS FlexDAQ VI

The ADS FlexDAQ VI stores the configuration of an instance in the path of the current project. Therefore, it is necessary to save the project beforehand.

Open ADS FlexDAQ VI in an accelerated way

✓ The library must be precompiled.

1. Open the settings for "Mass Compile" in the LabVIEW™ settings at **Tools > Advanced**.
2. Select the folder of the TwinCAT 3 Interfaces for LabVIEW™ library, e.g. *C:\Program Files\ National Instruments\LabVIEW 2023\user.lib\Beckhoff-LabVIEW-Interface* .
3. Start "Mass Compile".



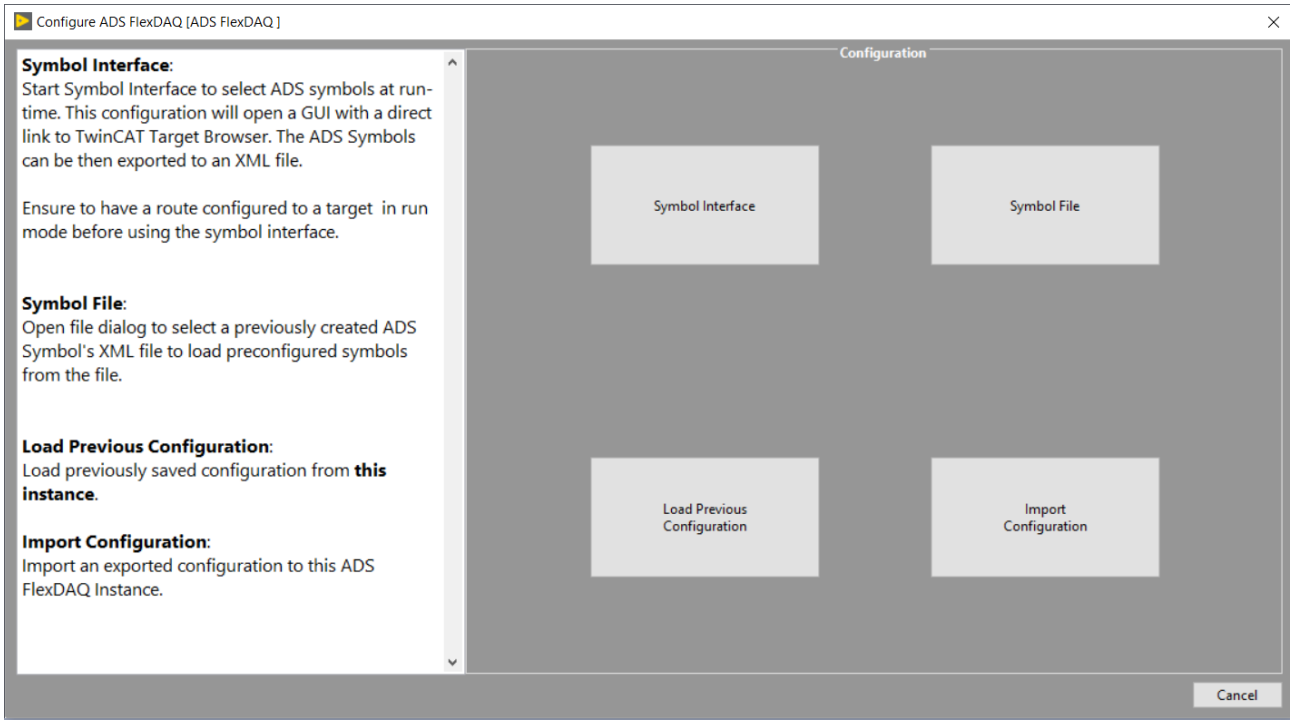
Input/output	Meaning
[1] Reader handles	Handles on the reading symbols
[20] Handle	Handle to the ADS client
[26] Selection	LabVIEW™ cluster consisting of two elements: • SymbolName: The name of the ADS symbol • Notification Mode: LabVIEW™ Enum ◦ Single TypeResolved Queue: Reads only one sample as notification and adds the sample to the LabVIEW™ queue (only in LabVIEW™ 32-bit). ◦ Single TypeResolved Control: Reads only one sample as notification and writes the sample to the LabVIEW™ display element (only in LabVIEW™ 32-bit). ◦ Buffered TypeResolved Queue: Reads a number of samples as described by **LVBufferSize**, and adds the samples to the LabVIEW™ queue. ◦ Buffered TypeResolved Control: Reads a number of samples as described by **LVBufferSize**, and writes the samples to the LabVIEW™ display element.

TwinCAT 3 data type generation

To support all notification modes, all generated types are converted to arrays. Notifications Buffered as well as Notifications Single are supported in the same way.

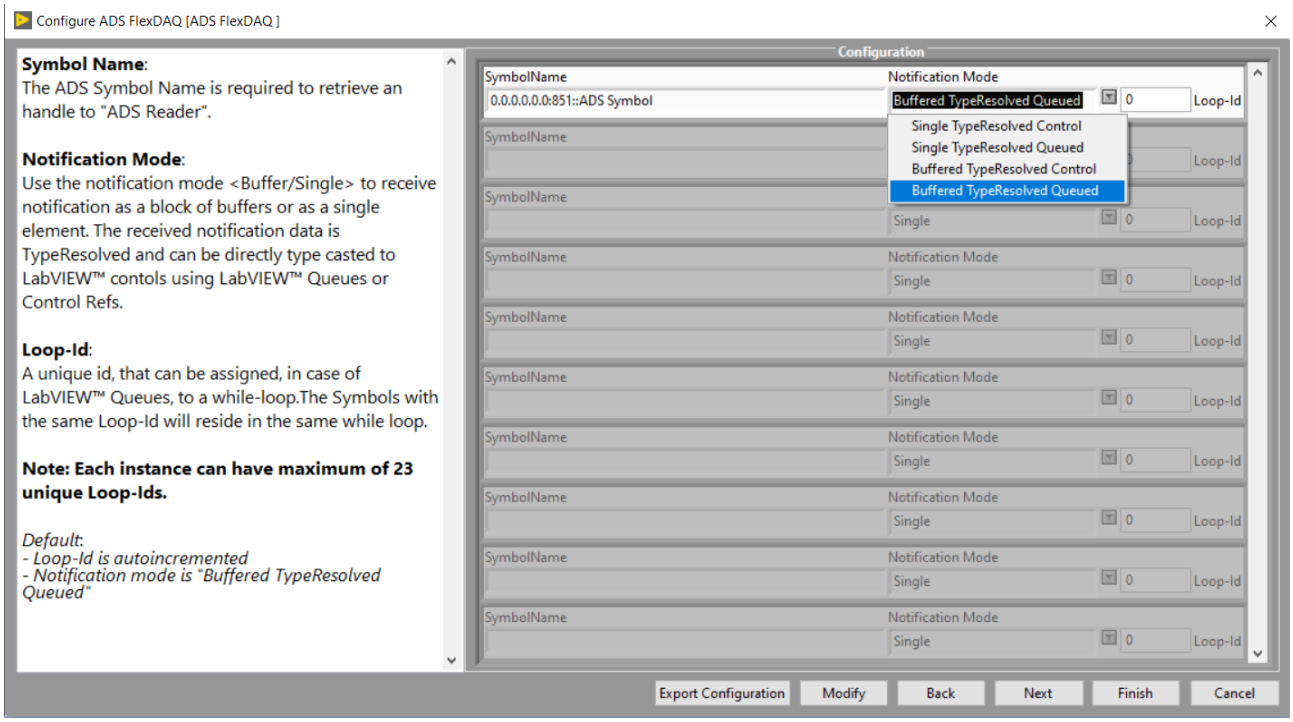
Symbol selection window

Just as with ADS DAQ [▶ 45], the Symbol selection window is used to select the ADS symbols to be read with the ADS Flex DAQ instance. In addition, there is an option **Import Configuration,** which can be used to import a saved configuration**.** For this purpose, a LabVIEW™ file dialog box opens, in which a path to an exported configuration can be selected.



Notification and Loop-ID selection window

In this window the Notification Mode and the Loop-ID are selected. The possible notification modes can be found in section Notification selection window [▶ 47]. The Loop-ID is used to distribute the reading of the symbols to while loops. With the ADS Flex DAQ it is possible to assign the same Loop-ID to several symbols. These are then processed in the same while loop. This reduces the number of loops.

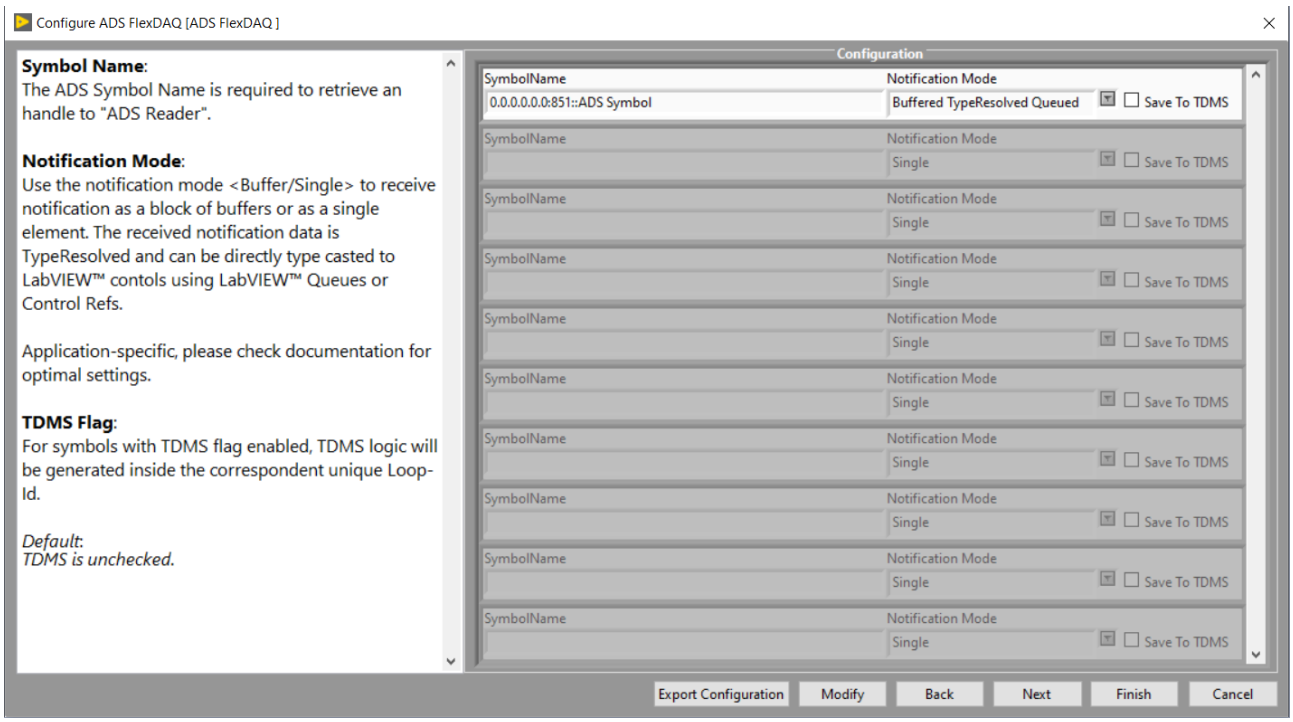


Maximum Loop-IDs

The number of inputs is limited to 23 for the ADS Flex DAQ instance. Consequently, a maximum of 23 different Loop-IDs can also be assigned.

Save measured data

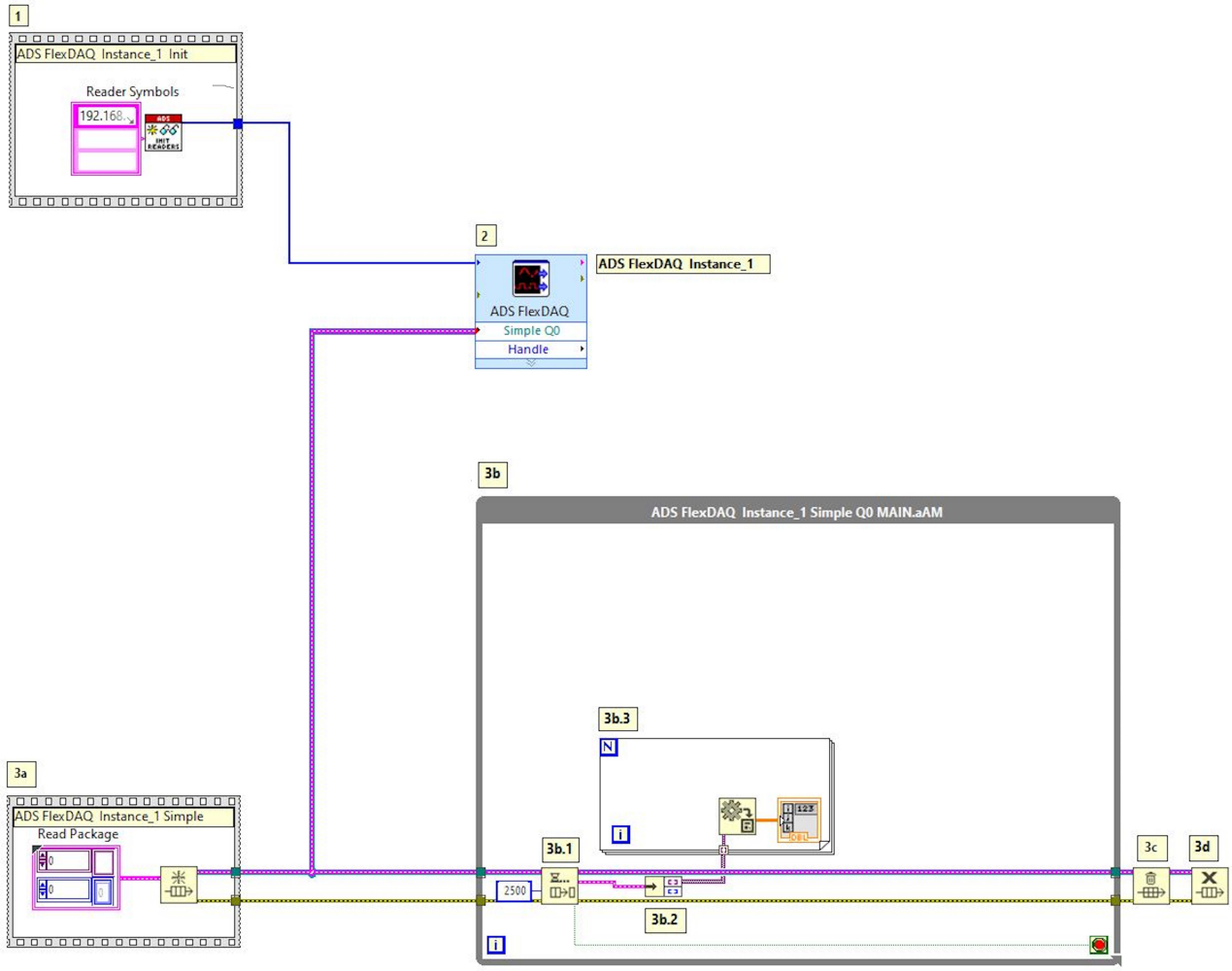
For each symbol there is the option "Save To TDMS", which saves the respective symbol in a TDMS file when reading from TwinCAT. The block Notes on the "To TDMS" in the block diagram [▶ 52] is used for this purpose.



Automatically generated code in the block diagram

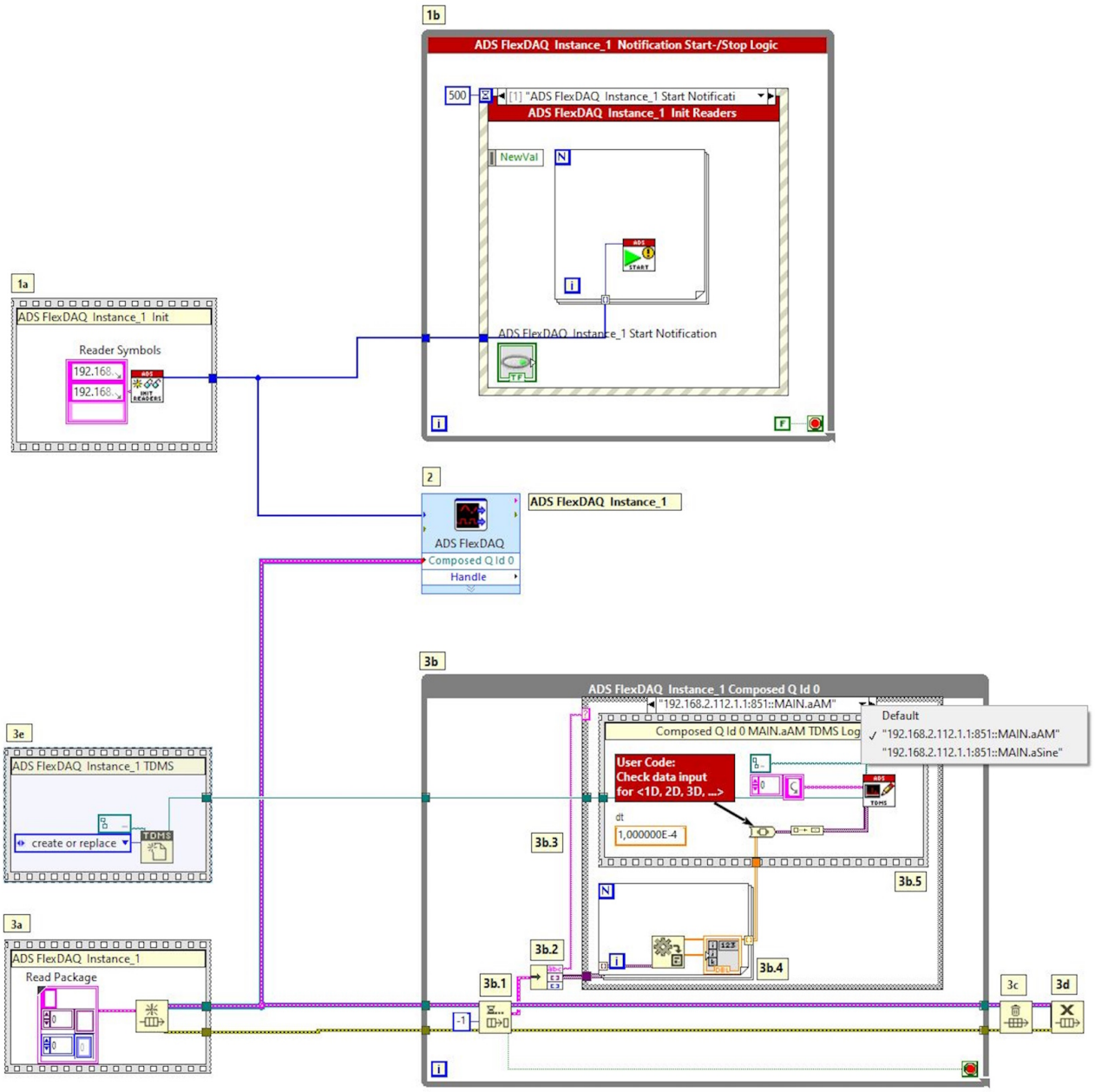
In the following, two variants of the automatically generated code are explained as examples.

In the first example, the ADS FlexDAQ VI is generated with **default settings**, i.e. Buffered Type Resolved Queue, no storage of measured data, Start with LabVIEW™ Run, Stop with LabVIEW™ Stop. An ADS symbol MAIN.aAM is read.



- **1:** ADS Reader handles are initialized for the selected ADS symbols. The ADS reader handles are regenerated with each new configuration of the ADS FlexDAQ VI. The obsolete reader handles are automatically released from memory.
- **2:** A handle from Simpe Queue1 goes from the instance of ADS FlexDAQ to the underlying queue blocks. The queue of the ADS FlexDAQ VI already contains Type-Resolved data packets. Each data packet is of size *LvBufferSize* (cf. settings in the Communication modes [▶ 23]), as **Buffered Type Resolved Queue** has been configured.
- **3a:** The queue is initialized with a LabVIEW™ Cluster, **Read Package**. The cluster contains a data array as a variant (TypeResolved ADS notification data) and a numeric array (notification timestamps) as elements.
- **3b:** While loop with start condition "LabVIEW™ Run" and without end condition.
- **3b.1:** Dequeue element: Waits for received data packets for an infinite time (customizable).
- **3b.2**: Unbundle element: Splits the cluster into data and ADS timestamps.
- **3b.3:** For loop: Takes the individual elements of the Variant array and converts the data type to the corresponding LabVIEW™ data type. Here, for example, the user can directly access the converted type and continue working with it.
- **3c and 3d:** The queue memory is released. After that the queue is released from the LabVIEW™ memory. Optionally, after these steps, the user can release the ADS client handle from memory.

In the second example, the start condition is changed from LabVIEW™-Run to "Trigger on LabVIEW™ Event" and the data storage to a TDMS file is enabled. Two symbols are read: MAIN.aAM and MAIN.aSine. These are assigned the same Loop-ID. Thus, both symbols are read in the same while loop.



The blocks **1a, 2, 3b, 3b.2, 3b.4, 3c** and **3d** already described in the first example remain identical in their function.

- **1b:** Logic is created that generates a trigger-event when a Boolean value change occurs. This is used to start the ADS notifications.
- **3a:** The queue is initialized with a LabVIEW™ cluster as data type with the member variables:
  - SymbolName: Identifies the read data packet
  - Data: TypeResolved ADS notification data packet
  - Timestamp: ADS notification timestamps
- **3b.2:** The read data packet is unpacked and distributed to the corresponding case structure.
- **3b.3:** Case structure that passes the ADS data for the unpacked symbol data to LabVIEW™. In the example, the case structure manages two ADS symbols MAIN.aAM and MAIN.aSine.
- **3b.5:** The TDMS block is generated automatically based on the setting that the data should be saved. The data to be saved is automatically linked to the To Variant block. If the received data correspond to a LabVIEW™ signal, they can be converted to a LabVIEW™ waveform before the To Variant. The user can use the automatically generated constant *dt* which corresponds to the time distance between two data points (sampling period duration).
- **3e:** Initialization of the handle to the TDMS file.

6.3 ADS Write Assistant

Like the ADS FlexDAQ [▶ 54], the ADS Write Assistant VI is a LabVIEW™ Express VI that simplifies the configuration of transfer tasks. The ADS Write Assistant VI can be used to write data from LabVIEW™ to TwinCAT 3.

The user interface of the ADS Write Assistant VI guides you step by step through the configuration of your transfer task:

- Selection of the data points to be written (ADS symbols)
- Configuration of the Loop-IDs
- Configuration of start and end condition of the write operation using the transfer job selection window

The configuration window opens after placing the ADS Write Assistant instance in the LabVIEW™ block diagram or by double-clicking. The configurations can be made with the help of the selection windows described below. After the configuration is complete, the instance creates all the necessary resources for writing the data.

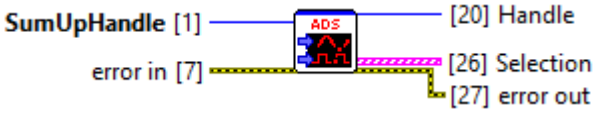
Save VI before using the ADS Write Assistant VI

The ADS Write Assistant VI saves the instance configuration in the path of the current project. Therefore it is necessary that the project has been saved before.

Open ADS Write Assistant VI in an accelerated way

✓ The library must be precompiled.

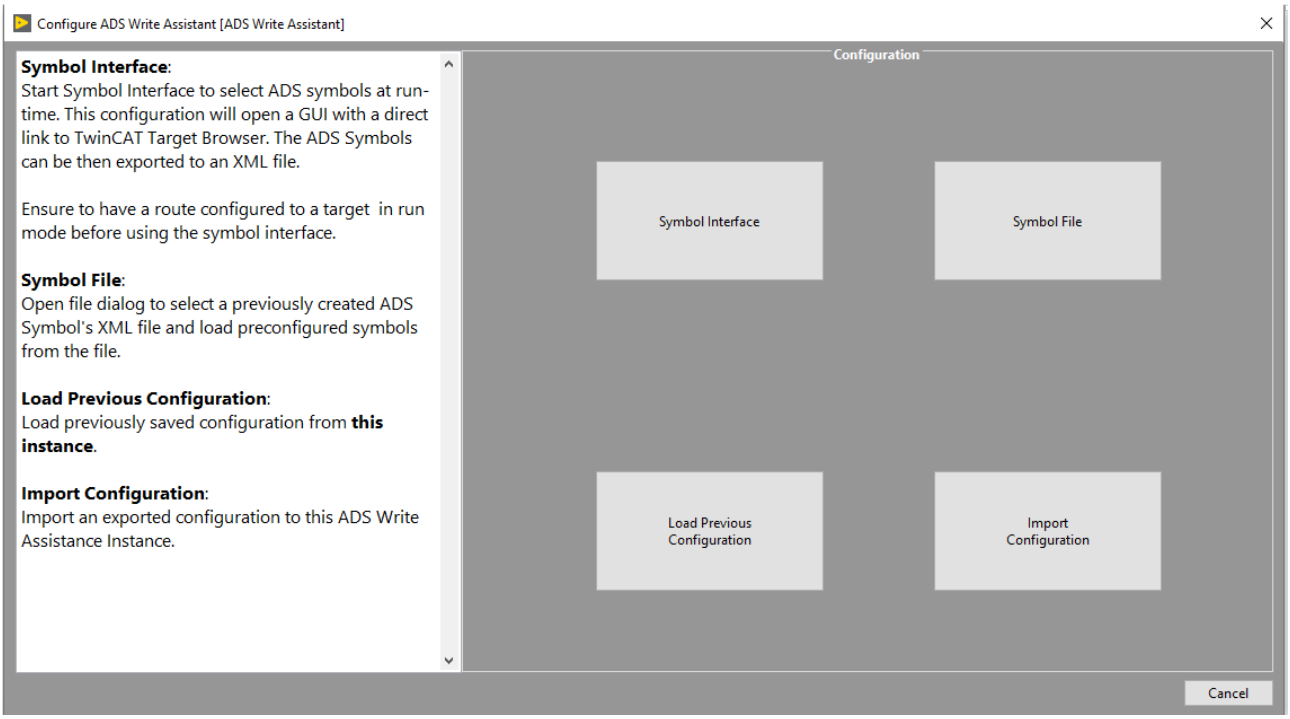
1. Open the settings for "Mass Compile" in the LabVIEW™ settings at **Tools > Advanced**.
2. Select the folder of the TwinCAT 3 Interfaces for LabVIEW™ library, e.g. *C:\Program Files\ National Instruments\LabVIEW 2024\user.lib\Beckhoff-LabVIEW-Interface*.
3. Start "Mass Compile".



Input/output	Meaning
[1] SumUp handle	Handle to the SumUp Writer
[20] Handle	Handle to the ADS client
[26] Selection	Control for Express VI

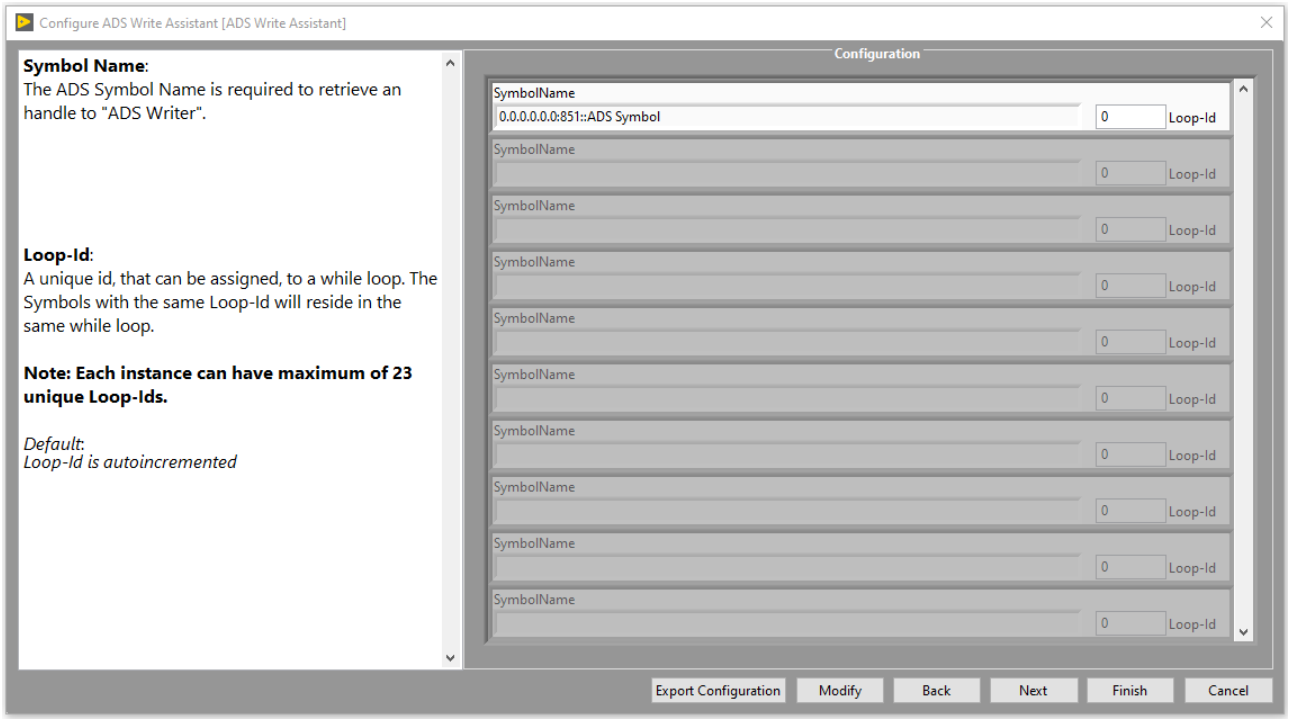
Symbol selection window

The symbol selection window at the ADS Write Assistant offers the same functionality as at ADS FlexDAQ [▶ 55]. Here one of the options can be selected to start a new configuration or to apply the settings of an existing configuration. If you start a new configuration via Symbol Interface, you will automatically move on to the next window.



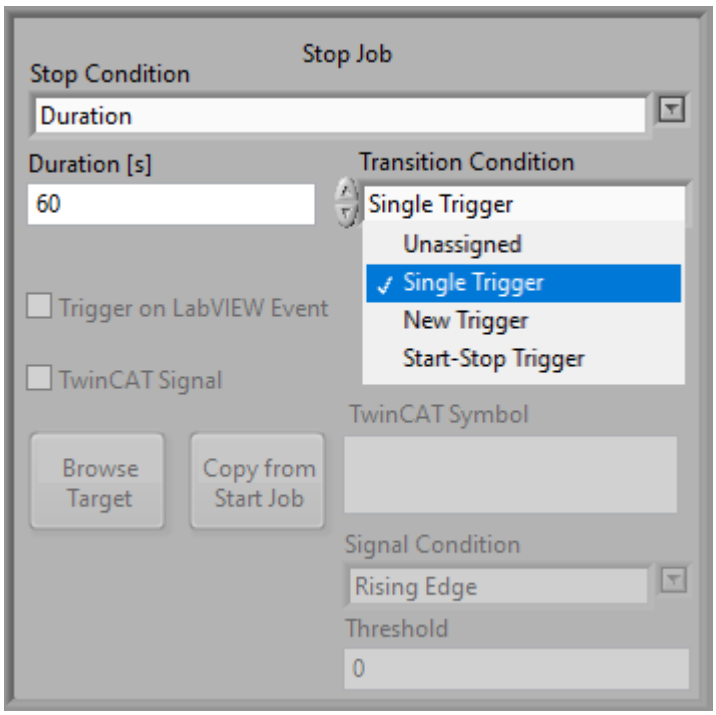
Loop-ID window

In this window, the symbols are assigned a unique Loop-ID. This distributes the writing of the symbols to different while loops. If two symbols have the same Loop-ID, both symbols have used the same loop. The number of Loop-IDs determines the number of generated while loops in the LabVIEW™ block diagram.



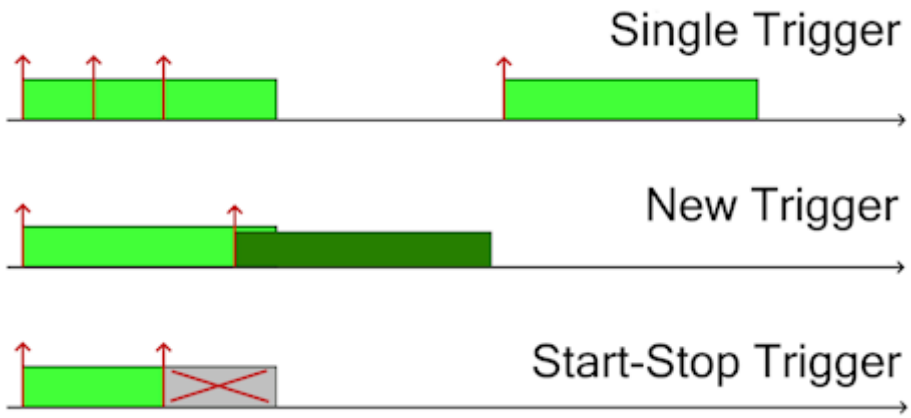
Write operation configurator (selection window)

In this window the start/stop condition for the data transport from LabVIEW™ to TwinCAT 3 can be configured. The configurator behaves identically to Measurement Job selection window [▶ 48] except for the stop condition.



With the control **Stop Job** you configure the stopping of the writing process. The parameter **Stop Condition** describes the stop condition.

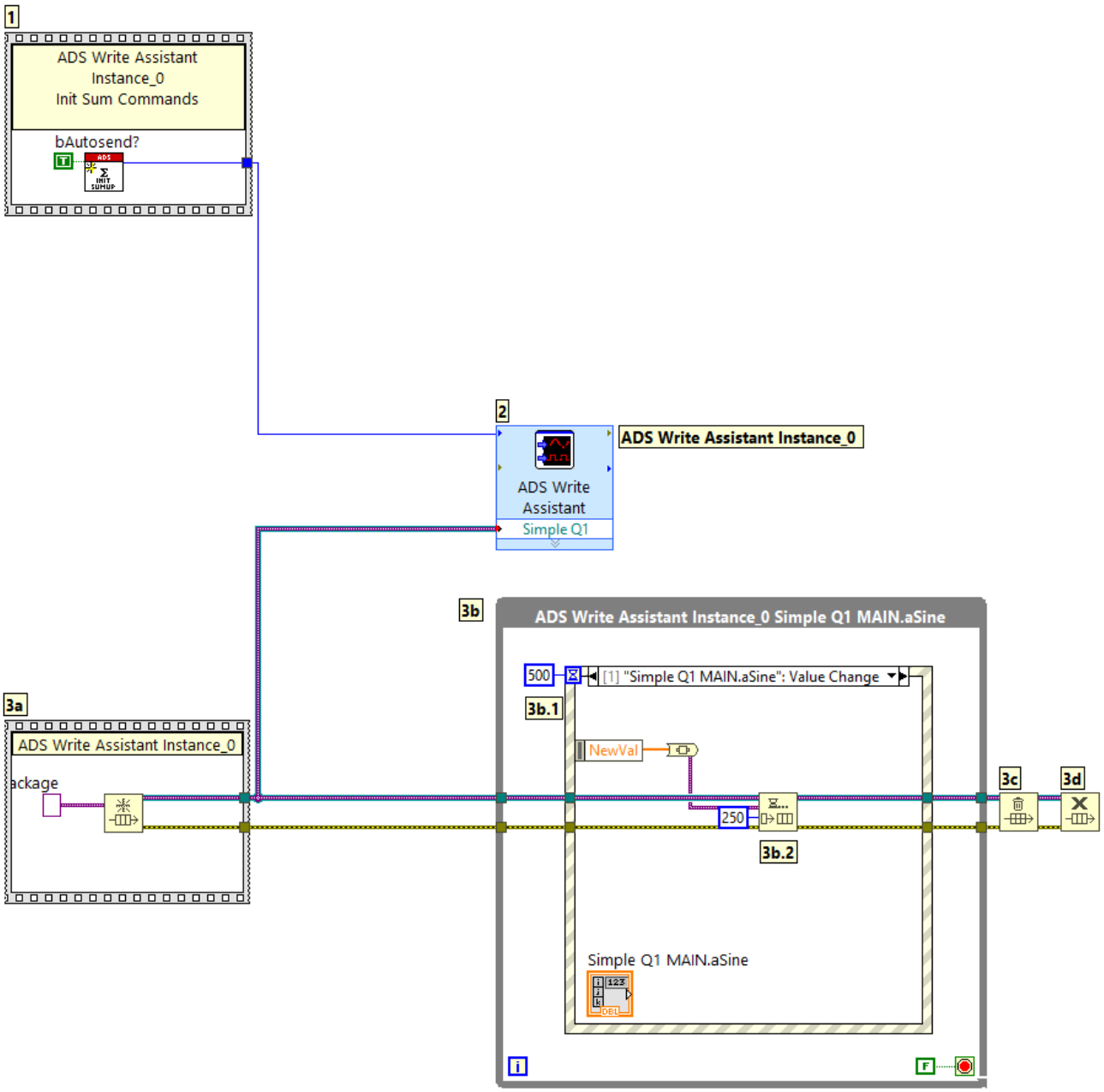
- The options **LabVIEW™ Abort** and **On Signal** are described in Measurement Job selection window [▶ 48].
- **Duration**: If Duration is selected as a stop condition, then the write operation is determined after a time. With this condition, the "Transition Condition" option is also available. This allows the write duration to be influenced by a trigger. The image below shows the different "Transition Condition" triggers.
  - Single Trigger: Starts a new write duration only if no other write operation is present.
  - New Trigger: Always starts a new write operation.
  - Start-Stop Trigger: Starts a write operation if no other is present or, on the contrary, stops an existing write operation.



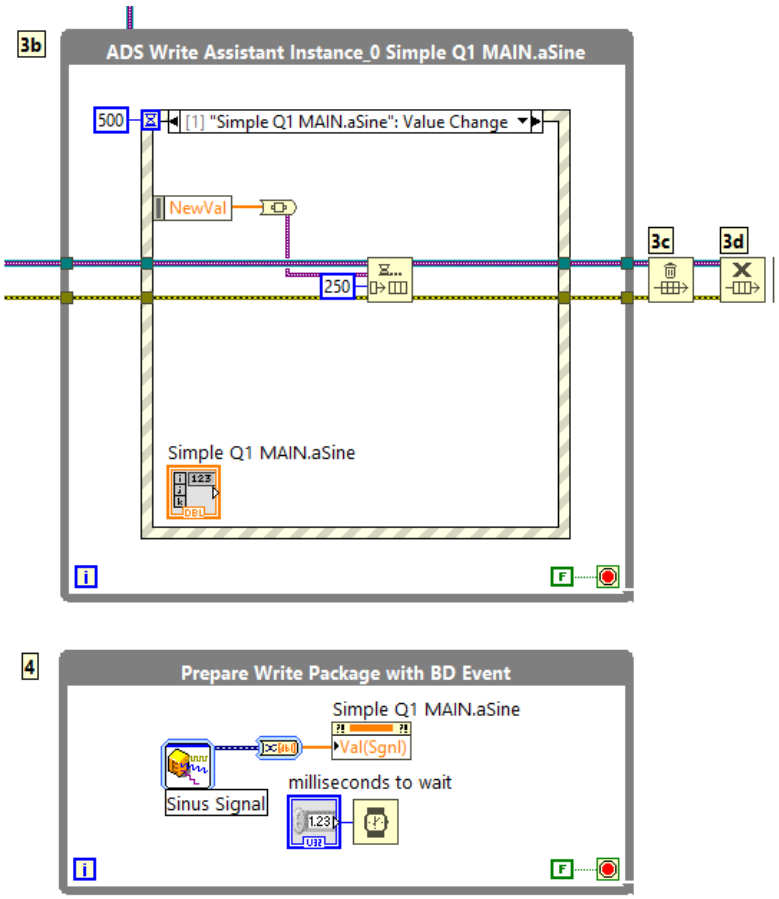
Automatically generated code in the block diagram

In the following, two variants of the automatically generated code are explained as examples.

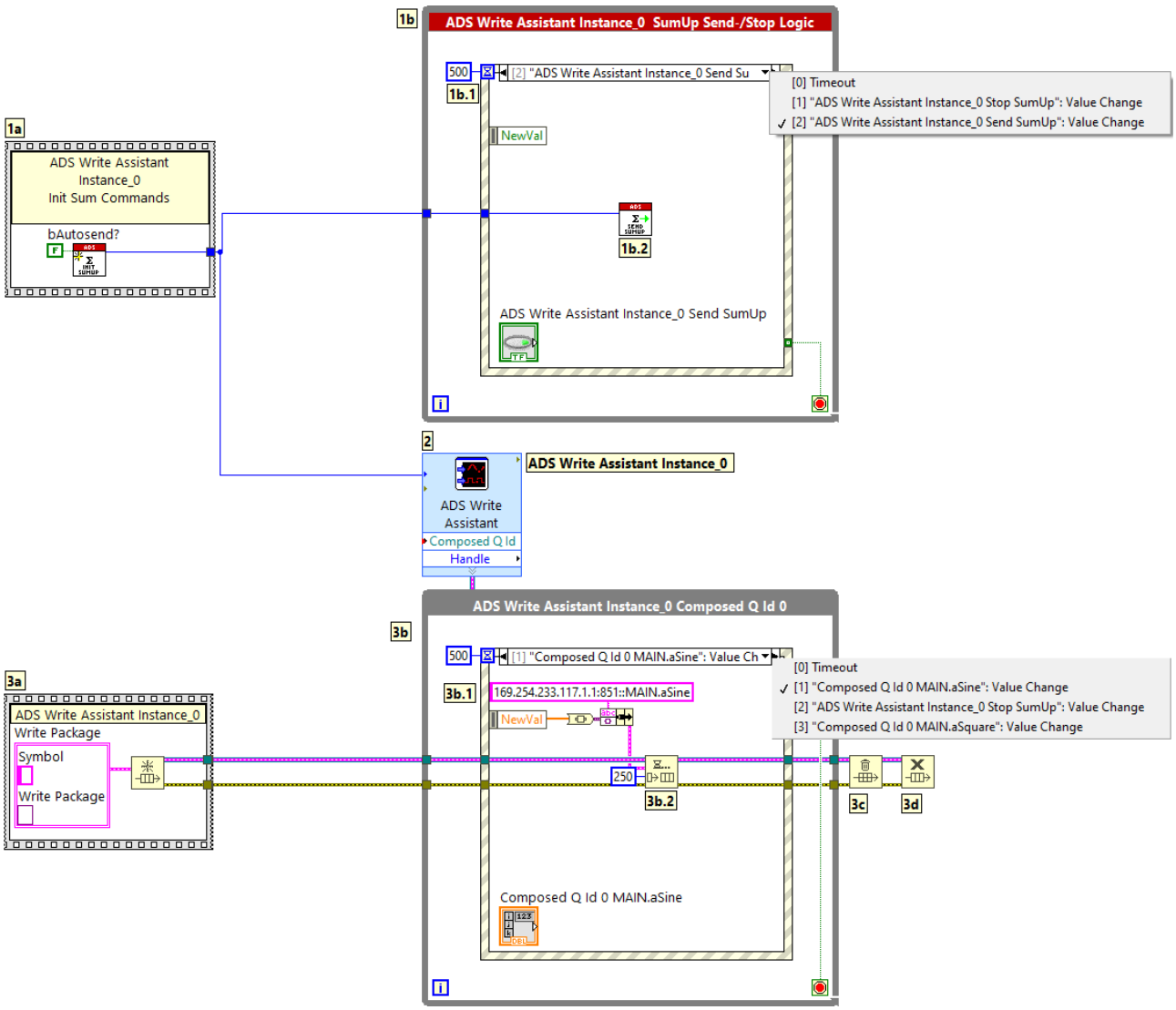
In the first example, the ADS Write Assistant VI is generated with **default settings**, i.e. Start with LabVIEW™ Run, Stop with LabVIEW™ Stop. An ADS symbol MAIN.aSine is written. A sine signal generated by LabVIEW™ and transferred to the TwinCAT 3 Runtime.



- **1:** The ADS Writer SumUp handle is initialized and the flag *bAutosend?* is set to **True**. Thus, each new packet is automatically transported from LabVIEW™ to the TwinCAT 3 Runtime.
- **2:** A handle goes from the ADS Write Assistant to the underlying queue blocks. The queue forwards each new data packet to the ADS Write Assistant. Until then, the data packet only contains raw data, which is then automatically converted into a TwinCAT 3 data type by the ADS Write Assistant using the TypeResolver and then transferred to TwinCAT using the ADS SumUp.
- **3a:** The queue is initialized with a LabVIEW™ variant as data type.
- **3b:** While loop with start condition "LabVIEW™ Run" without end condition.
- **3b.1**: The ADS Write Assistant generates an event case with the property "value change" for each selected symbol. Only value changes are made available to the SumUp handle. The image below shows an example of how the new data can be generated. This is about the so-called LabVIEW™ block diagram events.
- **4**: The ADS symbol MAIN.aSine is in this case a LREAL array of 20 elements. The generated sine signal therefore also contains only 20 data points.
- **3b.2**: Enqueue element: Waits for new data packets and inserts each new data packet into the queue.
- **3c and 3d**: Releases the queue memory. After that the queue is released from the LabVIEW™ memory. Optionally, after these steps, the user can release the ADS client handle from memory.



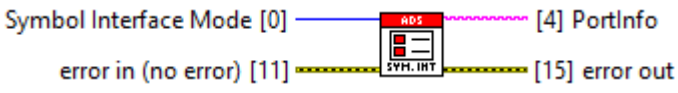
In the second example, the start/stop conditions are set to Trigger on LabVIEW™ Event. In addition, two symbols are written here with the SumUp handle: MAIN.aSquare and MAIN.aSine. The two symbols are assigned the same Loop-ID. Thus, both symbols use a while loop and the same event structure.



- The blocks **2**, **3b**, **3b.1**, **3b.2**, **3c** and **3d** already described in the first example remain identical in their function.
- **1a:** The ADS Writer SumUp handle is initialized and the flag *bAutosend?* is set to **False** . The new packet must be sent explicitly to TwinCAT 3. In the example this is done by the separate event "Send SumUp".
- **1b:** Logic is created that generates a trigger-event when a Boolean value change occurs. The value change starts the write process.
- **1b.2:** A write operation is started using a SumUp handle.
- **3a:** The queue is initialized with a LabVIEW™ cluster as data type with the following member variables:
  - SymbolName: Identifies the data packet to be written.
  - Write Package: Contains the data packet to be transferred.

6.4 Symbol Interface

The *Symbol Interface* block provides a graphical user interface (UI) for browsing ADS symbols. It simplifies the selection of different ADS symbols as read or write symbols and the parameterization of the symbols. The block creates, based on the configuration created by the user, a LabVIEW™ string which can be passed to the Init VI. Likewise, an XML file of the created configuration can be saved from the UI so that the configuration can be loaded and passed to the Init VI without repeated manual configuration.

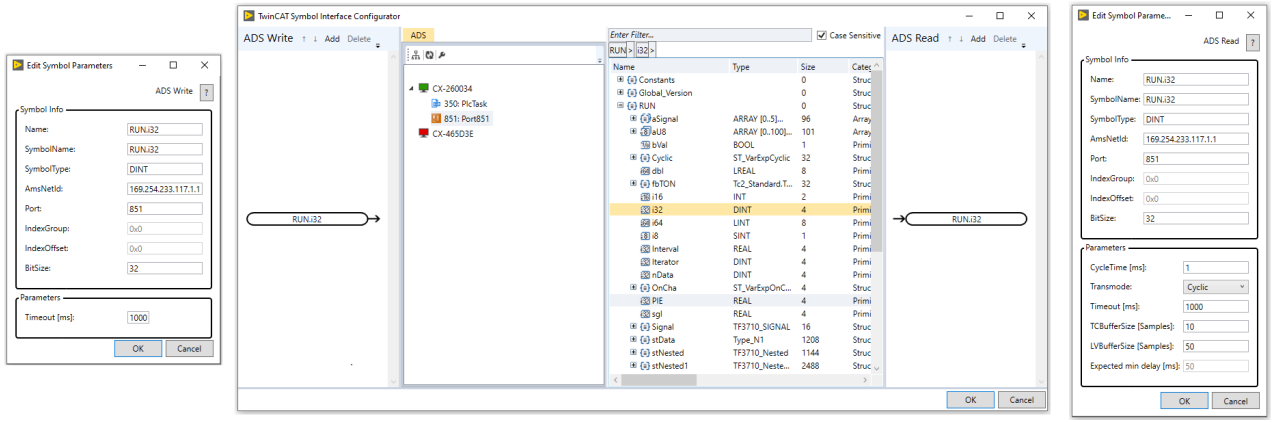


Input/output	Meaning
[0] Symbol Interface Mode	LabVIEW™ Enum consisting of three modes: • ADS symbols for read access only • ADS symbols for write access only • Read & Write: Reading and writing are allowed.
[4] PortInfo	LabVIEW™ string in XML with ADS read and/or write symbols.

Microsoft Windows only

The *Symbol Interface* block can be accessed only with a Windows operating system.

Running the *Symbol Interface* will open the graphical user interface shown below. The TwinCAT Target Browser is integrated in the middle part. This is used to browse target systems or their ADS symbols. By drag and drop an ADS symbol (or by multi-select several ADS symbols at the same time) can be dragged to the right area for read accesses or to the left area for write accesses. For each selected ADS symbol a graphic element appears in the Write or Read area. By double-clicking on these graphical elements (multi-select is also possible here) another window appears. The window describes the information of the ADS symbol and offers the option to attach parameters to the ADS symbol, which are used for certain read or write commands.



Manual addressing via AmsNetId, Port, Index Group and Index Offset is also possible. To do this, create a new element under ADS-Read or ADS-Write with **New** and you can fill in the "Symbol Info" area manually.

CycleTime parameter

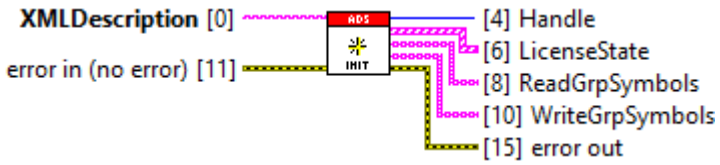
The cycle time in which the selected ADS symbol is executed in the TwinCAT runtime is always offered as the default value for the CycleTime. Only multiples of this cycle time are allowed, since an ADS notification with the new value is sent after each completed cycle. If a value other than CycleTime is entered, it is always rounded up to the next permissible value.

6.5 Init

The main task of the *Init* block is to initialize and connect the ADS client to the ADS router. After successful initialization, LabVIEW™ receives a handle back to the ADS client. For this purpose the *Init* block uses the low-level Init [▶ 92] blocks in the background.

In addition to its main task, the *Init* block performs the following other tasks:

- Creates a list of ADS target systems based on the input XMLDescription.
- Checks each individual target system for a valid TF3710 TwinCAT 3 Interface for LabVIEW™ license.
- Creates a list of target systems on which a valid TF3710 TwinCAT 3 Interface for LabVIEW™ could be retrieved. Reading and writing is only possible on target systems with a valid license.
- Creates a sorted list of ADS read symbols based on the input *XMLDescription*.
- Creates a sorted list of ADS write symbols based on the input *XMLDescription*.



Input/output	Meaning
[0] XMLDescription	LabVIEW™ XML string with ADS read and write symbols **or** the path as a string to an existing, already created (exported) XML file.
[4] Handle	Handle to the ADS client
[6] LicenseState	List of license states of the TwinCAT target systems
[8] ReadGrpSymbols	List of symbols that are read via ADS.
[10] WriteGrpSymbols	List of symbols that are written via ADS.

Client Handle

The handle to the ADS client in LabVIEW™ is only released if a valid TF3710 TwinCAT 3 Interface for LabVIEW™ license is found on at least one of the selected target systems. If no license could be found, the Init block returns an error via error out [15].

6.6 ADS-Read

The *ADS-Read*- block is a polymorphic VI and supports the following ADS communication modes [▶ 23] to read data from TwinCAT:

- Sync Single [▶ 26]
- Async Single [▶ 27]
- Noti. Single [▶ 27]
- Noti. Buffered [▶ 27]
- E-Noti. Sinlge [▶ 31]
- E-Noti Buffered [▶ 33]
- E-Noti. Multiple Symbols [▶ 71]
- LVB-Noti. Single Symbol [▶ 71]
- LVB-Noti. Multiple Symbols [▶ 72]

All modes use the low-level Read [▶ 94] blocks in the background to:

1. Initialize the ADS Reader.
2. To send the ADS request.
3. To wait for answer and
4. finally release the reader from the memory.

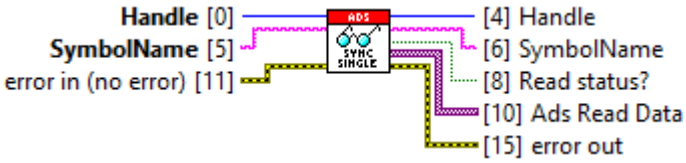
The section Communication modes [▶ 23] provides further information as well as practical application recommendations. In the chapter Examples [▶ 111] you will find exemplary implementations in LabVIEW™.

Depending on the selected communication mode, the parameters configured in Symbol Interface [▶ 65] are used or remain unused. The relevant parameters are named below.

Sync Single [▶ 26]

In this operation mode, the ADS client (LabVIEW™) sends a request to the ADS server (TwinCAT) and waits for a response from the server in the program sequence.

Relevant parameter: Timeout

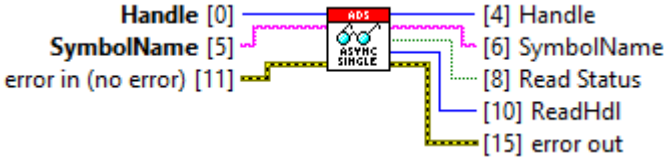


Input/output	Meaning
[0] [4] Handle	Handle to the ADS client
[5] [6] SymbolName	Describes an ADS symbol consisting of AMS address and symbol designation: • AMS address: A compilation of AMS NetId and AMS port. • Symbol designation: Symbol name or index group/offset Examples of the symbol name: • AMS address and symbol name: 127.0.0.1.1.1:851::MAIN.Var • AMS address and index group/offset: 127.0.0.1.1.1:851::0x1234/0x12345
[8] Read status?	Read status
[10] Ads Read Data	ADS raw data

Async Single [▶ 27]

In this operation mode, the ADS client (LabVIEW™) sends a request to the ADS server (TwinCAT) and does not wait for a response from the server in the program sequence.

Relevant parameter: Timeout

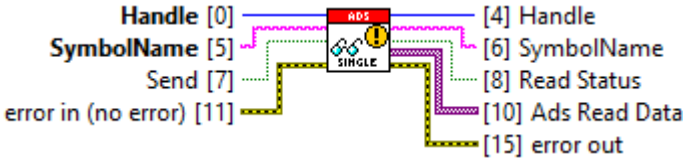


Input/output	Meaning
[0] [4] Handle	Handle to the ADS client
[5] [6] SymbolName	Describes an ADS symbol consisting of AMS address and symbol designation: • AMS address: A compilation of AMS NetId and AMS port. • Symbol designation: Symbol name or index group/offset Examples of the symbol name: • AMS address and symbol name: 127.0.0.1.1.1:851::MAIN.Var • AMS address and index group/offset: 127.0.0.1.1.1:851::0x1234/0x12345
[8] Read Status	Read status
[10] ReadHdl	Handle to the ADS reader

Noti. Single [▶ 27]

In this operation mode of ADS Read.vi, ADS notifications are registered to the relevant ADS symbols in TwinCAT. Only "on change" registration is possible. The ADS notification is unregistered again from the ADS server when the first "on change" notification is received in LabVIEW™.

Relevant parameters: Timeout, Transmode (= on change)



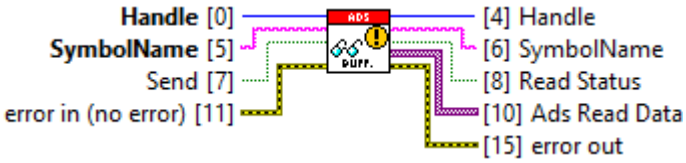
Input/output	Meaning
[0] [4] Handle	Handle to the ADS client
[5] [6] SymbolName	Describes an ADS symbol consisting of AMS address and symbol designation: • AMS address: A compilation of AMS NetId and AMS port. • Symbol designation: Symbol name or index group/offset Examples of the symbol name: • AMS address and symbol name: 127.0.0.1.1.1:851::MAIN.Var • AMS address and index group/offset: 127.0.0.1.1.1:851::0x1234/0x12345
[7] Send	Send flag, TRUE registers the ADS notification
[8] Read Status	Read status
[10] Ads Read Data	ADS raw data (array of two entries, previous value and current value)

Noti. Buffered [▶ 27]

In this communication mode of ADS Read.vi, ADS notifications are registered on the relevant ADS symbols in TwinCAT. The ADS client in LabVIEW™ also uses a buffer memory, which is first filled with feedback messages from TwinCAT before the entire buffer memory is transferred to LabVIEW™. After passing the buffered data to LabVIEW™, the ADS notification is unregistered from the server. Both ADS notification types, "on change" and "cyclic", are supported.

Relevant parameters: Timeout, Transmode, SampleTime, LVBufferSize, TCBufferSize

With Transmode = "on change" the SampleTime and the TCBufferSize are not relevant.



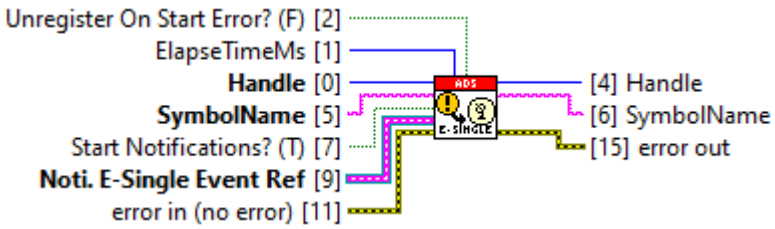
Input/output	Meaning
[0] [4] Handle	Handle to the ADS client
[5] [6] SymbolName	Describes an ADS symbol consisting of AMS address and symbol designation: • AMS address: A compilation of AMS NetId and AMS port. • Symbol designation: Symbol name or index group/offset Examples of the symbol name: • AMS address and symbol name: 127.0.0.1.1.1:851::MAIN.Var • AMS address and index group/offset: 127.0.0.1.1.1:851::0x1234/0x12345
[7] Send	Send flag, TRUE registers the ADS notification
[8] Read Status	Read status
[10] Ads Read Data	Buffered ADS raw data

E-Noti. Single [▶ 31]

In this operation mode of ADS Read.vi, ADS notifications are registered on the relevant ADS symbols in TwinCAT and then continuously transmitted to LabVIEW™ as LabVIEW™ event. The ADS notification reports after a defined time if ElapseTimeMs is greater than zero, or remains until it is actively unregistered. For the use of user events in LabVIEW™, see LabVIEW™ documentation. In contrast to E-Noti. Buffered (see below), no LabVIEW™-side buffer memory is used in this case.

Relevant parameters: Timeout, Transmode, SampleTime, TCBufferSize

With Transmode = "on change" the SampleTime and the TCBufferSize are not relevant.



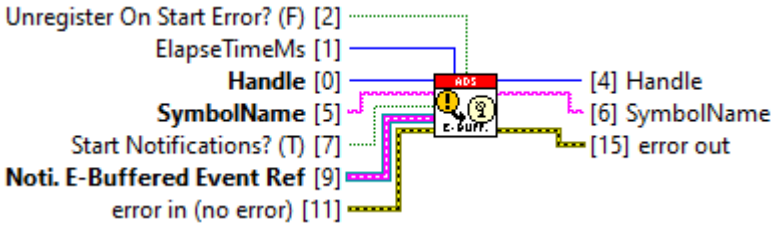
Input/output	Meaning
[0] [4] Handle	Handle to the ADS client
[1] ElapseTimeMs	Measurement duration in milliseconds: • ElapseTimeMs > 0: The notifications stop after the time has elapsed. • ElapseTimeMs = 0: The notifications must be stopped from outside.
[5] [6] SymbolName	Describes an ADS symbol consisting of AMS address and symbol designation: • AMS address: A compilation of AMS NetId and AMS port. • Symbol designation: Symbol name or index group/offset Examples of the symbol name: • AMS address and symbol name: 127.0.0.1.1.1:851::MAIN.Var • AMS address and index group/offset: 127.0.0.1.1.1:851::0x1234/0x12345
[7] Start Notifications	Send Flag, TRUE starts the ADS notification
[9] Noti. E-Single Event Ref	Reference to user event

E-Noti Buffered [▶ 33]

In this operation mode of ADS Read.vi, ADS notifications are registered on the relevant ADS symbols in TwinCAT and then continuously transmitted to LabVIEW™ as LabVIEW™ event. The ADS notification reports after a defined time if ElapseTimeMs is greater than zero, or remains until it is actively unregistered. For the use of user events in LabVIEW™, see LabVIEW™ documentation. In contrast to E-Noti. Single (see above), a LabVIEW™-side buffer memory of size LVBufferSize is used in this case. When the buffer is filled the collected data are passed to LabVIEW™ via an event.

Relevant parameters: Timeout, Transmode, SampleTime, TCBufferSize, LVBufferSize

With Transmode = "on change" the SampleTime and the TCBufferSize are not relevant.



Input/output	Meaning
[0] [4] Handle	Handle to the ADS client
[1] ElapseTimeMs	Measurement duration in milliseconds: • ElapseTimeMs > 0: The notifications stop after the time has elapsed. • ElapseTimeMs = 0: The notifications must be stopped from outside.
[5] [6] SymbolName	Describes an ADS symbol consisting of AMS address and symbol designation: • AMS address: A compilation of AMS NetId and AMS port. • Symbol designation: Symbol name or index group/offset Examples of the symbol name: • AMS address and symbol name: 127.0.0.1.1.1:851::MAIN.Var

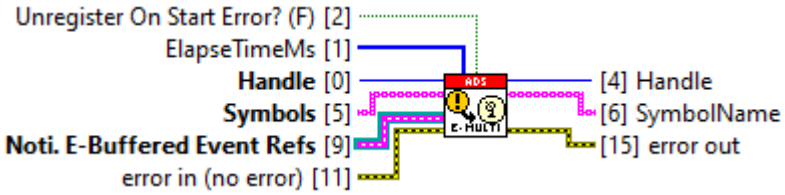
Input/output	Meaning
	• AMS address and index group/offset: 127.0.0.1.1.1:851::0x1234/0x12345
[7] Start Notifications	Send Flag, TRUE starts the ADS notification
[9] Noti. E-Single Event Ref	Reference to user event

E-Noti. Multiple Symbols

This operation mode of ADS Read.vi works like E-Noti. Buffered [▶ 70], except that this VI can be used for several ADS symbols at the same time.

Relevant parameters: Timeout, Transmode, SampleTime, TCBufferSize, LVBufferSize

With Transmode = "on change" the SampleTime and the TCBufferSize are not relevant.



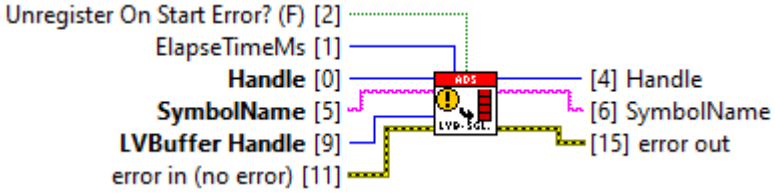
Input/output	Meaning
[0] [4] Handle	Handle to the ADS client
[1] ElapseTimeMs	An array of U32 consisting of measurement duration in milliseconds: • Element of ElapseTimeMs > 0: The notifications stop after the time expires. • Element of ElapseTimeMs = 0: The notifications must be stopped externally.
[5] [6] Symbols	An array of LabVIEW™ strings that specifies the ADS symbols. A symbol is named by the following elements: • AMS address: A compilation of AMS NetId and AMS port. • Symbol designation: Symbol name or index group/offset Examples of the symbol name: • AMS address and symbol name: 127.0.0.0.1.1:851::MAIN.Var • AMS address and index group/offset: 127.0.0.0.1.1:851::0x1234/0x12345
[9] Noti. E-Buffered Event Refs	An array of references to user event

LVB-Noti. Single Symbol

This operation mode of ADS Read.vi works like E-Noti. Buffered [▶ 70], except that this VI provides direct access to LVBuffer [▶ 79] instead of using LabVIEW™ events. As a result, no LabVIEW™ events are required to read the notifications.

Relevant parameters: Timeout, Transmode, SampleTime, LVBufferSize, TCBufferSize

With Transmode = "on change" the SampleTime and the TCBufferSize are not relevant.



Input/output	Meaning
[0] [4] Handle	Handle to the ADS client
[1] ElapseTimeMs	An array of U32 consisting of measurement duration in milliseconds:

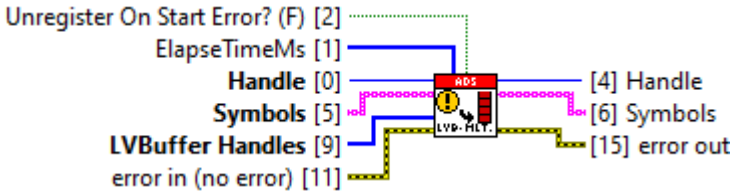
Input/output	Meaning
	• Element of ElapseTimeMs > 0: The notifications stop after the time expires. • Element of ElapseTimeMs = 0: The notifications must be stopped externally.
[5] [6] SymbolName	Describes an ADS symbol consisting of AMS address and symbol designation: • AMS address: A compilation of AMS NetId and AMS port. • Symbol designation: Symbol name or index group/offset Examples of the symbol name: • AMS address and symbol name: 127.0.0.1.1.1:851::MAIN.Var • AMS address and index group/offset: 127.0.0.1.1.1:851::0x1234/0x12345
[9] LVBuffer Handle	An array of U32 consisting of handle on LVBuffer

LVB-Noti. Multiple Symbols

This operation mode of ADS Read.vi works exactly like LVB-Noti. Single Symbol [▶ 71], with the difference that this VI can be used for several ADS symbols at the same time.

Relevant parameters: Timeout, Transmode, SampleTime, LVBufferSize, TCBufferSize

With Transmode = "on change" the SampleTime and the TCBufferSize are not relevant.



Input/output	Meaning
[0] [4] Handle	Handle to the ADS client
[1] ElapseTimeMs	An array of U32 consisting of measurement duration in milliseconds: • Element of ElapseTimeMs > 0: The notifications stop after the time expires. • Element of ElapseTimeMs = 0: The notifications must be stopped externally.
[5] [6] Symbols	An array of LabVIEW™ strings that specifies the ADS symbols. A symbol is named by the following elements: • AMS address: A compilation of AMS NetId and AMS port. • Symbol designation: Symbol name or index group/offset Examples of the symbol name: • AMS address and symbol name: 127.0.0.0.1.1:851::MAIN.Var • AMS address and index group/offset: 127.0.0.0.1.1:851::0x1234/0x12345
[9] LVBuffer Handle	An array of U32 consisting of handle on LVBuffer

6.7 ADS-Write

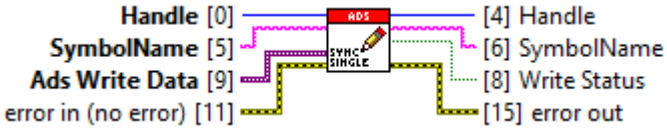
The *ADS-Write* block is a polymorphic VI and supports the following ADS communication modes [▶ 23] to write data to TwinCAT:

- Sync Single [▶ 28]
- Async Single [▶ 28]

For ADS-Write only one parameter is relevant, which can be set in the Symbol Interface [▶ 65] and that is the parameter *Timeout*.

Sync Single [▶ 28]

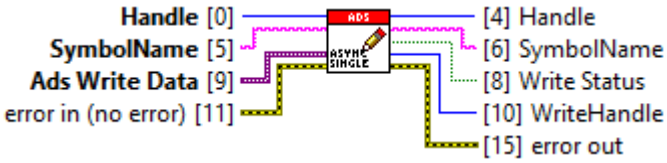
In this operation mode of ADS Write.vi, the ADS client in LabVIEW™ sends a write request with the type-resolved ADS value to the ADS server in TwinCAT and waits for a response as to whether the value has been written.



Input/output	Meaning
[4] Handle	Handle to the ADS client
[5] [6] SymbolName	Describes an ADS symbol consisting of AMS address and symbol designation: • AMS address: A compilation of AMS NetId and AMS port. • Symbol designation: Symbol name or index group/offset Examples of the symbol name: • AMS address and symbol name: 127.0.0.1.1.1:851::MAIN.Var • AMS address and index group/offset: 127.0.0.1.1.1:851::0x1234/0x12345
[9] Ads Write Data	Type-resolved ADS value
[8] Write Status	Write status

Async Single [▶ 28]

In this operation mode of ADS Write.vi, the ADS client in LabVIEW™ sends a write request with the type-resolved ADS value to the ADS server in TwinCAT and does **not** wait for a response as to whether the value has been written.



Input/output	Meaning
[0] [4] Handle	Handle to the ADS client
[5] [6] SymbolName	Describes an ADS symbol consisting of AMS address and symbol designation: • AMS address: A compilation of AMS NetId and AMS port. • Symbol designation: Symbol name or index group/offset Examples of the symbol name: • AMS address and symbol name: 127.0.0.1.1.1:851::MAIN.Var • AMS address and index group/offset: 127.0.0.1.1.1:851::0x1234/0x12345
[9] Ads Write Data	Type-resolved ADS value
[8] Write Status	Write status
[10] WriteHandle	Handle to the ADS Writer

6.8 TypeResolver

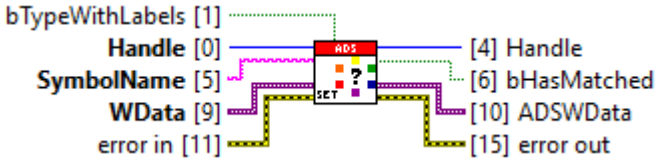
The *TypeResolver* block is a polymorphic VI and supports the following operation modes to convert data types:

- To TC (for writing from LabVIEW™ to TwinCAT)
- From TC (for reading from TwinCAT to LabVIEW™)

To TC

The TypeResolver block in operation mode *To TC* initializes the TypeResolver, compares the LabVIEW™ data type with the TC3 data type, and converts the LabVIEW™ data to ADS raw data. The block then releases the TypeResolver from the memory.

Further information can be found in the section Resolving ADS data stream for TwinCAT 3 [▶ 36].



Input/output	Meaning
[0] Handle	Handle to the ADS client
[5] SymbolName	ADS symbol consisting of AMSNetId and symbol name
[9] WData	Raw data for ADS-Write
[4] Handle	Handle to the ADS client
[6] bHasMatched	Flag (TRUE if TwinCAT 3 and LabVIEW™ data type are identical, otherwise FALSE)
[10] ADSWData	Type-resolved ADS write value

From TC

The TypeResolver block in operation mode *From TC* initializes the TypeResolver, parses and converts the ADS data read by TwinCAT using ADS-Read into the LabVIEW™ data type *Variant*. The Type Release block releases the TypeResolver from memory.

Further information can be found in the section Resolving ADS data stream for TwinCAT 3 [▶ 35].



Input/output	Meaning
[0] Handle	Handle to the ADS client
[5] SymbolName	ADS symbol consisting of AMSNetId and symbol name
[9] ADSRData	Raw data from ADS-Read
[4] Handle	Handle to the ADS client
[8] RDataArray	Type-resolved ADS read array for the LabVIEW™ data type
[10] RData	Type-resolved ADS read value for the LabVIEW™ data type

6.9 Release

The *Release* block releases the handle to the ADS client from memory.



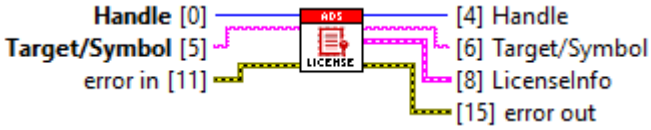
Input/output	Meaning
[0] Handle	Handle to the ADS client

6.10 Utilities

Additional VIs can be found under Utilities.

Check License

The *Check License* block checks the target system for a valid TF3710 TwinCAT 3 Interface for LabVIEW™ license.



Input/output	Meaning
[0] Handle	Handle to the ADS client
[5] Target/Symbol	ADS target as string consisting of AMSNetId or ADS symbol as string consisting of AMSNetId and symbol name
[6] Target/Symbol	ADS target as string consisting of AMSNetId or ADS symbol as string consisting of AMSNetId and symbol name
[4] Handle	Handle to the ADS client
[8] LicenseInfo	LabVIEW™ cluster consisting of three elements: • Boolean flag (TRUE if license is valid, FALSE otherwise) • AMSNetId of the target system • String with message about the license state, e.g. "license is valid" or "trial license denied“

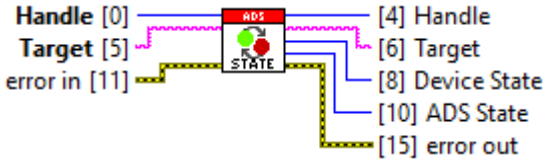
Get Device State

The *Device State* block is a polymorphic VI and can be operated as *Get Device State* and *Set Device State*.

The operation mode *Get Device State* reads the current state of the target system and the PLC.

The following table describes the ADS port and its meaning:

ADS Port	ADS State	State
TwinCAT 3 PLC Port > 851	ADS_STATE_RUN	PLC is running.
	ADS_STATE_STOP	PLC is stopped.
	ADS_STATE_RESET	PLC is reset.
System Port = 10000	ADS_STATE_RECONFIG	TwinCAT is in Config mode.
	ADS_STATE_RESET	TwinCAT is in Run mode.



Input/output	Meaning
[0] Handle	Handle to the ADS client
[5] Target	ADS target as string consisting of AMSNetId
[4] Handle	Handle to the ADS client
[6] Target	ADS target as string consisting of AMSNetId
[8] Device State	State of the target system (ENUM)

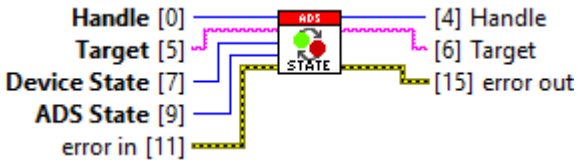
Input/output	Meaning
[10] ADS State	State of the PLC (ENUM)

Set Device State

The *Set Device State* block changes the current state of the target system and the PLC.

With a specific ADS Port and ADS State the PLC or TwinCAT can be set to a corresponding state. The following table describes the ADS port and its usage:

ADS Port	ADS State	State
TwinCAT 3 PLC Port > 851	ADS_STATE_RUN	Starts the PLC.
	ADS_STATE_STOP	Stops the PLC.
	ADS_STATE_RESET	Resets the PLC.
System Port = 10000	ADS_STATE_RECONFIG	Puts TwinCAT in Config mode.
	ADS_STATE_RESET	Puts TwinCAT in Run mode.



Input/output	Meaning
[0] Handle	Handle to the ADS client
[5] Target	ADS target as string consisting of AMSNetId
[7] Device State	New state of the target system (numeric)
[9] ADS State	New state of the PLC (ENUM)
[4] Handle	Handle to the ADS client
[6] Target	ADS target as string consisting of AMSNetId

Get Version Info

The Device State block in operation mode *Get Version Info* returns the current version of the TF3710 setup and the installed TF3710 library as a LabVIEW™ string.



Input/output	Meaning
[6] ProductVersion	Setup version as string
[8] FileVersion	Version of the installed TF3710 library as string

Generate Type

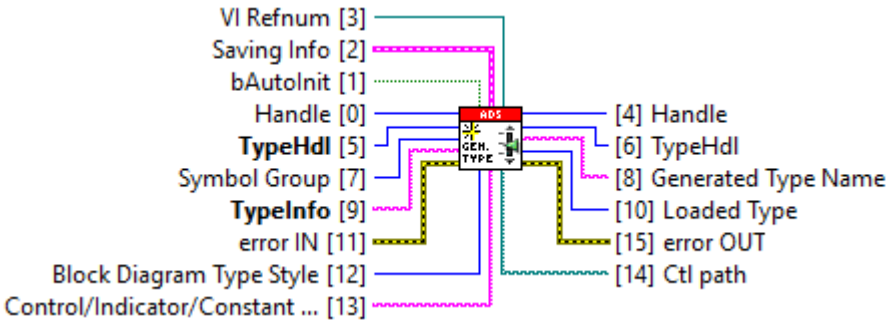
The *Generate Type* block automatically generates the LabVIEW™ data type based on a TypeInfo description from a TwinCAT 3 data type.

The LabVIEW™ type can be inserted as a constant or control on a block diagram or saved as a LabVIEW™ ctl file. The *Generate Type* block is a wrapper VI and uses low-level class objects from TypeGenerator [▶ 99] together with the TypeResolver [▶ 98]. There are several ways in which the wrapper block can be parameterized.

1. **The TypeResolver is already initialized externally and LabVIEW™ already has a TypeHandle. The bAutoInit flag has the value "False"**:
   - The TypeResolver already has information regarding the TwinCAT 3 data type.
   - The Wrapper block can generate the new type.

- The example Generate Type using Symbol Interface or Symbol's file [▶ 116] uses this method.

2. **The TypeResolver is not initialized externally and LabVIEW™ does not have a TypeHandle. The bAutoInit flag has the value "True"**:
   - The TypeResolver has no information regarding the TwinCAT 3 data type.
   - The Wrapper block needs the TypeInfo to generate the type.
   - The example Generate Type using TypeInfo file [▶ 117] uses this method.



Input/output	Meaning
[0] Handle	Handle to the ADS client
[1] bAutoInit	Boolean flag (True = TypeResolver is initialized internally, otherwise False)
[2] Saving Info	LabVIEW™ cluster consisting of three elements: • CustomDir: Boolean flag TRUE = LabVIEW™ type is saved in a specific folder (Dir Name). If this folder does not exist, the block creates the folder. FALSE = LabVIEW™ type is saved in the project folder, subfolder Ctl. • Dir Name: LabVIEW™ string (name of the folder) • Control FileName: LabVIEW™ string (name of the new Ctl. If empty, the block uses the name from the TypeInfo)
[3] VI Refnum	A static VI Refnum The block uses the VI Refnum to insert the LabVIEW™ type as a Constant/Control/Indicator on the block diagram.
[5] TypeHdl	The handle to the TypeResolver
[7] Symbol Group	LabVIEW™ Enum: • Read symbol: The block generates a control. • Write symbol: The block generates a control.
[9] TypeInfo	LabVIEW™ string as XML description of TwinCAT 3 data type
[12] Block Diagram Type Stype	LabVIEW™ Enum: • Constant • Control/Indicator
[13] Control/Indicator/Constant Name	LabVIEW™ string. Empty string: Default name of the TypeInfo. Otherwise: Name of the Control/Indicator/Constant.
[8] Generated Type Name	LabVIEW™ string as the name of the generated type
[10] Loaded Type	LabVIEW™ Enum Describes which data type was generated.
[14] Ctl path	File path to the generated type control

"Symbol Group" parameter

The Symbol Group parameter can be used to select whether a control/display element must be generated. For the control, the TypeGenerator generates the type without name, because the TypeResolver supports the control without name. If the control is a LabVIEW™ cluster, then the cluster members are also generated without names.

6.10.1 Notification

The *Notification* folder contains additional blocks that can be used with ADS notifications when reading. The folder contains the following VIs:

- ADS To LabVIEW Timestamp
- Notification Data To Variant Array
- Stop Notification
- Unregister Notification

ADS To LabVIEW Timestamp

The *ADS To LabVIEW Timestamp* block is a polymorpohic VI and supports single and multiple ADS timestamps (as array). The block converts ADS timestamps, as transmitted by an ADS notification from TwinCAT, into LabVIEW™ timestamps.

Notification Timestamp Single

The *Notification Timestamp Single* block converts single ADS timestamps to LabVIEW™ timestamps.



Input/output	Meaning
[1] Ads Timestamp	Single ADS timestamp
[0] LabVIEW Timestamp	Single converted LabVIEW™ timestamp

Notification Timestamp Buffered

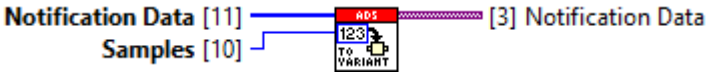
The *Notification Timestamp Buffered* block converts multiple ADS timestamps to LabVIEW™ timestamps.



Input/output	Meaning
[1] Ads Timestamp	Array of ADS timestamps
[0] LabVIEW Timestamp	Array of converted LabVIEW™ timestamps

Notification Data To Variant Array

The *Notification Data To Variant Array* block converts ADS raw data that has been read with ADS notification into a suitable LabVIEW™ Variant.



Input/output	Meaning
[11] Notification Data	ADS raw data
[10] Samples	Number of samples
[3] Notification Data	Converted raw data in LabVIEW™ variant

Stop Notification

The *Stop Notification* block stops receiving ADS notifications.

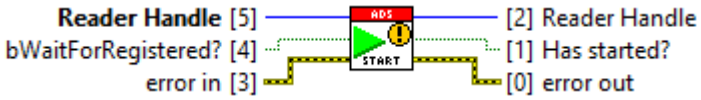


Input/output	Meaning
[5] Notification/Reader handle	Handle to the ADS Notification/Reader

Input/output	Meaning
[2] Notification/Reader handle	Handle to the ADS notification/ADS reader
[1] Has stopped?	Boolean flag (TRUE if notification is stopped, FALSE otherwise)

Start Notification

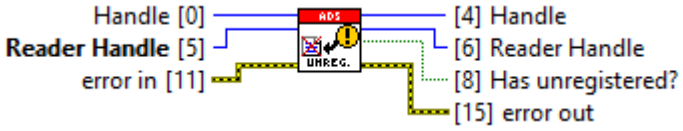
The *Start Notification* block starts the registered notification.



Input/output	Meaning
[5] Reader Handle	Handle to the ADS reader
[4] bWaitForRegistered?	True if the notification is to be started.
[2] Reader Handle	Handle to the ADS reader
[1] Has started?	Boolean flag (TRUE if notification is started, FALSE otherwise)

Unregister Notification

The *Unregister Notification* block unregisters the notification on the ADS server.



Input/output	Meaning
[0] Handle	The handle to the client (can also be left blank)
[5] Reader Handle	Handle to the ADS reader
[4] Handle	Handle to the ADS client
[6] Reader Handle	The handle to the ADS reader
[8] Has unregistered?	Boolean flag (TRUE if notification is registered, FALSE otherwise)

6.10.1.1 Notification controls

These controls can be found at:
Front Panel Palette > User Controls > Beckhoff LabVIEW Interface > Notification.

Single Buffer Info

The *Single Buffer Info* control describes/defines the information needed for reading the Noti. Buffered [▶ 69].

Single User-Event Data

The *Single User-Event Data* control describes/defines information needed for continuous reading of the E-Noti. Single [▶ 69] with LabVIEW™ events.

Buffered User-Event Data

The *Buffered User-Event Data* control describes/defines information needed for continuous reading of the E-Noti. Buffered [▶ 70] with LabVIEW™ events.

6.10.2 LVBuffer

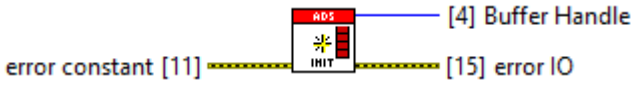
The LVBuffer folder contains blocks that can be used when reading an ADS notification. The folder contains the following VIs:

- Init LVBuffer Handle

- Read From LVBuffer
- LVBuffer status
- Release LVBuffer Handle

Init LVBuffer Handle

The *Init LVBuffer Handle* block initializes a handle on the LVBuffer.

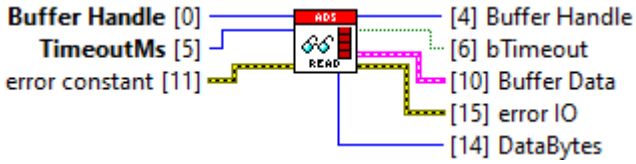


Input/output	Meaning
[4] Buffer handle	The handle to the LVBuffer

Read From LVBuffer

The *Read from LVBuffer* block waits for samples in the LVBuffer (LabVIEW™-side data buffer, cf. Communication modes [▶ 23]). The TimeoutMs influences the waiting. The block waits for a defined time if TimeoutMs > 0, otherwise the block waits forever. If the LVBuffer receives a sample while waiting, the block reads the sample and passes it to LabVIEW™.

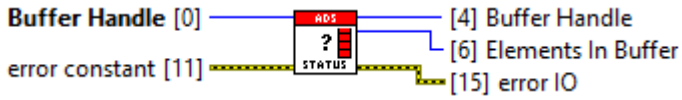
Relevant parameters: LVBufferSize



Input/output	Meaning
[0] Buffer handle	The handle to the LVBuffer
[5] TimeoutMs	The waiting time in milliseconds: • TimeoutMs>0: *Read From Buffer* block waits for the defined time. • TimeoutMs<0: *Read From Buffer* block waits forever.
[4] Buffer handle	The handle to the LVBuffer
[6] bTimeout	Boolean flag: • True: if there is no sample in the LVBuffer in the defined time. • False: LVBuffer continuously gets new samples and the *Read From Buffer* block can read them.
[10] Buffer Data	Samples from the LVBuffer
[14] DataBytes	Number of bytes in LVBuffer

LVBuffer status

The *LVBuffer Status* block returns the current state of the LVBuffer regarding samples in the LVBuffer.



Input/output	Meaning
[0] Buffer handle	The handle to the LVBuffer
[4] Buffer handle	The handle to the LVBuffer
[6] Elements In Buffer	Number of samples in LVBuffer that are still to be read.

Release LVBuffer Handle

The release LVBuffer Handle block releases the handle to the LVBuffer from memory.



Input/output	Meaning
[0] Buffer handle	The handle to the LVBuffer

6.10.3 CoE

Some input and output components of control systems can be parameterized. For EtherCAT components, parameterization is based on the Can-Application-protocoll-over-EtherCAT (CoE) standard, see information on the CoE interface. VIs that enable you to read and write CoE objects from LabVIEW™ are described below. The AMS address of the EtherCAT device is used here. The address is made up of the master AMS NetId and AMS port of the client. The folder contains the following VIs:

- Read CoE List
- Read CoE Description
- Read CoE Entry
- Read CoE Value
- Write CoE Value

The example CoE Read or Write [▶ 118] describes the use of CoE blocks.

Read CoE List

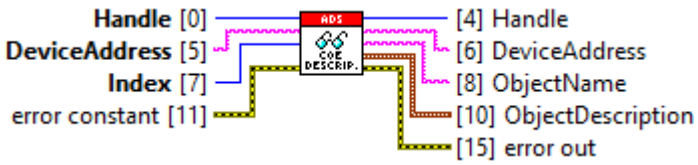
The VI *Read CoE List* reads a CoE directory of a subscriber and lists all objects in an array that are available for the selected device. The objects are identified via indices that are used by subsequent CoE VIs for access.



Input/output	Meaning
[0] [4] Handle	Handle to the ADS client
[5][6] DeviceAddress	AMS address of the device consisting of: • AMS NetId of the master • AMS port of the client
[7] ListType	LabVIEW™ Enum Describes which indices from the directory are to be listed: • TotalNumberOfLists: Lists the number of elements in the AllCoEObjects, RxPDOs, TxPDOs, StoredForDevice and StartUp directories. • AllCoEObjects: List of all CoE objects • RxPDOs: List of all RxPDO objects • TxPDOs: List of all TxPDO objects • SettingObjects: List of all objects that are relevant when the device is replaced. • StartUp: List of objects that can be used as StartUp parameters.
[8] List	1D array of the indices of the objects

Read CoE Description

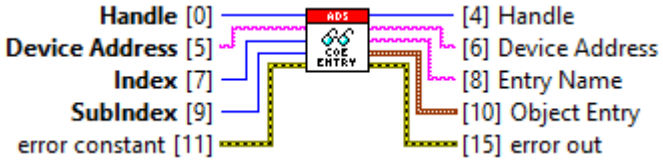
The VI *Read CoE Description* calls up the CoE object description of the device and sends it to LabVIEW™. The description includes the object name and the number of entries or subindices of the object.



Input/output	Meaning
[0] [4] Handle	Handle to the ADS client
[5][6] DeviceAddress	AMS address of the device consisting of: • AMS NetId of the master • AMS port of the client
[7] Index	Index of the object
[8] ObjectName	Name of the object
[10] ObjectDescription	LabVIEW™ cluster consisting of the following elements: • Index: Index of the object • DataType: Data type of the object • MaxSubIndex: Number of subindices of the object

Read CoE Entry

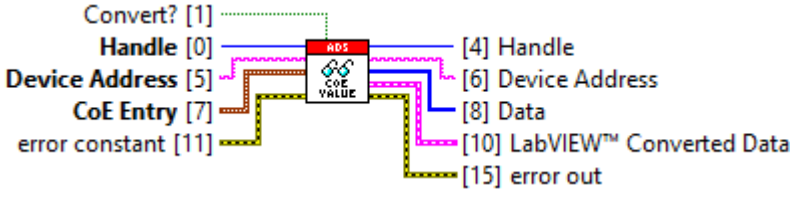
The VI *Read CoE Entry* reads a CoE entry of an object. The object is referenced via the index and subindex. Information is returned, such as the name of the entry and the access.



Input/output	Meaning
[0] [4] Handle	Handle to the ADS client
[5][6] DeviceAddress	AMS address of the device consisting of: • AMS NetId of the master • AMS port of the slave
[7] Index	Index of the object
[8] Entry Name	Type of entry
[9] SubIndex	Subindex of the entry
[10] CoE Entry	LabVIEW™ cluster consisting of the following elements: • Index: Index of the object • SubIndex: Subindex of the entry • DataType: Data type of the entry • BitLength: The length of the entry in bits • ObjectAccess: Information on changeability, e.g. whether it can only be read or also written.

Read CoE Value

The VI *Read CoE Value* reads the value of a CoE entry. An array of the length BitLength in bytes is returned. The array specifies the value in hexadecimal numbers in little-endian format.

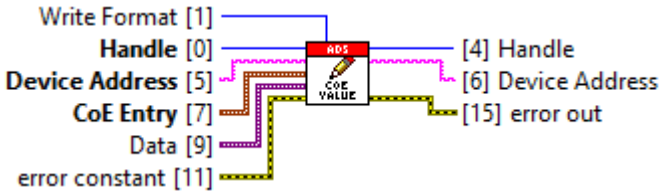


Input/output	Meaning
[1] Convert?	Switches on/off the conversion of the read CoE entry into the corresponding LabVIEW™ data type. If switched on, the converted data is located in "LabVIEW™ Converted Data [10]".
[0] [4] Handle	Handle to the ADS client
[5][6] DeviceAddress	AMS address of the device consisting of: • AMS NetId of the master • AMS port of the slave
[7] CoE Entry	LabVIEW™ cluster consisting of the following elements: • Index: Index of the object • SubIndex: Subindex of the entry • DataType: Data type of the entry • BitLength: The length of the value of the entry in bits • ObjectAccess: Describes the EtherCAT state in which the object can be accessed.
[8] Data	1D byte array with the value of the CoE entry
[10] LabVIEW™ Converted Data	Only available if "[1] Convert?" is set to True. LabVIEW™ Cluster, which contains the CoE data converted to LabVIEW™, consisting of: • Data as string: Data as string (e.g. numerical value 42 becomes "42") • Data as variant: Data as LabVIEW™ Variant, which can be converted into the original LabVIEW™ data type using "Variant To Data".

The value of the CoE entry is displayed in hexadecimal and the bytes are transferred in little-endian order. Strings and arrays are transferred from left to right, whereby the elements of the array are stored in the little-endian format.

Write CoE Value

The *Write CoE Value* block writes the value of the CoE entry. The value to be written can be entered either as a byte array, string or variant. Byte arrays must be specified in hexadecimal numbers in little-endian format.



Input/output	Meaning
[1] Write Format	LabVIEW™ Enum Selects the format of the data to be written (CoE entry), which is provided via the "[9] Data" input: • Byte array • String • Variant
[0] [4] Handle	Handle to the ADS client
[5][6] DeviceAddress	AMS address of the device consisting of: • AMS NetId of the master • AMS port of the slave
[7] CoE Entry	LabVIEW™ cluster consisting of the following elements: • Index: Index of the object • SubIndex: Subindex of the entry

Input/output	Meaning
	• DataType: Data type of the entry • BitLength: The length of the value of the entry in bits • ObjectAccess: Describes the EtherCAT state in which the object can be accessed.
[9] Data	LabVIEW™ Variant that contains the data to be written (CoE entry). The data can be available in the following formats: • Byte array: 1D byte array of the length "CoE Entry.BitLength". The values must be specified in hexadecimal numbers in little-endian format. • String: The value is specified as a string and converted internally to "CoE Entry.DataType". • Variant: The value is transferred as LabVIEW™ Variant and converted internally to "CoE Entry.DataType".

6.10.4 Heartbeat

In this chapter you will find a description of the VIs with which you can create a Heartbeat connection monitoring between LabVIEW™ and TwinCAT. Further information on the parameters can be found in the Heartbeat Connection Monitoring section in the technical introduction [▶ 37]. Connection monitoring can be created from the following Low-Level VIs:

- Init Heartbeat
- Run Heartbeat
- Stop Heartbeat
- Release Hearbeat
- Get RTD Estimate

The folder also contains the Heartbeat basic VI, in which the connection monitoring is summarized in a VI.

System requirements

The Heartbeat function is available from TwinCAT 3.1 Build 4026.

Init Heartbeat

The "Init Heartbeat" block creates a Heartbeat instance. Based on the client handle (see Init), a scan is performed for each registered target to determine whether an instance of FB_Heartbeat exists. If this is the case, connection monitoring is prepared for each heartbeat found. If no instance of FB_Heartbeat is found at any target, the VI returns the error -8114.

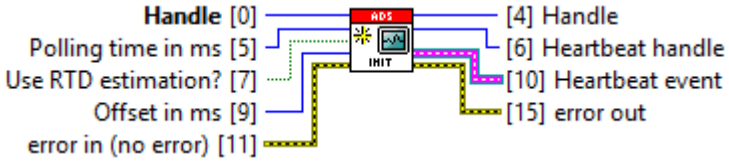
FB_Heartbeat

The FB_Heartbeat function block can be found in the Tc3_HeartbeatConnectionMonitoring library and must be created and called in the project for which connection monitoring is being set up.

The block also creates a Heartbeat event. This can then be used to query the status of the connection. As soon as the status of the connection changes, a Heartbeat event is generated. The Heartbeat event has the following event dates:

- Heartbeat handle: Handle to the Heartbeat instance from which the event was generated.
- TargetAddr: String containing the AMS address (NetId:Port) of the target from which the Heartbeat event was triggered.
  - Connection State: Enum that specifies the reason for triggering the Heartbeat event.
  - Waiting: Heartbeat initialized, connection monitoring not (yet) started.
  - Connected: Connection monitoring started.
  - Timeout detected: Connection monitoring has detected an interruption.
  - Stopped: Connection monitoring has been stopped.

- Error: An error has occurred during connection monitoring (error code is provided at the Heartbeat output. If more than one target is monitored, only the connection monitoring for the corresponding target is stopped; the others continue to run).
- Reserved: reserved (ignore)



Input/output	Meaning
[0] [4] Handle	Handle to the ADS client
[5] Polling time in ms	Parameter Polling time in ms
[6] Heartbeat handle	Handle to the Heartbeat instance
[7] Use RTD estimation?	Parameter UseRtdEstimation
[9] Offset in ms	Parameter offset
[10] Heartbeat event	User Event Refnum on the generated Heartbeat event

Run Heartbeat

The Run Heartbeat block is used to start and continuously execute connection monitoring. This block is executed until the connection monitoring is stopped with the Stop Heartbeat.



Input/output	Meaning
[0] [4] Handle	Handle to the ADS client
[5] [6] Heartbeat handle	Handle to the Heartbeat instance

Stop Heartbeat

The Stop Heartbeat block has the task of stopping connection monitoring.



Input/output	Meaning
[0] [4] Heartbeat handle	Handle to the Heartbeat instance

Release Heartbeat

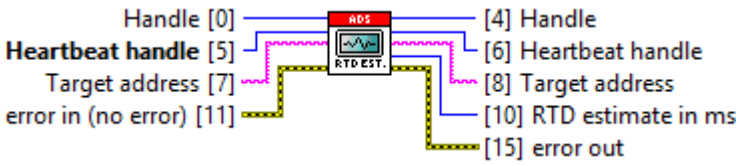
The Release Heartbeat block releases the handle to the Heartbeat connection monitoring from the memory.



Input/output	Meaning
[0] [4] Handle	Handle to the ADS client
[5] Heartbeat handle	Handle to the Heartbeat instance

Get RTD Estimate

The Get RTD Estimate block returns the current (maximum) roundtrip delay (RTD) estimate for the selected target. The target can be selected via the "TargetAddr" string. If the string is empty, the target found first is automatically selected. To use this block, it is necessary to execute the Heartbeat connection monitoring (see VIs described above).

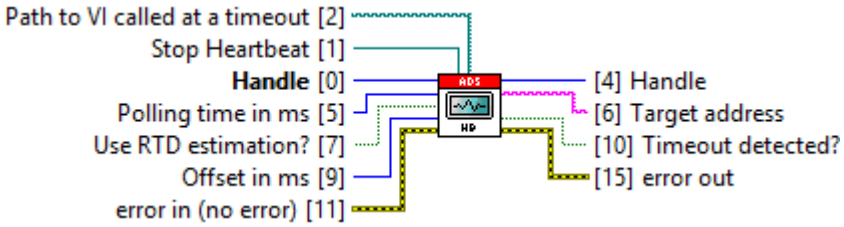


Input/output	Meaning
[0] [4] Handle	Handle to the ADS client
[5] [6] Heartbeat handle	Handle to the Heartbeat instance
[7][8] Target address	String with the address of the target
[10] RTD estimate in ms	Estimation of the maximum RTD in ms

Heartbeat

The Heartbeat block uses the Init Heartbeat, Run Heartbeat, Stop Heartbeat and Release Heartbeat blocks internally to enable the use of Heartbeat connection monitoring with just one block. The Heartbeat block also serves as an illustration of how Heartbeat connection monitoring can be used and edited.

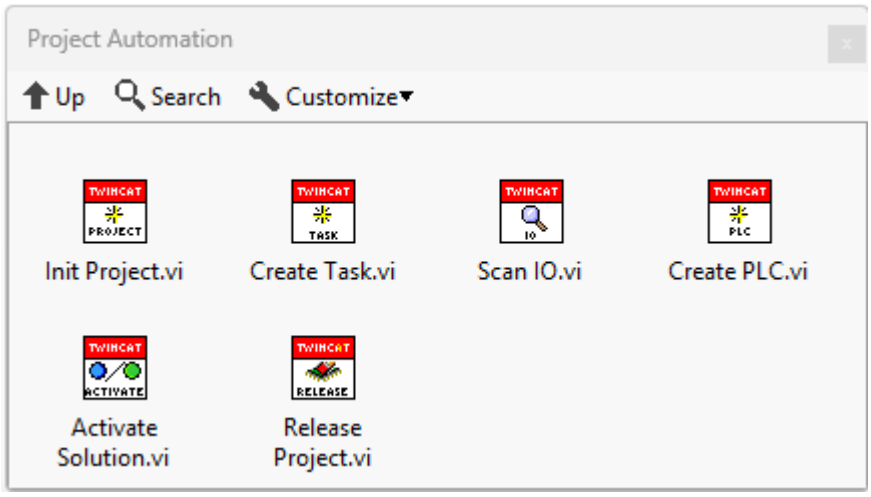
Heartbeat connection monitoring is carried out continuously until it is stopped via a control. If the connection monitoring detects an interruption (timeout), an attempt is made to call up a VI (vi path timeout detected). If it is not possible to call the VI, the block and Heartbeat connection monitoring are stopped. At the output of the VI, the target for which an interruption was detected is then displayed via Timeout detected and the TargetAddr. If there are several interruptions, only the first interruption is displayed.



Input/output	Meaning
[0] [4] Handle	Handle to the ADS client
[1] Stop Heartbeat	Reference to a Boolean control that is used to end the execution of the Heartbeat VI.
[2] Path to VI called at a timeout	Path to a VI that is to be called if a connection interruption is detected.
[5] Polling time in ms	Parameter Polling time in ms
[6] Target address	String that specifies the target to which the possible timeout belongs.
[7] Use RTD estimation?	Parameter UseRtdEstimation
[9] Offset in ms	Parameter offset
[10] Timeout detected?	Indicates whether a connection interruption was detected.

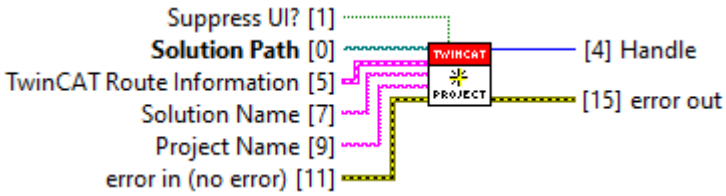
6.10.5 Project Automation

The Project Automation section contains VIs that can be used to create, configure and activate a TwinCAT 3 project. The associated example is particularly suitable if you have no previous knowledge of TwinCAT 3. The folder contains the following blocks:



Init Project

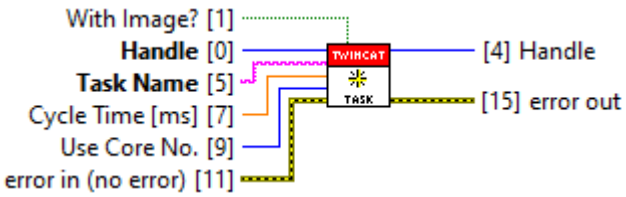
The VI **Init Project** initializes a TwinCAT 3 Engineering project. If there is no solution file (*.sln) and no TwinCAT 3 project of the same name under the specified path, these are created, otherwise the existing ones are called. Each call of the block opens a new TwinCAT 3 XAE Shell, unless this is prevented via the *Suppress UI* parameter.



Input/output	Meaning
[0] Solution Path	An absolute path that specifies where the solution file should be stored or from where it should be opened. If the path is empty, the LabVIEW™ Data directory is selected.
[1] Suppress UI?	False (default): Open new XAE Shell. True: Prevents the opening of a new XAE Shell.
[4] Handle	The handle to the TwinCAT 3 Solution
[5] TwinCAT Route Information	LabVIEW™ cluster consisting of the following elements: • Route Identification: Target system designation consisting of: host name, IP address or AMS NetId • Route Authentification: Route authentication consisting of: user name and password
[7] Solution Name	Name of the solution file. If the name does not yet exist in the directory, a new solution is created; if the file already exists, it is used.
[9] Project Name	Name of the new TwinCAT 3 project. If the project already exists, no new one is created.

Create Task

The **Create Task** block initializes a TwinCAT 3 task with a name. If the task already exists in the project tree, the existing one is used.



Input/output	Meaning
[0] [4] Handle	Handle to the TwinCAT 3 Solution
[1] With Image?	True: Create TwinCAT 3 task with image, i.e. the task can contain one or more inputs/outputs that can be linked to real inputs/outputs.
[5] Task Name	The name of the task
[7] Cycle Time [ms]	The cycle time of the task in milliseconds
[9] Use Core No.	The number of the CPU core on which the task is executed.

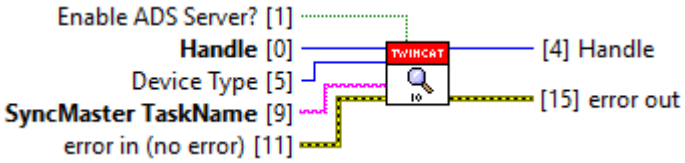
Isolated and split cores

Do one or more isolated cores already exist? These are then preferred over split cores.

Scan IO

The **Scan IO** block checks the target system for connected I/O devices (EtherCAT master or EtherCAT slave) in the network and adds them to the project tree. The following is taken into account:

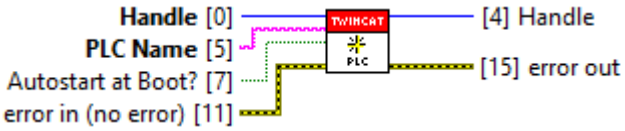
- If an EtherCAT master already exists in the project tree, the block checks whether the configuration of the master or the terminals has been changed.
- If there is a second EtherCAT master that is not yet present in the project tree, it is added to the project tree with the associated terminal configuration.
- If the route is changed that is not connected to any I/O devices but there are unnecessary I/O devices in the project tree, they are deleted.



Input/output	Meaning
[0][4] Handle	The handle to the TwinCAT 3 Solution
[1] Enable ADS Server?	True: Enables the ADS server for the process image so that all inputs and outputs can be accessed via ADS. This can be used, for example, to read terminal values with the TF3710.
[5] Device Type	LabVIEW™ Enum consisting of two elements: • EtherCATMaster: Check new or existing EtherCAT masters and their terminal configuration • EtherCATSlave: Check changed terminal configuration of existing masters
[9] SyncMaster TaskName	Name of the task that is responsible for the cyclic execution of the EtherCAT frame.

Create PLC

The block creates a new PLC project within the TwinCAT 3 project. If a PLC with the same name already exists in the directory, it will be used.



Input/output	Meaning
[0] [4] Handle	Handle to the TwinCAT 3 Solution
[5] PLC Name	The name of the TwinCAT 3 PLC project
[7] Autostart at Boot?	True: After restarting the computer, the PLC program is started automatically.

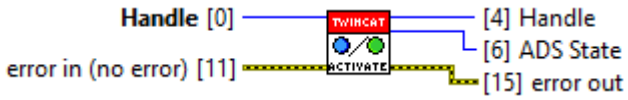
Activate Solution

The block activates the TwinCAT configuration and switches from Config to Run Mode.

⚠

Controller is activated

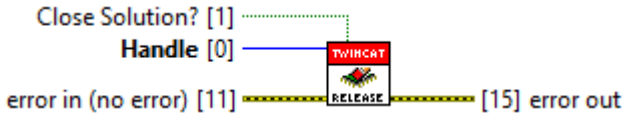
The controller is activated and the program code is executed by the hardware.



Input/output	Meaning
[0] [4] Handle	Handle to the TwinCAT 3 Solution
[6] ADS State	LabVIEW™ enum: Describes the current state of the TwinCAT system

Release Project

The block releases the handle to the TwinCAT 3 Solution from the memory.



Input/output	Meaning
[0] Handle	Handle to the TwinCAT 3 Solution
[1] Close Solution?	True: Close the TwinCAT 3 XAE Shell

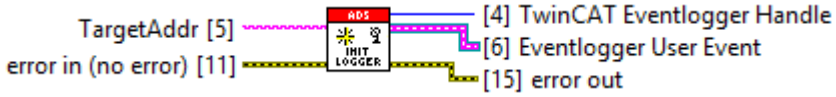
6.10.6 TwinCAT 3 EventLogger Interface

The TwinCAT 3 EventLogger technology facilitates the exchange of events between various TwinCAT components and other systems. The events are created and triggered in the TwinCAT system. The LabVIEW™ blocks enable current TwinCAT events to be received or logged TwinCAT events to be read from the cache of the TwinCAT target system. The target system is identified via the AMS NetId. The read events can be of the *message* or *alarm* type. The main difference between the event types is that messages are stateless, while alarms have the following states: `Not-Raised`, `Raised`, (optional) `WaitForConfirmation` and (optional) `Confirmed` or `Reset`. For more detailed information on the TwinCAT 3 EventLogger, as well as for the TwinCAT-side implementation, see "Technical introduction" of the TE1000 | TwinCAT 3 EventLogger.

The example TwinCAT 3 EventLogger Interface.vi describes the use of the TwinCAT EventLogger blocks.

Init TwinCAT EventLogger

The VI Init TwinCAT EventLogger initializes the TwinCAT 3 EventLogger interface in LabVIEW™ for a target system. An EventLogger User Event is also provided, which enables newly generated events to be received dynamically. The received User Event must be released by the user after the release of the TwinCAT 3 EventLogger Handle (see below “Release TwinCAT Eventlogger”).



Input/output	Meaning
[4] TwinCAT EventLogger Handle	Handle to the TwinCAT EventLogger
[5] TargetAddr	AMS NetID of the TwinCAT target system
[6] EventLogger User Event	User Event reference number to the EventLogger event generated by Init

Run TwinCAT EventLogger

The VI Run TwinCAT EventLogger starts the TwinCAT 3 EventLogger interface. This block is executed until the TwinCAT 3 EventLogger interface is stopped (see “Stop TwinCAT EventLogger” as follows).



Input/output	Meaning
[0][4] TwinCAT EventLogger Handle	Handle to the TwinCAT EventLogger

Stop TwinCAT EventLogger

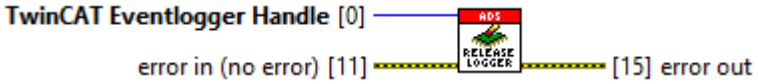
The VI Stop TwinCAT EventLogger stops the retrieval of TwinCAT 3 Eventlogs.



Input/output	Meaning
[0][4] TwinCAT EventLogger Handle	Handle to the TwinCAT EventLogger

Release TwinCAT EventLogger

The VI Release TwinCAT EventLogger releases the handle to the TwinCAT 3 EventLogger from the memory.



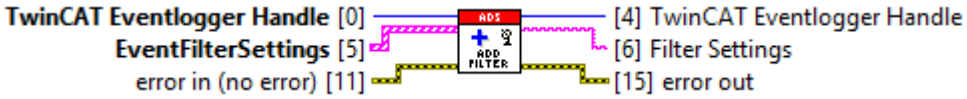
Input/output	Meaning
[0] TwinCAT EventLogger Handle	Handle to the TwinCAT EventLogger

TwinCAT EventLogger Add Filter

The VI TwinCAT EventLogger Add Filter adds a filter to the TwinCAT 3 EventLogger interface. The VI is a polymorphic VI. This allows flexible use of the VI depending on the filter to be added. If several filters are to be added, the "FilterSettings" version of the VI can be used (alternatively, the VI can also be called several times sequentially). A description of the corresponding versions of the VI can be found in the LabVIEW™ Help window for the corresponding VIs.

Availability (from TwinCAT 3.1 Build 4026)

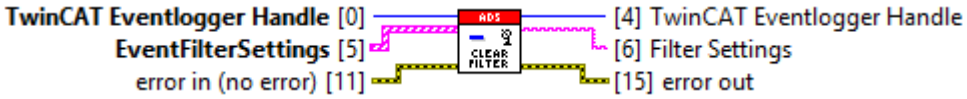
The EventLogger filters are available for target systems from TwinCAT 3.1 Build 4026.



Input/output	Meaning
[0][4] TwinCAT EventLogger Handle	Handle to the TwinCAT EventLogger
[5] EventFilterSettings	This input depends on the selected option of the polymorphic VI.
[6] Filter Settings	String that describes the applied Filter Settings (in the form (COND1 AND COND2 AND ...). Can be used to check whether the desired filters are applied.

TwinCAT EventLogger Clear Filter

The VI TwinCAT EventLogger Clear Filter removes a filter from the TwinCAT 3 EventLogger interface. The VI is a polymorphic VI. This allows the VI to be used flexibly depending on the filter to be removed. If you want to remove all filters, use the "All" option of the VI. A description of the corresponding options can be found in the LabVIEW™ Help window.



Input/output	Meaning
[0][4] TwinCAT EventLogger Handle	Handle to the TwinCAT EventLogger
[5] EventFilterSettings	This input depends on the selected option of the polymorphic VI.
[6] Filter Settings	String that describes the applied Filter Settings (in the form (COND1 AND COND2 AND ...). Can be used to check whether the desired filters are applied.

TwinCAT EventLogger Get Filter Settings

The VI TwinCAT EventLogger Get Filter Settings reads the currently applied filter settings from the TwinCAT 3 EventLogger interface.



Input/output	Meaning
[0][4] TwinCAT EventLogger Handle	Handle to the TwinCAT EventLogger
[6] Filter Settings	String that describes the applied Filter Settings (in the form (COND1 AND COND2 AND ...). Can be used to check whether the desired filters are applied.

TwinCAT EventLogger Get Logged Events

The VI TwinCAT EventLogger Get Logged Events reads the past logged events from the configured TwinCAT target system. The configured filter settings are taken into account.



Input/output	Meaning
[0][4] TwinCAT EventLogger Handle	Handle to the TwinCAT EventLogger
[5] Max entries	Maximum number of events to be read
[6] Logged Events	Array containing the logged events

TwinCAT EventLogger Get Active Alarms

The VI TwinCAT EventLogger Get Active Alarms reads the active alarms from the configured TwinCAT target system. The unused filter settings are taken into account here.



Input/output	Meaning
[0][4] TwinCAT EventLogger Handle	Handle to the TwinCAT EventLogger

Input/output	Meaning
[6] Active Alarms	Array containing the active alarms

TwinCAT EventLogger Clear Logged Events Cache

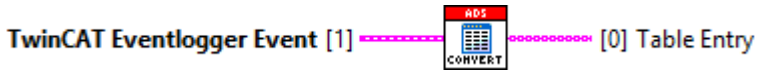
The VI Clear Logged Events Cache clears the Event Cache on the configured TwinCAT target system. This operation cannot be undone and all events in the cache are deleted.



Input/output	Meaning
[0][4] TwinCAT EventLogger Handle	Handle to the TwinCAT EventLogger

TwinCAT EventLogger Event to Table Entry

The VI TwinCAT EventLogger Event to Table Entry converts a structure of the type "TwinCAT EventLogger Event" into a table entry. All elements of the structure are converted into strings and merged into a string array.



Input/output	Meaning
[1] TwinCAT EventLogger Event	Event data of type "TwinCAT EventLogger Event"
[0] Table Entry	String array containing the respective elements of the TwinCAT EventLogger event as a string.

6.11 Low-Level

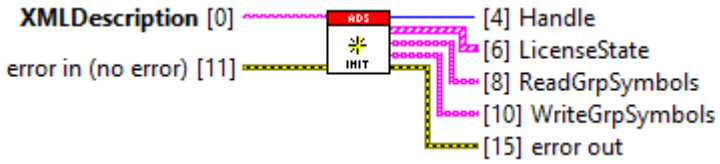
Low-Level VIs [▶ 40] can be used to provide more programming flexibility in certain situations, or to increase read or write speed.

6.11.1 Init

The *Init* subfolder contains low-level blocks that are necessary for initializing and connecting the ADS client.

Base Init

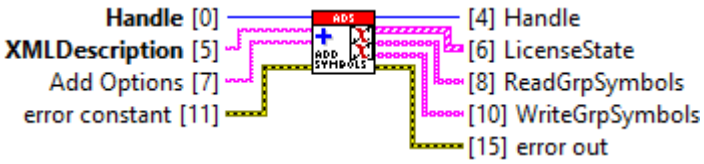
The *Base Init* block initializes and establishes a connection between ADS client and ADS router. The block checks the licenses on the target systems after a successful connection. After successful verification the block returns a valid handle to the ADS client.



Input/output	Meaning
[0] XML Description	LabVIEW™ XML string with ADS read and write symbols **or** the path as a string to an existing, already created (exported) XML file.
[4] Handle	Handle to the ADS client
[6] LicenseState	List of license states of the TwinCAT target systems
[8] ReadGrpSymbols	List of symbols that are read via ADS.
[10] WriteGrpSymbols	List of symbols that are written via ADS.

AddWrite Symbols

The *Add ReadWrite Symbols* block adds further symbols to the Client Handle. If new TwinCAT target systems are added, the block checks whether a valid license is available for each of the target systems. The result can be checked via the *LicenseState* output.



Input/output	Meaning
[0] [4] Handle	Handle to the ADS client
[5] XMLDescription	LabVIEW™ XML string with symbols to be read or written, **or** string of the path to an existing XML file.
[6] LicenseState	List of license states of the TwinCAT target systems
[7] Add Options	Options for dealing with multiple symbols: - NoDuplicateDefault: Checks whether a symbol with the same **name, symbol name and symbol type** already exists**.** If yes, this symbol is skipped. - DuplicateDefault: Adds the new symbols to the list without checking for duplicates. - UpdateDuplicateDefault: Checks whether a symbol with the same **name, symbol name and symbol type** already exists**.** If so, this symbol is updated. - NoDuplicateByName: Checks whether a symbol with the same **name** already exists**.** If yes, this symbol is skipped. - UpdateDuplicateByName: Checks whether a symbol with the same **name** already exists**.** If so, this symbol is updated. - ...
[8] ReadGrpSymbols	List of symbols that are read via ADS.
[10] WriteGrpSymbols	List of symbols that are written via ADS.

Get List of Registered Targets

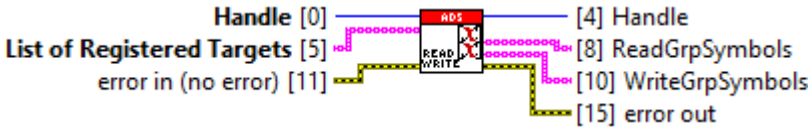
The *Get List of Registered Targets* block creates a list of ADS target systems which have been selected by the user in Symbol Interface [▶ 65] or entered in the directly loaded XML file.



Input/output	Meaning
[0] [4] Handle	Handle to the ADS client
[6] List of Registered Targets	List of registered target systems

Get List of ReadWrite Symbols

The *Get List of ReadWrite Symbols* block creates a list of registered ADS read and write symbols, which have been selected by the user in Symbol Interface [▶ 65] or entered in the directly loaded XML file.



Input/output	Meaning
[0] [4] Handle	Handle to the ADS client

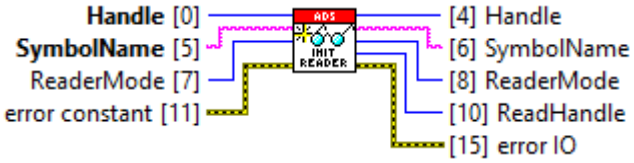
Input/output	Meaning
[5] List of Registered Targets	List of registered target systems
[8] ReadGrpSymbols	List of symbols that are read via ADS.
[10] WriteGrpSymbols	List of symbols that are written via ADS.

6.11.2 Read

The subfolder *Read* contains low-level blocks which are necessary for reading via ADS.

Init Reader

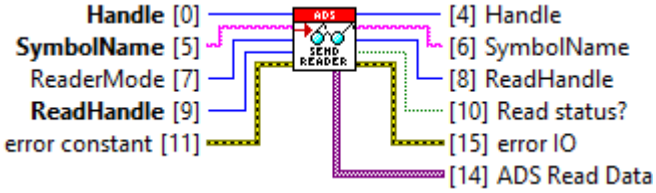
The *Init Reader* block initializes the ADS reader. When a call is successful the block returns a handle to the ADS reader.



Input/output	Meaning
[0] [4] Handle	Handle to the ADS client
[5] [6] SymbolName	Describes an ADS symbol consisting of AMS address and symbol designation: • AMS address: A compilation of AMS NetId and AMS port. • Symbol designation: Symbol name or index group/offset Examples of the symbol name: • AMS address and symbol name: 127.0.0.1.1.1:851::MAIN.Var • AMS address and index group/offset: 127.0.0.1.1.1:851::0x1234/0x12345
[7] [8] ReaderMode	The reading mode (ENUM: Sync/Async)
[10] ReadHandle	Handle to the ADS reader

Send Reader-Request

The *Send Reader-Request* block sends a read request to the ADS server. The block waits for a response from the server if the reader has been initialized with the "Synchronous" operation mode. Otherwise the block does not wait for the response and passes the ReadHandle.

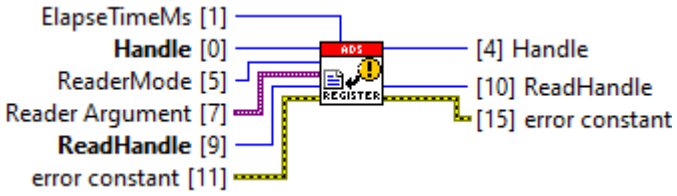


Input/output	Meaning
[0] [4] Handle	Handle to the ADS client
[5] [6] SymbolName	Describes an ADS symbol consisting of AMS address and symbol designation: • AMS address: A compilation of AMS NetId and AMS port. • Symbol designation: Symbol name or index group/offset Examples of the symbol name: • AMS address and symbol name: 127.0.0.1.1.1:851::MAIN.Var • AMS address and index group/offset: 127.0.0.1.1.1:851::0x1234/0x12345

Input/output	Meaning
[7] ReaderMode	The reading mode (ENUM: Sync/Async)
[8] [9] ReadHandle	Handle to the ADS reader
[10] Read status?	Read status
[14] ADS-Read Data	ADS raw data

Register Notification

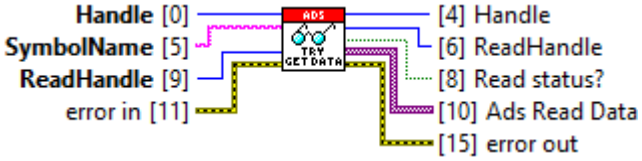
The *Register Notification* block registers the ADS notification with the ADS server and waits until the notification is explicitly unregistered from the outside. The registration does not start the notifications automatically. They have to be started explicitly.



Input/output	Meaning
[0] [4] Handle	Handle to the ADS client
[1] ElapseTimeMs	ADS symbol consisting of AMSNetId and symbol name
[5] ReaderMode	The type of reading: LabVIEW™-ENUM: • Noti. Single [▶ 68] • Noti. Buffered [▶ 69] • E-Noti. Single [▶ 69] • E-Noti. Buffered [▶ 70] • LVB-Noti. Single Symbol [▶ 71]
[7] Reader Argument	Reader arguments vary with the ReaderMode: • Noti. Single: no argument • Noti. Buffered: Single Buffer Info [▶ 79] • E-Noti. Single: LabVIEW™ Event Ref to Single User-Event Data [▶ 79] • E-Noti.Buffered: LabVIEW™ Event Ref to Buffered User-Event Data [▶ 79] • LVB-Noti.Single Symbol: Handle to LVBuffer to type (UINT32)
[9] [10] ReadHandle	Handle to the ADS reader

TryReadData

In connection with the "Asynchronous" operation mode the *TryReadData* block checks the successful receipt of a response (ADS data packet) from the ADS server.



Input/output	Meaning
[0] [4] Handle	Handle to the ADS client
[5] SymbolName	Describes an ADS symbol consisting of AMS address and symbol designation: • AMS address: a compilation of AMS NetId and AMS port. • Symbol designation: Symbol name or index group/offset

Input/output	Meaning
	Examples of the symbol name: • AMS address and symbol name: 127.0.0.0.1.1:851::MAIN.Var • AMS address and index group/offset: 127.0.0.0.1.1:851::0x1234/0x12345
[6] [9] ReadHandle	Handle to the ADS reader
[8] Read status?	Read status
[10] ADS-Read Data	ADS raw data

Release Reader

The *Release Reader* releases the handle to the reader from memory.



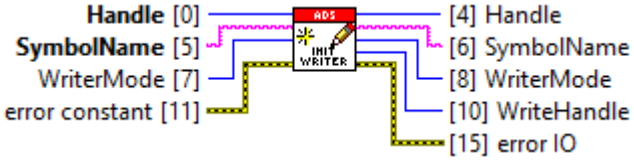
Input/output	Meaning
[0] [4] Handle	Handle to the ADS client
[5] ReadHandle	Handle to the ADS reader

6.11.3 Write

The subfolder *Write* contains low-level blocks that are necessary for writing via ADS.

Init Writer

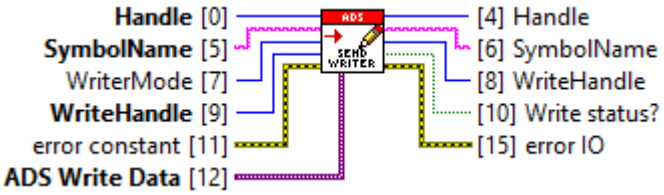
The *Init Writer* block initializes the ADS writer. When a call is successful the block returns a handle to the ADS writer.



Input/output	Meaning
[0] [4] Handle	Handle to the ADS client
[5] SymbolName	Describes an ADS symbol consisting of AMS address and symbol designation: • AMS address: a compilation of AMS NetId and AMS port. • Symbol designation: Symbol name or index group/offset Examples of the symbol name: • AMS address and symbol name: 127.0.0.0.1.1:851::MAIN.Var • AMS address and index group/offset: 127.0.0.0.1.1:851::0x1234/0x12345
[7] [8] WriterMode	Writing mode (Sync/Async)
[10] WriteHandle	Handle to the ADS Writer

Send Writer Request

The *Send Writer-Request* block sends a write request to the ADS server. The block waits for a response from the server if the writer has been initialized with the "Synchronous" operation mode. Otherwise the block does not wait for the response and passes the WriteHandle.



Input/output	Meaning
[0] [4] Handle	Handle to the ADS client
[5] [6] SymbolName	Describes an ADS symbol consisting of AMS address and symbol designation: • AMS address: A compilation of AMS NetId and AMS port. • Symbol designation: Symbol name or index group/offset Examples of the symbol name: • AMS address and symbol name: 127.0.0.1.1.1:851::MAIN.Var • AMS address and index group/offset: 127.0.0.1.1.1:851::0x1234/0x12345
[7] WriterMode	Writing mode (Sync/Async)
[8] [9] WriteHandle	Handle to the ADS writer
[12] ADS-Write Data	TypeResolved ADS data packet
[10] Write status?	Write status

CheckWriteStatus

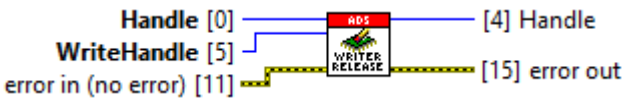
The *CheckWriteStatus* block checks for successful write access to the ADS server in connection with the "Asynchronous" operation mode.



Input/output	Meaning
[0] [4] Handle	Handle to the ADS client
[5] SymbolName	Describes an ADS symbol consisting of AMS address and symbol designation: • AMS address: a compilation of AMS NetId and AMS port. • Symbol designation: Symbol name or index group/offset Examples of the symbol name: • AMS address and symbol name: 127.0.0.0.1.1:851::MAIN.Var • AMS address and index group/offset: 127.0.0.0.1.1:851::0x1234/0x12345
[6] [9] WriteHandle	Handle to the ADS writer
[8] Write status?	Write status

Release Writer

The *Release Writer* releases the handle to the writer from memory.



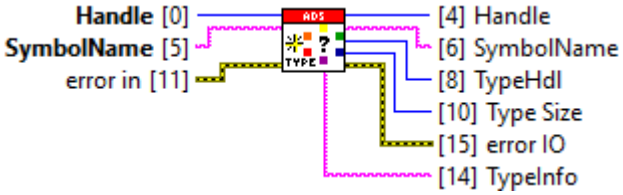
Input/output	Meaning
[0] [4] Handle	Handle to the ADS client
[5] WriteHandle	Handle to the ADS writer

6.11.4 TypeResolver

The subfolder *TypeResolver* contains low-level blocks necessary for converting and comparing between the LabVIEW™ data type and the TwinCAT 3 data type.

Init Type

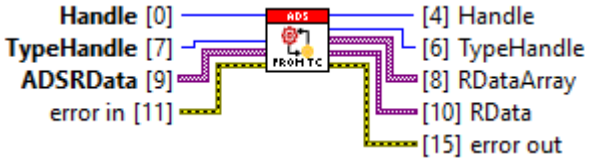
The *Init Type* block initializes the TypeResolver based on SymbolName and Handle. Upon successful initialization, the block passes the handle to the TypeResolver and the TwinCAT 3 data type of the ADS symbol as a LabVIEW™ string in XML description to the LabVIEW™ process.



Input/output	Meaning
[0] [4] Handle	Handle to the ADS client
[5] [6] SymbolName	Describes an ADS symbol consisting of AMS address and symbol designation: • AMS address: A compilation of AMS NetId and AMS port. • Symbol designation: Symbol name or index group/offset Examples of the symbol name: • AMS address and symbol name: 127.0.0.1.1.1:851::MAIN.Var • AMS address and index group/offset: 127.0.0.1.1.1:851::0x1234/0x12345
[8] TypeHdl	Handle to the TypeResolver
[10] Type Size	Data type size in bytes
[14] TypeInfo	Type Description in XML as LabVIEW™ string

Resolve From TC Type

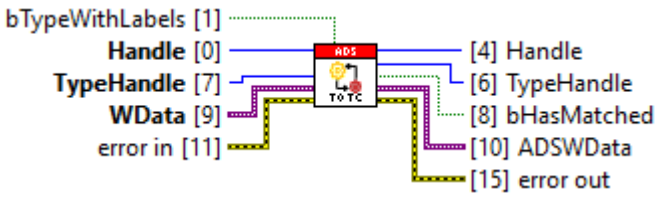
The *Resolve From TC Type* block compares and converts the raw data from the ADS read to the corresponding LabVIEW™ data type "Variant". The conversion only takes place if the comparison between the two data types was successful.



Input/output	Meaning
[0] [4] Handle	Handle to the ADS client
[6] [7] TypeHdl	Handle to the TypeResolver
[9] ADSRData	ADS data packet as raw data
[8] RDataArray	TypeResolved ADS raw data as Variant array
[10] RData	TypeResolved ADS raw data as Variant

Resolve To TC Type

The block *Resolve To TC Type* converts the raw data for ADS-Read from a LabVIEW™ data type "Variant" to the appropriate TwinCAT 3 data type. The conversion only takes place if the comparison between the two data types was successful.



Input/output	Meaning
[0] [4] Handle	Handle to the ADS client
[6] [7] TypeHdl	Handle to the TypeResolver
[9] WData	ADS data packet as raw data
[8] bHasMatched	Flag (True if TwinCAT 3 and LabVIEW™ data type are identical, otherwise False)
[10] ADSWData	TypeResolved ADS raw data as Variant

Release Type

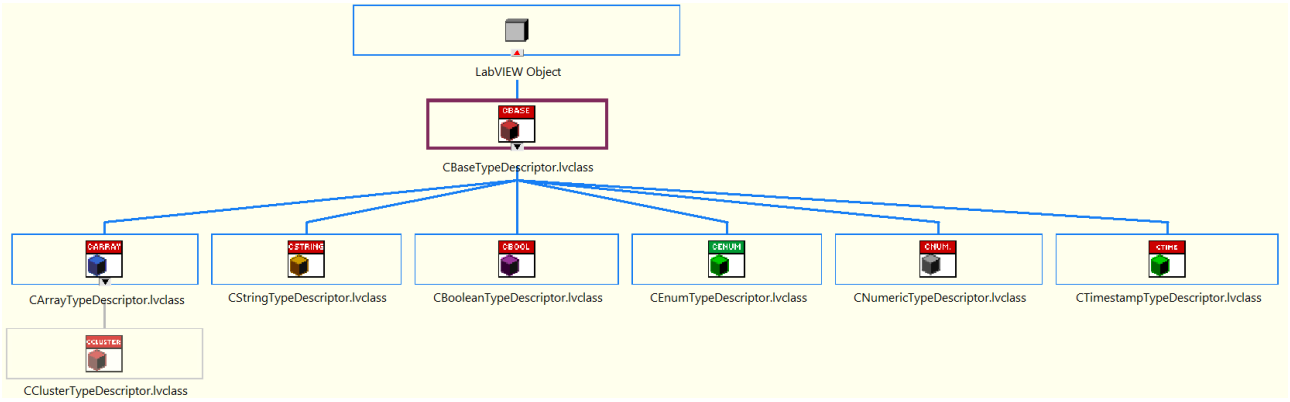
The *Release Type* releases the handle to the TypeResolver from memory.



Input/output	Meaning
[0] [4] Handle	Handle to the ADS client
[5] TypeHandle	Handle to the TypeResolver

6.11.4.1 TypeGenerator

The subfolder *TypeGenerator* contains LabVIEW™ classes that are responsible for generating a supported LabVIEW™-Data types [▶ 127]. The following graphic shows the relationship or inheritance between the *TypeGenerator* classes.

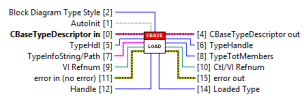
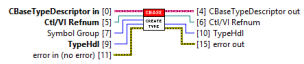
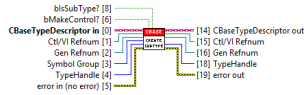


Use of the classes

The TypeGnerator classes of the respective data types were derived from the Base class and have inherited the functions. See also the information about the wrapper block Generate Type [▶ 76].

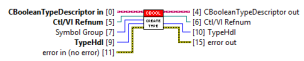
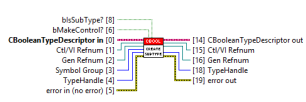
CBaseTypeDescriptor class

The CBaseTypeDescriptor class is the top class in the hierarchy. This class provides general methods and properties for the other classes that inherit from it. The table presents general information about the public properties and methods. The LabVIEW™ blocks provide further detailed information.

VI	Property/method	Access (Scope)	Meaning
	Read m_TCName	Public	Reads the property m_TCName (name of the loaded type)
	Write m_TCName	Public	Writes the property m_TCName (name of the loaded type)
	Read m_TypeCode	Public	Reads the property m_TypeCode (defines a numeric value for the loaded type)
	Read m_TypeID	Public	Reads the property m_TypeID (ID as GUID of the loaded type)
	Read m_TypeStyle	Public	Reads the property m_TypeStyle (LabVIEW™ Enum: Describes how the loaded type should be generated)
	Write m_TypeStyle	Public	Writes the property m_TypeStyle (LabVIEW™ Enum: Describes how the loaded type should be generated).
	Load	Public, Static Dispatch	Loads TwinCAT3 type information into memory based on TypeResolver or **TypeInfo.**
	Create Type	Public, Dynamic Dispatch	The method is empty. The classes that inherit from this class implement this method.
	Create SubType	Public, Dynamic Dispatch	The method is empty. The classes that inherit from this class implement this method.
	Unload and Save	Public, Static Dispatch	Unloads the loaded type from memory and saves the new type (if Ctl).

CBooleanTypeDescriptor class

The CBooleanTypeDescriptor class generates a Boolean type as a constant or control (in a block diagram) or as a ctl on the hard disk. The table presents general information about public methods. The LabVIEW™ blocks provide further detailed information.

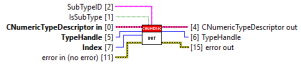
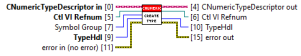
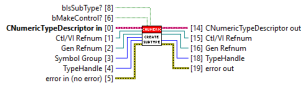
	Method	Access (Scope)	Meaning
	Init	Public, Static Dispatch	Initializes a Boolean type based on the loaded TypeInfo.
	Create Type	Public, Static Dispatch	Generates a Boolean type based on the loaded TwinCAT 3 type information.
	Create SubType	Public, Static Dispatch	Generates a Boolean type as a sub-type, e.g. the element of a cluster.

IEC 61131-3 BIT data type

The CBooleanTypeDescriptor class supports BOOL and BIT data types and can therefore be used to generate both.

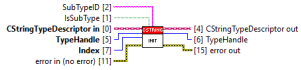
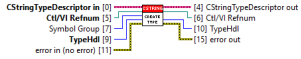
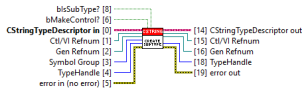
CNumericTypeDescriptor class

The CNumericTypeDescriptor class generates a numeric type with proper representation (I8, I16, I32, Single, ...) as constant or controls (in a block diagram) or as ctl on the hard disk. The table presents general information about public methods. The LabVIEW™ blocks provide further detailed information.

	Method	Access (Scope)	Meaning
	Init	Public, Static Dispatch	Initializes the numeric type, based on the loaded TypeInfo.
	Create Type	Public, Static Dispatch	Generates a numeric type based on the loaded TypeInfo.
	Create SubType	Public, Static Dispatch	Generates a numeric type as a sub-type, e.g. the element of a cluster.

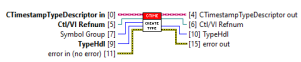
CStringTypeDescriptor class

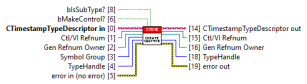
The CStringTypeDescriptor class generates a LabVIEW™ string as a constant or control (in a block diagram) or as a ctl on the hard disk. The table presents general information about public methods. The LabVIEW™ blocks provide further detailed information.

	Property/method	Access (Scope)	Meaning
	Read m_Length	Public	Reads the property m_Length (length of the TwinCAT 3 string of the loaded TypeInfo).
	Init	Public, Static Dispatch	Initializes a LabVIEW™ string based on the loaded TypeInfo.
	Create Type	Public, Static Dispatch	Generates a LabVIEW™ string based on loaded TypeInfo.
	Create SubType	Public, Static Dispatch	Generates a LabVIEW™ string as a sub-type, e.g. an element of a LabVIEW™ cluster.

CTimestampTypeDescriptor class

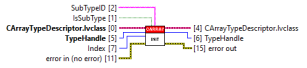
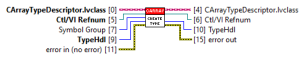
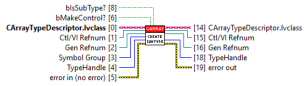
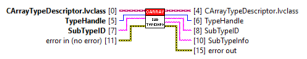
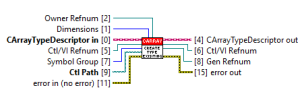
The CTimestampTypeDescriptor class generates a timestamp LabVIEW™ type as a constant or control (in a block diagram) or as a ctl on the hard disk. The table presents general information about public methods. The LabVIEW™ blocks provide further detailed information.

	Method	Access (Scope)	Meaning
	Init	Public, Static Dispatch	Initializes the timestamp based on the loaded TypeInfo.
	Create Type	Public, Static Dispatch	Generates a timestamp based on the loaded TypeInfo.

	Method	Access (Scope)	Meaning
	Create SubType	Public, Static Dispatch	Generates a timestamp as a sub-type, e.g. an element of a LabVIEW™ cluster.

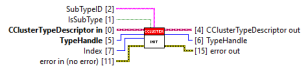
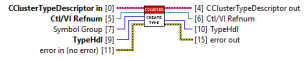
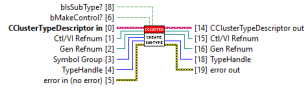
CArrayTypeDescriptor class

The CArrayTypeDescriptor class generates an array LabVIEW™ type as a constant or control (in a block diagram) or as a ctl on the hard disk. The table presents general information about public methods. The LabVIEW™ blocks provide further detailed information.

	Property/method	Access (Scope)	Meaning
	Read m_ArrayInfo	Public	Reads the property m_ArrayInfo (information regarding TwinCAT 3 Array Dimension elements)
	Read m_BaseTypeID	Public	Reads the property m_BaseTypeID (ID as GUID of the loaded base type)
	Read m_Dimensions	Public	Reads the property m_Dimensions (number of TwinCAT 3 Array Dimensions)
	Read m_SubTypes	Public	Reads the property m_SubTypes (for arrays of LabVIEW™ clusters, this property specifies the number of cluster elements generated)
	Init	Public, Static Dispatch	Initializes the array based on the loaded TypeInfo.
	Create Type	Public, Static Dispatch	Generates an array based on the loaded TypeInfo.
	Create SubType	Public, Static Dispatch	Generates an array as a sub-type, e.g. an element of a LabVIEW™ cluster.
	Get Tot SubTypes	Public, Static Dispatch	Returns information about the number of complex sub-types of the array and their IDs as GUID (e.g.: array with complex BaseType, where the BaseType contains further arrays with complex BaseTypes).
	Get Sub-TypeInfo	Public, Static Dispatch	Reads TypeInfo of the complex BaseType or SubType with entered ID as GUID.
	Create Type From Existing BaseType	Public, Static Dispatch	Generates an array of a ctl from a path.

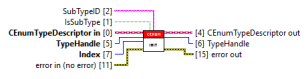
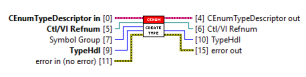
CClusterTypeDescriptor class

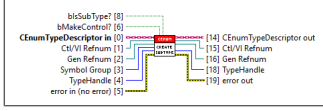
The CClusterTypeDescriptor class generates a cluster LabVIEW™ type as a constant or control (in a block diagram) or as a ctl on the hard disk. The table presents general information about public methods. The LabVIEW™ blocks provide further detailed information.

	Property/method	Access (Scope)	Meaning
	Read m_SubItems	Public	Reads the property m_SubItems (the number of elements in the current cluster).
	Read m_GeneratedSubItems	Public	Reads the property m_GeneratedSubItems (the number of generated elements in the current cluster).
	Read m_NestedLevel	Public	Reads the property m_NestedLevel (the nested level of the current cluster).
	Read m_RemainingSubItems	Public	Reads the property m_RemainingSubItems (the number of elements of the current cluster that are yet to be generated).
	Read m_ActualPosition	Public	Reads the property m_ActualPosition (iterator of the SubTypes in the cluster).
	InitC	Public, Static Dispatch	Initializes the clusters, based on the loaded TypeInfo.
	Create Type	Public, Static Dispatch	Generates a cluster based on the loaded TypeInfo.
	Create SubType	Public, Static Dispatch	Generates a cluster as a sub-type, e.g. an element of a LabVIEW™ cluster.

CEnumTypeDescriptor class

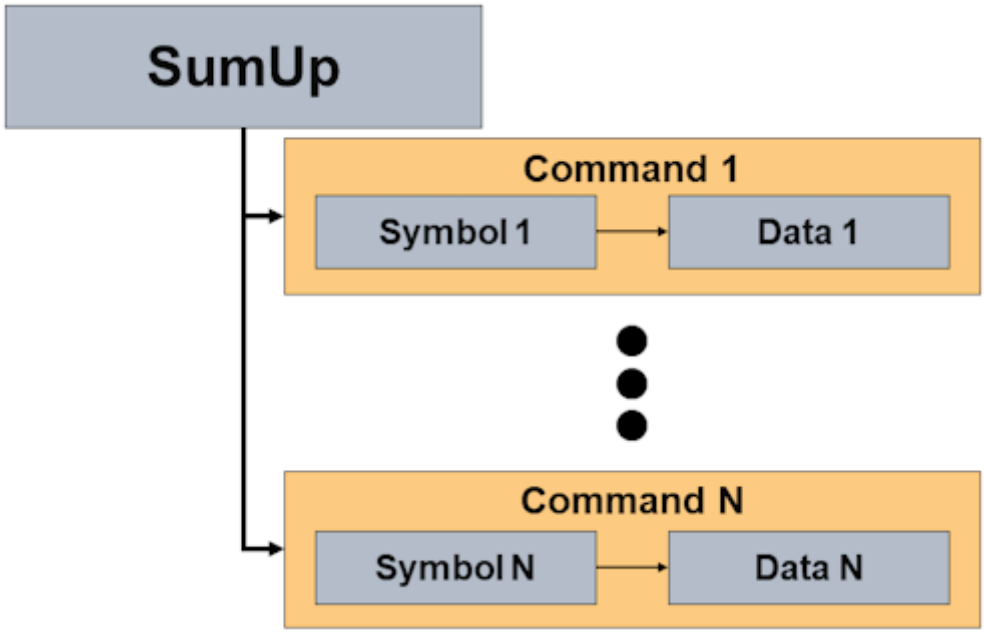
The CEnumTypeDescriptor class generates an Enum LabVIEW™ type as a constant or control (in a block diagram) or as a ctl on the hard disk. The table presents general information about public methods. The LabVIEW™ blocks provide further detailed information.

	Property/method	Access (Scope)	Meaning
	Read m_Keys	Public	Reads the property m_Keys (All text elements of the Enum).
	Read m_Representation	Public	Reads the property m_Representation (The underlying memory of the Enum).
	Init	Public, Static Dispatch	Initializes the Enum, based on the loaded TypeInfo.
	Create Type	Public, Static Dispatch	Generates an Enum based on the loaded TypeInfo.

	Property/method	Access (Scope)	Meaning
	Create SubType	Public, Static Dispatch	Generates an Enum as a sub-type, e.g. an element of a LabVIEW™ cluster.

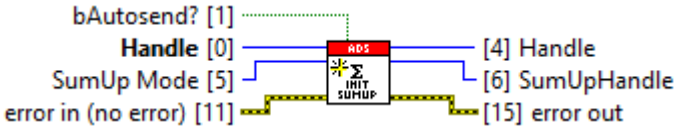
6.11.5 SumUp

The *SumUp* subfolder contains low-level blocks that allow writing or reading multiple ADS symbols with one API call. Compared to simple ADS Read or Write statements, only one handle is needed for writing or reading multiple symbols with the SumUp. The handle can in turn contain several subcommands that are sent simultaneously to the TwinCAT 3 Runtime.



Init SumUp

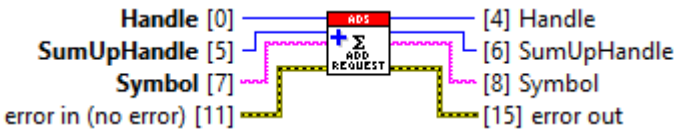
The block *Init SumUp* initializes the ADS SumUp. If the call is successful, the block returns a handle to the ADS SumUp.



Input/output	Meaning
[0] [4] Handle	Handle to the ADS client
[1] bAutosend?	Autosend flag enables automatic sending of the SubCommand
[5] SumUp mode	SumUp modes: • Write • Read
[6] SumUp handle	Handle on the SumUp

Add SubCommand

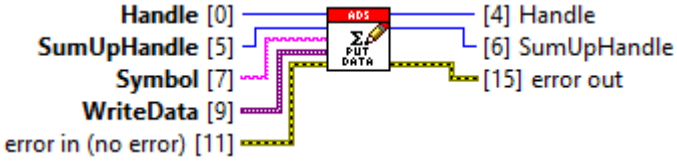
The block *Add SubCommand* initializes a new subcommand and adds it to the SumUp handle.



Input/output	Meaning
[0] [4] Handle	Handle to the ADS client
[5] [6] SumUpHandle	Handle on the SumUp
[7] [8] Symbol	ADS symbol consisting of AMS address and symbol designation: • AMS address: a compilation of AMS NetId and AMS port. • Symbol designation: Symbol name or index group/offset Examples of the symbol name: • AMS address and symbol name: 127.0.0.0.1.1:851::MAIN.Var • AMS address and index group/offset: 127.0.0.0.1.1:851::0x1234/0x12345

Put Data

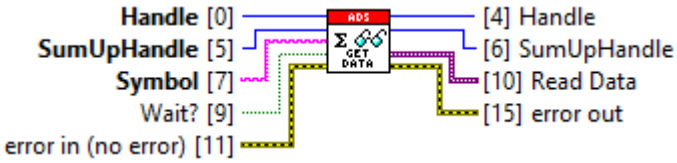
The block *Put Data* adds new data to the initialized sub-command. For this, the block needs the symbol name to identify the sub-command. The block can only be used for writing SumUp commands.



Input/output	Meaning
[0] [4] Handle	Handle to the ADS client
[7] Symbol	ADS symbol consisting of AMS address and symbol designation: • AMS address: a compilation of AMS NetId and AMS port. • Symbol designation: Symbol name or index group/offset Examples of the symbol name: • AMS address and symbol name: 127.0.0.0.1.1:851::MAIN.Var • AMS address and index group/offset: 127.0.0.0.1.1:851::0x1234/0x12345
[9] WriteData	Data that is to be written

Get Data

The block *Get Data* retrieves new data from the initialized sub-command. For this, the block needs the symbol name to identify the sub-command. The block can only be used for reading SumUp commands.

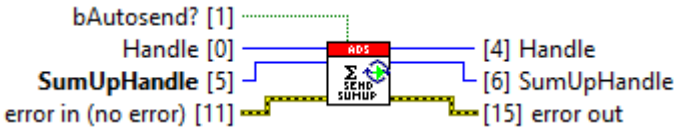


Input/output	Meaning
[0] [4] Handle	Handle to the ADS client
[5] [6] SumUpHandle	Handle on the SumUp
[7] Symbol	ADS symbol consisting of AMS address and symbol designation: • AMS address: a compilation of AMS NetId and AMS port. • Symbol designation: Symbol name or index group/offset Examples of the symbol name: • AMS address and symbol name: 127.0.0.0.1.1:851::MAIN.Var • AMS address and index group/offset: 127.0.0.0.1.1:851::0x1234/0x12345
[9] Wait?	Wait flag that defines whether to wait for the data to be read.

Input/output	Meaning
	• True: The block waits until the timeout to see if the new data has come in. • False: The block does not wait for the new data.
[15] Read Data	The data to be read

Enable Autosend

The block *Enable Autosend* enables the automatic sending of the SumUp command.



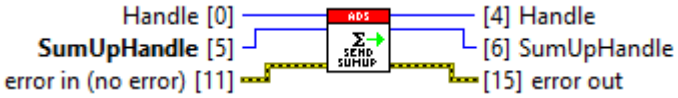
Sub-commands without data

Automatic sending fails for initialized sub-commands that do not contain any data.

Input/output	Meaning
[0] [4] Handle	Handle to the ADS client
[1] bAutosend	The Autosend flag enables automatic sending of the SubCommand
[5] [6] SumUpHandle	Handle on the SumUp

Send

The *Send* block sends the sub-commands added to the SumUp handle to the TwinCAT 3 runtime. In contrast to Enable Autosend [▶ 106], the Send SumUp must be explicitly called cyclically to send the data to TwinCAT.



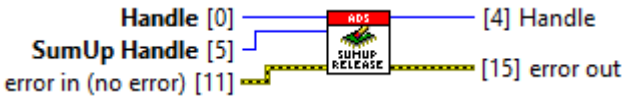
Sub-commands without data

Sending fails for initialized sub-commands that do not contain any data.

Input/output	Meaning
[0] [4] Handle	Handle to the ADS client
[5] [6] SumUpHandle	Handle on the SumUp

Release SumUp

The block *Release SumUp* releases the SumUp handle from memory.



Input/output	Meaning
[0] [4] Handle	Handle to the ADS client
[5] SumUp handle	Handle on the SumUp

6.12 With TypeResolving

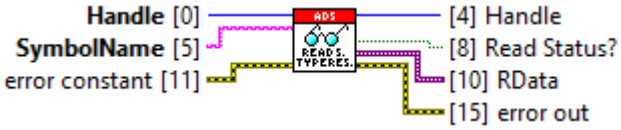
The folder *With TypeResolving* contains blocks that integrate reading and writing via ADS with TypeResolver and, accordingly, further simplify programming in LabVIEW™.

Read TypeResolved

The *Read TypeResolved* block is a polymorphic block that integrates reading via ADS with TypeResolver. The block offers synchronous and asynchronous reading via ADS.

Read Sync Single TypeResolved

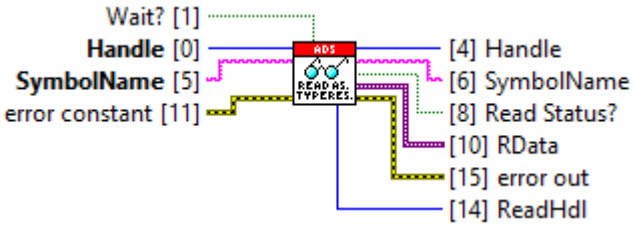
The Read TypeResolved block in operation mode *Read Sync Single TypeResolved* calls the ADS-Read [▶ 67] block to receive the ADS data packet (as ADS raw data) **synchronously** from the ADS server and then convert it into a LabVIEW™ data type using the Get TypeResolver [▶ 74] block.



Input/output	Meaning
[0] [4] Handle	Handle to the ADS client
[5] SymbolName	ADS symbol consisting of AMSNetId and symbol name
[8] Read status?	Read status
[10] RData	TypeResolved ADS raw data as Variant

Read Async Single TypeResolved

In the operation mode *Read Async Single TypeResolved*, the Read TypeResolved block calls the Async Single [▶ 68] block to receive the ADS data packet (as ADS raw data) **asynchronously** from the ADS server and then convert it to a LabVIEW™ data type using the Get TypeResolver [▶ 74] block.



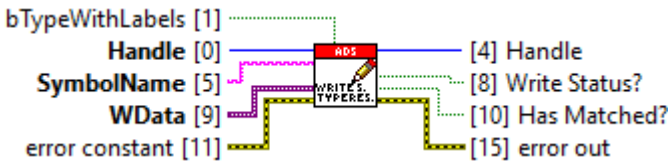
Input/output	Meaning
[0] [4] Handle	Handle to the ADS client
[1] Wait?	TRUE = wait for server response (sync) FALSE (default) = do not wait for server response (async)
[5] [6] SymbolName	ADS symbol consisting of AMSNetId and symbol name
[8] Read status?	Read status
[10] RData	TypeResolved ADS raw data as Variant
[14] ReadHdl	Handle to the ADS reader

Write TypeResolved

The *Write TypeResolved* block is a polymorphic block that integrates writing via ADS with TypeResolver. The block offers synchronous and asynchronous writing via ADS.

Write Sync Single TypeResolved

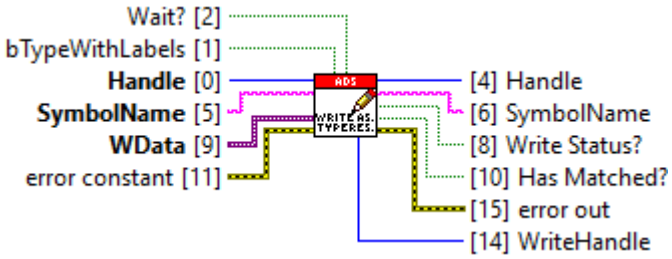
The Write TypeResolved block in operation mode *Write Sync Single TypeResolved* calls the Set TypeResolver [▶ 74] block to convert the LabVIEW™ data type to a TwinCAT 3 data type and then sends the converted data packet to the ADS server **synchronously** by calling the Sync Single [▶ 73] block.



Input/output	Meaning
[1] bTypeWithLabels	True if the data type has a label.
[0] [4] Handle	Handle to the ADS client
[5] SymbolName	ADS symbol consisting of AMSNetId and symbol name
[9] WData	ADS data packet as raw data
[8] Write Status?	Write status
[10] Has Matched?	Flag indicating whether conversion and comparison between LabVIEW™ and TwinCAT 3 data type was successful.

Write Async Single TypeResolved

In operation mode *Write Async Single TypeResolved*, the Write TypeResolved block calls the Set TypeResolver [▶ 74] block to convert the LabVIEW™ data type to a TwinCAT 3 data type and then sends the converted data packet **asynchronously** to the ADS server with the call from the Sync Single [▶ 73] block.



Input/output	Meaning
[1] bTypeWithLabels	True if the data type has a label.

[0] [4] Handle	Handle to the ADS client
[1] Wait?	TRUE = wait for server response (sync) FALSE (default) = do not wait for server response (async)
[5] [6] SymbolName	ADS symbol consisting of AMSNetId and symbol name
[9] WData	ADS data packet as raw data
[8] Write Status?	Write status
[10] Has Matched?	Flag indicating whether conversion and comparison between LabVIEW™ and TwinCAT 3 data type was successful.
[14] WriteHandle	Handle to the ADS writer

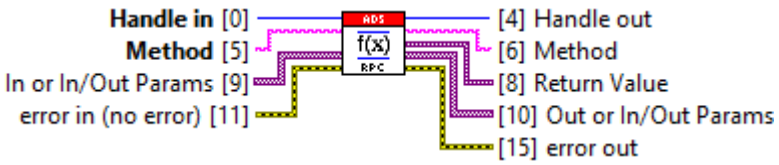
6.13 ADS Remote Procedure Call (RPC)

The **ADS RPC** block can be used to call TwinCAT 3 methods. The methods must be declared accordingly and are executed when called in the TwinCAT runtime. After execution, the return values of the function and/or function parameters are returned to LabVIEW™.

⚠

Access to the control

The methods are executed in runtime. When using the real-time runtime, the cycle time must be selected sufficiently.



Input/output	Meaning
Handle	Handle to the ADS client
Method	Describes an ADS method consisting of the AMS address, the name of the function block and its method: • AMS address: a compilation of AMS NetId and AMS port. • Name of the function block • Name of the method e.g.: 127.0.0.1.1.1:851::MAIN.fbTest#Method1
In or In/Out Params	LabVIEW™ Variant array with all parameters of the method that have the attribute TcRpcDirection := 'in' or TcRpcDirection := 'inout'.
Out or In/Out Params	LabVIEW™ Variant array with all parameters of the method that have the attribute TcRpcDirection := 'out' or TcRpcDirection := 'inout'.
Return value	LabVIEW™ Variant as return value from the method, if available. Remains empty for methods without a return value (void).

In TwinCAT 3, there are many ways to instantiate/define a method, such as in function blocks within the PLC, in an interface, or in a C++ module.

Below, we will look at calling a TwinCAT 3 PLC method called *AddNumbers*, which adds two positive integers and writes the result to an output parameter. If negative numbers are transferred, the function returns an HRESULT error.

A potential implementation of the method under a function block could look like this:

```
{attribute 'TcRpcEnable'}
METHOD AddNumbers : HRESULT
VAR_INPUT
{attribute 'TcRpcDirection' := 'in'}
Param1: INT;
{attribute 'TcRpcDirection' := 'in'}
Param2: INT;
END_VAR
VAR_OUTPUT
{attribute 'TcRpcDirection' := 'out'}
Result: INT;
END_VAR
```

```
IF Param1 < 0 OR Param2 < 0 THEN
AddNumbers := E_INVALIDARG; // HRESULT error code
Result := 0;
ELSE
Result := Param1 + Param2;
AddNumbers := S_OK; // Success
END_IF
```

A potential implementation as a C/C++ function signature could look like this:

```
HRESULT AddNumbers(int16 Param1, int16 Param2, int16& Result);
```

In this case, the ADS RPC block expects exactly two parameters of type int16, as only Param1 and Param2 have been assigned the attribute '**TcRpcDirection=in**'.

Choice of methods in the symbol interface

Methods that are defined in Function Blocks (FB) or interfaces are not displayed directly in the Symbol Interface. Only the function block or the interface is visible (e.g. FB_Test).

If the function block or the interface contains methods, these can be selected using the **TwinCAT Symbol Interface Configurator**:

1. Move symbol: Drag the function block or the interface into the Read/Write window.
2. Pop-up dialog: A dialog box appears with the message: "This symbol contains public methods. Would you like to make a choice?"
3. Confirmation: If **Yes** is selected, a second window opens for method selection.
4. Make your choice: Mark the desired methods and confirm with **OK**.

The selected methods now appear in the Read window.

Using string parameters

The direct use of string parameters can lead to initialization errors in TwinCAT 3 C++ modules. As a **best practice**, strings should be encapsulated in structured data types, e.g. structs.

7 Samples

The TF3710 TwinCAT 3 Interface for LabVIEW™ product categorizes the examples into two different groups, which are described as follows:

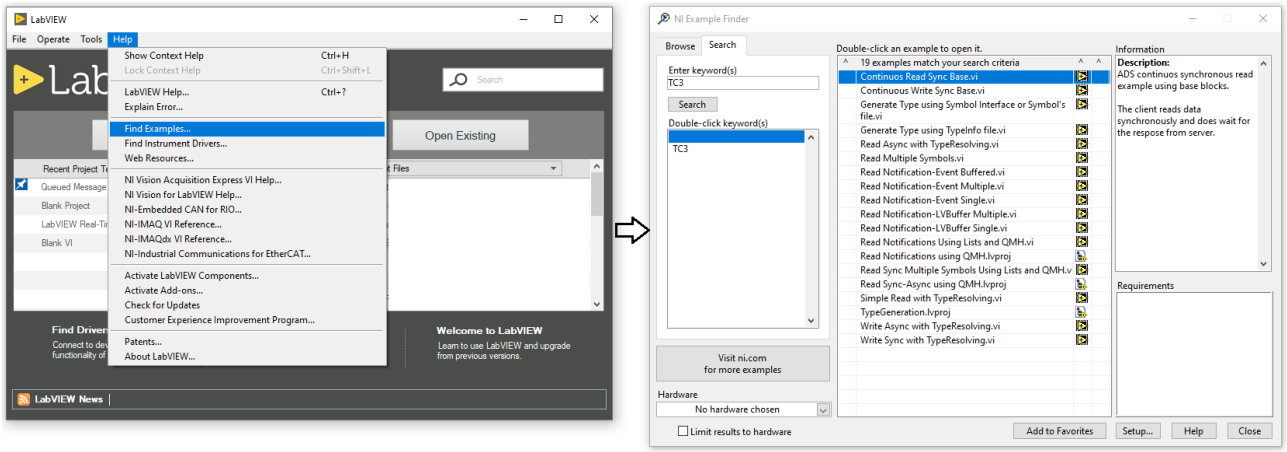
7.1 Basic examples

A large number of basic examples are included in the LabVIEW™ environment during the installation of the Interface for LabVIEW™. These can be found with the help of the **NI™ Example Finder**.

The following keywords facilitate the search for the examples:

- TC3
- Beckhoff
- Beckhoff LabVIEW™ Interface
- ADS
- TwinCAT
- TF3710

The NI™ Example Finder can be started in LabVIEW™ under **Menu > Help > Find Examples**.



Likewise, you can find all examples via the *Directory Structure* at **Beckhoff Automation > Beckhoff LabVIEW™ Interface**.

The examples are categorized into the communication modes [▶ 23] described at the beginning of the documentation:

- One-time reading
- One-time writing
- Reading data continuously
- Writing data continuously
- Generate LabVIEW™ type
- SumUp Read or Write
- CoE Read or Write

One-time reading

Simple Read with TypeResolving.vi

The example, like the other examples in Basic examples [▶ 111], uses the basic VIs from the LabVIEW™ VIs [▶ 40] in the following steps:

1. In the first step, the Symbol Interface [▶ 65] basic block is called. When the example is executed, the user interface opens and the user can select an ADS symbol in the TwinCAT runtime.

- If you have already exported an XML description of your ADS symbols from the Symbol Interface, you can select it as a path so that the XML is loaded rather than calling the UI.

2. In the second step, the Init [▶ 66] basic block is called and the ADS client is initialized.
3. The ReadGrpSymbols output may contain more than one ADS symbol (if you selected more than one symbol in step 1). In the example, *Index Array* selects the first ADS symbol in the list.
4. After initialization the polymorphic ADS-Read [▶ 67] block is called. The **Read Choice** input can be used to select which read mode is to be used. The Read Choice input is a LabVIEW™ Enum and provides the following options:
   - **Read Sync**: For this, a synchronous read request is first sent to the TwinCAT runtime and a response is waited for. The received ADS data packet is then converted from raw data to the LabVIEW™ data type "Variant" using the TypeResolver.
   - **Noti. Single**: For this, an ADS notification is registered and individual received notifications are passed on to LabVIEW™. After reading the notification, the notifications are unregistered again. The received ADS data is then converted from raw data to the LabVIEW™ data type "Variant" with the help of the TypeResolver.
   - **Noti. Buffered**: For this, an ADS notification is registered and a block of received notifications is passed on to LabVIEW™. The number of notifications buffered in the block is determined by the `LVBufferSize` symbol parameter. After reading the notifications, they are unregistered. The received ADS data is then converted from raw data to the LabVIEW™ data type "Variant" with the help of the TypeResolver.
5. **Interaction**: Variant-to-Data can be used to customize the example to translate the Variant to an appropriate data type. The data type depends on the ADS symbol selected in the Symbol Interface. See Type Resolving [▶ 35].
6. In the last step, the Release [▶ 74] basic block is called and the ADS client is released from memory.

Noti. Single

Noti. Single supports only the `Transmode`=OnChange

Read Async with TypeResolving.vi

The example, like the other examples in Basic examples [▶ 111], uses the basic VIs and the low-level VIs from LabVIEW™ VIs [▶ 40] in the following steps:

1. In the first step, the basic Symbol Interface [▶ 65] block is called. When the example is executed, the user interface opens and the user can select an ADS symbol in the TwinCAT runtime.
   - If you have already exported an XML description of the ADS symbols from the Symbol Interface, you can select it as a path so that the XML is loaded rather than calling the UI.
2. In the second step, the Init [▶ 66] basic block is called and the ADS client is initialized.
3. The ReadGrpSymbols output may contain more than one ADS symbol (if you selected more than one symbol in step 1). In the example, *Index Array* selects the first ADS symbol in the list.
4. After initialization, the block Read Async Single TypeResolved [▶ 107] is called. An asynchronous read request is sent to the TwinCAT runtime for this purpose. The program code continues running without waiting for feedback.
5. In this example the low-level block TryReadData [▶ 95] is called in a while loop and waits for a response from the requested ADS server. Please note that this is an illustrative example. The implementation in actual applications may differ considerably. Once feedback has been received, the received data packet is converted to the appropriate LabVIEW™ data type "Variant".
6. **Interaction**: Variant-to-Data can be used to customize the example to translate the Variant to an appropriate data type. The data type depends on the ADS symbol selected in the Symbol Interface. See Type Resolving [▶ 35].
7. The next step is to call the low-level block Release Reader [▶ 96] to release the reader from memory.
8. In the last step the Release [▶ 74] basic block is called and thus the ADS client is released from memory.

One-time writing

Write Sync with TypeResolving.vi

The example, like the other examples in Basic examples [▶ 111], uses the basic VIs from LabVIEW™ VIs [▶ 40] in the following steps:

1. In the first step, the basic Symbol Interface [▶ 65] block is called. When the example is executed, the user interface opens and the user can select an ADS symbol in the TwinCAT runtime.
   - If you have already exported an XML description of your ADS symbols from the Symbol Interface, you can select it as a path so that the XML is loaded rather than calling the UI.
2. In the second step, the Init [▶ 66] basic block is called and the ADS client is initialized.
3. The WriteGrpSymbols output may contain more than one ADS symbol (if you selected more than one symbol in step 1). In the example, *Index Array* selects the first ADS symbol in the list.
4. **Interaction**: Insert the appropriate data type in the place marked in red. The data type depends on the ADS symbol selected in the Symbol Interface. See Data types [▶ 127].
5. After initialization, the block Write Sync Single TypeResolved [▶ 107] is called. For this purpose, the raw data is first converted from the LabVIEW™ data type to a suitable TwinCAT 3 data type and then a synchronous write request is sent to TwinCAT. The block waits for a response from the server to ensure that the data was sent successfully.
6. In the last step the Release [▶ 74] basic block is called and thus the ADS client is released from memory.

Write Async with TypeResolving.vi

The example, like the other examples in Basic examples [▶ 111], uses the basic VIs and the low-level VIs from LabVIEW™ VIs [▶ 40] in the following steps:

1. In the first step, the basic Symbol Interface [▶ 65] block is called. When the example is executed, the user interface opens and the user can select an ADS symbol in the TwinCAT runtime.
   - If you have already exported an XML description of your ADS symbols from the Symbol Interface, you can select it as a path so that the XML is loaded rather than calling the UI.
2. In the second step, the Init [▶ 66] basic block is called and the ADS client is initialized.
3. The WriteGrpSymbols output may contain more than one ADS symbol (if you selected more than one symbol in step 1). In the example, *Index Array* selects the first ADS symbol in the list.
4. **Interaction**: Insert the appropriate data type in the place marked in red. The data type depends on the ADS symbol selected in the Symbol Interface. See Data types [▶ 127].
5. After initialization, the block Write Async Single TypeResolved [▶ 108] is called. For this purpose, first the raw data is converted from the LabVIEW™ data type to a suitable TwinCAT 3 data type and then an asynchronous write request is sent to TwinCAT. The block does not wait for a response from the ADS server.
6. In this example the low-level block CheckWriteStatus [▶ 97] is called in a While loop and waits for a response from the requested ADS server. Please note that this is an illustrative example. The implementation in actual applications may differ considerably.
7. The next step is to call the low-level block Release Writer [▶ 97] to release the writer from memory.
8. In the last step the Release [▶ 74] basic block is called and thus the ADS client is released from memory.

Reading data continuously

The examples in this group use the polling procedure or LabVIEW™ event-based procedures to request the data packets cyclically.

Continuos Read Sync Base.vi

The example, like the other examples in Basic examples [▶ 111], uses the basic VIs and the low-level VIs from LabVIEW™ VIs [▶ 40] in the following steps:

1. In the first step, the basic Symbol Interface [▶ 65] block is called. When the example is executed, the user interface opens and the user can select an ADS symbol in the TwinCAT runtime.
   - If you have already exported an XML description of your ADS symbols from the Symbol Interface, you can select it as a path so that the XML is loaded rather than calling the UI.
2. In the second step, the low-level blocks Base Init [▶ 92], Get List of ReadWrite Symbols [▶ 93] and Get List of Registered Targets [▶ 93] are called to initialize the ADS client.
3. **To speed up reading and type resolving**, the ADS Reader, with the call of Init Reader [▶ 94], and the TypeResolver, with the call of Init Type [▶ 98], are initialized in advance and only once in the next step.
4. In the next step the Send Reader-Request [▶ 94] block requests a new data packet from TwinCAT with each cycle of the loop. The received data packet is converted to a suitable LabVIEW™ data type "Variant" using the call Resolve From TC Type [▶ 98].
5. **Interaction**: Variant-to-Data can be used to customize the example to translate the Variant to an appropriate data type. The data type depends on the ADS symbol selected in the Symbol Interface. See Type Resolving [▶ 35].
6. When the termination condition for the loop is reached, the ADS reader is released from memory with the call Release Reader [▶ 96] and the TypeResolver with the call Release Type [▶ 99].
7. In the last step the Release [▶ 74] basic block is called and thus the ADS client is released from memory.

Read Notification-Event Single

The example, like the other examples in Basic examples [▶ 111], uses the basic VIs and the low-level VIs from LabVIEW™ VIs [▶ 40] in the following steps:

1. In the first step, the basic Symbol Interface [▶ 65] block is called. When the example is executed, the user interface opens and the user can select an ADS symbol in the TwinCAT runtime. When selecting, note the parameter settings of the symbols for this operation mode: E-Noti. Single [▶ 69].
   - If you have already exported an XML description of your ADS symbols from the Symbol Interface, you can select it as a path so that the XML is loaded rather than calling the UI.
2. In the second step, the Init [▶ 66] basic block is called and the ADS client is initialized.
3. The ReadGrpSymbols output may contain more than one ADS symbol (if you selected more than one symbol in step 1). In the example, *Index Array* selects the first ADS symbol in the list.
4. A User Event is created and registered. See also the LabVIEW™ documentation on user events.
5. When E-Noti. Single [▶ 69] is called, an ADS notification is registered on the ADS server for the selected ADS symbol.
6. The next step is to wait for the ADS notifications in the event structure. If no notification is received, the event structure times out. Calling the block Stop Notification [▶ 78] stops the ADS notification. The stop occurs at the ADS server so that no more messages are sent.
7. **Interaction**: Type Resolving [▶ 35] is not implemented in the example. Two options are available. The Resolving type is either realized within the event loop or after the ADS notification has been completed/unregistered. The following items in the event structure are relevant: Data and TimeStamps. The latter are the ADS timestamps.
8. When the block Unregister Notification [▶ 79] is called, the ADS notification of the symbol is unregistered at the ADS server and the handle to the ADS notification is released from memory.
9. In the last step the Release [▶ 74] basic block is called and thus the ADS client is released from memory.

Read Notification-Event Buffered.vi

The example, like the other examples in Basic examples [▶ 111], uses the basic VIs and the low-level VIs from LabVIEW™ VIs [▶ 40]. The example is identical in structure to Read Notification-Event Single [▶ 114]. The only differences are:

- The polymorphic VI is set to E-Noti. Buffered [▶ 70].
- At the Data point in the event loop an array of size `LVBufferSize` is always passed.

Read Notification-Event Multiple

The example, like the other examples in Basic examples [▶ 111], uses the basic VIs and the low-level VIs from LabVIEW™ VIs [▶ 40]. The example is identical in structure to Read Notification-Event Buffered [▶ 114]. The only differences are:

- The polymorphic VI is set to E-Noti. Multiple Symbols [▶ 71].
- For each ADS symbol a corresponding measurement duration *ElapseTimeMs* (in milliseconds) can be selected.
- Reading from each ADS symbol requires corresponding LabVIEW™ event case.

Read Notification-LVBuffer Multiple

The example, like the other examples in Basic examples [▶ 111], uses the basic VIs and the low-level VIs from LabVIEW™ VIs [▶ 40] in the following steps:

1. In the first step, the basic Symbol Interface [▶ 65] block is called. When the example is executed, the user interface opens and the user can select an ADS symbol in the TwinCAT runtime. When selecting, observe the parameter settings of the symbols for the operation mode LVB-Noti. Multiple Symbols [▶ 72].
   - If you have already exported an XML description of your ADS symbols from the Symbol Interface, you can select it as a path so that the XML is loaded rather than calling the UI.
2. In the second step, the Init [▶ 66] basic block is called and the ADS client is initialized.
3. The ReadGrpSymbols output may contain more than one ADS symbol (if you selected more than one symbol in step 1). In the example, *Array Subset* selects the first 3 ADS symbols in the list.
4. For each selected ADS symbol, a handle to the LVBuffer is initialized with the call to Init LVBuffer Handle [▶ 80].
5. When LVB-Noti. Multiple Symbols [▶ 72] is called, ADS notifications are registered on the ADS server for the selected ADS symbols. For this purpose the block takes an array of ADS symbols as input. In addition, the duration of the individual notification can be entered in milliseconds. Thus, the notifications are automatically stopped after the time expires.
6. In the next step, the individual symbols are read in a while loop. Reading can be waited for a defined time (timeout) or infinitely long.
7. When the block Unregister Notification [▶ 79] is called, the ADS notification of the symbol is unregistered at the ADS server and the handle to the ADS notification is released from memory.
8. When the block Release LVBuffer Handle [▶ 80] is called, the handle to the LVBuffer is released from memory.
9. In the last step the Release [▶ 74] basic block is called and thus the ADS client is released from memory.

Read Notification-LVBuffer Single

The example, like the other examples in Basic examples [▶ 111], uses the basic VIs and the low-level VIs from LabVIEW™ VIs [▶ 40] in the following steps:

1. In the first step, the basic Symbol Interface [▶ 65] block is called. When the example is executed, the user interface opens and the user can select an ADS symbol in the TwinCAT runtime. When selecting, observe the parameter settings of the symbols for the operation mode LVB-Noti. Single Symbol [▶ 71].
   - If you have already exported an XML description of your ADS symbols from the Symbol Interface, you can select it as a path so that the XML is loaded rather than calling the UI.
2. In the second step, the Init [▶ 66] basic block is called and the ADS client is initialized.
3. The ReadGrpSymbols output may contain more than one ADS symbol (if you selected more than one symbol in step 1). In the example, *Index Array* selects the first ADS symbol in the list.
4. A handle to the LVBuffer is initialized for the selected ADS symbol, with the call of Init LVBuffer Handle [▶ 80].

5. When LVB-Noti. Single Symbol [▶ 71] is called, ADS notifications are registered on the ADS server for the selected ADS symbol. In addition, the block allows to enter a measurement duration in milliseconds. Thus, the notifications are automatically stopped after the time expires.
6. In the next step, the selected symbol can be read in a while loop. Reading can be waited for a defined time (timeout) or infinitely long. It is important to consider whether the ADS notifications are sent Cyclic or OnChange from the server.
7. When the block Unregister Notification [▶ 79] is called, the ADS notification of the symbol is unregistered at the ADS server and the handle to the ADS notification is released from memory.
8. When the block Release LVBuffer Handle [▶ 80] is called, the handle to the LVBuffer is released from memory.
9. In the last step the Release [▶ 74] basic block is called and thus the ADS client is released from memory.

Writing data continuously

Continuos Write Sync Base.vi

The example, like the other examples in Basic examples [▶ 111], uses the basic VIs and the low-level VIs from LabVIEW™ VIs [▶ 40] in the following steps:

1. In the first step, the basic Symbol Interface [▶ 65] block is called. When the example is executed, the user interface opens and the user can select an ADS symbol in the TwinCAT runtime.
   - If you have already exported an XML description of your ADS symbols from the Symbol Interface, you can select it as a path so that the XML is loaded rather than calling the UI.
2. In the second step, the Init [▶ 66] basic block is called and the ADS client is initialized.
3. The WriteGrpSymbols output may contain more than one ADS symbol (if you selected more than one symbol in step 1). In the example, *Index Array* selects the first ADS symbol in the list.
4. To speed up writing and type resolving, the ADS Writer, with the call of Init Writer [▶ 96], and the TypeResolver, with the call of Init Type [▶ 98], are initialized in advance and only once in the next step.
5. In the next step the Send Writer-Request [▶ 96] block sends a new data packet to TwinCAT with each cycle of the loop. Before writing, the LabVIEW™ data type Variant is converted to a suitable TwinCAT 3 data type by calling Resolve To TC Type [▶ 98] block.
6. After reaching the termination condition of the loop, the ADS reader is released from memory by calling Release Writer [▶ 97] and the TypeResolver by calling Release Type [▶ 99].
7. In the last step, the Release [▶ 74] basic block is called and the ADS client is released from memory.

Generate LabVIEW™ type

To automatically generate a LabVIEW™ data type to an ADS symbol, the TypeResolver [▶ 98] and the Generate Type [▶ 76] wrapper block are used. The following examples describe two different ways to implement this:

- Generate Type using Symbol Interface or Symbol's file
- Generate Type using TypeInfo file

Generate Type using Symbol Interface or Symbol's file

The example, like the other examples in Basic examples [▶ 111], uses the basic VIs and the low-level VIs from LabVIEW™ VIs [▶ 40] in the following steps:

1. In the first step, the basic Symbol Interface [▶ 65] block is called. When the example is executed, the user interface opens and the user can select an ADS symbol in the TwinCAT runtime.
   - If you have already exported an XML description of your ADS symbols from the Symbol Interface, you can select it as a path so that the XML is loaded rather than calling the UI.
2. In the second step, the Init [▶ 66] basic block is called and the ADS client is initialized.
3. The ReadGrpSymbols and WriteGrpSymbols output may contain more than one ADS symbol (if you selected more than one symbol in step 1).

4. In a For loop is executed:
   - TypeResolver: With Init Type [▶ 98] the type description is loaded in the TypeResolver and passed to the LabVIEW™ process as XML string (**TypeInfo**).
   - TypeGenerator: Calling the Generate Type [▶ 76] Wrapper VI generates a LabVIEW™ type.
   - Release-Type: Releases the handle to the TypeResolver from memory.
5. Releases the handle to the client.

Generate Type using TypeInfo file

In contrast to Generate Type using Symbol Interface or Symbol's file [▶ 116], the example uses a pre-generated **TypeInfo** file to generate a LabVIEW™ type.

SumUp Read or Write

Read SumUp with TypeResolving.vi

The example, like the other examples in Basic examples [▶ 111], uses the basic VIs from the LabVIEW™ VIs [▶ 40] in the following steps:

1. In the first step, the Symbol Interface [▶ 65] basic block is called. When the example is executed, the user interface opens and the user can select an ADS symbol from his TwinCAT runtime.
   - If you have already exported an XML description of your ADS symbols from the Symbol Interface, you can also select the exported file in the Front Panel before starting the VI, so that the XML is loaded rather than calling the UI.
2. In the second step, the Init [▶ 66] basic block is called and the ADS client is initialized.
3. The ReadGrpSymbols output may contain more than one ADS symbol (if you selected more than one symbol in step 1). In the example, *Index Array* selects the first and second ADS symbols of the list.
4. After initialization of the ADS client, the TypeResolvers [▶ 35] are initialized with the selected symbols.
5. After that the SumUp handle is initialized for reading with the help of Init SumUp [▶ 104]. Here the flag *bAutosend* is set to true so that new data is read automatically.
6. The Add Request block assembles the SumUp sub-commands.
7. After that, the example splits into two different threads:
   - **Loop 1**: Reads cyclically the new data one after the other from the sub-commands. The read data is converted to corresponding LabVIEW™ data types using the initialized TypeResolver.
   - **Loop 2:** Enables or disables the automatic sending of ADS requests.
8. In the last step, all initialized handles are released from memory.

ADS error message when reading symbols with bAutosend set to false

ADS error messages may occur in Loop 1 if the flag *bAutosend* is set to false when initializing the SumUp handle. In this case, the block **Get Data** starts reading the data from the sub-commands even though they have not yet been sent to the ADS server and therefore do not contain any data.

Write SumUp with TypeResolving.vi

The example, like the other examples in Basic examples [▶ 111], uses the basic VIs and the low-level VIs from LabVIEW™ VIs [▶ 40] in the following steps:

1. In the first step, the basic Symbol Interface [▶ 65] block is called. When the example is executed, the user interface opens and the user can select an ADS symbol from his TwinCAT runtime.
   - If you have already exported an XML description of the ADS symbols from the Symbol Interface, you can select it as a path so that the XML is loaded rather than calling the UI.
2. In the second step, the basic Init [▶ 66] block is called and the ADS client is initialized.

3. The WriteGrpSymbols output may contain more than one ADS symbol (if you selected more than one symbol in step 1). In the example two ADS symbols of the types timestamp and string are expected. With *Index Array* the first and the second ADS symbol of the list are selected.
4. After initialization, the TypeResolvers [▶ 35] are initialized to the selected symbols. Then it is checked whether the first symbol has the timestamp data type and the second has the string data type.
5. After that the SumUp handle is initialized for writing with the help of Init SumUp [▶ 104]. Here the flag *bAutosend* is set to false so that the new data is not written automatically.
6. After that, the example splits into two different threads:
   - **Loop 1**: Waits in an event structure for new data generated by Loop 2. The new data is distributed to the appropriate sub-commands and waited to be sent. Before that the raw data is transformed into a TwinCAT 3 data type. If the sub-command already contains data, these are replaced by the new data. In the event structure, automatic sending can be enabled or disabled.
   - **Loop 2:** Generates the block diagram events in a cycle of 250 ms.
7. In the last step, all initialized handles are released from memory.

CoE Read or Write

The example, like the other examples in this chapter, uses the basic VIs and the low-level VIs from LabVIEW™ VIs [▶ 40] in the following steps:

1. In the first step, the basic Init [▶ 66] block is called up with an empty character string and the ADS client is initialized.
2. The CoE blocks [▶ 81] expect the AMS address as a string of the EtherCAT device from which the CoE configuration is to be read. The Device Address control element expects a character string with the following format: AMS Net-Id of the master and AMS Port of the device (AMS NetId:Port).

If the control element is left empty, the example opens a dialog box for selecting an EtherCAT device.

1. The example reads the CoE list from the EtherCAT device.
2. **Interaction**:
   - Double-click on an object in the CoE list to select it and read the object description. This calls up the low-level VI.
   - An entry with a subindex can be read using the **Check Entry** button.
   - In the next step, various CoE entries can be read or written.
3. In the last step, the client handle is released from the memory.

Heartbeat Connection Monitoring

Continuous Read with Heartbeat.vi

The example is based on the "Continuous Read Sync Base" example and extends it to include Heartbeat connection monitoring. The example, like the other examples, uses the basic VIs and the low-level VIs in the following steps:

1. In the first step the basic Symbol Interface block is called. When the example is executed, the user interface opens and the user can select an ADS symbol in the TwinCAT runtime.
   - If you have already exported an XML description of your ADS symbols from the Symbol Interface, you can select it as a path so that the XML is loaded rather than calling the UI.
2. In the second step, the ADS client is initialized using Init.
3. Connection monitoring is then initialized using Init Heartbeat [▶ 84]. The parameters for connection monitoring are already defined here. Changing the parameters at runtime is not possible. The Init Heartbeat returns a Heartbeat event, which is used to query the status of the connection.
4. To speed up reading and type resolving, the ADS Reader, with the call of Read, and the TypeResolver, with the call of Init Type [▶ 98], are initialized in advance and only once in the next step.
5. In the next step, the While loop and the Run Heartbeat [▶ 85] block are executed in parallel. The Run Heartbeat block carries out connection monitoring and runs until the Stop Heartbeat block is called. Continuous reading and the reaction to Heartbeat events take place in the while loop.

6. There is an event structure in the While loop. This waits for the duration defined by the user in "Polling time read in ms" for an event to occur. If no Heartbeat event occurs, the timeout case of the event structure is executed.
7. If the event structure times out, a read operation is carried out. In the event of an error, connection monitoring is aborted by calling the Stop Heartbeat [▶ 85] (timeout errors from the ADS read are ignored, as Heartbeat connection monitoring).
8. **Interaction**: Variant-to-Data can be used to customize the example to translate the Variant to an appropriate data type. The data type depends on the ADS symbol selected in the Symbol Interface. See Type Resolving.
9. If a Heartbeat event occurs, the Event data (Target address, Connection state, Heartbeat handle and Timestamp) is used to display the status of the connection in the front panel of the example VI.
10. **Interaction**: In the Heartbeat event case of the *event structure*, code can be implemented which is to be executed in the event of a change in the connection state. For example, it could be implemented here that a warning is issued to the user in the event of a timeout.
11. Once the termination condition of the loop has been reached, the ADS reader is released from the memory with the Read call, the Heartbeat connection monitoring with the Release Heartbeat [▶ 85] call and the TypeResolver with the Release Type [▶ 99] call.
12. In the last step the basic Release block is called to release the ADS client from memory.

Continuous Write with Heartbeat.vi

The example is based on the "Continuous Write Sync Base" example and extends this with Heartbeat connection monitoring. A minimum configuration for connection monitoring is shown here using the Heartbeat Wrapper VI. The process is carried out in the following steps:

1. In the first step the basic Symbol Interface block is called. When the example is executed, the user interface opens and the user can select an ADS symbol in the TwinCAT runtime.
   - If you have already exported an XML description of your ADS symbols from the Symbol Interface, you can select it as a path so that the XML is loaded rather than calling the UI.
2. In the second step, the basic Init block is called and the ADS client is initialized.
3. The WriteGrpSymbols output may contain more than one ADS symbol (if you selected more than one symbol in step 1). In the example, Index Array selects the first ADS symbol in the list.
4. To speed up writing and type resolving, the ADS Writer, with the call of Write, and the TypeResolver, with the call of Init Type [▶ 98], are initialized in advance and only once in the next step.
5. In the next step, connection monitoring is started using the Heartbeat Wrapper VI [▶ 86] and called up until either a connection interruption is detected or the value of the "boolean Controls Stop Heartbeat" is set to True.
6. At the same time, continuous writing takes place in the While loop. In the next step, the Write block sends a new data packet to TwinCAT with each cycle of the loop. Before writing, the LabVIEW™ data type Variant, with the call to TypeResolver block, is converted to a suitable TwinCAT 3 data type.
7. Once the termination condition of the loop has been reached, the ADS reader is released from memory by calling Write, the Heartbeat connection monitoring is released via the Heartbeat Wrapper VI and the TypeResolver is released by calling Release Type [▶ 99].
8. In the last step the basic Release block is called and the ADS client is released from memory.

Read Notification-Event Single with Heartbeat.vi

The example is based on the "Read Notification-Event Single" example and extends it to include Heartbeat connection monitoring. The example, like the other examples, uses the basic VIs and the low-level VIs in the following steps:

1. In the first step the basic Symbol Interface block is called. When the example is executed, the user interface opens and the user can select an ADS symbol in the TwinCAT runtime. When selecting, note the parameter settings of the symbols for this operation mode: ADS-Read.
   - If you have already exported an XML description of your ADS symbols from the Symbol Interface, you can select it as a path so that the XML is loaded rather than calling the UI.
2. In the second step, the basic Init block is called and the ADS client is initialized.

3. The ReadGrpSymbols output may contain more than one ADS symbol (if you selected more than one symbol in step 1). In the example, Index Array selects the first ADS symbol in the list.
4. Connection monitoring is then initialized using Init Heartbeat [▶ 84]. The parameters for connection monitoring are already defined here. Changing the parameters at runtime is not possible. The Init Heartbeat returns a Heartbeat event, which is used to query the status of the connection.
5. A user event is created for the notifications and then registered together with the Heartbeat event.
6. Calling ADS-Read registers an ADS notification at the ADS server for the selected ADS symbol. At the same time, connection monitoring is started with the Run Heartbeat [▶ 85] block.
7. The next step is to wait for the ADS notifications or a Heartbeat event in the event structure. If no event occurs, the event structure enters a timeout. If data arrives, a notification event is triggered and the received data is displayed in the front panel. If the connection status changes (Heartbeat event), the status of the connection is displayed. In addition, the notifications are stopped when an interruption is detected and restarted when the connection is re-established.
8. **Interaction**: Type Resolving is not implemented in the example. There are two ways: The Resolving type is either realized within the event loop or after the ADS notification has been completed/ unregistered. The points of contact in the event structure are Data and TimeStamps. The latter are the ADS timestamps.
9. When the Unregister Notification [▶ 79] block is called, the ADS notification of the symbol is unregistered at the ADS server and the handle to the ADS notification is released from memory.
10. In the last step, the Heartbeat connection monitoring is released from the memory with the Release Heartbeat [▶ 85] call and the ADS client with the basic Release block call.

Read QMH with Heartbeat.vi

The example is based on the example "Read Notifications using QMH" and extends it to include Heartbeat connection monitoring. It shows the integration of the Heartbeat low-level VIs into a more complex LabVIEW™ project.

In the example, the user can select any number of symbols to read. Either an xml file with the symbols is specified before the VI is started or the symbols are selected in the symbol interface when the VI is started. The symbols are then displayed in a list. Double-click on an icon to select it for reading. The "Read Noti Sgl." and "Read Noti Buff." buttons can be used to select between the continuous, event-controlled read modes Read Notification Single [▶ 31] and Read Notifications Buffered [▶ 33]. The corresponding SubVI then opens and the data is read accordingly.

Connection monitoring is initialized in "Initialize.vi". The Heartbeat event received during initialization is registered when the "Register List Events.vi" block is called and evaluated in the "Producer Loop".

ADS Remote Procedure Call (RPC)

The example, like the other examples in Basic examples [▶ 111], uses the basic VIs from the LabVIEW™ VIs [▶ 40] in the following steps:

1. In the first step, the basic Symbol Interface [▶ 65] block is called. When the example is executed, the user interface opens and the user can select an ADS symbol from his TwinCAT runtime.
2. If you have already exported an XML description of your ADS symbols from the Symbol Interface, you can also select the exported file in the Front Panel before starting the VI, so that the XML is loaded rather than calling the UI.
3. In the second step, the Init [▶ 66] basic block is called and the ADS client is initialized.
4. The ReadWriteGrpSymbols output may contain more than one ADS symbol (if you selected more than one symbol in step 1). In the example, *Index Array* selects the first and second ADS symbols of the list.
5. After initializing the ADS client, the ADS RPC [▶ 108] block is called by passing the IN/INOUT parameters together with the method name to the block. After a successful call, the block returns the following outputs:
6. Return value of the method (if available)
7. OUT parameters
8. Modified INOUT parameters

9. In the last step, all initialized handles are released from memory.

7.2 Application example

The TwinCAT Solution and the corresponding VI (for x86 and x64 LabVIEW™ bit version) can be downloaded here: https://infosys.beckhoff.com/content/1033/TF3710_TC3_Interface_for_LabVIEW/Resources/10083995531.zip.

The VIs are currently compiled with LabVIEW™ 2017 and can be used for subsequent LabVIEW™ versions (2018, 2019, 2020, 2021, 2022, 2023, 2024, 2025) as well.

Description of the example

TwinCAT: The TwinCAT project generates signals in the PLC via signal generator function blocks (sine, amplitude modulated signal, triangle signal, ...). The PLC runs with a cycle time of 5 ms. The signal generator generates 10 values per cycle for the generated signals. Accordingly, an oversampling of 10 is simulated here with the Beckhoff measurement terminals.

LabVIEW™: The LabVIEW™ project uses the Read Notification-Event Buffered [▶ 114] example to read the ADS notifications as LabVIEW™ events. Two while loops are used in parallel.

1. The first while loop receives the ADS notifications and converts the raw ADS data to an appropriate LabVIEW™ data type "Variant" using the TypeResolver. The converted raw data is then inserted into a queue.
2. The second while loop reads the elements from the queue and uses the "Variant to Data" block to finally display the data in a graph. The received data and the ADS timestamps are displayed in two different graphs.

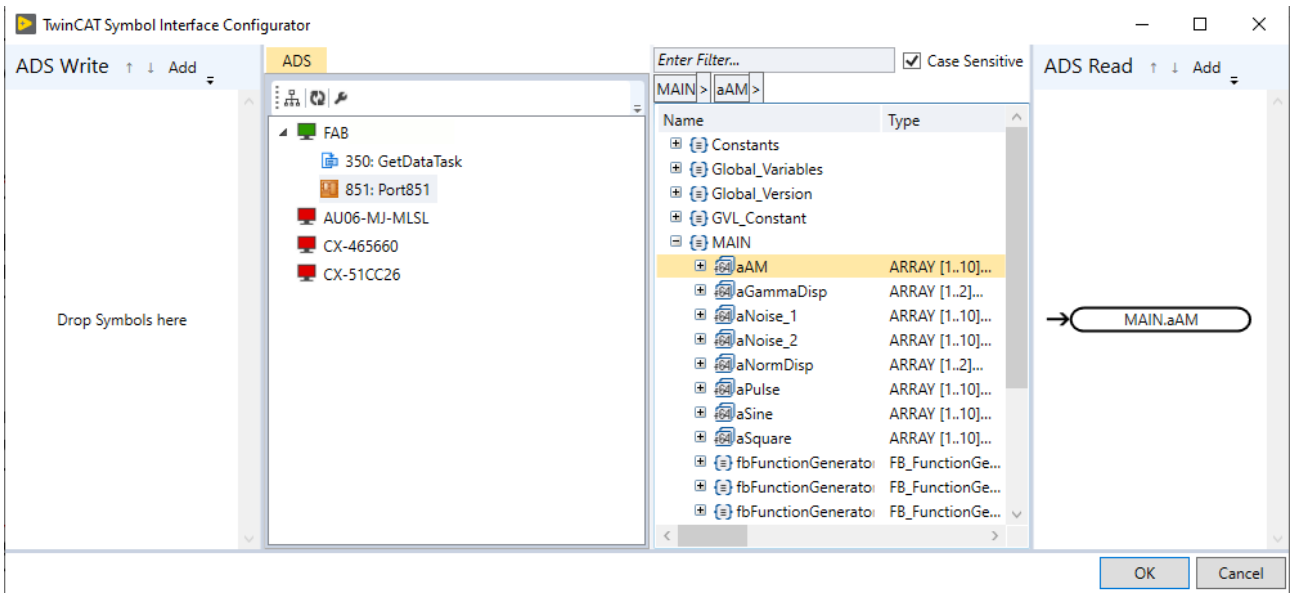
Opening and starting the example

The example contains a TwinCAT 3 folder and a LabVIEW™ folder. In the TwinCAT 3 folder there is a tnzip, which you can open with TwinCAT 3 via **File > Open > Open Solution From Archive** and then save as TwinCAT Solution on your PC.

You can start the TwinCAT Solution with *activate configuration* on the target system of your choice. Make sure that a TF3710 license is available on your target system. If you do not have a valid license, you can create a 7-day trial license.

If your TwinCAT runtime is active, you can display the generated signals in TwinCAT with the TwinCAT 3 Scope. Depending on the selected target system, you only have to select your target system in the properties under
TC Signals > DataPool > aAM (aNoise_1, aNoise_2, aSine).

In the LabVIEW folder select either 32bit (x86) or 64bit (x64) and open the VI contained in it. When you start the program the interface icon opens. Navigate to your target system and select one of the signals.


TwinCAT Symbol Interface Configurator
ADS Write
Add
Drop Symbols here
ADS
FAB
350: GetDataTask
851: Port851
AU06-MJ-MLSL
CX-465660
CX-51CC26
Enter Filter...
Case Sensitive
MAIN
aAM
Name
Type
Constants
Global_Variables
Global_Version
GVL_Constant
MAIN
aAM ARRAY [1..10]...
aGammaDisp ARRAY [1..2]...
aNoise_1 ARRAY [1..10]...
aNoise_2 ARRAY [1..10]...
aNormDisp ARRAY [1..2]...
aPulse ARRAY [1..10]...
aSine ARRAY [1..10]...
aSquare ARRAY [1..10]...
fbFunctionGenerator FB_FunctionGe...
ADS Read
Add
MAIN.aAM
OK
Cancel

8 Appendix

8.1 Overview of error codes

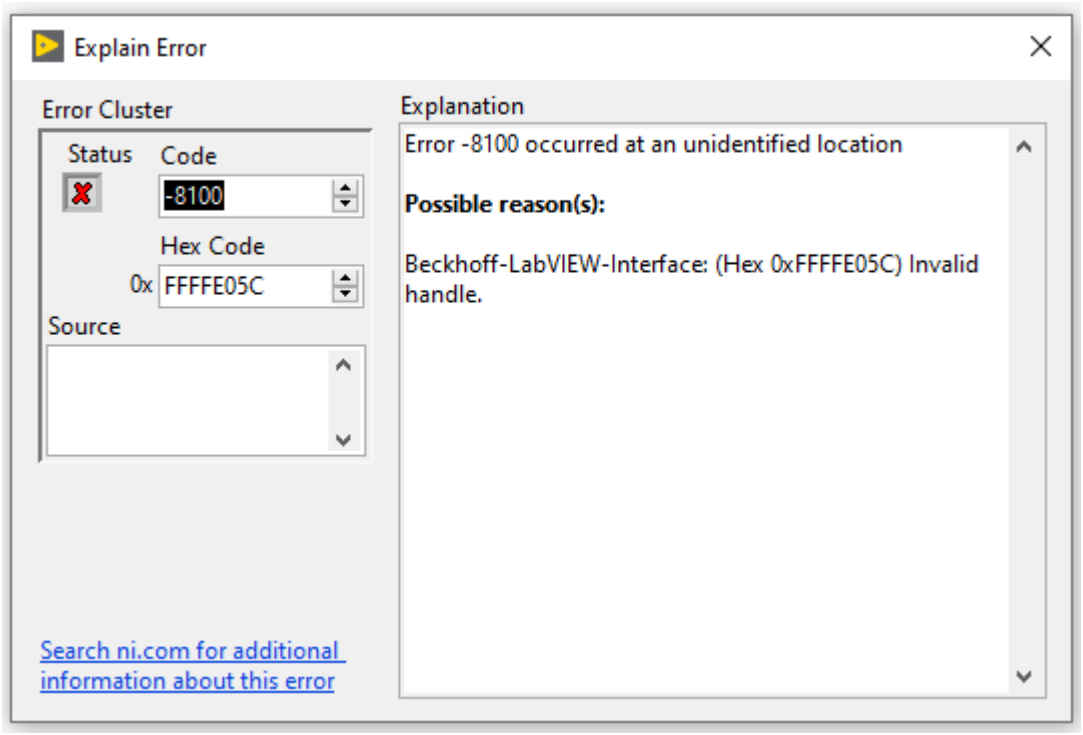
Error checking in the TF3710 TwinCAT 3 interface for LabVIEW™ product follows the standard LabVIEW™ data flow model. During the execution of LabVIEW™ VIs [▶ 40], error checking takes place at each executed node. The blocks only continue if no errors occur. If an error occurs in a block, it is passed to the next block and the affected part of the function or the block is no longer executed accordingly.

The LabVIEW™ error clusters are used for error checking, which then forwards the following information accordingly:

- Status: a Boolean value that returns TRUE if an error occurred.
- Error code: an error identifier in the form of a signed 32-bit integer. If the error code is not 0 and the status output is FALSE, it is a warning rather than an error.
- Source: specifies the source of the error in the form of a LabVIEW™ string.

The description of the error code can be found in LabVIEW™

Menu > Help > Describe Error.



To avoid collision with existing LabVIEW™ errors, all error codes are described in user-defined error codes. The following error codes may occur when running LabVIEW™ VIs [▶ 40].

error value	See…
16#FFFF_DE04 - 16#FFFF_DCD9	Listed in ADS Return Codes [▶ 124] (there without high-order WORD). Further information below on this page.
16#0000_1414 - 16#0000_1432	Listed in Support Return Codes [▶ 133].
16#0000_1464 - 16#0000_157C 16#FFFF_E05C – 16xFFFF_DFF9	Listed in Runtime Return Codes [▶ 129].

If an error occurs during initialization, the function block cannot be used.

Further information on standard TwinCAT Error Codes:

error value	symbol	Error description	Remedy option
16#FFFF_DDFA	NOMEMORY	No memory	Incorrect memory settings => Increase router memory.
16#FFFF_DDEB	TIMEOUT	Device has a timeout	A timeout can occur while waiting for feedback or sending a request. This can often be the case if the network is overloaded. With One-time reading [▶ 26] or One-time writing [▶ 28] the timeout is not so critical, because here the data packet is read or written in one pass.

8.2 ADS Return Codes

Grouping of error codes:
Global error codes: 0x0000 [▶ 124]... (0x9811_0000 ...)
Router error codes: 0x500 [▶ 125]... (0x9811_0500 ...)
General ADS errors: 0x700 [▶ 125]... (0x9811_0700 ...)
RTime error codes: 0x1000 [▶ 126]... (0x9811_1000 ...)

Global error codes

Hex	Dec	HRESULT	Name	Description
0x0	0	0x98110000	ERR_NOERROR	No error.
0x1	1	0x98110001	ERR_INTERNAL	Internal error.
0x2	2	0x98110002	ERR_NORTIME	No real time.
0x3	3	0x98110003	ERR_ALLOCLOCKEDMEM	Allocation locked – memory error.
0x4	4	0x98110004	ERR_INSERTMAILBOX	Mailbox full – the ADS message could not be sent. Reducing the number of ADS messages per cycle will help.
0x5	5	0x98110005	ERR_WRONGRECEIVEHMSG	Wrong HMSG.
0x6	6	0x98110006	ERR_TARGETPORTNOTFOUND	Target port not found – ADS server is not started, not reachable or not installed.
0x7	7	0x98110007	ERR_TARGETMACHINENOTFOUND	Target computer not found – AMS route was not found.
0x8	8	0x98110008	ERR_UNKNOWNCMDID	Unknown command ID.
0x9	9	0x98110009	ERR_BADTASKID	Invalid task ID.
0xA	10	0x9811000A	ERR_NOIO	No IO.
0xB	11	0x9811000B	ERR_UNKNOWNAMSCMD	Unknown AMS command.
0xC	12	0x9811000C	ERR_WIN32ERROR	Win32 error.
0xD	13	0x9811000D	ERR_PORTNOTCONNECTED	Port not connected.
0xE	14	0x9811000E	ERR_INVALIDAMSLENGTH	Invalid AMS length.
0xF	15	0x9811000F	ERR_INVALIDAMSNETID	Invalid AMS Net ID.
0x10	16	0x98110010	ERR_LOWINSTLEVEL	Installation level is too low –TwinCAT 2 license error.
0x11	17	0x98110011	ERR_NODEBUGINTAVAILABLE	No debugging available.
0x12	18	0x98110012	ERR_PORTDISABLED	Port disabled – TwinCAT system service not started.
0x13	19	0x98110013	ERR_PORTALREADYCONNECTED	Port already connected.
0x14	20	0x98110014	ERR_AMSSYNC_W32ERROR	AMS Sync Win32 error.
0x15	21	0x98110015	ERR_AMSSYNC_TIMEOUT	AMS Sync Timeout.
0x16	22	0x98110016	ERR_AMSSYNC_AMSERROR	AMS Sync error.
0x17	23	0x98110017	ERR_AMSSYNC_NOINDEXINMAP	No index map for AMS Sync available.
0x18	24	0x98110018	ERR_INVALIDAMSPORT	Invalid AMS port.
0x19	25	0x98110019	ERR_NOMEMORY	No memory.
0x1A	26	0x9811001A	ERR_TCPSEND	TCP send error.

Hex	Dec	HRESULT	Name	Description
0x1B	27	0x9811001B	ERR_HOSTUNREACHABLE	Host unreachable.
0x1C	28	0x9811001C	ERR_INVALIDAMSFRAGMENT	Invalid AMS fragment.
0x1D	29	0x9811001D	ERR_TLSSEND	TLS send error – secure ADS connection failed.
0x1E	30	0x9811001E	ERR_ACCESSDENIED	Access denied – secure ADS access denied.

Router error codes

Hex	Dec	HRESULT	Name	Description
0x500	1280	0x98110500	ROUTERERR_NOLOCKEDMEMORY	Locked memory cannot be allocated.
0x501	1281	0x98110501	ROUTERERR_RESIZEMEMORY	The router memory size could not be changed.
0x502	1282	0x98110502	ROUTERERR_MAILBOXFULL	The mailbox has reached the maximum number of possible messages.
0x503	1283	0x98110503	ROUTERERR_DEBUGBOXFULL	The Debug mailbox has reached the maximum number of possible messages.
0x504	1284	0x98110504	ROUTERERR_UNKNOWNPORTTYPE	The port type is unknown.
0x505	1285	0x98110505	ROUTERERR_NOTINITIALIZED	The router is not initialized.
0x506	1286	0x98110506	ROUTERERR_PORTALREADYINUSE	The port number is already assigned.
0x507	1287	0x98110507	ROUTERERR_NOTREGISTERED	The port is not registered.
0x508	1288	0x98110508	ROUTERERR_NOMOREQUEUES	The maximum number of ports has been reached.
0x509	1289	0x98110509	ROUTERERR_INVALIDPORT	The port is invalid.
0x50A	1290	0x9811050A	ROUTERERR_NOTACTIVATED	The router is not active.
0x50B	1291	0x9811050B	ROUTERERR_FRAGMENTBOXFULL	The mailbox has reached the maximum number for fragmented messages.
0x50C	1292	0x9811050C	ROUTERERR_FRAGMENTTIMEOUT	A fragment timeout has occurred.
0x50D	1293	0x9811050D	ROUTERERR_TOBEREMOVED	The port is removed.

General ADS error codes

Hex	Dec	HRESULT	Name	Description
0x700	1792	0x98110700	ADSERR_DEVICE_ERROR	General device error.
0x701	1793	0x98110701	ADSERR_DEVICE_SRVNOTSUPP	Service is not supported by the server.
0x702	1794	0x98110702	ADSERR_DEVICE_INVALIDGRP	Invalid index group.
0x703	1795	0x98110703	ADSERR_DEVICE_INVALIDOFFSET	Invalid index offset.
0x704	1796	0x98110704	ADSERR_DEVICE_INVALIDACCESS	Reading or writing not permitted. Several causes are possible. For example, an incorrect password was entered when creating routes.
0x705	1797	0x98110705	ADSERR_DEVICE_INVALIDSIZE	Parameter size not correct.
0x706	1798	0x98110706	ADSERR_DEVICE_INVALIDDATA	Invalid data values.
0x707	1799	0x98110707	ADSERR_DEVICE_NOTREADY	Device is not ready to operate.
0x708	1800	0x98110708	ADSERR_DEVICE_BUSY	Device is busy.
0x709	1801	0x98110709	ADSERR_DEVICE_INVALIDCONTEXT	Invalid operating system context. This can result from use of ADS blocks in different tasks. It may be possible to resolve this through multitasking synchronization in the PLC.
0x70A	1802	0x9811070A	ADSERR_DEVICE_NOMEMORY	Insufficient memory.
0x70B	1803	0x9811070B	ADSERR_DEVICE_INVALIDPARM	Invalid parameter values.
0x70C	1804	0x9811070C	ADSERR_DEVICE_NOTFOUND	Not found (files, ...).
0x70D	1805	0x9811070D	ADSERR_DEVICE_SYNTAX	Syntax error in file or command.
0x70E	1806	0x9811070E	ADSERR_DEVICE_INCOMPATIBLE	Objects do not match.
0x70F	1807	0x9811070F	ADSERR_DEVICE_EXISTS	Object already exists.
0x710	1808	0x98110710	ADSERR_DEVICE_SYMBOLNOTFOUND	Symbol not found.
0x711	1809	0x98110711	ADSERR_DEVICE_SYMBOLVERSIONINVALID	Invalid symbol version. This can occur due to an online change. Create a new handle.
0x712	1810	0x98110712	ADSERR_DEVICE_INVALIDSTATE	Device (server) is in invalid state.
0x713	1811	0x98110713	ADSERR_DEVICE_TRANSMODENOTSUPP	AdsTransMode not supported.
0x714	1812	0x98110714	ADSERR_DEVICE_NOTIFYHNDINVALID	Notification handle is invalid.
0x715	1813	0x98110715	ADSERR_DEVICE_CLIENTUNKNOWN	Notification client not registered.
0x716	1814	0x98110716	ADSERR_DEVICE_NOMOREHDLS	No further handle available.

Hex	Dec	HRESULT	Name	Description
0x717	1815	0x98110717	ADSERR_DEVICE_INVALIDWATCHSIZE	Notification size too large.
0x718	1816	0x98110718	ADSERR_DEVICE_NOTINIT	Device not initialized.
0x719	1817	0x98110719	ADSERR_DEVICE_TIMEOUT	Device has a timeout.
0x71A	1818	0x9811071A	ADSERR_DEVICE_NOINTERFACE	Interface query failed.
0x71B	1819	0x9811071B	ADSERR_DEVICE_INVALIDINTERFACE	Wrong interface requested.
0x71C	1820	0x9811071C	ADSERR_DEVICE_INVALIDCLSID	Class ID is invalid.
0x71D	1821	0x9811071D	ADSERR_DEVICE_INVALIDOBJID	Object ID is invalid.
0x71E	1822	0x9811071E	ADSERR_DEVICE_PENDING	Request pending.
0x71F	1823	0x9811071F	ADSERR_DEVICE_ABORTED	Request is aborted.
0x720	1824	0x98110720	ADSERR_DEVICE_WARNING	Signal warning.
0x721	1825	0x98110721	ADSERR_DEVICE_INVALIDARRAYIDX	Invalid array index.
0x722	1826	0x98110722	ADSERR_DEVICE_SYMBOLNOTACTIVE	Symbol not active.
0x723	1827	0x98110723	ADSERR_DEVICE_ACCESSDENIED	Access denied. Several causes are possible. For example, a unidirectional ADS route is used in the opposite direction.
0x724	1828	0x98110724	ADSERR_DEVICE_LICENSENOTFOUND	Missing license.
0x725	1829	0x98110725	ADSERR_DEVICE_LICENSEEXPIRED	License expired.
0x726	1830	0x98110726	ADSERR_DEVICE_LICENSEEXCEEDED	License exceeded.
0x727	1831	0x98110727	ADSERR_DEVICE_LICENSEINVALID	Invalid license.
0x728	1832	0x98110728	ADSERR_DEVICE_LICENSESYSTEMID	License problem: System ID is invalid.
0x729	1833	0x98110729	ADSERR_DEVICE_LICENSENOTIMELIMIT	License not limited in time.
0x72A	1834	0x9811072A	ADSERR_DEVICE_LICENSEFUTUREISSUE	Licensing problem: time in the future.
0x72B	1835	0x9811072B	ADSERR_DEVICE_LICENSETIMETOLONG	License period too long.
0x72C	1836	0x9811072C	ADSERR_DEVICE_EXCEPTION	Exception at system startup.
0x72D	1837	0x9811072D	ADSERR_DEVICE_LICENSEDUPLICATED	License file read twice.
0x72E	1838	0x9811072E	ADSERR_DEVICE_SIGNATUREINVALID	Invalid signature.
0x72F	1839	0x9811072F	ADSERR_DEVICE_CERTIFICATEINVALID	Invalid certificate.
0x730	1840	0x98110730	ADSERR_DEVICE_LICENSEOEMNOTFOUND	Public key not known from OEM.
0x731	1841	0x98110731	ADSERR_DEVICE_LICENSERESTRICTED	License not valid for this system ID.
0x732	1842	0x98110732	ADSERR_DEVICE_LICENSEDEMODENIED	Demo license prohibited.
0x733	1843	0x98110733	ADSERR_DEVICE_INVALIDFNCID	Invalid function ID.
0x734	1844	0x98110734	ADSERR_DEVICE_OUTOFRANGE	Outside the valid range.
0x735	1845	0x98110735	ADSERR_DEVICE_INVALIDALIGNMENT	Invalid alignment.
0x736	1846	0x98110736	ADSERR_DEVICE_LICENSEPLATFORM	Invalid platform level.
0x737	1847	0x98110737	ADSERR_DEVICE_FORWARD_PL	Context – forward to passive level.
0x738	1848	0x98110738	ADSERR_DEVICE_FORWARD_DL	Context – forward to dispatch level.
0x739	1849	0x98110739	ADSERR_DEVICE_FORWARD_RT	Context – forward to real-time.
0x740	1856	0x98110740	ADSERR_CLIENT_ERROR	Client error.
0x741	1857	0x98110741	ADSERR_CLIENT_INVALIDPARM	Service contains an invalid parameter.
0x742	1858	0x98110742	ADSERR_CLIENT_LISTEMPTY	Polling list is empty.
0x743	1859	0x98110743	ADSERR_CLIENT_VARUSED	Var connection already in use.
0x744	1860	0x98110744	ADSERR_CLIENT_DUPLINVOKEID	The called ID is already in use.
0x745	1861	0x98110745	ADSERR_CLIENT_SYNCTIMEOUT	Timeout has occurred – the remote terminal is not responding in the specified ADS timeout. The route setting of the remote terminal may be configured incorrectly.
0x746	1862	0x98110746	ADSERR_CLIENT_W32ERROR	Error in Win32 subsystem.
0x747	1863	0x98110747	ADSERR_CLIENT_TIMEOUTINVALID	Invalid client timeout value.
0x748	1864	0x98110748	ADSERR_CLIENT_PORTNOTOPEN	Port not open.
0x749	1865	0x98110749	ADSERR_CLIENT_NOAMSADDR	No AMS address.
0x750	1872	0x98110750	ADSERR_CLIENT_SYNCINTERNAL	Internal error in Ads sync.
0x751	1873	0x98110751	ADSERR_CLIENT_ADDHASH	Hash table overflow.
0x752	1874	0x98110752	ADSERR_CLIENT_REMOVEHASH	Key not found in the table.
0x753	1875	0x98110753	ADSERR_CLIENT_NOMORESYM	No symbols in the cache.
0x754	1876	0x98110754	ADSERR_CLIENT_SYNCRESINVALID	Invalid response received.
0x755	1877	0x98110755	ADSERR_CLIENT_SYNCPORTLOCKED	Sync Port is locked.
0x756	1878	0x98110756	ADSERR_CLIENT_REQUESTCANCELLED	The request was canceled.

RTime error codes

Hex	Dec	HRESULT	Name	Description
0x1000	4096	0x98111000	RTERR_INTERNAL	Internal error in the real-time system.
0x1001	4097	0x98111001	RTERR_BADTIMERPERIODS	Timer value is not valid.
0x1002	4098	0x98111002	RTERR_INVALIDTASKPTR	Task pointer has the invalid value 0 (zero).
0x1003	4099	0x98111003	RTERR_INVALIDSTACKPTR	Stack pointer has the invalid value 0 (zero).
0x1004	4100	0x98111004	RTERR_PRIOEXISTS	The request task priority is already assigned.
0x1005	4101	0x98111005	RTERR_NOMORETCB	No free TCB (Task Control Block) available. The maximum number of TCBs is 64.
0x1006	4102	0x98111006	RTERR_NOMORESEMAS	No free semaphores available. The maximum number of semaphores is 64.
0x1007	4103	0x98111007	RTERR_NOMOREQUEUES	No free space available in the queue. The maximum number of positions in the queue is 64.
0x100D	4109	0x9811100D	RTERR_EXTIRQALREADYDEF	An external synchronization interrupt is already applied.
0x100E	4110	0x9811100E	RTERR_EXTIRQNOTDEF	No external sync interrupt applied.
0x100F	4111	0x9811100F	RTERR_EXTIRQINSTALLFAILED	Application of the external synchronization interrupt has failed.
0x1010	4112	0x98111010	RTERR_IRQLNOTLESSOREQUAL	Call of a service function in the wrong context
0x1017	4119	0x98111017	RTERR_VMXNOTSUPPORTED	Intel® VT-x extension is not supported.
0x1018	4120	0x98111018	RTERR_VMXDISABLED	Intel® VT-x extension is not enabled in the BIOS.
0x1019	4121	0x98111019	RTERR_VMXCONTROLSMISSING	Missing function in Intel® VT-x extension.
0x101A	4122	0x9811101A	RTERR_VMXENABLEFAILS	Activation of Intel® VT-x fails.

Specific positive HRESULT Return Codes:

HRESULT	Name	Description
0x0000_0000	S_OK	No error.
0x0000_0001	S_FALSE	No error. Example: successful processing, but with a negative or incomplete result.
0x0000_0203	S_PENDING	No error. Example: successful processing, but no result is available yet.
0x0000_0256	S_WATCHDOG_TIMEOUT	No error. Example: successful processing, but a timeout occurred.

TCP Winsock error codes

Hex	Dec	Name	Description
0x274C	10060	WSAETIMEDOUT	A connection timeout has occurred - error while establishing the connection, because the remote terminal did not respond properly after a certain period of time, or the established connection could not be maintained because the connected host did not respond.
0x274D	10061	WSAECONNREFUSED	Connection refused - no connection could be established because the target computer has explicitly rejected it. This error usually results from an attempt to connect to a service that is inactive on the external host, that is, a service for which no server application is running.
0x2751	10065	WSAEHOSTUNREACH	No route to host - a socket operation referred to an unavailable host.
More Winsock error codes: Win32 error codes			

8.3 Data types

TwinCAT data type	Memory requirement	LabVIEW™ data type
BIT	1 bit	LabVIEW™ Boolean
BOOL	8 bits	LabVIEW™ Boolean
BYTE/USINT	8 bits	LabVIEW™ U8
WORD/UINT	16 bits	LabVIEW™ U16
DWORD/UDINT	32 bits	LabVIEW™ U32
ULINT	64 bits	LabVIEW™ U64
SINT	8 bits	LabVIEW™ I8
INT	16 bits	LabVIEW™ I16
DINT	32 bits	LabVIEW™ I32

TwinCAT data type	Memory requirement	LabVIEW™ data type
LINT	64 bits	LabVIEW™ I64
REAL	32 bits	LabVIEW™ Single (sgl)
LREAL	64 bits	LabVIEW™ Double (dbl)
String	-	LabVIEW™ String
TIME	-	LabVIEW™ Timestamp
DATE	-	LabVIEW™ Timestamp
TOD/ TIME_OF_DAY	-	LabVIEW™ Timestamp
DT/DATE_AND_TIME	-	LabVIEW™ Timestamp
DCTIME	-	LabVIEW™ Timestamp
FILETIME	-	LabVIEW™ Timestamp
LTIME	-	LabVIEW™ Timestamp
ARRAY	-	LabVIEW™ ARRAY
STRUCT	-	LabVIEW™ Cluster
Enum	16 bits (Default)	LabVIEW™ Enum

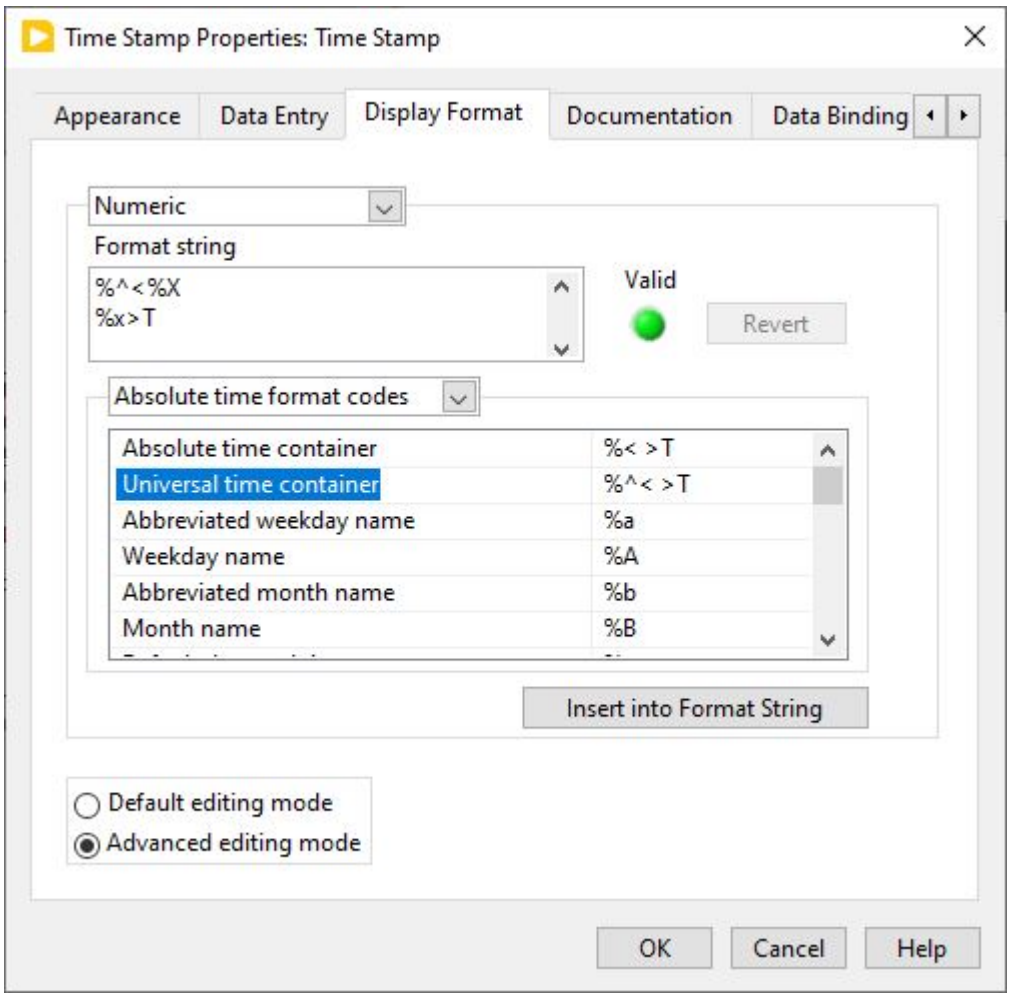
Display of timestamps in UTC time format

TwinCAT and LabVIEW™ generally measure time in Coordinated Universal Time (UTC). As soon as a timestamp is displayed in the LabVIEW™ front panels, it is converted to the local time by default.

✓ If you want to display the Coordinated Universal Time (UTC) instead of the local time, you can format it as follows via the control settings:

1. Open **Time Stamp Properties.**
2. Open the **Display Format** tab.
3. Activate **Advanced editing mode**.
4. To use the UTC time format, insert a ^ in the default format string before the time container.

⇨ The format string then looks like this, for example:
%^<%X
%x>T



8.4 Runtime Return Codes

Errors that occur during: initialization of the ADS client; establishing the connection; sending requests or receiving data packets; converting the LabVIEW™ data type to TwinCAT 3 data type or vice versa are described as runtime errors.

error value	symbol	Error description	Remedy option
16#00001464	5220	TF3710_ERR_E_POINTER	Invalid pointer 0 (null)
16#00001465	5221	TF3710_ERR_E_OUT_OF_RANGE	The value is not in the range.
16#00001466	5222	TF3710_ERR_E_INVALID_ARG	The parameters are invalid.
16#00001467	5223	TF3710_ERR_E_OUT_OF_MEMORY	The memory is full.
16#00001468	5224	TF3710_ERR_E_NOT_A_NUMBER	Is not a number
16#00001469	5225	TF3710_ERR_E_BAD_ALLOCATION	Wrong allocation
16#0000146A	5226	TF3710_ERR_E_NOT_A_STRING	Is not a string
16#0000146B	5227	TF3710_ERR_E_UNINITIALIZED_TIMESTAMP	Timestamp is not initialized.
16#0000146D	5229	TF3710_ERR_E_CONFIGURATION_NON_EXISTENT	The configuration was not found.
16#0000146E	5230	TF3710_ERR_E_FLOATING_OBJECT	Floating/Non-saved object (VI, Project, ...)

error value	symbol	Error description	Remedy option
16#0000146F	5231	TF3710_ERR_E_INVALID_CONFIGURATION	The configuration is incorrect.
16#0000146C	5228	TF3710_ERR_E_LOCKFAILED	Lock has failed.
16#00001479	5241	TF3710_ERR_FAILED_WAIT_ON_A_REQUEST	Waiting for the request has failed.
16#00001483	5251	TF3710_ERR_INVALID_PORT_GROUP	Invalid ADS port type
16#00001484	5252	TF3710_ERR_NEW_NODE_FAILED	The new node has failed
16#00001485	5253	TF3710_ERR_APPEND_BUFFER_FAILED	Writing to the node has failed
16#00001486	5254	TF3710_ERR_MISSING_NODE	Node missing
16#00001487	5255	TF3710_ERR_PARSE_ERROR	Parsing has failed
16#0000148D	5261	TF3710_ERR_PORT_ALREADY_EXIST	ADS port is still present.
16#0000148E	5262	TF3710_ERR_PORT_OPEN_FAILED	Opening the ADS port has failed.
16#0000148F	5263	TF3710_ERR_INVALID_TARGET_ID	AmsNetID of the target system is invalid.
16#00001490	5264	TF3710_ERR_UNDEFINED_TARGET_PORT	ADS port of the target system is invalid.
16#00001491	5265	TF3710_ERR_CONNECTION_FAILED	Failed to connect to the router.
16#00001492	5266	TF3710_ERR_NO_PORT_OPENED	Port is not opened.
16#00001493	5267	TF3710_ERR_DISCONNECT_FAILED	Disconnect from router has failed.
16#00001494	5268	TF3710_ERR_READ_STATE_REQUEST_FAILED	Read state request has failed.
16#00001495	5269	TF3710_ERR_VARHANDLE_FAILED	Loading Var handler has failed.
16#00001496	5270	TF3710_ERR_VARHANDLE_ALREADY_EXIST	Var handler already exists.
16#00001497	5271	TF3710_ERR_VARHANDLE_RELEASE_FAILED	Releasing Var handler has failed.
16#00001498	5272	TF3710_ERR_VARHANDLE_ALREADY_RELEASED	Var handler has already been released.
16#00001499	5273	TF3710_ERR_INVALID_VAR_HANDLER	Var handler is invalid.
16#0000149A	5274	TF3710_ERR_MISSING_ROUTE	ADS route is not available.
16#0000149B	5275	TF3710_ERR_CONNECTION_ALREADY_EXISTS	The connection is already established.
16#000014A6	5286	TF3710_ERR_EMPTY_SYMBOL_TABLE	Currently no symbols available in table.
16#000014A7	5287	TF3710_ERR_MISSING_TARGETS	Target systems are missing
16#000014A8	5288	TF3710_ERR_TYPESYSTEM_PROVIDER_FAILED	The TypeSystem provider has failed.
16#000014A9	5289	TF3710_ERR_TYPERESOLVERPROXY_FAILED	The TypeResolverProxy has failed.

error value	symbol	Error description	Remedy option
16#000014AA	5290	TF3710_ERR_TYPERESOLVERPROXY_LOAD_GUID_FAILED	Loading of TC-GUID has failed.
16#000014AB	5291	TF3710_ERR_TYPERESOLVERPROXY_LOAD_TYPE_COLLECTION_FAILED	Loading of TypeCollection has failed.
16#000014AC	5292	TF3710_ERR_TYPERESOLVERPROXY_SYMBOL_GUID_ERROR	GUID is invalid.
16#000014AD	5293	TF3710_ERR_LVTYPEDESCRIPTOR_INVALID_ELEMENT_RANGE	Number of elements in the TypeResolver do not match.
16#000014AE	5294	TF3710_ERR_LVTYPEDESCRIPTOR_TYPE_NOT_SUPPORTED	Data type is currently not supported.
16#000014AF	5295	TF3710_ERR_LVTYPEDESCRIPTOR_INVALID_ARRAY_SYMBOLINFO	Elements are missing in the array data type.
16#000014B0	5296	TF3710_ERR_LVTYPEDESCRIPTOR_MISSING_ARRAY_SYMBOLINFO_ELEMENTS	Symbol info for array data type is invalid.
16#000014B1	5297	TF3710_ERR_LVTYPEDESCRIPTOR_MISSING_ARRAY_BASE_TYPE	Array data type has no base type.
16#000014B2	5298	TF3710_ERR_LVTYPEDESCRIPTOR_LOAD_FAILDED	Loading of TypeResolver has failed
16#000014B3	5299	TF3710_ERR_LVTYPEDESCRIPTOR_INVALID_PRIMARY_TYPE	The primary data type is invalid.
16#000014B4	5300	TF3710_ERR_LVTYPEDESCRIPTOR_TYPE_UNINITIALIZED	TypeResolver is not initialized.
16#000014C4	5316	TF3710_ERR_LVTYPEDESCRIPTOR_INVALID_CLUSTER_SYMBOLINFO	Symbol info for cluster data type is invalid.
16#000014C5	5317	TF3710_ERR_LVTYPEDESCRIPTOR_NON_CLUSTER_TYPE	Data type is not of the LabVIEW™ cluster data type.
16#000014C6	5318	TF3710_ERR_LVTYPEDESCRIPTOR_INVALID_CLUSTER_SYMBOLPOSITION	Elements are missing in the cluster data type.
16#000014D4	5331	TF3710_ERR_LVTYPEDESCRIPTOR_INVALID_TIMESTAMP_VALUE	Invalid TC timestamp value
16#000014D5	5332	TF3710_ERR_LVTYPEDESCRIPTOR_UNSUPORTED_TIMESTAMP	This TC timestamp data type is currently not supported.
16#000014D9	5337	TF3710_ERR_LVTYPEDESCRIPTOR_TYPE_SIZE_MISMATCH	Data types are not identical.
16#000014DA	5338	TF3710_ERR_LVTYPEDESCRIPTOR_TYPE_MISMATCH	Data types are not identical.

error value	symbol	Error description	Remedy option
16#000014DB	5339	TF3710_ERR_LVTYPEDESCRIPTOR_DATA_SIZE_MISMATCH	Data type content does not match.
16#00001504	5380	TF3710_ERR_TYPEGENERATOR_SUBTYPE_NOT_FOUND	TypeGenerator cannot find the SubType or the BaseType.
16#00001519	5401	TF3710_ERR_READ_REQUEST_FAILED	Read request has failed
16#0000151A	5402	TF3710_ERR_INVALID_READ_BUFFER	ADS raw data incorrectly packed
16#0000151B	5403	TF3710_ERR_INVALID_READ_DATA_FORMAT	ADS raw data invalid
16#0000151C	5404	TF3710_ERR_NOTIFICATION_HANDLER_FAILED	Notification handler has failed.
16#0000151D	5405	TF3710_ERR_ATLEAST_ONE_MISSED_NOTIFICATION	At least one/several notifications is/are not received correctly
16#0000151E	5406	TF3710_ERR_NOTIFICATION_REQUEST_FAILED	Notification request has failed.
16#0000151F	5407	TF3710_ERR_UNSUPPORTED_NOTIFICATION_MODE	Notification mode is currently not supported.
16#00001520	5408	TF3710_ERR_NOTIFICATION_BUFFER_FAILED	Buffered notification has failed.
16#00001534	5428	TF3710_ERR_WRITE_REQUEST_FAILED	Write request has failed.
16#00001565	5429	TF3710_ERR_WRITE_BUFFER	Write buffer invalid
16#FFFFE05C	-8100	TF3710_ERR_INVALID_HANDLE	Invalid handle
16#FFFFE05B	-8101	TF3710_ERR_HANDLE_IN_USE	The handle is still in use.
16#FFFFE052	-8110	TF3710_ERR_INVALID_ARRAY	Invalid array
0xFFFFE051	-8111	TF3710_ERR_INVALID_ARRAY_SIZE	Array size invalid
0xFFFFE050	-8112	TF3710_ERR_SYMBOL_NOT_FOUND	Symbol not found.
0xFFFFE04F	-8113	TF3710_ERR_LVBUFFER_NOT_REGISTERED	LVBuffer was not registered.
16#FFFFE04E	-8114	TF3710_ERR_HEARTBEAT_NO_TARGET	No instance of FB_Heartbeat found at a target.
16#FFFFE04D	-8115	TF3710_ERR_HEARTBEAT_TARGET_NOT_FOUND	No instance of FB_Heartbeat with the specified target address found.
16#FFFFE04C	-8116	TF3710_ERR_HEARTBEAT_TARGET_ALREADY_IN_USE	The instance of FB_Heartbeat is already being used by another target.
16#FFFFE04B	-8117	TF3710_ERR_HEARTBEAT_ALREADY_INITIALIZED	Heartbeat has already been initialized. Only one instance is permitted per client handle.
16#FFFFE04A	-8118	TF3710_ERR_HEARTBEAT_ALREADY_RUNNING	Heartbeat has already been launched. Only one running instance is permitted per client handle.
16#FFFFE049	-8119	TF3710_ERR_HEARTBEAT_NOT_INITIALIZED	Heartbeat has not yet been initialized.

error value	symbol	Error description	Remedy option
16#FFFFE048	-8120	TF3710_ERR_HEARTBEAT_NOT_RUNNING	Heartbeat is not executed.

8.5 Support Return Codes

error value	symbol	Error description	Remedy option
16#00001414	5140	TF3710_ERR_INVALID_REQUEST	The request is not currently supported
16#00001415	5141	TF3710_ERR_BETA_TRIAL_EXPIRED	Beta trial period has expired
16#00001416	5142	TF3710_ERR_LICENSE_STATE_ISSUE	The license state is invalid

9 Support and Service

Beckhoff and their partners around the world offer comprehensive support and service, making available fast and competent assistance with all questions related to Beckhoff products and system solutions.

Download finder

Our download finder contains all the files that we offer you for downloading. You will find application reports, technical documentation, technical drawings, configuration files and much more.

The downloads are available in various formats.

Beckhoff's branch offices and representatives

Please contact your Beckhoff branch office or representative for local support and service on Beckhoff products!

The addresses of Beckhoff's branch offices and representatives round the world can be found on our internet page: www.beckhoff.com

You will also find further documentation for Beckhoff components there.

Beckhoff Support

Support offers you comprehensive technical assistance, helping you not only with the application of individual Beckhoff products, but also with other, wide-ranging services:

- support
- design, programming and commissioning of complex automation systems
- and extensive training program for Beckhoff system components

Hotline:	+49 5246 963-157
e-mail:	support@beckhoff.com

Beckhoff Service

The Beckhoff Service Center supports you in all matters of after-sales service:

- on-site service
- repair service
- spare parts service
- hotline service

Hotline:	+49 5246 963-460
e-mail:	service@beckhoff.com

Beckhoff Headquarters

Beckhoff Automation GmbH & Co. KG

Huelshorstweg 20
33415 Verl
Germany

Phone:	+49 5246 963-0
e-mail:	info@beckhoff.com
web:	www.beckhoff.com

Trademark statements

Beckhoff®, ATRO®, EtherCAT®, EtherCAT G®, EtherCAT G10®, EtherCAT P®, MX-System®, Safety over EtherCAT®, TC/BSD®, TwinCAT®, TwinCAT/BSD®, TwinSAFE®, XFC®, XPlanar® and XTS® are registered and licensed trademarks of Beckhoff Automation GmbH.

Third-party trademark statements

DALI, DALI-2, D4i, DALI+, and DiiA are trademarks in various countries in the exclusive use of the Digital Illumination Interface Alliance.

Excel, IntelliSense, Microsoft, Microsoft Azure, Microsoft Edge, PowerShell, Visual Studio, Windows and Xbox are trademarks of the Microsoft group of companies.

Intel, the Intel logo, Intel Core, Xeon, Intel Atom, Celeron and Pentium are trademarks of Intel Corporation or its subsidiaries.

The registered trademark Linux® is used pursuant to a sublicense from the Linux Foundation, the exclusive licensee of Linus Torvalds, owner of the mark on a worldwide basis.

National Instruments® and LabVIEW™ are trademarks of National Instruments. Neither Beckhoff, nor any software programs or other goods or services offered by Beckhoff, are affiliated with, endorsed by, or sponsored by National Instruments.

:

www.beckhoff.com/TF3710

: