Application for Communication

OPC Communication via the SEND/RECEIVE Protocol with a Visual Basic .NET OPC Client Extension
Warranty, Liability and Support

We accept no liability for information contained in this document.

Any claims against us - based on whatever legal reason - resulting from the use of the examples, information, programs, engineering and performance data etc., described in this document shall be excluded. Such an exclusion shall not apply in the case of mandatory liability, e.g. under the German Product Liability Act (“Produkthaftungsgesetz”), in case of intent, gross negligence, or injury of life, body or health, guarantee for the quality of a product, fraudulent concealment of a deficiency or breach of a condition which goes to the root of the contract (“wesentliche Vertragspflichten”). However, claims arising from a breach of a condition which goes to the root of the contract shall be limited to the foreseeable damage which is intrinsic to the contract, unless caused by intent or gross negligence or based on mandatory liability for injury of life, body or health. The above provisions do not imply a change in the burden of proof to your detriment.

The Application Examples are not binding and do not claim to be complete regarding the circuits shown, equipping and any eventuality. They do not represent customer-specific solutions. They are only intended to provide support for typical applications. You are responsible for ensuring that the described products are used correctly. These Application Examples do not relieve you of the responsibility in safely and professionally using, installing, operating and servicing equipment. By using this application example you accept that Siemens is not liable for any damages except for those specified in the above liability clause. We reserve the right to make changes to these Application Examples at any time without prior notice. If there are any deviations between the recommendations provided in this application example and other Siemens publications – e.g. Catalogs – the contents of the other documents have priority.

Copyright© 2005 Siemens A&D. It is not permissible to transfer or copy these Application Examples or excerpts of them without first having prior authorization from Siemens A&D in writing.

For questions about this document please use the following e-mail address:

mailto:csweb@ad.siemens.de
Foreword

Objectives of the application

This example application discusses the application of the SEND/RECEIVE protocol with the SIMATIC NET OPC server. The OPC server communicates with the controller via the block and variable services of the SEND/RECEIVE protocol.

It demonstrates how an OPC client, which is tailored to individual requirements, is programmed in the programming language Visual Basic .NET. The configuration of the PC station for using the SIMATIC NET OPC server with the SEND/RECEIVE protocol and Industrial Ethernet is also explained. This enables transferring structured data via the OPC interface.

Main contents of this application

The following main points are discussed in this application:

- Dealing with the communication services of the SEND/RECEIVE protocol with regards to program technology:
  - Block services (AG_SEND / AG_RECEIVE)
  - Variable services (FETCH / WRITE)
  for enabling exchange of larger data volumes between PC and mixed automation systems (i.e. SIMATIC S5 and SIMATIC T S7).
- Configuring the SEND/RECEIVE protocol when using Industrial Ethernet
- Generating an OPC client with graphical user interface
- Demonstrating the handling of the Data Access interface of the OPC server via the Custom Interface, and the integrating process into .NET-Framework using the OPC DA RC-Wrapper
- Displaying the procedure for implementing an additional acknowledgement mechanism (additional level 7 protocol).
Structure of the document

The documentation of this application is divided into the following main parts:

- **Introduction**
- **Extension**
- **Demonstration**

Additionally, the STEP7 code and the Visual Basic .NET code are available.

This second document, the **Extension**, is aimed at persons who want to have a more detailed overview.

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>Provides a general overview of the contents. You will learn about the components used (standard hardware and software components and the specially created software).</td>
</tr>
<tr>
<td>Extension</td>
<td>Discusses the detailed function processes of the involved hardware and software components, the solution structures, and where sensible the concrete implementation of this application. This part is necessary if you want to learn about the interaction of the solution components, for example in order to use them as the basis for own development.</td>
</tr>
<tr>
<td>Demonstration</td>
<td>This part leads you step by step through the structure, important configuration steps, commissioning and operation of the application.</td>
</tr>
</tbody>
</table>

An additional component available is the S7 program code.

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>S7 program code</td>
<td>The S7 program code includes the code and a user interface which may also be used for demonstration purposes.</td>
</tr>
<tr>
<td>Visual Basic .NET Code</td>
<td>Contains the program files of the OPC client and the Visual Studio .NET project.</td>
</tr>
</tbody>
</table>

Reference for Automation and Drives Service & Support

This entry originates from the internet application portal of the A&D Service and Support. The following link takes you directly to the download page of this document.

# Table of Contents

## Table of Contents

- Introduction ................................................................................................................... 7
- 1 Document on Hand ..................................................................................................... 7
- **Function Principles and Program Structures** ............................................................. 8
  - 2 **General Function Mechanisms** .............................................................................. 8
    - 2.1 OPC DA Interface (OPC Data Access) ................................................................. 8
    - 2.1.1 General ............................................................................................................. 9
    - 2.1.2 OPC Server ..................................................................................................... 10
    - 2.1.3 OPC Client ...................................................................................................... 10
    - 2.1.4 The role of variables ....................................................................................... 11
    - 2.1.5 Data source Cache and Device ...................................................................... 12
    - 2.1.6 Synchronous Read / Write jobs ................................................................... 13
    - 2.1.7 Asynchronous Read / Write jobs .................................................................. 13
    - 2.1.8 Monitoring variables (DataChange mechanism) ............................................ 14
    - 2.1.9 Active and passive items .............................................................................. 15
    - 2.2 Principal user model of the OPC DA interface in this application ................. 16
    - 2.3 The class model of OPC Data Access ............................................................... 17
    - 2.4 Communication forms between OPC and SIMATIC NET ................................. 19
    - 2.4.1 Block and variable services .......................................................................... 19
    - 2.4.2 Block services ............................................................................................... 21
    - 2.4.3 Mapping data buffers to OPC variables ....................................................... 22
    - 2.4.4 Using block-oriented services ..................................................................... 22
    - 2.4.5 Variable services .......................................................................................... 23
    - 2.5 Integrating COM components into the .NET environment ................................ 25
  - 3 **Function Mechanisms of this Application** ......................................................... 29
    - 3.1 Data transfer services between OPC client and controller ............................... 29
    - 3.1.1 Variable services ............................................................................................ 29
    - 3.1.2 Block-oriented services ................................................................................ 30
    - 3.2 Identification and connection of process variables .......................................... 30
    - 3.3 Sequence and data flow charts of this application ........................................... 34
    - 3.3.1 Data flow between user interface and controller ........................................... 34
    - 3.3.2 Group assignment and status of items in this application ............................ 35
    - 3.4 Class diagrams of the application ................................................................... 37
    - 3.4.1 Overview ........................................................................................................ 37
    - 3.4.2 The MainForm class ....................................................................................... 38
    - 3.4.3 The BlockServices class ............................................................................. 39
    - 3.4.4 The VariableServices class ......................................................................... 40
    - 3.4.5 Error handling in the application .................................................................. 40

---

V1.0 02.05.05 5/68
4 Explanations on OPC Client ................................................................. 42
  4.1 Structures in the OPC Client ........................................................... 42
  4.2 Connection to the OPC Server ....................................................... 43
  4.3 Block-oriented services ................................................................. 44
     4.3.1 Basic variables for connecting to the OPC server ....................... 44
     4.3.2 Creating OPC groups ............................................................... 45
     4.3.3 OPC Items ............................................................................. 46
     4.3.4 Registering Callback Interface ................................................... 48
     4.3.5 Receiving data from a PLC using AG_SEND/AG_RECV .................. 49
     4.3.6 Sending data to a PLC using AG_SEND/AG_RECV ....................... 52
  4.4 Variable services ............................................................................ 55
     4.4.1 Example: synchronous writing .................................................. 55
     4.4.2 Example: asynchronous reading ................................................ 56

5 Explanations on the S7 Program ......................................................... 58
  5.1 Basic structure ................................................................................. 58
  5.2 Program sequence .......................................................................... 60
  5.3 Calling the Send-/Receive blocks (FC 5 / FC 6) .................................. 63
     5.3.1 AG_SEND (FC 5) ........................................................................ 63
     5.3.2 AG_RECV (FC 6) ........................................................................ 63

6 Modifications on the Example Program ............................................. 64
  6.1 Changes in the STEP 7 program ....................................................... 64
  6.2 Changes at the Visual Basic .NET application .................................... 65

Appendix and List of Further Literature .............................................. 66

7 Glossary (optional) ............................................................................. 66

8 Literature ........................................................................................... 68
  8.1 Bibliographic references ................................................................. 68
  8.2 Internet links ................................................................................... 68
Introduction

1 Document on Hand

This document is the extension of the document “OPC Communication via SEND/RECEIVE Protocol with a Visual Basic .NET OPC-Client – Introduction”.

The Introduction part describes the basic application structure and you should be familiar with its contents before reading the Extension part.

Supplementary to the Introduction part, the document on hand includes further in-depth information on the application. However, it is not a substitute for a manual.
Function Principles and Program Structures

Content

Discusses the detailed function processes of the involved hardware and software components, the solution structures, and where sensible the concrete implementation of this application.

2 General Function Mechanisms

Here you will find further information on …

the general function mechanisms which apply. The main focus is on OPC and the function mechanisms with .NET.

2.1 OPC DA Interface (OPC Data Access)

OPC specifies various services for accessing information at the production process, such as variables, alarms and archive data. The first and most important specification for the variable access is the OPC Data Access Interface Specification. It defines standardized and producer-independent services for process connections. In this context, process connections refers to programmable logic controllers (PLC), field bus systems or other hardware on the field level of automation technology.

Overview

The following figure is meant to illustrate that a process architecture can become very complex and is connected up to the company management level. It becomes apparent, that standardized interfaces make sense for the large number of modules on field level and can save costs.
2.1.1 General

To make the process connection and the actual application as independent from each other as possible, the process connection is realized as an independent application, which is referred to as OPC server. An application which accesses data of such an OPC server is referred to as OPC client.

OPC server and OPC clients can run on different computers. The OPC server, for example, can be provided on a mainframe. The clients could be used for process visualization or recording of process data in an office application such as Microsoft Excel.

The communication between server and client occurs via Industrial Ethernet and the process hardware is connected to the server via producer-specific drivers. The communication between OPC server and OPC client is handled via the OLE services.
Overview

Figure 2-2

2.1.2 OPC Server

An OPC server for data access forms the connection between the OPC interface and one or several data sources.

This makes it the central communication unit between an OPC client and the respective controller.

The OPC server is connected to the respective controller via the implementation of the communication protocols.

Typically, the OPC server is based on special drivers which form the interface with the process hardware. The server provides the applications with an OPC interface which is based on COM/DCOM mechanisms. The applications using the service of the OPC server are referred to as OPC clients.

2.1.3 OPC Client

By generating COM references in the client to the respective interfaces, the OPC server is informed of client access to a certain variable of the configured controller being requested.

Further interfaces available via the "IOPCGroupStateMgt" Group(s) interface enable read and write access to these variables. The following access methods are possible:
- Synchronous reading / writing
- Asynchronous reading / writing
- Monitoring of variables

**Note**

Synchronous read or write jobs are blocking the further execution of the OPC client. For asynchronous read or write jobs, the OPC client remains responsive, i.e. the job is being executed in the background.

For further information please refer to the OPC Specification or the OPC Manual.

### 2.1.4 The role of variables

In order to receive process values, the variables must be specified when calling the OPC interface. The client can request the required values from the server by specifying variables. The client must register each required variable at the server in order to define which values are to be read and when. Variables can be read synchronously as well as asynchronously and be written.

The client can hand over the monitoring of variables to the server. As soon as the value of a variable changes, the server sends a respective message to the client.

#### Dividing the variables

The variables offered by the OPC server can be divided into:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process variables</td>
<td>Represent measuring and control parameters in a process, e.g. the status of input/output devices</td>
</tr>
<tr>
<td>Control variables</td>
<td>Using these variables triggers certain additional services, for example, the transfer of passwords</td>
</tr>
<tr>
<td>Information variables</td>
<td>These variables are provided by the communication system and notify of the status of connection, devices, etc.</td>
</tr>
</tbody>
</table>

**Example for variables**

Here some examples for variables of an OPC Data Access server:

- Control parameters of a programmable logic controller
- Data of a measured data recording system
- Status variables of the communication system
2.1.5 Data source Cache and Device

Differentiation

Two job types are differentiated during reading and writing of values:
- Cache.
- Device.

Device

Reading from or writing to the device means, that the respective read or write job is forwarded directly to the controller/hardware. The read job is being processed in the protocol level. Only after it has been processed there, or after an error, the function returns to the client.

Cache

The cache is a memory area on the OPC server in which the data are filed in their converted format as variant. Accessing this memory does not necessarily trigger a job to the PLC. This means the data are not necessarily up to date.

Overview

Figure 2-3

Note

Synchronous and asynchronous read jobs can be executed by the cache as well as by the device. Write jobs are always executed to the device.
2.1.6 Synchronous Read / Write jobs

Synchronous means that the client directs a read or write job to the server, and only after it has been completely processed is this job returned with a message. For reading from the hardware, with synchronous jobs this means that the job is only returned to the server after a successfully processed job or returned to the client after an error. For slow communication, this may cause considerable delays at the client.

Advantages and disadvantages of synchronous accesses

Table 2-2

<table>
<thead>
<tr>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Simple programming</td>
<td>• The application is interrupted for as long as the synchronous job is</td>
</tr>
<tr>
<td>• High data throughput, as only one process change per job occurs between OPC client and OPC server.</td>
<td>being processed. Only after all data has been read can the application continue. If the function is not called in a separate thread, the user interface of an interactive application, for example, is blocked during the function call.</td>
</tr>
</tbody>
</table>

2.1.7 Asynchronous Read / Write jobs

General information

If an asynchronous communication is established, the client transfers a connection point to the OPC server. The OPC server can now start sending asynchronous messages to the OPC client via this connection point.

A synchronous job returns to the client after registration at the server. Processing now occurs only in the OPC server. As soon as the job is terminated, the client receives data and information on success or failure of the respective job via the connection point.

In the above described manner, the OPC client every time sends a request to the OPC server, to receive the value of a process variable.
Advantages and disadvantages of asynchronous accesses

Table 2-3

<table>
<thead>
<tr>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>• The own application is only interrupted briefly, as the actual communication runs parallel to the application.</td>
<td>• Generating the application is slightly more complex. In the application a call-back mechanism must have been implemented which, at any time, can receive the result of the job processing. Windows programs have built-in asynchronous mechanisms, enabling them to respond to user requests.</td>
</tr>
<tr>
<td>• When transferring few variables in a job, an increased load occurs due to process transitions during call and call-back. Twice the amount as for synchronous accesses.</td>
<td></td>
</tr>
</tbody>
</table>

**Note**

In order to keep the OPC client operable during read/write operations, the asynchronous method must be used!!!

Synchronous calls block the OPC client which frequently causes the user to not receive any feedback from the OPC client and hence he believes the OPC client has crashed.

### 2.1.8 Monitoring variables (DataChange mechanism)

When monitoring variables, the OPC server continuously checks whether the value or the quality of variables has changed.

For this reason the OPC client adds active OPC items to a group and activates the group. Then all active OPC items are monitored in all active groups.

The OPC client provides the DataChange function for this. The OPC server calls this function if a change in value has occurred. As a parameter of DataChange, the OPC server transfers the changed values, qualities and time stamps of the OPC items.

Monitoring of the variables does not put any load on the OPC client. Only after a change has been recognized is the program of the client executed.

In order for the OPC client not to be overloaded with change messages during fast changing process variables, you can specify its minimum call rate via the group specific UpdateRate parameter. (See Figure 2-3).
Monitoring the application area of variables

Monitoring of variables is the ideal solution if a program continuously requires updated data of the process or parts of the process.

Note

The data throughput is determined by the UpdateRate parameter of the group and the cycle time defined in the configuration. The cycle time determines the smallest possible update rate. The update rate should be given as whole multiples of the cycle time.

Advantages and disadvantages of monitoring variables

Table 2-4

<table>
<thead>
<tr>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>• The application is only notified if the process data has changed</td>
<td>• The reaction time from changing a value in the process up to transfer of the new value to the client is larger than the update rate of the group</td>
</tr>
<tr>
<td>• High data throughput because few process changes occur. Good optimization is possible depending on the composition of the item structure.</td>
<td>• Generating the application is slightly more complex as the application requires an asynchronous part for receiving value changes. Windows programs have built-in asynchronous mechanisms, enabling them to respond to user requests.</td>
</tr>
<tr>
<td>• Monitoring of variables can be switched on and off items and group related by the client</td>
<td></td>
</tr>
</tbody>
</table>

2.1.9 Active and passive items

It is possible to create groups or items actively or passively.

For active variables the cache of the OPC server is updated. A continuous communication with the process occurs. This setting is necessary for receiving callbacks for the respective item.

For passive variables only a communication from the device is possible. The cache is not updated. If only individual variables are to be polled specifically, it is better to create these items passively, as the process is not polled continuously.
2.2 Principal user model of the OPC DA interface in this application

The application model of the OPC DA interface is briefly discussed now. A detailed description of the object model is available in the OPC DA Specification.

Figure 2-4
2.3 The class model of OPC Data Access

General information

The hierarchical class model of Data Access helps adjust time expenditure and textual result to the current requirements of an application during data access by the client. Data Access differentiates three classes:

- OPC server
- OPC group
- OPC item

Overview

The following figure illustrates the hierarchical structure of the class mode.

The client application only uses COM calls of the operating system for generating an object of the OPC server class. The other objects are generated by the respective OPC methods of the OPC server class or subordinate classes.

Figure 2-5

OPC server class

The OPC server class is the top class. Every OPC server is part of this class. This class represents the access point for all further services of the OPC Data Access server. Class specific attributes and methods enable you to receive information on status, version and (optionally) the name space of the available process variables. An object of the OPC server class manages the instances of the lower level OPC group class.
OPC group class

Using the OPC group class allows for classifying and structuring the process variables used by the OPC server, as the OPC server class has methods enabling a client to manage group objects (OPC group class). An OPC client can use several objects of this class simultaneously.

Using the objects of OPC Group enables a client to form sensible units of process variables, which are encapsulated into OPC items, and execute them with these operations. For example, all process variables of the screen page of an HMI system can be summarized into a group.

The OPC group defines methods via which the values of the process variables can be read and written. For some methods several variables can be summarized in one job and transferred simultaneously. This is referred to quantity operations, as with calling one method several objects can be manipulated. Especially when using an OPC server via the net these quantity operations enable high execution speeds, as they keep the number of process changes during calling methods of a software component low.

OPC item class

Objects of this class represent the actual process variables and enable aimed polling of individual data. Process variables are identified by an element (item) in the name space of the OPC server and are characterized by an item-ID. 1-1, 1-n or n-1 relationships between process variables and OPC item are possible. The item ID is defined by the producer of the server and must be unique within the name space of the server. The following properties are connected with each item:

Table 2-5

<table>
<thead>
<tr>
<th>Contents</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>last recorded value of the variable.</td>
</tr>
<tr>
<td>Quality</td>
<td>Significance of the value. If the quality is good, the value was determined</td>
</tr>
<tr>
<td></td>
<td>definitely.</td>
</tr>
<tr>
<td>Time stamp</td>
<td>Time at which the current value of a variable was first determined. With</td>
</tr>
<tr>
<td></td>
<td>each value change reported to the client, the time stamp is also updated.</td>
</tr>
<tr>
<td></td>
<td>If the value of a variable does not change, the time stamp also remains</td>
</tr>
<tr>
<td></td>
<td>equal.</td>
</tr>
</tbody>
</table>
Item-IDs

The item-ID is a character sequence which clearly identifies a process variable. It informs the server of which process variable is allocated to the OPC item. The process variable can then be accessed via the OPC item. The OPC server of SIMATIC NET maps the various communication services of the protocols via OPC items, by using parts of the item-ID as parameter for calling a communication function.

For the OPC server of Simatic NET the following syntax applies for the item IDs:

<Protocol-ID>: [<Connection name>] <Variable name>

Table 2-6

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Protocol-ID</td>
<td>Specifies the protocol to be used.</td>
</tr>
<tr>
<td>Connection name</td>
<td>Name of the configured connection via which the device can be reached.</td>
</tr>
<tr>
<td>Variable name</td>
<td>Determines the variable of the device to be used.</td>
</tr>
</tbody>
</table>

2.4 Communication forms between OPC and SIMATIC NET

2.4.1 Block and variable services

The values of the variables are read out via the connections configured in the OPC server (with NetPro) using the respective protocol blocks.

It must be noted, that the protocol blocks are not standardized interfaces, but proprietary protocols implemented for the respective underlying controller.

Reading and updating the variables can be performed in two ways:
### Via variable services

**Table 2-7**

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Variable services diagram" /></td>
<td>For variable services the variables to be monitored are polled from the OPC server cyclically. The OPC server <strong>actively</strong> updates the process variables to be monitored.</td>
</tr>
</tbody>
</table>

### Via block services

**Table 2-8**

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Block services diagram" /></td>
<td>If the variables to be monitored are transferred to the OPC server in a program controlled manner by means of larger data blocks, then this is referred to as block service. Thereby the control program triggers the variable transfer via communication blocks. Here the controller is the active partner.</td>
</tr>
</tbody>
</table>
Note
In this application both services are introduced. Block services should be used for large data volumes. Variable services for individual requests. A comparison between variable and block services is available in the Introduction document of this application.

Supported services
For block-oriented services, the contents of a data buffer are sent to a receiver by a sender via the communication system.
The following protocols support block-oriented services:

- S7 communication service (BSEND / BRCV)
- S5-compatible communication via Ethernet (SEND/RECEIVE)
- S5-compatible communication via PROFIBUS (SDA, SDN/Indication)

Note
Characteristic for block-oriented services is that data are only transferred when the sender triggers the transfer process. The receiver cannot trigger the communication.

2.4.2 Block services

Actions by the sender during exchanging of data buffers
- Compiles a send buffer with the contents.
- Sends the buffer to a communication partner on a connection.
- Receives an acknowledgement via the result of the data transmission.

Actions by the receiver during exchanging of data buffers
- Provides a receive buffer for a connection.
- Will be notified if a partner sends him a data buffer.
- Evaluates the received data.

Selection criteria for selecting the communication service

<table>
<thead>
<tr>
<th>Criterion</th>
<th>AG_SEND</th>
<th>BSEND</th>
</tr>
</thead>
<tbody>
<tr>
<td>S5-compatible communication</td>
<td>+</td>
<td></td>
</tr>
<tr>
<td>S7 communication</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>Implied acknowledgement</td>
<td>-</td>
<td>+</td>
</tr>
</tbody>
</table>
2.4.3 Mapping data buffers to OPC variables

The OPC Data Access interface only knows process variables. In order for the advantages of block-oriented services also being available with OPC, there must be a mapping of data buffers on OPC items.

Differences between send and receive items:

Send item
- S7 communication: "BSEND" S5-compatible communication: “SEND”
  - An OPC item represents a send buffer or a part of a send buffer.
  - If the OPC item is written (synchronous/asynchronous), a write job is triggered on the network.
  - If several items, which represent a partial area of a buffer, are written at once in a quantity operation, the complete send buffer is first formed of all partial sections and the send.
  - The read access delivers the data sent last from the send buffer. If not yet sent, the item can be read under the circumstances mentioned, however, it has the quality “BAD”.

Receive item
- S7 communication: “BRCV”, S5-compatible communication: “RECEIVE”.
  - An OPC item represents a receive buffer.
  - If the OPC item is read by the device (synchronous/asynchronous), the module becomes ready for receiving. This state remains until a data package was received or the connection specific timeout has elapsed. If no data package was received during the timeout interval, the quality of the OPC item is "BAD".
  - If the OPC item is monitored (active receive item in active group), a receive buffer is established permanently within the communication module. If a data package is received, the OnDataChange callback signals this to the application – as long as the data differ from the previously received data.

2.4.4 Using block-oriented services

Handling of send and receive items:

Send items
- They should only be described, the reading or monitoring of send items only delivers the previously written data.
Receive item

They should be monitored, i.e. the receive item should exist as active item within an active group. This updates the cache of the OPC server with every arrival of a data package - irrespective of the implementation of a callback function.

Advantages / disadvantages during communication with the block services of the SEND/RECEIVE protocol

Table 2-10

<table>
<thead>
<tr>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Larger data blocks up to 64 Kbytes can be transferred with the OPC server.</td>
<td>• Receiver cannot trigger the data transfer. It must wait for the transmission of data by the sender.</td>
</tr>
<tr>
<td>• No net load as there is no data transfer unless triggered by the user program (no polling).</td>
<td>• The data must be located in a buffer or be copied to a buffer by a user program in the partner device.</td>
</tr>
<tr>
<td>• When using the OPC server, a structured access to the data blocks is possible.</td>
<td></td>
</tr>
<tr>
<td>• Communication with S5 and S7 devices as well as PCs possible.</td>
<td></td>
</tr>
</tbody>
</table>

2.4.5 Variable services

Variables of an S5 or S7 automation system can simply, via icons be accessed with the S5-compatible communication on Industrial Ethernet. The services "FETCH" and "WRITE" are used for this.

When executing the "FETCH" service a job is sent to the partner device which contains a specification of the requested variable as source parameter. The receiver of the job evaluates the source parameter and returns the current content of the requested variable with the confirmation of the job.

With a job for the "WRITE" service the value which the variable is to take on at the target device is transmitted as well as the source parameter. The receiver evaluates the source parameter and sets the given variable accordingly. Then the job confirmation is sent back.

Variables are normally sections of an S7 or S5 object from a certain storage address of an object onward, whom a certain file type is assigned to. For example, the variable with the notation "DB 5, B10" accesses the byte 10 of the data block 5 and represents the content as a word. The formation of field variables – several elements of the same type in one variable – is also possible.
The objects available on most S5 and S7 devices are:

- Data blocks
- Inputs/outputs
- Distributed I/O
- Memory bits
- Timer
- Counter
- System area
- Extended distributed I/O

Advantages / disadvantages during communication with the variable services

<table>
<thead>
<tr>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flexible access</td>
<td>Low data throughput compared with block services for SEND/RECEIVE.</td>
</tr>
<tr>
<td>Automatic optimization with the OPC server possible</td>
<td>For monitoring variable changes the partner station must be accessed cyclically.</td>
</tr>
</tbody>
</table>
2.5 Integrating COM components into the .NET environment

The COM components origin from the times before .NET-Framework, therefore adjustments are necessary for integrating them into .NET applications and being able to use them.

Overview

Using a COM component in a .NET application/component:

Figure 2-6

RCW and the relationship with the COM component

Installing a COM component, generated for the previous Windows world, into a .Net application, requires a "wrapper", a type of mask which encapsulates all interface definitions for the .NET application.

This is necessary as the previous interface definitions located within so-called IDL files are not supported by .NET. This wrapper is also referred to as Runtime Callable Wrapper (RCW).

For COM components offering the automation interface, Visual Studio can generate these wrappers automatically.

For COM component providing the custom interface, such wrappers must be generated manually.

Note

The differences between automation and custom interface are not discussed here in greater detail.
Memory management when using COM components

Memory management for .NET applications is performed by the Garbage Collector, however, COM components require an explicit memory management. Therefore, the following points must be considered:

- Data exchange from .NET to COM
- Data exchange from COM to .NET

Data exchange from .NET to COM

In .NET, all variables are objects. Due to the fact that generally speaking, COM servers do not synchronize with COM clients (a .NET-RCW in this case), transfer values should be protected from the access of the Garbage Collector ("pinning" of objects).

Data exchange from COM to .NET

COM components deliver returned values in form of COM pointers. Due to the fact that in the managed code of .NET clients no such pointers exist, the returned values must first be saved into .NET objects. This is performed via the .NET data type "IntPtr" and methods / objects of the "System.Marshal" classes.

RCW and OPC

Due to the OPC specification, OPC components offer a Custom Interface; an RCW must be generated manually for performant applications. For the SIMATIC NET OPC server an RC Wrapper is delivered for the Data Access interface V2.05 which only needs to be integrated into the respective Visual Studio project.

If the automation interface of OPC is used with .NET, then the RCW is automatically generated by Visual Studio .NET as soon as a reference to the OPC automation interface is added to the .NET project. The resulting double encapsulation of the OPC custom interface however, may cause a reduction in performance.

Note
This application deals with the usage of the custom interface, as a performant connection is the focus of this application.

Note
The reverse application case – using a .NET component as COM component – is possible using a Com Callable Wrapper (CCW). This however, is not further discussed within the framework of this document, as it is not necessary for comprehension of this application.
Integrating the SIMATIC NET OPC DA interface into Visual Studio .NET

The following prerequisites must be fulfilled:

- SIMATIC NET V6.2 has been installed
- Visual Studio .NET started with a .NET-Windows application

Installation of .NET RCW

In order to access the OPC-DA interface, a reference to OPC-DA-RC-Wrapper must be generated within the project.

**Note**
The necessity of an RCW and what it represents is explained in chapter 2.4.

Table 2-12

<table>
<thead>
<tr>
<th>No.</th>
<th>Instructions</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>In the project you click Add reference...</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>In the following dialog you click Browse and select the files OpcRcw.Comn.dll, OpcRcw.Da.dll and DataConversion.dll which are located at in your project directory. Acknowledge with Open and then with OK. <strong>Note</strong> The files OpcRcw.Comn.dll und OpcRcw.Da.dll are part of the SIMATIC NET software and are also available at: &lt;LW&gt;:...\Siemens\SIMATIC.NET\opc2\bin</td>
<td></td>
</tr>
</tbody>
</table>
The RCW as well as the DataConversion library now appear at the references and can be used in your .NET project.

Chapter 4 contains a description of how the respective OPC-DA interfaces must be used.
3 Function Mechanisms of this Application

This chapter explains ...

how the classes interact in the OPC client and which tasks the most important components fulfill, as well as the realization of block and variable-oriented communication.

First the logic data connection between controller and OPC client is described. Hereby, the corresponding data structures in controller and OPC client are listed. Then you will learn more about the data flow within the application using data flow charts. Also, the structure of the OPC client will be explained to you with class diagrams.

3.1 Data transfer services between OPC client and controller

3.1.1 Variable services

Variable services enable direct access to and monitoring of variables in the automation device. Addressing of the variables occurs symbolically. For direct read access to variables the OPC server transfers the desired address information to the PLC and then returns the requested data. For a write access, the OPC server transfers the address information, together with the value to be written, to the partner station (PLC).

Syntax of the process variables for SEND/RECEIVE variable services

SR: [<Connection name>] <Area>{,}<Type><Address>{,<Number>}

As an example, the declaration of two Visual Basic .NET variables is mentioned here which are used for the write job. The process variable is referred to via the declared constant. As a fixed expression, SR stands for the SEND/RECEIVE protocol. In squared brackets behind it there is the name of the configured connection. After that respectively one memory byte is being referred to, which is the data object to be addressed.

```
Private Const m_RecvDoSendItemName As String = "SR:[Write]MB1,1";
Private Const m_RecvAckItemName As String = "SR:[Write]MB2,1";
```
3.1.2 Block-oriented services

The block-oriented services enable program controlled transmission of larger data blocks. These services are also referred to as SEND/RECEIVE services. The transmission with the OPC server is realized with variables:

- Variables which receive data blocks.
- Variables which send data blocks.

Default variable names

The following variable names are default for every connection:

- RECEIVE
- SEND

Syntax of the process variables for block-oriented services

There are the following options:

\[
\text{SR:}[\langle\text{Connection name}\rangle]\text{receive}\{,\langle\text{Type}\rangle<\text{Address}\}{,\langle\text{Number}\rangle}\}
\]

\[
\text{SR:}[\langle\text{Connection name}\rangle]\text{send}\{\langle\text{n}\rangle}\{,\langle\text{Type}\rangle<\text{Address}\}{,\langle\text{Number}\rangle}\}
\]

As an example, the following declaration of a variable is displayed in the Visual Basic .NET program, which can receive data blocks (receive). We are dealing with a byte addressing.

```
Private Const m_SendAckItemName As String = SR:|[PC=>S7]receive,B0,12";
```

3.2 Identification and connection of process variables

If a user wishes to implement an OPC client himself, it is important to understand how process variables are identified at OPC. Furthermore, it is explained how the variables are connected in the example application.

Meaning of OPC server and client handles

In order for the created OPC items and OPC groups to be uniquely identified, each OPC client must manage the so-called handles (e.g. in form of member variables or arrays).

Thereby it is differentiated between client and server handles. This is necessary in order to receive unique assignment for several OPC clients as well (two OPC clients can have identical ClientHandles). Thereby it is important for the OPC client how it wants to request data at the OPC server.
Client handles

- Are transferred from the OPC client during the generation process.
- Are used for later identification of OPC groups and items, if the call direction is OPC server → OPC client (necessary for callbacks).
- Are necessary if the OPC client has registered for a callback at the OPC server. This means that it is notified during value changes of an item in a callback function. In this function it can identify the respective item with the Client handle.

Using Client handles in the application

If a message is sent from OPC server to OPC client, the OPC client must identify and allocate the items using the client handles. Such a usage corresponds to the receiving of OPC results.

For asynchronous write/read calls and for "Active groups / items" OPC callback methods are called.

Note

Client handles should as a rule correspond with array indices of the respective variables in the OPC client or identify control elements in the visualization user interface.

Server handles:

- Are returned from the OPC server after creating an object (group or item).
- Are used for later identification of OPC groups and items, if the call direction is OPC client → OPC server, and are required for explicit Read and Write calls of the OPC client, irrespective of whether they are synchronous or asynchronous. For these calls the client sends the server handles with the items for which it wants to send a call. In the server, the respective items can be returned using these transferred server handles.
Using Server handles in the application

For requests from the OPC client to the OPC server it may be a write or read call in synchronous as well as asynchronous form. The following figure illustrates which assignment of variables exist between OPC client and OPC server and in which variables the respective server handles are memorized in the OPC client.

Figure 3-2

Connecting the process variables with visualization elements

The following figures illustrate the connections between variable table and user interface.

Connection during receiving

The variables pressure, temperature and humidity in UDT 1 are filed as byte-array in the DB20 with a header consisting of an ID and a time stamp. This array is separated after a RECEIVE in the Visual Basic .NET application and displayed accordingly in the text fields.

Figure 3-3
Connection during sending

The variables name, ValveState and voltage on the Visual Basic .NET user interface can be sent block-oriented to the controller.

Procedure during time measurement

In order to design the program in the PLC as simple as possible, the average transaction time is determined in the PC program. Below it is schematically illustrated which time is measured in the send/receive job for the block services.

Table 3-1

<table>
<thead>
<tr>
<th>Direction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>S7=&gt;PC</td>
<td>The time t1 corresponds to the measured time for a write job.</td>
</tr>
<tr>
<td>PC=&gt;S7</td>
<td>The time t2 corresponds to the measured time for a read job.</td>
</tr>
</tbody>
</table>
3.3 Sequence and data flow charts of this application

The dynamic model of this application is described below. It shows which time processes occur and how the application behaves.

3.3.1 Data flow between user interface and controller

Send connection

The figure below also shows the data flow during a send connection.

![Diagram showing data flow](image)

Explanation

<table>
<thead>
<tr>
<th>No.</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>When pressing the “S7-&gt;PC” button the “DoSend” byte in the controller is set to “1”</td>
</tr>
<tr>
<td>2</td>
<td>This causes the FC1 to generate a telegram and file it in the DB20.</td>
</tr>
<tr>
<td>3</td>
<td>Using the AG_SEND the data is then sent to the application.</td>
</tr>
<tr>
<td>4</td>
<td>It confirms the reception of the data package by setting the variable “Ack_Send.Done” to “1”.</td>
</tr>
</tbody>
</table>
Receive connection

The figure below also shows the data flow during a receive connection.

Figure 3-6

```
<table>
<thead>
<tr>
<th>No.</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>When pressing the “PC-&gt;S7“ button, the PC generates a telegram with the above data and sends it to the controller.</td>
</tr>
<tr>
<td>2</td>
<td>It receives the data package via the AG_RECV and files it in the DB30.</td>
</tr>
<tr>
<td>3</td>
<td>The FC2 extracts the telegram as header and user data.</td>
</tr>
<tr>
<td>4</td>
<td>Then it sends the extracted header back to the application via the AG_SEND, which serves as an acknowledgement.</td>
</tr>
</tbody>
</table>
```

Note

It is also possible to refrain from FETCH / WRITE and work exclusively with AG_SEND / AG_RECV.

However, a higher programming workload must be taken into account.

3.3.2 Group assignment and status of items in this application

In the table below it is illustrated which items were added to the groups in which state, or respectively which group status prevailed.
Block services

2 groups are created for the block services:

- Group ActiveGroupSR
- Group PassiveGroupSR

Generally, the items are created active or passive depending on the group status.

<table>
<thead>
<tr>
<th>Item</th>
<th>Group</th>
<th>State</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>SR:[S7 =&gt; PC]receive,B0, +Address</td>
<td>ActiveGroupSR</td>
<td>active</td>
<td></td>
</tr>
<tr>
<td>SR:[PC =&gt; S7]receive,B0, 12</td>
<td>ActiveGroupSR</td>
<td>active</td>
<td></td>
</tr>
<tr>
<td>SR:[Write]MB1,1&quot;</td>
<td>PassiveGroupSR</td>
<td>passive</td>
<td>Flow control S7 (\rightarrow) PC</td>
</tr>
<tr>
<td>SR:[Write]MB2,1</td>
<td>PassiveGroupSR</td>
<td>passive</td>
<td>Acknowledgment S7 (\rightarrow) PC</td>
</tr>
<tr>
<td>SR:[PC =&gt; S7]send,B0, +Address</td>
<td>PassiveGroupSR</td>
<td>passive</td>
<td></td>
</tr>
</tbody>
</table>

Note: Address refers to the beginning of the ByteArray, which is created from the size of the telegram header and the received UDT size.

Variable services

For the variable services 2 groups are also created:

- Group FetchGroup
- Group WriteGroup

A group for the write variables is created as well as a group for the read variables. Both groups are passive.

<table>
<thead>
<tr>
<th>Item</th>
<th>Group</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>SR:[Fetch]MD200</td>
<td>FetchGroup</td>
<td>passive</td>
</tr>
<tr>
<td>SR:[Fetch]MD204</td>
<td>FetchGroup</td>
<td>passive</td>
</tr>
<tr>
<td>SR:[Fetch]MD208</td>
<td>FetchGroup</td>
<td>passive</td>
</tr>
<tr>
<td>SR:[Write]MD200</td>
<td>WriteGroup</td>
<td>passive</td>
</tr>
<tr>
<td>SR:[Write]MD204</td>
<td>WriteGroup</td>
<td>passive</td>
</tr>
<tr>
<td>SR:[Write]MD208</td>
<td>WriteGroup</td>
<td>passive</td>
</tr>
</tbody>
</table>
3.4 Class diagrams of the application

In the following chapter the static structure of the application is described. It shows how the classes of the Visual Basic .NET application are designed and which object-oriented structuring of the task was made.

3.4.1 Overview

Class diagram

The figure below illustrates the inter-dependencies between the classes.

Figure 3-7
Explanation of the class diagram

It is apparent that a MainForm class exists which collects the interactions with the user and passes them on to the respective class responsible for the action.

Here the specialized classes are firstly BlockServices, which are responsible for managing the block-oriented services and secondly the VariablenServices classes responsible for variable-oriented services. Between the MainForm class and the class responsible for the service there is a forward and backward reference.

3.4.2 The MainForm class

Tasks

The tasks of this class are:

- Connecting with the SIMATIC OPC server
- Separating the connection from the OPC server for as long as there is a connection with the server.
- Initializing the BlockServices and VariableServices classes
- Receiving the input and output parameters
- Form management of the graphical user interface such as events handling on button-click, activating and deactivating the buttons, display and reading of text boxes. Most important methods

Table 3-6

<table>
<thead>
<tr>
<th>Method name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>btnConnect_Click</td>
<td>Clicking the &quot;Connect&quot; button calls this routine. Only possible if no connection has been established yet. Generating the connection with the SIMATIC NET OPC server. Initializing the BlockServices and VariableServices classes with the required parameters from the established connection</td>
</tr>
<tr>
<td>btnDisconnect_Click</td>
<td>Clicking the &quot;Disconnect&quot; button calls this routine. Only possible after successfully established connection. Terminating the server and disconnecting. Calling the disconnect method from the BlockServices and VariableServices classes in which the clean-up works are processed.</td>
</tr>
</tbody>
</table>
3.4.3 The BlockServices class

Tasks

The tasks of this class consist of:

- OPC connection management (adding of groups and items)
- Sending and receiving with block services (synchronous reading, asynchronous writing)
- Calculating the transaction time
- Using the conversion-dll (DataConversion.dll) for data conversion between S7 and PC data type.

Most important methods

Table 3-7

<table>
<thead>
<tr>
<th>Method name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Connect</td>
<td>Adding groups and items. Creating the callback object for the synchronous operations.</td>
</tr>
<tr>
<td>Disconnect</td>
<td>Clean-up works during disconnecting from the server. Unregistering the callback and releasing memory of no longer required objects.</td>
</tr>
<tr>
<td>ReceiveData</td>
<td>Synchronous read job with block services is called.</td>
</tr>
<tr>
<td>ProcessReceivedData</td>
<td>Calling a DataConversion.dll method for converting received data in the respective data type and transfer to MainForm for display. Calculating the transaction time</td>
</tr>
<tr>
<td>ProcessSendDataReceipt</td>
<td>Calling a DataConversion.dll method for transforming the data to be sent into a byte array. Calculating the transaction time</td>
</tr>
<tr>
<td>SendData</td>
<td>Method for telegram generation is called and an asynchronous write job to the OPC server (by means of block services) is triggered.</td>
</tr>
<tr>
<td>OnDataChange</td>
<td>Callback function for the active items. During value change and elapsed update rate there is a call to this method.</td>
</tr>
</tbody>
</table>
3.4.4 The VariableServices class

Tasks

The tasks of this class are:

- OPC connection management (adding of groups and items)
- Sending and receiving with variable services (asynchronous reading, synchronous writing)

Most important methods

Table 3-8

<table>
<thead>
<tr>
<th>Method name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Connect</td>
<td>Adding groups and items. Creating the callback object and registering at the callback for the asynchronous operations.</td>
</tr>
<tr>
<td>Disconnect</td>
<td>Clean-up works during disconnecting from the server. Unregistering the callback and releasing memory of no longer required objects.</td>
</tr>
<tr>
<td>FetchData</td>
<td>Asynchronous read call using variable services</td>
</tr>
<tr>
<td>OnFetchedData</td>
<td>Callback method for asynchronous read job. Calling the respective display methods in the MainForm class</td>
</tr>
<tr>
<td>WriteData</td>
<td>Synchronous write job</td>
</tr>
</tbody>
</table>

3.4.5 Error handling in the application

General information

The error handling in this application is performed by means of "unstructured exception handling". In Visual Basic occurring errors can be delegated to a label with the "On Error Goto Label" instruction, with the respective instructions in case of an error. Using "On Error Resume Next" the application can be made more secure in case of an error.

During an error in this application it is ensured that objects no longer used are released and further operability of the application is provided.

Another option for error handling is the "structured exception handling". The commands "Try", "Catch" and "Finally" are used for this. This type of error handling is not used in this application, therefore it is not further discussed here.

Example

If establishing the connection with the SIMATIC NET OPC server fails for certain reasons, it is ensured in the error handler that the "Connect" button is activated again. This ensures, that the user can reinitiate a connection.
Resolving an exception

Should errors for jobs occur during communication, for example during a write job, an exception is thrown with the respective description as parameter. In Visual Basic this occurs with the keyword “throw”:

```vbnet
If Errors(0) <> 0 Then
    Throw New ApplicationException("Writing AckByte failed.")
End If
```

Using error codes

As return values of the OPC functions, certain error codes (available in the OPC Specification) can provide information on the status of the method. With aimed polling of the error code a message can be sent to the user during call failure, or a different reaction of the application be initiated.
4 Explanations on OPC Client

Here you will find information on …

the structures realized in the OPC client. Mainly creating of groups and items is discussed here.

Note

For further information, please refer to the commented source code.

4.1 Structures in the OPC Client

the structures realized in the OPC client.

The OPC client uses classes instead of structures.

As far as sensible within the framework of this application, object-oriented methods are used.

The class diagrams as well as the tasks of the most important classes have already been discussed in chapter 3.4 Class diagrams of the application.

The MainForm class represents the implementation of the dialog and is at the same time used as central container for the other classes used. Instancing of the OPC server is a central task as well, therefore, it is also performed here.

In the BlockServices class, the AG_SEND/AG_RECEIVE functionality is encapsulated. It manages own groups, has own callback interface and handles the creating and releasing of resources.

The counterpart is the VariableServices class. It implements the variable services and has, like the BlockServices class, its own resources.
4.2 Connection to the OPC Server

how to connect to the OPC server with .NET.

The connection to the OPC server occurs with the "btnConnect_Click" method of the "MainForm" dialog:

Figure 4-1

```vbnet
Private Sub btnConnect_Click(ByVal sender As System.Object, ByVal e As _
    System.EventArgs) Handles btnConnect.Click
    On Error GoTo CleanUp
    txtConnect.Enabled = False
    txtUpdateRate.Enabled = False

    ' Create the OPC server object and query for the IOPCServer interface
    Dim o OPC Server As Type = type.GetTypeFromProgID(txtServerName.Text)
    m_OPC Server = Activator.CreateInstance(type OPC Server)

    Dim UpdateRate As Integer = System.Convert.ToInt32(txtUpdateRate.Text)
    Dim RevisedUpdateRate As Integer = 0
    Dim LCID As Integer = Thread.CurrentThread.CurrentCulture.LCID + &H0000

    ' Initialize the BlockServices class
    m_Block Services.Connect(m_OPC Server, UpdateRate, LCID, RevisedUpdateRate)

    ' Initialize the VariableServices class
    m_Variable Services.Connect(m_OPC Server, LCID, tx Item Definition 1.Text,
        tx Item Definition 2.Text, tx Item Definition 3.Text)

    ' If the revised update rate differs from the requested one...
    If RevisedUpdateRate <> UpdateRate Then
        txtUpdateRate.Text = System.Convert.ToString(RevisedUpdateRate)
        System.Windows.Forms.MessageBox.Show(m_LanguageManager.GetString("UpdateRateWarning"), _
            m_LanguageManager.GetString("Warning"), MessageBoxButtons.OK, MessageBoxIcon.Warning)
    End If

    btnDisconnect.Enabled = True
    btnFetch.Enabled = True
    txtWrite.Enabled = True
    btnSelObj.Enabled = True
    Exit Sub

CleanUp:

    ' Free OPC server
    If Not m_OPC Server Is Nothing Then
        m_OPC Server.Release COM object
    End If

    ' Free the BlockServices etc.
    m_Block Services = Nothing

    ' Here you could add a message for your purpose

    ' To ensure operability enable buttons
    btnConnect.Enabled = True
    txtUpdateRate.Enabled = True

End Sub
```
4.3 **Block-oriented services**

The block-oriented services, i.e. AG_SEND/AG_RECV, are implemented in the "BlockServices" class.

The OPC-specific topics represented below refer to the source code of the "BlockServices.vb" file.

4.3.1 **Basic variables for connecting to the OPC server**

The class requires the following values for connecting to the OPC server:

**Connection to the OPC server incl. groups**

- `Private m_Server As IOPCServer = Nothing`
- `Private m_ActiveGroupHandle As Integer = 0`
- `Private m_PassiveGroupHandle As Integer = 0`
- `Private m_ActiveGroup As IOPCGroupStateMgt = Nothing`
- `Private m_PassiveGroup As IOPCGroupStateMgt = Nothing`

Furthermore, the following variables are required for the provision of the functionalities.

**Memory bit for connecting to the ConnectionPointContainer (so-called cookies)**

- `Private m_ActiveGroupCallbackCookie As Integer = 0`
- `Private m_PassiveGroupCallbackCookie As Integer = 0`

**Handles**

<table>
<thead>
<tr>
<th>Handle</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generally used handles</td>
<td>These handles are used for the send as well as the receive direction.</td>
</tr>
<tr>
<td></td>
<td><code>Private Const m_AckItemClientHandle As Integer = 1</code></td>
</tr>
<tr>
<td></td>
<td><code>Private Const m_DataItemClientHandle As Integer = 2</code></td>
</tr>
<tr>
<td>Handles, used for receiving</td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>Private Const m_DoSendItemClientHandle As Integer = 3</code></td>
</tr>
<tr>
<td></td>
<td><code>Private m_RecvDoSendItemServerHandle As Integer = 0</code></td>
</tr>
<tr>
<td></td>
<td><code>Private m_RecvAckItemServerHandle As Integer = 0</code></td>
</tr>
<tr>
<td></td>
<td><code>Private m_RecvDataItemServerHandle As Integer = 0</code></td>
</tr>
<tr>
<td>Handles, used for sending</td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>Private m_SendAckItemServerHandle As Integer = 0</code></td>
</tr>
<tr>
<td></td>
<td><code>Private m_SendDataItemServerHandle As Integer = 0</code></td>
</tr>
</tbody>
</table>
4.3.2 Creating OPC groups

Code excerpt

how to proceed for creating OPC groups in .NET.

Creating of groups was encapsulated with the “Connect(...)” method.

Figure 4-2

```csharp
' add an active group to accommodate the receive items
m_Server.AddGroup("ActiveGroup", 1, UpdateRate, 1, pTimeBias, _
  pDeadband, LocaleId, m_ActiveGroupHandle, _
  RevisedUpdateRate, GetType(IOPCGroupStateMgt).GUID, _
  m_ActiveGroup)

' add a passive group to accommodate the other items
m_Server.AddGroup("PassiveGroup", 0, UpdateRate, 2, pTimeBias, _
  pDeadband, LocaleId, m_PassiveGroupHandle, _
  RevisedUpdateRate, GetType(IOPCGroupStateMgt).GUID, _
  m_PassiveGroup)
```

Explanation

The code excerpt only shows the call of the OPC method “AddGroup”.

Please note that the required update rate need not correspond with the actual update rate. Latter one is returned with the RevisedUpdateRate variable.

Note

An overview of the used groups and items and their state is available in chapter 3.3.2.
4.3.3 OPC Items

Creating OPC items

how to create .NET OPC items and what needs to be observed.
Creating of items was encapsulated within the “Connect” method.
The following code section illustrates how item definitions are created in
darrays and then transferred to the “AddItems” method.

Figure 4-3

```
Dim ItemDefinitions(3) As OPCITEMDEF
Dim pResults As IntPtr = IntPtr.Zero
Dim pErrors As IntPtr = IntPtr.Zero
Dim ItemMgt As OPCItemMgt = Nothing

' set up item 0
ItemDefinitions(0).szAccessPath = ""
ItemDefinitions(0).szItemID = m_SendAckItemName
ItemDefinitions(0).bActive = 1
ItemDefinitions(0).hClient = m_AckItemClientHandle
ItemDefinitions(0).vtRequestedDataType = VarEnum.VT_ARRAY Or VarEnum.VT_UI1
ItemDefinitions(0).dwBlobSize = 0
ItemDefinitions(0).pBlob = IntPtr.Zero

' set up item 1
ItemDefinitions(1).szAccessPath = ""
ItemDefinitions(1).szItemID = m_RecvDataItemName & 
    Convert.ToString(m_RecvDataItemSize + (m_RecvDataItemSize + 
    m_RecvDataItemCount) / 2))
ItemDefinitions(1).bActive = 1
ItemDefinitions(1).hClient = m_DataItemClientHandle
ItemDefinitions(1).vtRequestedDataType = VarEnum.VT_ARRAY Or VarEnum.VT_UI1
ItemDefinitions(1).dwBlobSize = 0
ItemDefinitions(1).pBlob = IntPtr.Zero

' adding two items to the active group
ItemMgt = m_ActiveGroup
ItemMgt.AddItem(2, ItemDefinitions, pResults, pErrors)
ItemMgt = Nothing
```

Properties of OPC items

For each item the following properties must be set:

<table>
<thead>
<tr>
<th>Property</th>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>szAccessPath</td>
<td>not used</td>
</tr>
<tr>
<td>szItemID</td>
<td>e.g. &quot;SR:[MyConnection]receive,B0,12&quot;</td>
</tr>
<tr>
<td>bActive</td>
<td>if True, then the item is actively created</td>
</tr>
<tr>
<td>hClient</td>
<td>ClientHandle; can be assigned by the user</td>
</tr>
<tr>
<td>vtRequestedDataType</td>
<td>Requested data type</td>
</tr>
<tr>
<td>dwBlobSize</td>
<td>not used</td>
</tr>
<tr>
<td>pBlob</td>
<td>not used</td>
</tr>
</tbody>
</table>
Evaluating the error array

The following code extract shows how the error array can be evaluated and which measures must be taken regarding the COM interoperability.

```vba
Dim Errors(9) As Integer
Dim Result As OPCITEMRESULT
Dim pos As IntPtr = pResults
Dim i As Integer

' Copy the errors array
Marshal.Copy(pErrors, Errors, 0, 2)

' Check for errors
For i = 0 To 2
    If Errors(i) <> 0 Then
        Dim ex As Exception = New Exception("Error while adding an Item")
        Throw ex
    Else
        Result = Marshal.PtrToStructure(pos, GetType(OPCITEMRESULT))
        Select Case i
            Case 0
                m_SendAckItemServerHandle = Result.hServer
            Case 1
                m_RecvDataItemServerHandle = Result.hServer
        End Select
    End If
    'Move to the next element in the array
    pos = New IntPtr(pos.ToInt32() + Marshal.SizeOf(GetType(OPCITEMRESULT)))
Next

' Free allocated COM-resources
Marshal.FreeCoTaskMem(pResults)
pResults = IntPtr.Zero
Marshal.FreeCoTaskMem(pErrors)
Errors = IntPtr.Zero
```

Notes for using the .NET Wrappers

The fact that the .NET Wrapper requires only a thin layer between .NET and COM, requires some converting work. The error array for example must be copied from the unmanaged to the managed memory of .NET.

Furthermore, access to the OPCITEMRESULT is only possible with copying the data. This data will give us the ServerHandle returned by the server. For an active access to an item at a later time, the ServerHandle for it must be memorized. If an item is only to be monitored, the ServerHandle is not required.

The error and result arrays generated by the OPC server must be released from the client. This occurs via FreeCoTaskMem. Setting the invalid pointer to 0 is no obligation, however, it is decent programming style and facilitates the search for errors.
4.3.4 Registering Callback Interface

Code excerpt

how to create .NET OPC items and what needs to be observed.
Registering the callback interface was encapsulated within the “Connect“
method.

Figure 4-5

`Query interface for callbacks on active group object
Dim ConnectionContainer As IConnectionPointContainer = m_ActiveGroup

`Establish Callback for all async operations
Dim ConnectionPoint As IConnectionPoint = Nothing
ConnectionContainer.FindConnectionPoint(GetType(IOPCDataCallback).GUID, _
    ConnectionPoint)
ConnectionContainer = Nothing

`Creates a connection between the OPC servers's connection point and
' this client's sink (the callback object).
ConnectionPoint.Advise(Me, m_ActiveGroupCallbackCookie)
ConnectionPoint = Nothing

Explanation

The procedure is quite clear here. The "ConnectionPoint" for the
IOPCDatacallback interface can be fetched via the
"ConnectionPointContainer" interface of an OPC group. The own
IOPCDatacallback interface is then registered there with the Advise call.
For disconnecting the own Callback interface from the server, the
"Unadvise" method of the ConnectionPoints is used.

Note

The Unadvise method at the ConnectionPoint must be called when
disconnecting from the server, as otherwise not all resources are being
released.
4.3.5 Receiving data from a PLC using AG_SEND/AG_RECV

Initiating the PLC to send the data

In order to start the transfer from the PLC towards the PC, the PC writes a byte, the “DoSend” byte, via the “Write” connection. As soon as the value of this byte changes from 0 to 1 the PLC starts the transfer (see chapter 5 Explanations on the S7 Program).

Code extract – Starting the transfer

Figure 4-6

```
Public Sub ReceiveData()

' Query for IOPCsyncIO interface
Dim SyncIO As IOPCsyncIO = m_PassiveGroup

Dim ServerHandles(1) As Integer
Dim Values(1) As Object
Dim pErrors As IntPtr = IntPtr.Zero
Dim Errors(1) As Integer

' Set server handle of the "DoSend" item in order to to get the plc
'simulation to send data with AG_SEND
ServerHandles(0) = m_RecvDoSendItemServerHandle

' Set value to 1 => PLC will send data
Values(0) = 1

' Prepare time measurement
m_RecvStartTime = DateTime.Now
'Data can be received
m_RecvInProgress = True

' Write value to PLC
SyncIO.Write(1, ServerHandles, Values, pErrors)
SyncIO = Nothing

' Copy the errors array
Marshal.Copy(pErrors, Errors, 0, 1)
Marshal.FreeCoTaskMem(pErrors)

' Check for errors
If Errors(0) <> 0 Then
    Throw New ApplicationException("Writing DoSend failed.")
End If

End Sub
```
Explanation – Starting the transfer

In the above code section the “m_RecvInProgress” variable is set to true. It is used for determining whether the application waits for data or not. This is an advantage because this function is processed asynchronously. The data requested here are received at a later time via the “OnDataChange” function and processed in the “ProcessReceivedData” function.

Apart from the actual data, error codes are also received via OnDataChange. The error codes, however, are only evaluated if data is actually expected. For example, an exceeded time is only relevant if the connection is currently needed for transferring data.

Code extract – Receiving the data in the OPC client

For receiving the data from the PLC an active OPC variable named "SR:[S7 => PC]receive,B0,xxx" was created at the beginning. The dummy xxx stands for the size of the byte array.

For this OPC item, the SIMATIC NET OPC server creates a receive buffer on the “S7 => PC” connection and reports the arrival of new data with OnDataChange.

In the ProcessReceivedData method the byte array is analyzed, the data converted to the format required by Visual Basic .NET and forwarded to the user interface.

```
' Prepare the data converter
Dim Data As S7DataConverter = New S7DataConverter(Value, True, True)

' Get timestamp of header
Offset = Data.GetValue(Offset, S7Types.DATE_AND_TIME, DateTimeStamp)

' Get values of first data array entry
Offset = Data.GetValue(Offset, S7Types.WORD, Pressure)
Offset = Data.GetValue(Offset, S7Types.REAL, Temperature)

' Calculate average transfer time
If m_RecvInProgress = True Then
    TimeDiff = m_RecvEndTime.Subtract(m_RecvStartTime).TotalMilliseconds
    m_RecvDuration = m_RecvDuration * m_RecvDurationCount + TimeDiff
    m_RecvDurationCount = 1
    m_RecvDuration /= m_RecvDurationCount
End If

' Pass data to the dialog
m_Dialog.OnReceivedData(DateTimeStamp, m_RecvDuration, Pressure, Temperature, Humidity)
```
Explanation – Receiving the data in the OPC client

In the above code section, the received byte array (Value) is initially transferred to the conversion-dll. Here it must be specified whether the array starts with an even address or whether an automatic offset correction is necessary. Subsequently, the individual data are extracted from the byte array using the “GetValue” method of the DataConversion-dll, and divided into the respective variables (DataTimeStamp, Pressure, Temperature, Humidity).

Then, given the data has arrived (m_RecvInProgress), the average performance time is calculated.

Afterwards, the data are transferred to the MainForm object (m_Dialog) and displayed there.

Code extract – Sending the acknowledgement

In order to indicate to the PLC that the data were transferred successfully and processed by the PC, an acknowledgement byte is sent to the PLC. This was realized on the Write connection by means of an asynchronous write command.

Figure 4-8

```
Dim AsyncIO2 As IOPCAsyncIO2 = m_PassiveGroup
Dim ServerHandles(1) As Integer
Dim Values(1) As Object
Dim CancelId As Integer
Dim pErrors As IntPtr = IntPtr.Zero
Dim Errors(1) As Integer

m_RecvInProgress = False

' Set server handle of the receive acknowledgement
   ServerHandles(0) = m_RecvAckItemServerHandle

' Set it to 'true'
   Values(0) = 1

' Write it asynchronously
   AsyncIO2.Write(1, ServerHandles, Values, 1, CancelId, pErrors)
   AsyncIO2 = Nothing

' Copy errors array
   Marshal.Copy(pErrors, Errors, 0, 1)
   Marshal.FreeCoTaskMem(pErrors)

' Check for errors
   If Errors[0] <> 0 Then
      Throw New ApplicationException("Writing AckByte failed.")
   End If
```

Should the Write call fail, an exception is thrown with the respective error message. See chapter 3.4.5, Error handling in the application.
4.3.6 Sending data to a PLC using AG_SEND/AG_RECV

Sending data to the PLC with the AG_SEND-OPC item

For sending the data to the PLC, a passive OPC item with named "SR:[PC => S7]send,B0,xxx" was created. The dummy xxx stands for the size of the byte array. The sending of the data was implemented in the SendData method.

Code extract – Sending the data

Figure 4-9

```
Dim ByteArray() As Byte

' Let's generate the telegram (header and data array)
Me.GenerateTelegram(Name, ValveState, Voltage, ByteArray)

Dim AsyncIO2 As IOPCAsyncIO2 = m_PassiveGroup
Dim ServerHandles(1) As Integer
Dim Values(1) As Object
Dim CancelId As Integer
Dim pErrors As IntPtr = IntPtr.Zero
Dim Errors(1) As Integer

' Set server handle for the send byte array
ServerHandles(0) = m_SendDataItemServerHandle

' Set the byte array as value
Values(0) = ByteArray

' Start time measurement
m_SendInProgress = True
m_SendStart = DateTime.UtcNow

' Send the byte array asynchronously
AsyncIO2.Write(1, ServerHandles, Values, 1, CancelId, pErrors)
AsyncIO2 = Nothing

' Copy errors array
Marshal.Copy(pErrors, Errors, 0, 1)
Marshal.FreeCoTaskMem(pErrors)

' Check for errors
If Errors(0) <> 0 Then
    Throw New ApplicationException("Writing data failed.")
End If
```
Explanation

As soon as the data was written, we receive an acknowledgement from the OPC server with the callback of the OnWriteComplete method. Here the error array must be evaluated in order to receive information on the quality of the transaction.

Code extract – Telegram generation

The following code extract focuses on telegram generation. The “GenerateTelegram” method of the BlockServices class is responsible for this.

```vba
Figure 4-10
Dim Offset As Int32 = 0
CREATE_BYTE_ARRAY(OFġeġt, _
Value = Nothing
Value = Array.CreateInstance(?Type.GstType("System.Byte"), _
m_HeaderSize + (m_SendUdtSize * m_SendUdtCount))|

'Here's the layout of the header and the data entry:

'Header:
'<FieldOffset(0)> TelegramId  INT
'<FieldOffset(2)> TimeStamp  DATE_AND_TIME

'SendDataEntry:
'<FieldOffset(0)> Name  STRING[254]
'<FieldOffset(4)> ValveState  BYTE
'<FieldOffset(8)> Voltage  REAL

'Prepare the data converter
Dim Data As S7DataConverter = New S7DataConverter(Value, True, True)

'Set telegram ID and timestamp
m_SendTelegramId += 1
Offset = Data.SetValue(Offset, S7Types.INT, m_SendTelegramId)
Offset = Data.SetValue(Offset, S7Types.DATE_AND_TIME, DateTime.Now)

'Set values for all configured data entries
' To simplify the process we set the same data for all data entries
For i As Integer = 1 To m_SendUdtCount
    Offset = Data.SetValue(Offset, S7Types.STRING, Name, 254)
    Offset = Data.SetValue(Offset, S7Types.BYTE, ValveState)
    Offset = Data.SetValue(Offset, S7Types.REAL, Voltage)
Next i
```
Code extract – Receiving the acknowledgement

After the PLC has received and saved the data it sends an acknowledgement package to the PC. We receive it via an OnDataChange and evaluate it in ProcessSendDataReceipt method.

![Figure 4-11](image)

```vbnet
If ErrorCode = eRC And Quality = qRCO Then
    Dim Offset As Int32 = 2
    Dim DateTimeStamp As DateTime
    Dim TimeDiff As Integer = 0

    ' End time measurement
    m_SendEndTime = DateTime.UtcNow

    ' Calculate average sending time
    If m_SendInProgress = True And m_SendDurationCount > 0 Then
        TimeDiff = m_SendEndTime.Subtract(m_SendStartTime).TotalMilliseconds
        m_SendDuration = (m_SendDuration * m_SendDurationCount) + TimeDiff
        m_SendDurationCount = 1
    ElseIf m_SendInProgress = False And m_SendDurationCount < 0 Then
        m_SendDurationCount = 1
    End If

    m_SendInProgress = False

    ' Prepare data converter
    Dim Data As S7DataConverter = New S7DataConverter(Value, True, True)

    ' Get timestamp of packet header
    Data.GetValue(Offset, S7Types.DATE_AND_TIME, DateTimeStamp)

    ' Pass data to dialog
    m_Dialog.OnSendData(DateTimeStamp, m_SendDuration)
Else
    If ErrorCode <> eEC046003 Then
        System.Windows.Forms.MessageBox.Show(
            m_LanguageManager.GetString("NoReceiptError", m_CultureInfo), _
            m_LanguageManager.GetString("Warning", m_CultureInfo), _
            MessageBoxButtons.OK, MessageBoxIcon.Warning)
    End If
End If
```

Explanation – Receiving the acknowledgement

A memory bit is used here as already done for receiving the data from the PLC. In this case it is named "m_SendInProgress". It is set to True when data are sent and to False when the process is completed.

Furthermore, a counter named "m_SendDurationCount" is used, which contains the number of time measurements. Due to the fact that the first transfer of data takes disproportionately long compared with all following transfers, no time measurement will occur for it. Therefore, this counter is initialized with -1.
4.4 Variable services

4.4.1 Example: synchronous writing

The synchronous writing of OPC items was for example implemented in the WriteData method of the "VariableServices" class.

This requires a started OPC server, a created group and at least one created OPC item.

**Code excerpt**

```vba
Public Sub WriteData(ByVal Val1 As UInt32, ByVal Val2 As UInt32, ByVal Val3

' Query IOPCSyncIO interface from the OPC group
Dim SyncIO As IOPCSyncIO = m_WriteGroup

Dim serverHandles(3) As Integer
Dim values(3) As Object
Dim pErrors As IntPtr = IntPtr.Zero
Dim errors(3) As Integer

' Set server handles within the server handles array
ServerHandles(0) = m_WriteItemServerHandle1
ServerHandles(1) = m_WriteItemServerHandle2
ServerHandles(2) = m_WriteItemServerHandle3

' Set values to write within the values array
Values(0) = Val1
Values(1) = Val2
Values(2) = Val3

' Start the write operation
SyncIO.Write(3, ServerHandles, Values, pErrors)
SyncIO = Nothing

' Copy the error array to managed memory and free unmanaged memory
Marshal.Copy(pErrors, Errors, 0, 1)
Marshal.FreeCoTaskMem(pErrors)

' Check for errors
For i As Integer = 0 To 2
    If Errors(i) <> 0 Then
        Throw New ApplicationException("Writing data failed.")
    End If
Next

End Sub
```

**Explanation**

The OPC function "Write" expects an array with server handles as well as an array with the respective values to be written.

After "Write" has been called, the error array is copied and the original is released in the unmanaged memory.
4.4.2 Example: asynchronous reading

Asynchronous reading occurs via the OPC-IAsyncIO2 interface. This can be realized as follows:

Code extract

First the values are read asynchronously from the controller:

```
Public Sub FetchData()

   ' Query IOPCAsyncIO2 interface from the OPC group
   Dim AsyncIO As IOPCAsyncIO2 = m_fetchGroup

   Dim ServerHandles(3) As Integer
   Dim Values(3) As Object
   Dim pErrors As IntPtr = IntPtr.Zero
   Dim Errors(3) As Integer
   Dim CancelId As Integer

   ' Set server handles within the server handles array
   ServerHandles(0) = m_fetchItemServerHandle1
   ServerHandles(1) = m_fetchItemServerHandle2
   ServerHandles(2) = m_fetchItemServerHandle3

   ' Start the read operation
   AsyncIO.Read(3, ServerHandles, 1, CancelId, pErrors)
   AsyncIO = Nothing

   ' Copy the error array to managed memory and free unmanaged memory
   Marshal.Copy(pErrors, Errors, 0, 3)
   Marshal.FreeCoTaskMem(pErrors)

   ' Check for errors
   For i As Integer = 0 To 2
      If Errors(i) <> 0 Then
         Throw New ApplicationException("Fetching data failed.")
      End If
   Next

End Sub
```

Explanation

Like for all read/write operations, the ItemServer handles must also be specified here. In the above case these are transferred via the parameter "ServerHandles".

In this code example the transaction number (parameter dwTransactionID) is always transferred to the read method. This is possible because the "FetchData" method is the only function which uses the asynchronous Read. If there are several callers it is advisable to assign a unique transaction number for each caller or each individual read job respectively.
Displaying the values

Due to the fact that this is an asynchronous call, the actual values are only transferred to the OPC client in the callback method OnReadComplete.

Due to all groups having a callback method, it must now be differentiated between groups, client handles and transaction numbers in order to be able to assign the delivered values to the objects and calls in the OPC client (see chapter 3.2).

Code extract – Displaying of values

This procedure has been displayed in simplified form in the following code fragment. This is possible here as there is only one function which uses the asynchronous Read. It also cannot be executed parallel several times.

Figure 4-14

```vbscript
Public Sub OnReadComplete(ByVal dwTransid As Integer, ByVal hGroup As Integer, _
ByVal hrMasterquality As Integer, ByVal hrMastererror As Integer, _
ByVal dwCount As Integer, ByVal pClientItems() As Integer, _
ByVal pvValues() As Object, ByVal pvQualities() As Short, _
ByVal pftTimeStamps() As OpcRei.Oa.FILETIME, _
ByVal pErrors() As Integer) _
Implements OpcRei.IOpcDataCallback.OnReadComplete

If dwTransid = 1 Then
    Me.OnFetchedData(dwCount, pClientItems, pvValues, pvQualities, pErrors)
End If

End Sub
```
5 Explanations on the S7 Program

Here you will find information on the functionality of the S7 program.

5.1 Basic structure

The program on the controller generates and sends a data block if requested by the OPC client (DO_SEND). At the same time it waits for data, transmitted by the PC, and confirms their reception by sending the telegram header (acknowledgement).

The following structures and data blocks are being used:

Structure “Header” (UDT 0)

<table>
<thead>
<tr>
<th>Address</th>
<th>Name</th>
<th>Type</th>
<th>Initial value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td></td>
<td>STRUCT</td>
<td></td>
</tr>
<tr>
<td>+0.0</td>
<td>TelegramID</td>
<td>INT</td>
<td>0</td>
</tr>
<tr>
<td>+2.0</td>
<td>TimeStamp</td>
<td>DATE_AND_TIME</td>
<td>DT#90-1-1-0:0:0:000</td>
</tr>
<tr>
<td>=10.0</td>
<td>END_STRUCT</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Structure “Measured_Values” (UDT 1):

The structure contains simulated measured values which serve as data for the communication from controller to PC (S7 => PC). At each request of this data they are incremented by a certain value in order to simulate continuously changing values.

<table>
<thead>
<tr>
<th>Address</th>
<th>Name</th>
<th>Type</th>
<th>Initial value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td></td>
<td>STRUCT</td>
<td></td>
</tr>
<tr>
<td>+0.0</td>
<td>Pressure</td>
<td>DWORD</td>
<td>DM#16#0</td>
</tr>
<tr>
<td>+4.0</td>
<td>Temperature</td>
<td>REAL</td>
<td>0.0000000E+000</td>
</tr>
<tr>
<td>+8.0</td>
<td>Humidity</td>
<td>INT</td>
<td>0</td>
</tr>
<tr>
<td>=10.0</td>
<td>END_STRUCT</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Structure “Machine_Data” (UDT 2):
Simulated machine data which serve as data for the communication from PC to controller (PC => S7).

Figure 5-3

<table>
<thead>
<tr>
<th>Address</th>
<th>Name</th>
<th>Type</th>
<th>Initial value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td></td>
<td>STRUCT</td>
<td></td>
</tr>
<tr>
<td>+0.0</td>
<td>Machine_Name</td>
<td>STRING[254]</td>
<td>’’</td>
</tr>
<tr>
<td>+256.0</td>
<td>Valvestate</td>
<td>BYTE</td>
<td>E#16#0</td>
</tr>
<tr>
<td>+258.0</td>
<td>Voltage</td>
<td>REAL</td>
<td>0.000000e+00C</td>
</tr>
<tr>
<td>=252.0</td>
<td></td>
<td>END_STRUCT</td>
<td></td>
</tr>
</tbody>
</table>

Data blocks “SND_DB” (DB 20):
If data are requested by the controller (DO_SEND), the values to be transferred are assembled into this block and then send as data block.

Figure 5-4

<table>
<thead>
<tr>
<th>Address</th>
<th>Name</th>
<th>Type</th>
<th>Initial value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td></td>
<td>STRUCT</td>
<td></td>
</tr>
<tr>
<td>+0.0</td>
<td>HEADER</td>
<td>’’HEADER”</td>
<td></td>
</tr>
<tr>
<td>+10.0</td>
<td>DataArray</td>
<td>ARRAY[0..817]</td>
<td></td>
</tr>
<tr>
<td>‘10.0</td>
<td>'Measured_Values’</td>
<td>‘’</td>
<td></td>
</tr>
<tr>
<td>=8190.0</td>
<td></td>
<td>END_STRUCT</td>
<td></td>
</tr>
</tbody>
</table>

Data block “RCV_DB” (DB 30):

Figure 5-5

<table>
<thead>
<tr>
<th>Address</th>
<th>Name</th>
<th>Type</th>
<th>Initial value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td></td>
<td>STRUCT</td>
<td></td>
</tr>
<tr>
<td>+0.0</td>
<td>HEADER</td>
<td>’’HEADER”</td>
<td></td>
</tr>
<tr>
<td>+10.0</td>
<td>DataArray</td>
<td>ARRAY[0..30]</td>
<td></td>
</tr>
<tr>
<td>‘252.0</td>
<td>Machine_Data</td>
<td>‘’</td>
<td></td>
</tr>
<tr>
<td>+632.0</td>
<td>DummyBytes</td>
<td>ARRAY[0..59]</td>
<td></td>
</tr>
<tr>
<td>’1.0</td>
<td>BYTE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>=8192.0</td>
<td></td>
<td>END_STRUCT</td>
<td></td>
</tr>
</tbody>
</table>

Warning
In this data block the received data block is filed.
This DB must be as large as the expected data package (⇒ Setting the OPC send buffer and the RECV block). If this value cannot be reached directly through the data array, a respective amount of dummy bytes needs to be added.
Data block “Data” (DB 10):
The function (FC 2) “Fractionize_Telegram” copies the received values from the receive data block to the respective locations. DB 10 is an example target here. Other data blocks or memory areas could also be used.

Figure 5-6

<table>
<thead>
<tr>
<th>Address</th>
<th>Name</th>
<th>Type</th>
<th>Initial value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td></td>
<td>STRUCT</td>
<td></td>
</tr>
<tr>
<td>+0.0</td>
<td>HEADER</td>
<td>”HEADER”</td>
<td></td>
</tr>
<tr>
<td>+10.0</td>
<td>DataArray</td>
<td>ARRAY[0..817]</td>
<td></td>
</tr>
<tr>
<td>#10.0</td>
<td></td>
<td>”Measured_Values'</td>
<td></td>
</tr>
<tr>
<td>=8190.0</td>
<td></td>
<td>END_STRUCT</td>
<td></td>
</tr>
</tbody>
</table>

5.2 Program sequence

The processes within the controller are quite clear.
Overview

The following diagram illustrates the call process:

Figure 5-7
Network 1

The first part of the program (network 1) sends the data from the controller to the PC.

Table 5-1

<table>
<thead>
<tr>
<th>No.</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Setting the Do_Send-Bits via Write connection.</td>
</tr>
<tr>
<td>2</td>
<td>Setting this bit causes a call of the function block “Built_Telegram” (FC1), which assembles the data block in the data block 20 to be transferred. Then the send block (FC 5) is run through enough times until the data package was transferred (Send_Done = 1).</td>
</tr>
<tr>
<td>3</td>
<td>Finally an acknowledgement from the PC is waited for (Ack_Bit), before a new send request is waited for.</td>
</tr>
</tbody>
</table>

Network 2

The second part of the program (network 2) waits for data which are transferred from the PC.

Table 5-2

<table>
<thead>
<tr>
<th>No.</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>The Receive block (FC 6) is run through enough times until it reports via NewDataReceive (NDR = 1) that data has been received.</td>
</tr>
<tr>
<td>2</td>
<td>Then the function “Fractionize_Telegram” (FC 2) is called, which dissembles the received data block. As an example, the data are copied there from the first three array elements to an other data block (DB 10).</td>
</tr>
<tr>
<td>3</td>
<td>Finally, the header of the received data package is returned as confirmation to the PC</td>
</tr>
</tbody>
</table>
5.3 Calling the Send- / Receive blocks (FC 5 / FC 6)

A brief summary of the interface of block FC5 and FC6 is given below. A call is performed as an example and the function of the parameter is described.

5.3.1 AG_SEND (FC 5)

CALL "AG_SEND"
ACT :=TRUE
ID :=1
LADDR :=W#16#100
SEND :=P#DB20.DBX0.0 BYTE 8010
LEN :=8010
DONE :="SEND_DONE"
ERROR :=M20.1
STATUS:=M022

Note
An exact description how the length of the data package can be determined is available in chapter 6.1.

5.3.2 AG_RECV (FC 6)

CALL "AG_RECV"
ID :=2
LADDR :=W#16#100
RECV :=P#DB30.DBX0.0 BYTE 8192
NDR :="NEW_DATA_RECEIVE"
ERROR :=M30.1
STATUS:=M032
LEN :=M034
6 Modifications on the Example Program

Here you will find information on …

where in the Visual Basic .NET or S7 program you need to make adjustments in order to change the data block size.

6.1 Changes in the STEP 7 program

General information

In this example, the data block to be transferred consists of the header (UDT0) and an array of user-defined structures (UDT1 or 2). This means that the length of the data blocks should always be a multiple of the UDT size + header.

Example:
An array with 800 UDT1 is to be transferred from the controller. The data block size to be adjusted is calculated as follows:

\[ \text{Number} \times \text{UDT size} + \text{header} \]

\[ 800 \times 10 \text{ bytes} + 10 \text{ bytes} = 8010 \text{ bytes} \]

S7 => PC

The data block “SND_DB” (DB 20) must be made large enough so it can hold all data to be sent, which are filed there by the function (FC 1) “Built_Telegram”. And it must be at least as large as the data area given at the send call.

When calling the Send block (FC 5) the length of the data block must be specified in byte at “SEND” and “LEN”:

\[
\begin{align*}
\text{SEND} & : = \#\text{DB20}.\text{DBX0.0 BYTE} \ 8010 \\
\text{LEN} & : = 8010
\end{align*}
\]

PC => S7

The data block “RCV_DB“ (DB 30) must be large enough so it can hold all data to be received.

When calling the Receive block (FC 6) the length of the data block must be specified at “RECV” in byte:

\[
\begin{align*}
\text{RECV} & : = \#\text{DB30}.\text{DBX0.0 BYTE} \ 8192
\end{align*}
\]

Note

This size must also have been configured as send buffer of the OPC server on the PC/PG side. How to make this setting is described in the "Demonstration" document of this application.
6.2 Changes at the Visual Basic .NET application

At this point we explain to you where in the source text of the application the settings need to be made so the application can respond correctly to the data block size.

Code extract

The following code section is at the beginning of the “BlockServices.vb” source file. Constants are created in it which roughly outline the data package regarding size and quality.

Figure 6-1

```vbs
' Items that describe the incoming/outgoing SR packets
Private Const m_HeaderSize As Integer = 10
Private Const m_SendUdtSize As Integer = 252
Private Const m_SendUdtCount As Integer = 30
Private Const m_RecvUdtSize As Integer = 10
Private Const m_RecvUdtCount As Integer = 800
```

Adjustments

To change the number of user data to be sent, please adjust the variable “m_SendUdtCount”. The size of the data block to be sent is calculated as follows: m_HeaderSize + m_SendUdtSize * m_SendUdtCount.

Accordingly, the number of data records to be received can be adjusted with the variable “m_RecvUdtCount”.

⚠️ Warning

The values of the variable m_HeaderSize, m_SendUdtSize and m_RecvUdtSize result from the UDTs used in the controller. It should therefore only be changed if there are changes on the controller side!
Appendix and List of Further Literature

7 Glossary (optional)

COM
Component Object Model: Software model for communication between components based on a standardized interface; → DCOM

DCOM
Distributed Object Model: Software model for communicating beyond computer boundaries based on → COM.

Eventhandler
An Event handler processes occurring events or Windows messages.

Exception
An Exception is referred to as an exceptional situation. It can be generated either by the operating system (e.g. Division by zero) or by the user program.

Exception handler
An Exception handler processes occurring exceptional situations. Generally a secured error behavior and/or a message to the user.

HRESULT
Return data type of COM objects.

IDL
Interface Definition Language: A Microsoft standard language for definition of function and parameter interfaces.

OPC
OLE for process control. A standardization for process control created for the automation sector. Maintained by the OPC - Foundation. Several interfaces with different tasks exist.

OPC client
A program which uses the services of an OPC server by sending requests to the server via existing OPC interfaces.

OPC DA
OPC Data Access interface. The interface enables monitoring and changing of process values.
OPC with SEND/RECEIVE Protocol  Entry-ID: 21523291

OPC server
A program which provides one of the OPC interfaces and responds to requests of an OPC client.

Polling
Term referring to (mostly cyclical) polling of certain values or states.

RCW
Runtime Callable Wrapper. A wrapper which encapsulates a Custom interface for the .NET Framework. The RCW enables accessing older COM ports for the .NET Framework.

Sink interface
Using the sink interface, messages between components can be sent. The sink interface is based on COM mechanisms.

Windows message
The standard Microsoft Windows operating systems exchange messages to notify of events, e.g. the paint event.

Wrapper
The term Wrapper normally refers to a class group which encapsulates other class groups for data conversion or easier handling. It can be considered an “envelope” enclosing the “wrapped” classes, covering them from the outside.
8 Literature

8.1 Bibliographic references

This list is by no means exhaustive and only gives a selection of appropriate sources.

Table 8-1

<table>
<thead>
<tr>
<th>Topic</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>/2/</td>
<td>.NET</td>
</tr>
<tr>
<td>/3/</td>
<td>Manual for industrial communication on PG/PC with SIMATIC NET.</td>
</tr>
</tbody>
</table>

8.2 Internet links

This list is by no means exhaustive and only gives a selection of appropriate sources.

<always write in plain text>

<table>
<thead>
<tr>
<th>Topic</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>/1\</td>
<td>Reference to the documentation</td>
</tr>
<tr>
<td>/2\</td>
<td>Siemens A&amp;D Customer Support</td>
</tr>
<tr>
<td>/3\</td>
<td>OPC</td>
</tr>
</tbody>
</table>