SIEMENS

Michael Braun | Wolfgang Horn

Object-oriented Programming in SIMOTION

Fundamentals, Program Examples and Software Concepts according to IEC 61131-3

Braun / Horn
Object-Oriented Programming with SIMOTION

Michael Braun is a product manager for Motion Control Engineering at SIEMENS AG in Erlangen. Among other responsibilities, he is tasked with communicating customer requirements to software developers/designers, monitoring the implementation of these requirements and launching new software on the market.

Dr. Wolfgang Horn is a software manager and developer at the Gesellschaft für Industrielle Steuerungstechnik in Chemnitz, Germany. He is a leading specialist in the architecture and programming of SIMOTION.

Object-Oriented Programming with SIMOTION

Basic Principles, Program Examples
and Software Concepts
according to IEC 61131-3

By Michael Braun and Wolfgang Horn

Bibliographic information published by the Deutsche Nationalbibliothek

The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data are available in the Internet at http://dnb.d-nb.de.

www.publicis-books.de

Print ISBN 978-3-89578-456-9
ePDF ISBN 978-3-89578-947-2

Publisher: Publicis Publishing, Erlangen, Germany

Printed in Germany

Table of contents

List of Figures

List of Tables

Information for readers

The demand for ever more flexible solutions in the field of mechanical engineering is also changing the methods by which the control systems themselves are programmed. Since we have already decided that mechatronic systems are the right way to go, the need to develop highly modular software and the programming techniques suitable for software of this kind is posing tough challenges. As the trend in favor of creating modular functional units within machines increases, it is inevitable that this modularity will be reflected in the software. The extensions defined in IEC 61131-3 ED3 relating to object-oriented programming go a long way to support the ongoing efforts to achieve modularized software. Designers of automation engineering software will thus have to deal with changes similar to those experienced by the programmers of PC software from the mid-1980s onwards.

If we want to create application software for automation systems that is far superior in design and structure, easier to modify and, above all, modular, then there will be no alternative to object-oriented programming. With software version 4.5 of the SIMOTION system, it will become possible to use object-oriented programming mechanisms as defined in IEC 61131-3 ED3. The purpose of this book is to help programmers get to grips with this new way of thinking and programming. Illustrative examples have been provided for each separate topic to make the learning process easier. Each example is based on and relates to previous examples that have been provided to explain individual topics. At the end of the book, the reader will find a reusable machine module that is fully implemented in OOP.

This book will be useful for *anyone* who wants to learn about object-oriented programming for automation engineering applications. The first part of the book focuses on explaining the basic principles of object-oriented programming and is based on the implementation of OOP in SIMOTION according to IEC 61131-3 ED3 (chapters 1 to 6). The second part is a general introduction to the SIMOTION system itself (chapters 7 and 8).

We would advise readers who are not yet familiar with SIMOTION to start by reading the second part “Introduction to SIMOTION”. This explains the basic principles of the SIMOTION control system and its engineering system SIMOTION SCOUT.

For readers to fully understand and learn the content relating to object-oriented programming, they must already be familiar with high-level programming languages such as Structured Text or Pascal. They must also have a basic knowledge of programmable logic controllers and their system behavior.

Readers will also notice that descriptions of certain issues are repeated in different chapters. This approach was motivated by our desire to manage with as few cross references between chapters as possible. We have therefore made it possible for our readers to jump between chapters without losing track of the discussion.

The examples we have included were specially developed for the book and they all build on one another. We deliberately kept them simple because we wanted them to clearly demonstrate the potential uses of object-oriented mechanisms. While all our examples are based on this idea, some of them still managed to grow to a significant

size. We obviously realize that nobody wants to go to the trouble of typing out all the program code printed in this book. We have therefore made the examples from this book available to our readers as an Internet download. You will find corresponding links to them at www.siemens.com/simotion. Please note the conditions for use of the examples.

Personal comments by the authors

Michael Braun

I have thought a very great deal about this chapter and was determined for a long time that I wouldn't write it at all. Perhaps because I myself am someone who often skips this kind of chapter in books. But life is a learning process and after giving the matter some thought, I decided that this chapter would give me the opportunity to tell our readers something about myself and my motivation for writing this book.

From the very beginning of my career, I have followed developments in the field of automation engineering and found them to be extraordinarily exciting. My attention was primarily focused on the design and development of software. To develop programs that will ultimately allow a production plant to do its job properly was, and still is, an occupation that I find thoroughly exhilarating, but it is also an activity that keeps the programmer on a continual learning curve. The ability to write good software is not something that falls out the sky into your lap (it didn't fall into mine either!). In my experience, you go through three distinct phases as a programmer.

During the first phase, you focus your attention on learning the basics of a new system or programming language. Certain relationships are not quite clear and it is simply a question of taking the first tentative steps. You are learning the basics and writing your first programs. As a general rule, you will later throw these into the waste bin because you have implemented them ineffectively or perhaps in an overcomplicated manner. But because you have got them to work, you make the transition into the second phase.

During this phase, you are reaching the point where you are familiar with all the elements of the language and can use "clever" tricks to formulate solutions. The fact that nobody can actually understand the solution is something that you deliberately ignore, such is your pride in the ingenuity and brilliance that have flowed into the creation of this software. Any pangs of conscience sink without trace in this mood of euphoria! This is the most dangerous time in your life as a programmer because you are writing unreadable code. It is now time for a helpful colleague to come along and tell you in no uncertain terms that your programs are rubbish (happened to me as well). You'll get another chance to see the error of your ways if you find you cannot get rid of this "brilliant" code and are obliged to take on the responsibility of maintaining it (this is also a great opportunity to prove that you are "indispensable"). These shocks will help you get through the second phase.

If you have reached the third phase, you will be over the worst and finally capable of writing comprehensible program code that is easy to maintain – in other words, you are creating reusable software. Every programmer should endeavor to reach this phase as quickly as possible.

Creating software for modern mechanical engineering applications is a team task. The times when a developer hatched software in a quiet little room behind closed doors are long gone. The creation process is driven by a continuous exchange of information between different members of the team, but also communication between different engineering disciplines. The more efficient the communication between all the participants, the more effectively individual developers can complete the tasks specifically assigned to them. Software development is an iterative process that undergoes repeated rounds of improvement. With each improvement, the development team gets closer to some imaginary optimum. I say "imaginary" because the definition of "optimum" also changes over time.

It is thus a fact that the software never actually gets finished. As development work progresses, it is inevitable that a deliverable version of the software will sooner or later emerge, but this delivery state is actually the basis for the next development stage.

Like the user software, the development system is itself also a kind of software that undergoes changes. Object-oriented programming is an extension of the development system for automation engineering software which can, and should, make life much easier for developers of application software than the conventional development environments in which they have previously worked. But for this to succeed, it is essential that programmers become familiar with and thoroughly understand the mechanisms of this programming method.

It was for this purpose that a description and explanation of object-oriented programming mechanisms was added to the SIMOTION documentation. The volume of this documentation ultimately became so extensive that it led to the idea of writing a book about OOP. You are now holding this book in your hands and I sincerely wish that it will help you to think up many new ideas for improving your own software. Learning and implementing what you have learned is something you have to do yourself, but please don't forget to have fun as well!

Acknowledgements

Object-oriented programming is not yet as widely established in automation engineering as it is in the PC environment. This book has been written with the intention of helping anyone who wishes to learn this new programming method. It would never have come into being without the help of many colleagues. It was the discussions I had with some of these colleagues that helped me most when I found myself in need of creative inspiration. I would like to express my gratitude to all helpers and supporters.

Before I could get started, I needed to abandon my "procedural mindset" and it was our software architect Dr. Michael Schlereth, my colleague and fellow author Dr. Wolfgang Horn, and Thomas Hennefelder from our application center who helped me most in this respect. They gave freely of their time to discuss my programs and basic ideas with me. But when someone occasionally said "But you just don't do it like that!", I was forced to reevaluate and change course and it was precisely this kind of exchange that helped me to move forwards. I would also like to express my gratitude to Rumwald Hermann, Klaus Lummer and Nils Focke from the SIMOTION development team for their assistance and encouragement.

I also received valuable support from my colleagues from Product Management and System Management who willingly assisted with proofreading. For this reason, I

would like to make special mention of my colleagues Benno Bruss, Jürgen Büssert, Kai Flucke, Alexander Heider, Manfred Popp and Wolfgang Wiedemann from Integration Testing. Last but not least, a special note of thanks goes to my bosses Erwin Neis, Josef Hammer and Rudolf Teplitzky for their constant encouragement and the freedom I needed to write this book.

For his assistance with the creation and testing of the example programs, I would finally like to say a special word of thanks to my colleague Frank Becker from APC Cologne.

Wolfgang Horn

When Michael Braun first came to me with the idea of writing a book about object-oriented programming, my initial reaction was: Why do we need to write yet another book on this subject? After all, countless publications about this topic from the viewpoint of a myriad of different programming languages and applications already exist. But when I had given more thought to the matter, I quickly realized that there is not much literature available that specifically relates to automation engineering or gives adequate support to those who wish to learn object-oriented programming techniques for control systems.

Based on my own professional practice, I am well aware of the opportunities and potential that can be exploited when OOP is used. This is true, of course, only if OOP is directly supported by the programming environment and by the programming language used. With the implementation of the 3rd Edition of the IEC in SIMOTION, we have now given our users direct access to the world of OOP. But this alone is not enough.

In the course of many discussions with Michael Braun and other colleagues who are pursuing this development with enthusiasm, it became clear to me that simply learning the new language constructs is not sufficient. In order to achieve a sustained effect, it is also important to understand why one can or should use a particular language element or technology. Listening to the questions posed by my colleagues, it became apparent that simple examples of programming constructs would not be enough and we would also need to give guidance as to which new solutions for automation engineering tasks could be developed by using object-oriented programming.

We have discussed many of these aspects in this book. We have aimed to help readers get to grips with the subject of object orientation using examples from the field of control engineering. To those readers who already have experience with other object-oriented programming languages we will try to explain the specific requirements of control engineering. Control-specific languages are specially formulated to ensure that control system software can be programmed in such a way that program runtimes do not exceed certain limits, in other words, to ensure that the number of runtime errors during program execution are minimized. As a result, readers of this book will search in vain for any reference to constructs for dynamic object generation and destruction. The reason that constructs of this kind are not offered in the control programming environment is a simple one: they could have a significantly negative impact on the real-time capability of the application.

When I set out on my professional career path, the programming language Structured Text (ST) was still one of the rank outsiders in the field of control technology. The increasing complexity of programs of the kind we are now encountering in

the field of motion control in particular is literally forcing us to adopt high-level programming languages like ST. A logical continuation of this approach can be seen in the support for object orientation afforded by control systems. It is my opinion that this method of control system programming will have become the standard for automation solutions in just a few years time.

By writing this book, we hope to contribute in some way to helping object-oriented programming find broader acceptance in control engineering applications. With respect to the modularity and combinability of software modules, the potential benefits of object-oriented programming, particularly for more complex applications, are enormous. In the meantime, I would like to join my colleague Michael Braun in wishing our readers much enjoyment and patience in learning and trying out their new skills.

1 Developments in the Field of Control Engineering

One of the most important extensions to IEC 61131-3 ED3 describes the mechanisms for the object-oriented programming of control systems in automation applications. This development has provided a solid basis for standardizing programs used in automation systems and offers a solution for overcoming the limitations associated with procedural programming methods.

As a result of the increasing trend to make mechanically engineered systems as flexible as possible, it has become essential to change existing programs in such a way that modular machine concepts are also reflected in the software. As a result, modularization is becoming the guiding principle for designing the programs of the future. Modularized software comprises modules that are fully functional and tested as independent entities, but they can be combined to create a single functional unit within different machines.

Anyone wishing to attain, and then retain, a competitive position on the international market must be capable of minimizing commissioning times. This can be achieved only if standardized program modules function reliably when combined with other modules so that any corrective work during the commissioning phase is either unnecessary or reduced to an absolute minimum.

The requirements to be fulfilled by the application software architecture and the automation systems are therefore as follows:

- The software must have a modular structure. The modules are totally encapsulated with the ability to function fully independently.
- The independent design of the modules means that even data belong to a module, i.e. are an integral part of that module. In other words, it must be possible to link data to the module and to prevent any changes from being made to the data outside the module.
- The ability to test modules as independent entities is crucial if they are to be combined and assembled to create a functional unit. The modules must therefore be designed in such a way that they can be individually tested in a test environment.
- Combining different modules in an environment must involve only minimal software modification, or none at all. To achieve this goal, interaction between modules is implemented on the basis of neutral interfaces.
- Since the machinery as a whole including all its individual components will undergo modernization over its service life, it is absolutely imperative that the machine software can be adapted accordingly. However, any modernization process should, wherever possible, preclude the need to modify tried-and-tested, functionally reliable modules.
- It must be possible for the machine manufacturer's software module developers to work as independently of one another as possible. To this end, it is necessary to use programming languages with appropriate mechanisms

> to reduce interdependencies between software modules. Agreement on the interfaces provided to allow data exchange between different software modules shall therefore be capable of being defined.

To satisfy all the requirements described above, we need to find a better programming model than the procedural methods presently used to write control engineering programs.

Modern automation systems have been evolving through a process of development for many years. During this time, the methods used to program them have also changed in various ways. It was advances in the field of automation engineering that necessitated these changes, and these in turn influenced users. Any forced change was met with a degree of resistance by some, and it was this attitude that blocked the acceptance of new approaches to programming. Programmers were only prepared to accept new methods if their concerns or misgivings were addressed and alleviated.

The advent of object-oriented programming hails another paradigm shift in automation engineering. As programmers experienced the advances made in the field of automation engineering, they developed a specific programming methodology for individual applications. These methods now need to be examined, changed where necessary, or even rejected altogether. But this could again engender misgivings or reservations, and if it is not possible to alleviate these, they will be an obstacle to the introduction of new methods. Programmers may well have adopted this mindset, for example, as a result of their past experience of change.

For this reason, we are going to allow ourselves a brief tour through the history of automation technology and pay special attention to the consequences of automation advances on programming. While the development stages described below are certainly representative, they are not necessarily given in the correct chronological sequence. Nor does the description claim to be complete. Nevertheless, each of these advances actually resulted in changes to programming methods. We now need to think about these consequences and, where misgivings and reservations from the past still exist, find effective arguments to counter them.

1.1 The early days of programmable logic controllers (PLCs)

A notable feature of early programmable logic controller (PLC) applications was the fact that users of this new control generation knew virtually nothing about programming. Before the days of the PLC, automation systems had been implemented using hard-wired relay controls, contactor controls or electronic components. Machine designers, commissioning engineers and service personnel were suddenly confronted with programming instead of wiring. For this reason, programming methods needed to be devised with a view to what users already knew.

At this time, users understood how to read circuit diagrams and use wiring in order to implement functions. It therefore made sense to design programming methods which supported these capabilities. Thus was born the “ladder logic” programming method (Figure 1) with resemblance to a circuit diagram, and the “function block diagram” system that is based on electronic diagrams.

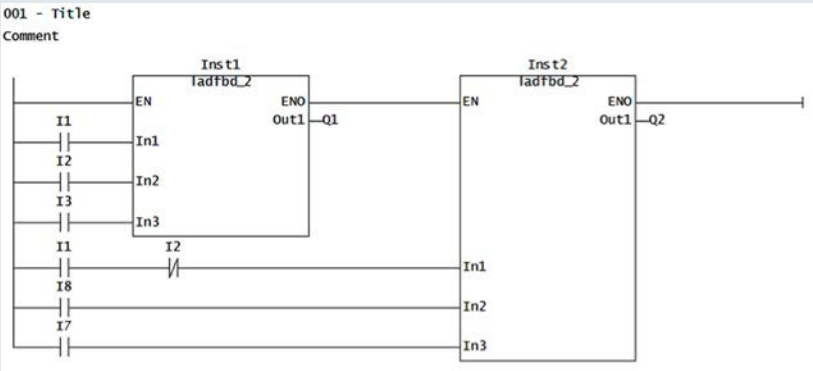


Figure 1 Example of a ladder logic program

More complex function elements of the system such as timers or counters were represented as a box with corresponding inputs and outputs. To allow users to create their own complex function modules, the system provided them with a tool to program their own function blocks or functions and these were represented in turn as complex elements (boxes) with inputs and outputs in the ladder diagram or function block diagram.

More complex elements of this kind needed to be programmed in a different way to the functionally limited ladder or function block diagram elements. Users were therefore provided with two programming languages with syntax similar to assembler language, i.e. Statement List (STL) (Figure 2) or Instruction List (IL).

```
U     #Switch_On              #Switch_On        -- Switch on engine
UN    "Automatic_Mode"        A4.2              -- Retentive output
S     #Engine_On              #Engine_On        -- Engine is switched on
U(
O     #Switch_Off             #Switch_Off       -- Switch off engine
ON    #Failure                #Failure          -- Engine failure, causes the engine to switch off
)
R     #Engine_On              #Engine_On        -- Engine is switched on
```

Figure 2 Example of SIEMENS STL

These enabled users to create programs of significant complexity which supported the programming, for example, of computation functions and branches within the program. However, each control system manufacturer created their own set of commands and there were wide variations between the command sets available. For this reason, it was extremely difficult to transfer programs from one control system to

another and users were required to learn the different “dialects” and approaches of each individual control system manufacturer.

Another disadvantage of these assembler-like mnemonics (= a plain-text, human-readable abbreviation for an assembler instruction) was that the scope for program structuring was extremely limited and the structuring tools were very laborious to use. Furthermore, many users also felt compelled to teach themselves how to program. As a result, some programs were readable to a greater or lesser extent, while others contained sections of code that were impossible to maintain.

Despite all these problems, the ease with which programs could be changed and the flexibility this offered was an immense advantage over conventional wiring, and this was what eventually swept the programmable logic controller to triumph. Programming therefore became an established, and now indispensable, part of the mechanical engineering process.

Obstacles

The ease with which software programs could be changed also encouraged some programmers to work according to the “trial and error” principle. They were particularly susceptible to this temptation when working under time pressure to make a machine function ready for acceptance or delivery. This practice of putting the finishing touches to a program by testing it during commissioning resulted in an unacceptably large number of program variants and ultimately to software that had no structure. This problem had a lasting, negative impact when it came to reusing programs.

It can often be observed in companies today that the time originally planned for writing software is continuously squeezed as a machine construction project progresses. The company has agreed a delivery deadline, but further technical changes to the machinery (including those requested by the customer) lead to unplanned additional expenditure or labor and exacerbate the scheduling situation. Changes to the system design can probably never be avoided because they are generally justified for technical or other reasons. Nonetheless, investment of substantially more labor in a project should logically lead to an extension of the delivery deadline.

The end customer will only accept a deadline extension if it can be proved incontrovertibly that the extra outlay was unavoidable due to requests for changes made by the customer. But this proof can be provided only in cases where the scope of supply was clearly and unambiguously formulated. If the scope of supply is not clearly defined when a system is sold, it is inevitable that the scope promised by the seller will be open to interpretation. In such cases, the end customer will generally demand delivery by the agreed deadline. As a result, the time scheduled for programming software is squeezed. “It’s so easy to change software and you can do it so quickly”. This attitude often leads to the problem that unfinished software is handed over to the commissioning team and has to be finished within whatever time remains. The software cannot be made to conform to the specified design guidelines and becomes less reusable.

Solutions

Precise software planning is the key to success. Software functions can be planned efficiently only if the relevant requirements are identified in advance. On the basis of these requirements, it is possible to work out how the software must be implemented and structured. The time required to develop the software can be calculated and the relevant deadlines planned accordingly.

Regular consultation with the customer prevents any unpleasant surprises for either party. For this purpose, the implementation schedule must be discussed with the customer and recorded in writing at an early stage.

A process for implementing development of the software must also be set out. The status and progress of the development task is then clearly identifiable and any deviations from the agreed process can be picked up early. When deviations are identified early enough, there is still time to take corrective action.

1.2 The PLC learns to communicate

The early PLCs had limited resources (e.g. memory capacity or processing performance). Nor was fine scaling of the different performance classes of control systems possible as it is with modern systems. This lack of scalability meant that it was necessary to use multiple control systems in a single plant, and where several control systems were deployed, they needed to be synchronized with one another. One of the simplest methods, but also one with extremely limited possibilities, was to synchronize different controls via inputs and outputs. This option was not viable in cases where large volumes of data needed to be exchanged.

The problem was resolved by using special communication modules that could be inserted in the PLC. These “computer links” (e.g. RK512) utilized standardized protocol frames (e.g. 3964R) to exchange data and were operated by driver blocks in the PLC. Even though the links provided by these modules were essentially no more than point-to-point connections, their use increased the communication capability of PLCs. Figure 3 shows a compact control system dating from around 1980.

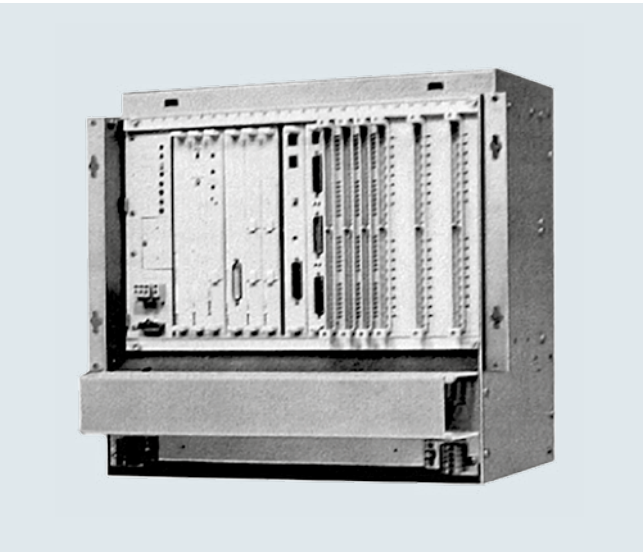


Figure 3 SIMATIC S5-150K

Another advantage of these computer links was that they provided a master computer interface. Using the same technology, therefore, it was possible to establish communication links between control system and master computer levels in order to record production data.

Since programmers were required to synchronize control systems by programming communication links between them, they were also forced to create programs that included the relevant communication mechanisms in addition to the implemented actual control task. This meant that they had to take the following aspects into account when designing the software:

- The link to individual devices/control systems needed to be connected and possibly disconnected. The system behavior when individual components were switched on or off also needed to be taken into account.
- Connections needed to be managed depending on the number of nodes. The data needed to be structured accordingly and transferred to / dispatched from the relevant communication module for the purpose of data exchange.
- Connection monitoring systems needed to be implemented and a suitable response programmed in the machine operating sequence.
- There was a risk of telegram loss under certain operating conditions. This could happen, for example, if a control system was unable to empty the telegram buffer of a communication module because its cycle time had been extended temporarily. In this instance as well, it was necessary to engineer suitable program responses.
- Production-relevant data needed to be collected in the control system and prepared for transfer to master computers.

Obstacles

Control system programs increased in size due to the addition of communication mechanisms. These bigger programs needed to remain manageable. Suitable structuring of the software and the modular programming this involved were the logical consequences of the drive towards program manageability. Not only were programmers required to concentrate on programming machine operating sequences, they also needed to consider the data structures in the control system. It seemed meaningful, on the one hand, to separate the communication functions from the machine operating sequences. On the other, however, these sequences were naturally influenced by the communication data. A way needed to be found to effectively combine communication functions with machine operating sequences. But it was also important to ensure that software changes in one area (e.g. machine operating sequence) would not automatically entail software changes in another area (e.g. communication).

Solutions

Without clear definitions and structures, software designs can become so idiosyncratic that they become difficult or even impossible to maintain and upgrade in the long term. Software design, modularization and standardization of software are still clearly defined objectives of the development process. By creating reusable software components and implementing a well-planned structure, it is possible to reduce the outlay for software development and plan deadlines with greater confidence.

1.3 Development of fieldbus systems

The centralized structure of programmable logic controllers with central processing units (CPU) mounted in the same rack as I/O modules made it necessary to increase the volume of wiring between the control cabinet and actuators (such as valves) installed in the machine or control components (such as switches and buttons) that were needed to control the machinery. It was the expense of installing this wiring that provided the impetus for change. The elements (actuators and sensors) were installed in the machine rather than in the control cabinet and the goal was to find a way of connecting them to the PLC using less wiring.

With this objective in mind, the control system manufacturers developed new communication modules that provided a serial bus link between the control system and field devices. By deploying these fieldbuses, it was possible to reduce the volume of wiring to actuators and sensors in the field.

Reducing the volume of cabling also had a further benefit. Since actuators and sensors were now linked to the PLC via a bus system, it was no longer necessary to have such a large number of I/O modules in the PLC rack. The terminal strip converters were relocated directly into local control boxes, allowing use of significantly smaller control cabinets.

But this development objective of achieving a substantial reduction in wiring ultimately increased the complexity of the software design process:

- It was necessary to provide systems to monitor proper booting when external I/O devices connected to the bus were powered up.
- Failure of a component during operation needed to be detected by the software and modeled by a suitable response in the process.
- The software developer needed to work out a substitute value strategy for inputs and outputs that would no longer be available if external I/O devices failed. This substitute value strategy had to be integrated into the relevant programs.

Obstacles

As I/O devices were relocated to external, bus-coupled components, the complexity of the software design process increased yet again, but remained relatively easy to manage as long as the I/O devices were purely digital or analog. As the technology continued to advance, however, ever more complex I/O devices were coupled to buses and needed to become a particular focus of attention for software developers. Implementing successful interaction with I/O components is not easy, especially when some of them are complex and capable of independent operation. This task becomes even more difficult if the components have their own independent operating sequence that needs to be synchronized with the machine process. If the link to a component of this kind fails, the stop response that may be required is relatively easy to manage. However, system restart after the stop command may well involve significantly more complex programs. To achieve a successful system restart, the main process requires more information and this needs to be acquired by an additionally programmed information exchange with the affected I/O component.

Solutions

A well-planned software design is absolutely essential if these interrelationships are to be managed effectively. Without a suitable software design, many unique software versions are created over time as machines are delivered – an approach which makes software maintenance significantly more difficult and renders modularization and standardization completely impossible.

In conclusion, it is fair to say that fieldbus systems and their components had an enormous impact on programming, as we will see later on (see Chapter 1.6 “Drives become fully-fledged bus system nodes1.6”).

1.4 Integration of display systems in PLCs

As the complexity and size of installations increased, it became necessary to provide the machine operator with a clear overview of process events. This entailed a great deal more than just an indication of the machine status signaled via lamps and illuminated pushbuttons. Machine operating sequences were becoming ever more complex, necessitating a general improvement in the quality of display systems. Driven by this necessity and aided by continuous advances in computer technology, it became economical to deploy visualization systems in the form of plug-in modules in the programmable logic controller. Control system manufacturers developed single-board computers that could be plugged into the PLC and allowed screens to be connected as a Human Machine Interface (HMI). These mini-computers exchanged data with the PLC via the backplane bus and had their own driver blocks in the PLC CPU. Manufacturers developed configuring tools and supplied these to users so that they could configure their own displays. One of the first screen systems of this kind for SIMATIC controllers was the WS400 visualization system (comprising the WF470 display module and various operator panels) developed by Siemens (Figure 4).

Using this visualization system, it was possible to display the plant as a block graphic and show the status of individual machine modules. Further configurable detail views provided the machine operator with more precise information about individual modules. The visualization system was capable of displaying the machine operating sequences for executing processes and also featured a standardized fault diagnostics system that indicated any problems.

Programmers were therefore required to structure the software and data in such a way that information could be transferred to the visualization system:

- Data models had to be modularized for display purposes at least, and scalability for different plant sizes needed to be taken into account. It was only by structuring the software in this way that it could be reused across different plants.
- The operator needed to be supplied with detailed information if any faults developed in the plant. Programmers therefore needed to ensure that the relevant plant faults were transferred to the visualization system by means of flags in the control system and that the corresponding fault messages were assigned to the flags by text lists.

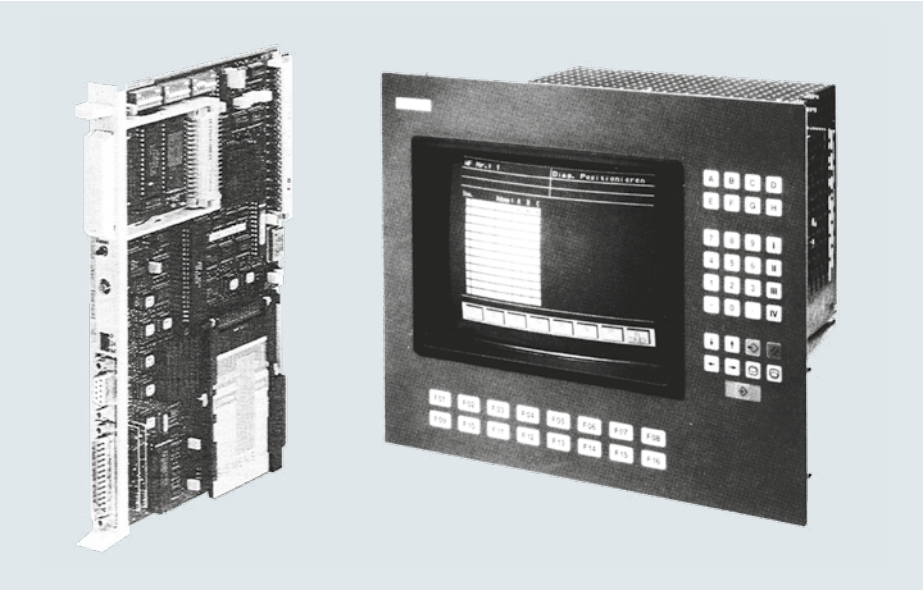


Figure 4 WF470 with compact operator panel

- They also had to program status messages for transfer to the HMI system.
- Since it was necessary to shut down plant sections or even the entire plant when serious faults occurred, fault-signaling processes often resulted in the display of many other follow-on faults in addition to the actual fault cause. This flurry of fault messages did not help the operator or service engineer to resolve the problem. The consequence for the software design process was that programmers now also needed to consider a means of evaluating initial and follow-on faults and to integrate a suitable evaluation strategy into the plant software.
- The integration of visualization systems into PLCs forced programmers to implement additional modules in the plant software, modules that were absolutely essential to effective operation of the machine but had little to do with the actual control task. The size and complexity of programs continued to increase as a consequence. Modularization and clear structures had become even more important as efforts were made to create software that could be maintained and upgraded. System programming nevertheless continued in the LAD/FBD or STL languages. Sequential processes were programmed with the GRAPH-5 language that had been specially developed for the purpose.

Modern visualization systems are linked to the control system via Industrial Ethernet. Configuring tools for HMI systems are a standard element of the engineering software and function as integral components of appropriate software suites. Achieving the required degree of software modularization and developing suitable data models in the control system still remain a key responsibility of software developers and designers.

Obstacles

Modern visualization systems allow direct use of PLC tags in configured plant displays. This is one of the positive features that is often highlighted when HMI systems are marketed. It seems simple enough and implies that users can easily create any plant display of their choice. The drawback of this system is that it creates tightly coupled software components. This means that changes to the machine program then also entail changes to the tag management system and thus to changed tag addresses. As a consequence, the HMI displays must at least be recompiled and reloaded. Loose coupling between software components and independent software development are then no longer an option.

Solutions

A much better solution is to define a well-planned interface to the HMI. The machine program transfers the necessary data to the interface and fetches from the interface the data required for an operational sequence. Only interface tags are used in plant displays. This approach ensures that data are loosely coupled. The HMI and control system can be loaded independently of one another, and the control program and display configuring software can be developed independently. Another advantage of this solution is that data do not need to be collected across the entire program, but can be transferred in a block to the HMI system, a solution that guarantees significantly faster transfer rates.

1.5 Integration of motion control in PLCs

The call for machine manufacturers to build machines that could be retooled quickly for manufacturing new products made it necessary to achieve a greater flexibility of machine motion. This resulted in an increasing trend to equip motion axes with electric drives rather than with the conventional mechanical or hydraulic solutions. The deployment of electric drives made it necessary in turn to use systems that supported flexible positioning of the drive. As with HMI systems, positioning modules with special microcomputer systems that could be plugged into the PLC were developed and so made it possible to flexibly position drives in the machine. To enable positioning modules of this kind to be connected to the drive systems used in those days, they needed to be equipped with analog setpoint outputs and be capable of detecting the axis position via connectable encoder systems.

The WS600 system (Figure 5) was one of the first positioning systems developed for the SIMATIC PLC. This system comprised the WF625 positioning module and the WS600G display system.

Communication between the user program and the positioning modules was handled by a standard software package. Data blocks with data content that needed to be administered by the user program acted as the interface to the user program.

The traversing programs were programmed via the WS600G operator panels and the programming methods were based on the semantics relating to numerical control of machines defined by DIN 66025/ISO 6983. It was thus possible to adapt the traversing movements more flexibly to the requirements of the production process.

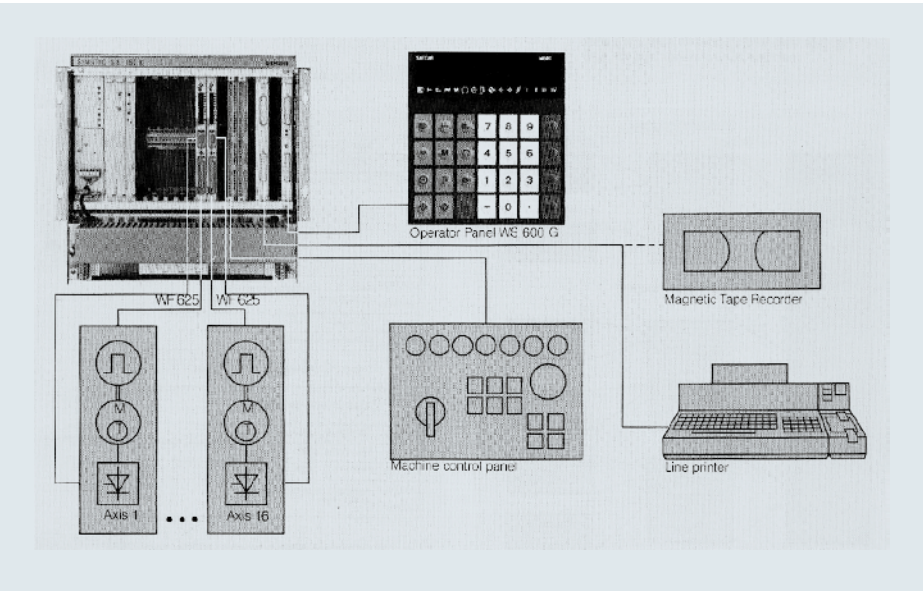


Figure 5 S5-150K with WF625 and WS600G

This trend towards integrating motion control functions into PLCs had a serious impact on the PLC control programs. The user program became responsible for managing the traversing movements in the form of CNC programs and the coordination of different modules. The fact that the positioning modules had their own cycle that was generally considerably shorter than the PLC cycle needed to be taken into account in the PLC program. Owing to these cycle time differences, synchronization routines had to be added to the PLC program.

If the traversing motion was executed faster than the time it took for the PLC to complete a scan cycle, then the signal changes of the positioning module could not be registered correctly in the PLC program. For the person programming the PLC, this made it difficult to determine whether or not the positioning process had actually taken place. When it came to fully automated processes, this lack of certainty was not acceptable. The PLC programmer therefore needed to take measures in the sequential program to ensure that positioning operations were clearly terminated. This could be done, for example, by implementing functions that compared the actual and target positions. Since it was possible to change the position values by entering programming commands at the WS600G operator panel, the programmer needed to ensure that the target positions required to perform the comparison were dynamically calculated so as not to impair the flexibility of the plant. The additional programming effort required to achieve this goal should not be underestimated.

These problems were solved by advances in the design of positioning modules which saw the implementation of handshake mechanisms at the interface. This development helped make PLC programs slightly simpler again.

Motion control functionality is fully integrated in modern PLC systems and an integral component of the PLC's operating system. Despite this integration of motion control functions, it is still necessary to invest programming time and effort in order to ensure effective organization of motion control movements:

- Programmers were forced to expand their domain knowledge in order to understand the processes managed by the motion control program. The knowledge they acquired lead inevitably to changes in the way the machine operating sequence was programmed.
- They also needed to understand the control processes of the positioned-controlled positioning modules and the subordinate drive control system. Errors occurring in these functional areas needed to be detected and managed by appropriate reactions in the machine operating sequence.
- Every positioning module works autonomously. As a result, it became necessary to synchronize positioning movements across multiple modules in the PLC. In this context, the most challenging task was to manage interruptions to the process caused by errors and restart the process smoothly again afterwards.
- In addition to organizing the motion control functionality itself, it was in some instances also necessary to create routines to manage the CNC programs in the PLC so that various retooling operations could be conducted quickly and automatically.

Integration of motion control functionality into programmable logic controllers did and does represent an important step in the process of enhancing plant flexibility. Motion control functionality has become an indispensable feature of modern machinery because the shift towards mechatronic systems is already in full swing.

Obstacles

Motion control functions are now an integral component of control systems. As in the past, it is essential for programmers to have the required level of domain knowledge if they are to create useful process plant programs. It is precisely because motion control functionality is integrated in the control system that software structuring and modularization have become so important. If we ignore this fact, it will simply be impossible to make further progress towards creating more abstract software modules.

Solutions

For programmers to write programs that are properly structured and modularized, they need to have an understanding of the domain engineering associated with motion control. Since this expertise does not come about by itself, the relevant personnel need to be given the time and opportunity to acquire the knowledge they need.

1.6 Drives become fully-fledged bus system nodes

The development of fieldbus systems for programmable logic controllers significantly reduced the volume of plant wiring required. As drive technology evolved from analog solutions to digital systems with processors, it was a completely logical step to implement the connection between controller and drive system via bus links. Bus systems had been a long-established engineering feature of control systems and use of this technology to couple several drives to a single control system was easy to accomplish. Standards committees like the PROFIBUS User Organization (PROFIBUS Nutzerorganisation PNO) had been working to establish standardized protocols for data exchange between control systems and drives. For example, they drew up the standardized PROFIdrive profile that was first defined for PROFIBUS and later for PROFINET (Industrial Ethernet).

As a consequence, control engineering software has been strongly influenced by the integration of motion control functionality in control systems and the use of digital buses to link drive systems. It is the responsibility of programmers to familiarize themselves with the specific characteristics of drive technology and to implement appropriate solutions in their software.

Obstacles

Modern drives are coupled to the control system via a bus or may even be an integral component of a motion control system. The software must be capable of supplying the drives with the data required for the process flow. It may also be necessary to display drive data on visualization systems. Detection of drive system faults for processing in the plant software is just as important as it is with positioning systems.

Programmers are therefore required to understand the basic principles of drive systems and know which drive data (including communication channels) are relevant from a programming perspective. This expertise is vital for designing suitable software concepts for plants equipped with drive systems and ensuring that software applications are modular and easily expandable.

Now that drive and motion control functionality form an integral component of the control system, it may be necessary to include further calculations in the programs to adapt the control process to certain operating situations. These might include, for example, dynamic calculation of correction values in response to changing circumstances or operator interventions. The use of high-level programming languages (e.g. Structured Text) has become indispensable as a means of keeping more complex calculation processes manageable.

Solutions

As mentioned in the previous chapter, it is essential to give the relevant personnel time and opportunity to acquire the required degree of domain knowledge. Motion control without drives doesn't exist. Knowledge about drive technology is just as important as an understanding of motion control.

1.7 PLC and PAC – what is the difference?

The history of development recounted in the previous chapters has been told from the perspective of the programmable logic controller (PLC). The first PLCs were born over 40 years ago and the scope of PLC functions has been expanding ever since.

The innovations that have emerged in the field of personal computer (PC) technology during the intervening period have given rise to another technical development. As the price of PC components gradually falls, it is becoming increasingly economical to integrate PC solutions in machinery. Rugged PCs suitable for industrial use have come onto the market to meet the demand for computers that can function reliably in harsh industrial environments. These PC-based controls have been expanded to feature connection technology for I/O components. Their appeal was further increased by the integration of visualization systems (HMI) and the ability to utilize existing communication mechanisms (Ethernet). Thus was born the Programmable Automation Controller (PAC). PACs combine the capabilities of a PC with the functional scope of a PLC.

The advantages of PC-based systems include extremely high performance, large memory capacity and outstanding ease of programming. Using appropriate software modules in the PAC it is possible to program control and PLC functions, motion control capabilities and much more besides. Since PACs are based on personal computer technology and therefore have multi-tasking-capable operating systems, they can be used to advantage to implement automation engineering solutions.

Thus, two different development routes have emerged in the field of automation engineering. The first route involves the expansion of PLCs through the integration of functions for communication, visualization, closed-loop control and motion control. The second route started with the PC and involves enhancement of this technology to include supplementary PLC programming and I/O connection options. It is now no longer possible to draw a clear boundary between PLC and PAC since both systems are offering an increasingly similar scope of functions.

When it comes to choosing the right control systems, users have to depend on the manufacturer's description. If a manufacturer describes his control system as a Programmable Automation Controller, then it is fairly safe to assume that it is a PC-based control system. If the manufacturer uses the term "PLC" to describe the control, it might very well be a system with PC-like functionality.

1.8 General conclusions about past developments

Every development increased the workload on programmers and entailed expansions to processing plant programs. Software developers needed to work continuously to keep abreast of advances in domain engineering and so ensure that modern-day plants can be automated by software. Control engineering has evolved into a highly complex technology and is a far cry from its beginnings in those early programmable logic controllers.

The multitude of tasks to be implemented in a modern control system requires efficient planning and proper design of the software. Modular programming and well-planned interfaces between modules ensure that the software can be easily

maintained and improved. Use of high-level programming languages such as Structured Text or SCL are absolutely essential to the successful management of complex software. High-level languages offer a significantly simpler, clearer means of describing a software solution. It becomes easier to read and interpret program codes.

If users regard classic PLC programming languages like LAD/FBD or STL as an essential standard, it will not be as easy to meet the challenges of the future because appropriate description languages will be needed if the full scope of functions integrated in control systems is to be used meaningfully. It is only by the use of high-level programming languages that more abstract programming methods will become possible.

That notwithstanding, it is still justifiable to use LAD or FBD for the purpose of expressing logic combinations because logic programs written in these graphical languages are extremely easy to understand and analyze. Each programming language should be used according to its own merits.

2 Basic Principles of Object-Oriented Programming

In this chapter we will briefly explain the basic principles of object-oriented programming (OOP) so as to provide readers with a sound foundation in the subject before they progress to further chapters. We have restricted our observations to the essentials and made a conscious effort to keep explanations as brief as possible. A broad range of books and information about object-oriented programming and the philosophies that underlie OOP are available on the Internet.

This book does not claim to give a comprehensive explanation of object-oriented programming nor a complete analysis of IEC 61131-3 ED3. Its primary goal instead is to give the reader a useful introduction to the subject of object-oriented programming based on the implementation of OOP in SIMOTION. The principles described are further illustrated by programming examples that have been tested on a SIMOTION system.

2.1 The basis of object-oriented programming

2.1.1 History

The principles of object-oriented programming [1] were formulated during the 1960s and 1970s. This new approach to programming emerged as a result of failed software developments during the mid-1960s. This was the first time that software costs had exceeded hardware costs, a situation that triggered the so-called “software crisis”.

But OOP did not gain in popularity until the mid-1980s and has become a widely accepted computer programming concept during the intervening years.

Most modern programs for personal computers are written in the language C++. These programs would be inconceivable today without object-oriented programming. When it was time to switch from C to C++, programmers were faced with the same paradigm shift as they are today in automation engineering.

With the Internet age came new programming languages. Java is generally the most popular of these because it was integrated very early on as a programming tool in WEB browsers. Java as well as C# both fundamentally support object-oriented programming.

[1] Source: page “History of programming languages“. In: Wikipedia – The Free Encyclopedia. Revision level: December 14, 2015. 10:38 UTC. URL: https://en.wikipedia.org/wiki/History_of_programming_languages (viewed on: March 22, 2016, 16:17 UTC)

2.1.2 What's different?

Object-oriented programming encapsulates functions and data in a single object. This means that data are tightly coupled with the object and the programmer is free to decide the means by which data may be accessed. Furthermore, in the event of a legal data access operation, it is possible to check whether the data have been changed to meaningful values. These check mechanisms are capable of preventing illegal changes to data and eliminating other error sources. The object has complete control over all its responsibilities at all times.

In Figure 6 you can see two objects in human form communicating with one another. Let's call our objects Michael and Manfred. Michael asks Manfred whether he's got a euro to spare. Manfred is not willing to give away a euro. Since Manfred always retains control over his responsibilities and data, Michael is not going to get a euro.

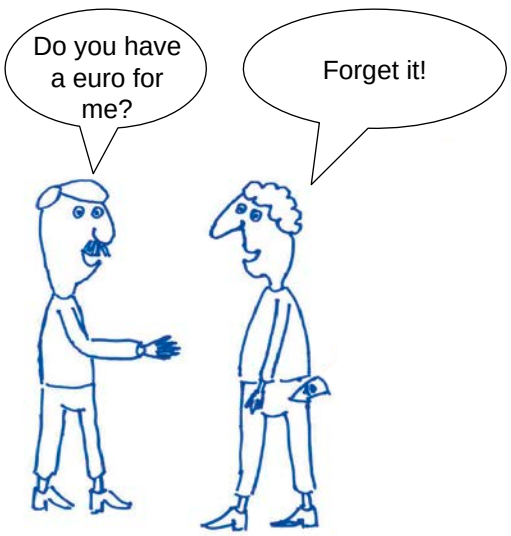


Figure 6 Communication between objects – object-oriented

The old procedural programming method that comprises programs, subprograms, functions and data breaks a task down into individual components. It thus uses a series of commands to define algorithms which ultimately solve the task. However, a basic principal feature of procedural programming is that data are not necessarily directly coupled with programs, subprograms or functions. In other words, data are defined by the programmer and made public for use by programs and functions. They can thus be globally accessed by any program. When the software is extended or modified, however, the overall contexts within the data mesh are often not documented, creating an error source which can lead to malfunctions in the processing plant.

In the example of communication between objects in human form, Manfred has now disclosed his data in accordance with the procedural method (note: the term

"objects" is not strictly correct when referring to procedural programming). He has disclosed where he keeps his money. Anyone now has access to it and can change the data (Figure 7). No communication takes place and Manfred is no longer fully in control. He's likely to have difficulties – the next time he goes shopping at the very latest.

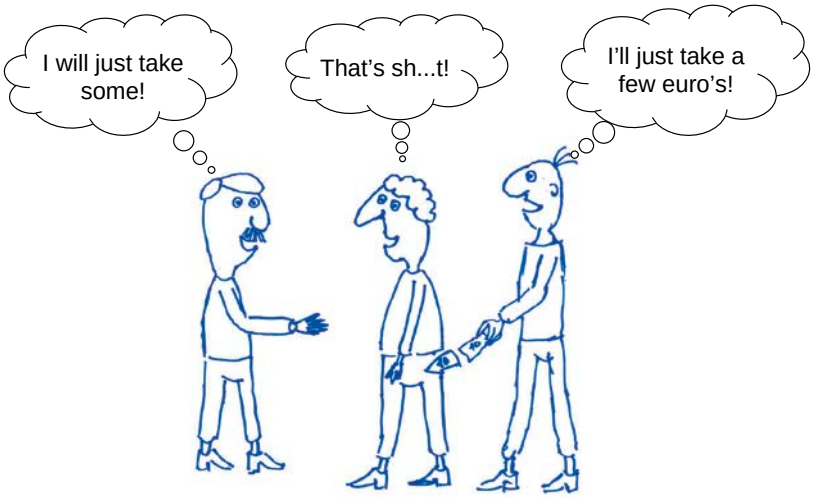


Figure 7 Communication between objects – procedural

This example illustrates a fundamental problem with procedural programming. Since there is no means of limiting or channeling access to data, changes to data can give rise to sporadic and often inexplicable errors that can be very difficult to locate.

With object-oriented programming, data and operations are uniquely assigned to the object. The programmer has the option of specifying how data can be accessed and can therefore protect the data. If data need to be changed for the purpose of the object function, this can only be done by specifically programmed methods. These methods can include a mechanism for checking whether data changes are meaningful. Erroneous transfer of data that can cause processing errors is prevented. An explanation (including examples) of how methods are programmed and used can be found in later chapters. First of all, however, we are going to lay the foundation for helping you to understand the basic principles of object-oriented programming.

2.1.3 What does object orientation mean?

The whole concept is actually pretty simple. The term "object orientation" is used because the language is entirely based on the concept of "objects". As human beings, we generally regard the world as a collection of objects. Everything that we perceive is an object. Houses, cars, people, plants and animals, in other words, all tangible things, are objects to us. At the same time, we can certainly view some of these things

as complex objects which may each comprise a collection of different objects. For example, a car consists of other objects such as wheels, seats, engine, bodywork, etc.

We also define more abstract groupings such as vehicles, for example, to which we then assign specialized derivations. A car is a vehicle, but a particular specialization of vehicle. A bicycle is not a car, but it is a vehicle. Since everybody knows the difference between a car and a bicycle, but both these objects can be assigned to the grouping "vehicle", we need to ask ourselves: How do we differentiate between these objects?

To a vehicle we attribute certain universal properties and potential functions that all vehicles must possess. Every vehicle has a number of wheels and some form of drive mechanism that renders it capable of movement (acceleration, braking and driving are all operations). There are of course other properties that can be used to describe a vehicle. These are attributable to cars, bicycles, motor bikes or even horse-drawn carriages.

It is therefore certainly true to say that when we consider properties, we describe vehicles in more generalized terms than specialized forms of vehicle such as cars, bicycles or HGVs.

Specializations thus inherit the properties (attributes) and the behavior of the original abstract object and tend to refine or expand these attributes.

Let's take a vehicle registration certificate as a useful example of how vehicle properties are described. This certificate defines all the properties of a vehicle that is authorized for road use according to road traffic regulations. These properties "include" specific values for the relevant vehicle such as length, width, engine type and capacity as well as registration number, owner and so on. The vehicle registration certificate therefore describes a particular vehicle, i.e. the specific object, for example, a car with all the defined attributes.

2.1.4 Objects and their interactions

Objects possess properties (attributes) and potential operations (functionalities). It therefore follows that objects of the same kind must have the same kind of functional scope. Individual vehicles are capable of movement, i.e. are capable of being driven. A vehicle can accelerate or brake in order to move or come to a stop. We need to remember that a car cannot move on its own, but simply has the potential to do so.

A car is generally controlled by another complex object, i.e. by its driver. The object "car driver" is completely different from the object "car" and there is no comparison between the two. For the car driver to be able to drive the car, some way needs to be found to coordinate the two objects.

The driver has hands, feet and eyes (and a brain as well of course, but we can ignore that for the moment). The car has a driver's seat, steering wheel, ignition lock, accelerator pedal and other control equipment. These have in turn been coordinated with the limbs of the human driver. When we consider this on a more abstract level, the object car and the object driver have mutually compatible interfaces and it is precisely these that allow the two objects to communicate.

To accelerate the car, the driver needs to press down the accelerator pedal. The car then accelerates at the rate determined by the position of the pedal. It is important

to understand, however, that acceleration is effected by the engine, gearbox and wheels and not by the accelerator pedal.

In other words, the car has a function (method) for accelerating and a method for commanding acceleration. Thus it can be said that objects of this kind possess not only methods to implement potential operations, but also methods to communicate.

The speedometer likewise displays the current traveling speed of the vehicle. The driver can therefore release the pressure on the accelerator pedal until the vehicle is moving at the desired speed. The speedometer is thus also a method possessed by the car to feed information back to the driver, i.e. a method of communication.

In our modern, technologically advanced world, we have a vast number of complex objects that possess executable operations (methods) as well as further methods for communicating with other objects. Objects always retain control over their own responsibilities and attributes. This encapsulation means that changes in the behavior or the attributes of an object can only be effected in response to requests (methods). The object itself decides according to existing (implemented) capabilities whether or not it can fulfill the request. Pressing the accelerator pedal for our object car will not directly cause it to accelerate if, for example, the engine is not running and/or the car is not in gear. To be able to drive a car, the driver needs to understand its fundamental behavior but does not need to know exactly how the car starts its own engine or accelerates. The only thing the driver needs to grasp is what has to be done to get the car to start its engine.

We can therefore deduce that a car possesses externally accessible (public) methods and internal methods that are not externally visible. A core principal of the object orientation concept is that access to the internal data or functions of objects is not permitted. Data can be changed only by the methods provided expressly for this purpose.

We have assimilated this view of the world as a series of objects and we find it easy to describe and categorize them. We learn to handle and interact with objects from an early age. Based on this insight, it is simply a question of logic when technical functions exploit this view of the world for their own benefit. By applying this model to software, it becomes easier for us to understand many aspects of software operation. This precisely was the driving force behind development of the object-oriented programming concept.

2.2 General principles of OOP

2.2.1 Objects

Object-oriented programming applies the human view of the world to the program engineering environment and so helps us to understand it better. Like a human being, OOP views the (software) world as one that comprises objects. An object is an entity that comprises properties (data) and potential operations (methods).

A hydraulic aggregate in a machine, for example, can be conceived as an object (Figure 8). This object supplies the oil pressure required for other sub-aggregates in the machine.

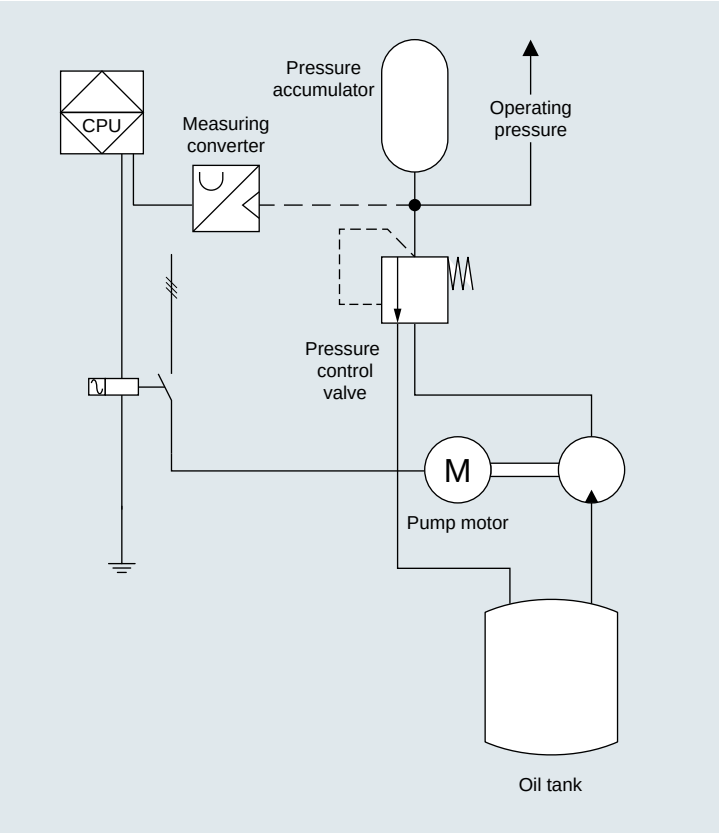


Figure 8 Hydraulic aggregate

This object is also mapped in the control system. The object possesses the following internal functions:

- The hydraulic aggregate must be switched on and off.
- The hydraulic pump must remain in operation until maximum pressure is reached. The pump then shuts down.
- If the pressure drops to the specified minimum pressure, the hydraulic pump is restarted so as to increase the pressure again.

The object “Hydraulic aggregate” therefore possesses the methods “Switch on” and “Switch off” and the properties “maximum pressure” and “minimum pressure”. The term “attributes” is also often used to refer to properties. These are represented by variables in control engineering applications.

Of crucial importance in this interpretation of software modules as objects is that the attributes (data) and the operations (methods) are inseparably linked to the object, i.e. they are encapsulated. An object operates autonomously with its implemented methods and always retains control over its own data.

This means, for example, that any change to the maximum pressure setting made necessary by other factors will be possible only if the object has implemented this

as an operation. This could be achieved for instance by additional methods such as SetMaxPressure or SetMinPressure. Overwriting values by accessing data directly is rendered impossible by the OOP concept. Since the control over all responsibilities rests entirely with the object itself, communication with the object is effected by means of methods.

This encapsulation makes the programs extremely secure because unintentional changes to public values as a result of programming errors are made impossible. Furthermore, it is easier to ensure compliance with specific plausibility rules (such as minimum pressure < maximum pressure) in value transfer methods (setter methods).

2.2.2 Classes

The object-oriented programming technique requires the definition of a blueprint for an object type before objects can be generated. This definition of an object type is referred to as a class. The class specifies the attributes (properties) and the methods (operations), i.e. it contains program code. Instances of this blueprint (class) can now be formed (a process referred to as "instantiation"). The instances of a class are the objects.

Once a class (blueprint) exists, instances of it can be created and thus several objects of the same type generated. With the example illustrated in Figure 9, it is thus possible to generate multiple hydraulic aggregates (e.g. Hydraulic_1, Hydraulic_2,... Hydraulic_n) by instantiation. Each unit is assigned a unique name by the application programmer. It has its own methods and attributes and can function independently of the other instances.

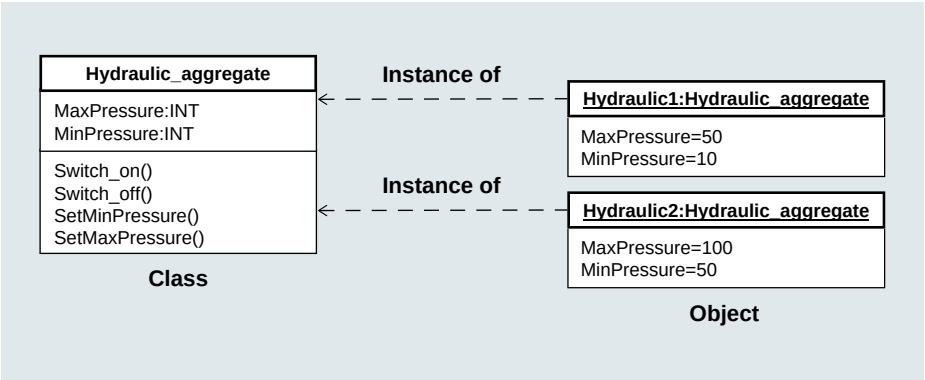


Figure 9 Class and object

2.2.3 Inheritance

After a unit and the associated software have been prepared for inclusion in the control software, it is often the case that new ideas, wishes and requirements emerge and result in extensions and modifications to this unit. It is generally true that multiple variants of the same unit need to be maintained in parallel. These requirements also need to be addressed in the control system software. In the past, it was common

practice to copy existing software modules and then extend or modify the program code. This process was repeated every time a new variant needed to be generated.

With conventional programming methods, this approach caused a variety of problems. Identical program code is duplicated in the sources. A large number of similar software derivations are created over time, making software maintenance extremely difficult. This situation becomes especially challenging with large-scale software projects.

By using the principal of inheritance, object-oriented programming offers a solution. It allows fully functional software components to be left intact while making it possible to integrate modifications as circumstances or requirements change.

However, the benefit of inheritance mechanisms needs to be considered in the context of increasing the specialization of the functions of a class. In other words, deriving a subclass from a base class is motivated by the desire to implement new functions rather than to introduce minor adaptations or advances. Minor adaptations are implemented in the program code of the base class. There is otherwise a risk that too many derivations will be created over time and the software will become difficult to manage. It is exactly this situation that we need to avoid. To obtain an optimum class design, i.e. to meaningfully divide functions into base classes and their derived classes, precise analysis of the commonalities and differences between classes is essential.

Once a class is fully defined and operational, it can pass on its methods and properties to a new class which utilizes these inherited methods and properties. To achieve further specialization, new methods and properties can be added or existing methods adapted (overridden).

The “Hydraulic_aggregate” class is a very simple implementation involving a two-step control system. The hydraulic pump remains in operation until the maximum pressure is reached and stays out of operation while the pressure remains above the minimum value.

We are now going to create a new hydraulic aggregate which is capable of tracking a defined pressure setpoint and maintaining the pressure until a new pressure setpoint is specified.

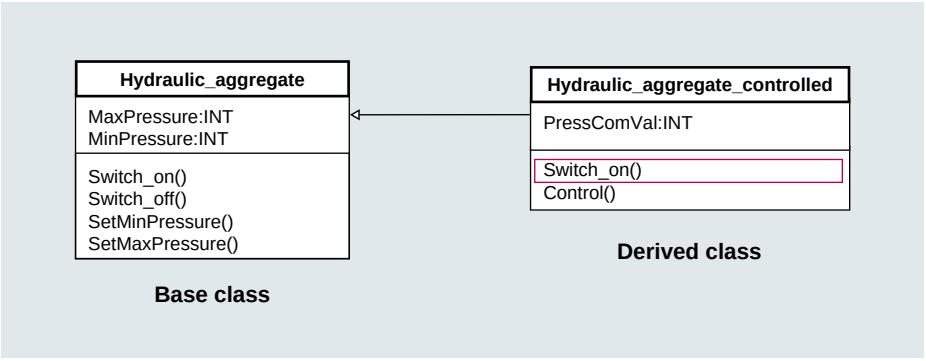


Figure 10 Inheritance principle with classes

To implement this new design in the software, the object-oriented programming principle allows us to derive a new class “Hydraulic_aggregate_controlled” from the base class “Hydraulic_aggregate”. The Hydraulic_aggregate_controlled class inherits the methods Switch_on(), Switch_off(), SetMinPressure() and SetMaxPressure() and the properties “MaxPressure” and “MinPressure”. To implement control according to a pressure setpoint in the new class, it needs to be assigned this new property. A new method “Control()” that provides control functionality also needs to be programmed. The programmer can choose various options for specifying the pressure setpoint. If a variable pressure setpoint needs to be specified during operation, an additional method, e.g. SetPressureSetpoint, has to be implemented. But if the pressure setpoint always remains constant, it is defined by an initialization value when Hydraulic_aggregate_controlled is instantiated. In our example, we have decided to choose the second method.

With the inheritance principle applied by deriving an extended class from a base class, it is not simply a question of copying the base class but of permanently maintaining the relationship between these classes. The new subclass is inseparably linked to the base class. This means that any changes to the programming of the base class will always have a corresponding effect on the derived subclasses because all subclasses will have inherited the attributes and methods of the base class.

It is possible to continue deriving other classes, either from the base class itself or from the extended subclasses derived from the base class.

2.2.4 Overriding

In addition to the inheritance principle, object-oriented programming also offers another means of adapting software. Adding new methods to a derived class does not achieve the desired result in many cases, and it becomes necessary to expand or modify methods that already exist. The mechanism used to implement changes of this kind is referred to as “overriding”. This mechanism is used to override the methods inherited by a derived class and to implement a different method in their place. Furthermore, the newly implemented method can be programmed to call the original method in the base class. In this instance, the program code originally implemented in the base class remains intact and continues to function as normal.

To give an example, the method “Switch_on()” inherited by the derived class Hydraulic_aggregate_controlled might not function properly because Hydraulic_aggregate_controlled requires a drive that is switched on with transfer of a speed setpoint. As a result, the method Switch_on() needs to be changed with new or additional program code. This problem can be best solved by overriding the Switch_on() method. Appropriate keywords exist for OOP in order to implement specific solutions. In order to override the “Switch_on()” method, the program code beginning with the keywords “METHOD OVERRIDE Switch_on()” is implemented in the class Hydraulic_aggregate_controlled. When overriding methods, it is essential to remember that only the program code for the method and possibly other temporary variables may be changed. Changing the name results in the creation of a new method and changing the defined external interface of the method is not permitted.

If an object of Hydraulic_aggregate_controlled is generated, it will possess a method with the same name as the corresponding method Switch_on() in the base class, but its method might have a completely different program code. This is why the method

Switch_on() is included in Figure 10 (UML representation – Unified Modeling Language) of the derived subclass Hydraulic_aggregate_controlled because the method inherited from the base class Hydraulic_aggregate has been overridden.

2.2.5 Interfaces for object interaction

With the mechanisms described above (class definition, inheritance and overriding), we focused our attention on how software can be structured so that it can be reused and expanded. In this context, a class always defines the overall functionality of an object and specifies how the relevant functions are implemented. In other words, a class formulates all the functions required for a specific object type. If we repeat this process for different objects in a given environment and compare a number of subfunctionalities, we will discover that there are no essential differences between them. In order to group these subfunctionalities so that they can be utilized in the programming of a class, we would need to be able to derive several base classes for inclusion in one class. According to IEC 61131-3, however, the multiple inheritance concept required to achieve this goal does not exist. The interface construct is used to address this problem.

The purpose of interfaces is to provide a separate definition of commonalities, i.e. subfunctionalities which are generally regarded as distinctive and form the basis for several different classes. The methods required to implement these subfunctionalities are grouped to form an interface. Attributes cannot be defined in an interface (as they can in a class) nor do the methods in an interface have any separate program code in that interface.

The objects of a class that implement an interface are still objects of the interface (as with the inheritance principle). Since a class is capable of implementing an unlimited number of interfaces, these offer a suitable method for compensating for the lack of multiple inheritance in Structured Text (by contrast with C++).

Interfaces play another important role by allowing interaction between objects. They are an efficient tool for implementing data exchange.

Let's use the example of the hydraulic aggregate described above in order to illustrate this mechanism. Let's assume that multiple hydraulic aggregates as well as other types of equipment can be deployed in the plant. We also want to make it possible to display the status of all the installed units on an HMI. In addition to the class of the hydraulic aggregate (and for all other equipment types), we also need a class to supply the display data for the HMI.

So that we can query the status of the aggregate, we need to program the method Status for the relevant class. We want to use this method to implement HMI diagnostics at all objects of type "Hydraulic_aggregate". However, since it is not only hydraulic aggregates that are installed in our plant, but also other types of equipment (e.g. electrical drive systems, final control elements, etc.) and we want to equip these with the same fault diagnostics functionality, we will define the interface IStatus. This interface includes the definition (but not the program code) of a method that will query the status of equipment.

Our hydraulic aggregate implements the interface IStatus and must therefore provide a status query method as defined in the interface.

All other equipment types in our plant are assigned by the same method a (different) implementation of the interface IStatus and can deposit their diagnostic functionality in a specific manner.

All aggregates that are to participate in the HMI diagnostics system must now be registered with the class HMI via the interface IStatus using the method registerObject() and administered in a list in the class HMI (Figure 11). Class HMI can now be used to diagnose all aggregates. The fact that different implementations of methods can be called depending on the type of aggregate is concealed from class HMI.

Interfaces thus allow the definition of neutral mechanisms so that information can be exchanged between different object types. In other words, methods are defined as prototypes (without implementation) in an interface. If an interface is implemented by a class, this class is responsible for providing the methods defined in the interface with the corresponding functionality. In turn, an interface can be used independently of the underlying implementation.

Interfaces significantly increase the potential of object-oriented programming and make it simpler to combine different object types within the same environment.

This brief description of interfaces may well be insufficient to give full clarity about their role, but we will discuss them in more detail in later chapters.

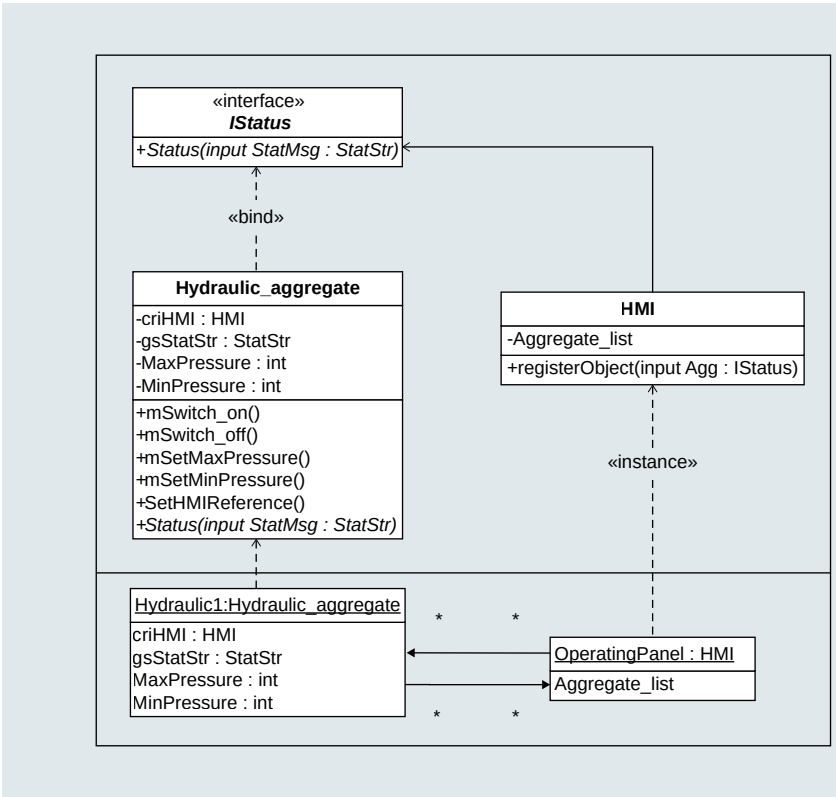


Figure 11 Hydraulic aggregate with HMI display

2.2.6 Summary

Object-oriented programming has used our object-focused view of the world to create a new software engineering concept.

- The software then consists of objects.
- Classes are the basis for objects. Attributes (properties) and methods (operations) are defined (or programmed) in the class.
- Objects are formed by creating instances (instantiation) of classes. An object thus represents an instance of a class.
- The internal functions of objects are provided in the form of internal methods.
- Objects interact with one another using public methods.
- Different objects can interact via defined interfaces.

The following also applies:

- Objects function independently of one another and each object always retains control over its own methods and attributes.
- Extended classes can be derived from existing classes. The derived class (subclass) inherits all the attributes and methods of the base class. Further attributes and/or methods can be added to derived classes in order to adapt them to new requirements.
- The override mechanism is another tool for adapting OOP software. Inherited methods can be overridden in derived classes. Using this mechanism, it is possible to change the existing program code of a method without altering the original program code.
- Inheritance is a powerful feature of the object-oriented programming concept. By deriving classes from a base class, it is possible to create specialized (refined) subclasses of the base class. The base class thus embodies the commonalities between classes and passes these commonalities on to the specialized subclasses. Despite its enormous potential, this mechanism should be used sparingly. Motto: “Avoid inheritance if you can find an alternative.” This is what Bjarne Stroustrup says in his book “The C++ Programming Language”. The reasons for this are simple: Design errors in the base class affect all other classes in the inheritance chain. This can happen if the software has not been planned with sufficient care.

OOP should be used whenever it can offer concrete benefits to programmers and users. In some situations, a procedural programming solution might be simpler and more efficient. The OOP programming language is not a panacea and does not speed up the software development process. It does however ensure that the software is better organized and easier to manage. When the OOP concept is applied correctly, it offers a number of advantages that can result in time savings.

As with all forms of expression (programming languages) in software engineering, poor program code can also be written in the OOP language. This problem can only be avoided by sensible planning of software. Programmers who work according to “straight from brain to terminal” should not bother with OOP.

2.2.7 Advantages of using OOP

- The encapsulation of objects is a feature of object-oriented programming that ensures highly reliable program execution.
- When the OOP method is used, it becomes significantly easier to design modularized software.
- Thanks to the encapsulation, inheritance and overriding mechanisms associated with OOP, the software is simpler to manage and easier to modify overall.
- When OOP is utilized in appropriate program modules or libraries, the degree to which the software can be standardized and reused is increased considerably.
- The amount of time and effort involved in programming can be reduced thanks to the inheritance principle (because program code does not need to be copied or adapted again).
- Larger-scale software projects are easier to implement with substantially fewer errors.
- By comparison with procedural programming, object-oriented programming makes it far simpler to develop software components independently. This can be achieved by consistently applying the principles of encapsulation and clear interface definition.
- Interfaces support generic programming, helping to reduce the time and effort involved in adapting software and making it significantly easier to combine objects of different types.
- OOP mechanisms permit the implementation of module tests without necessitating changes to the modules themselves. Software testing is made simpler.
- OOP has a much closer resemblance to the object-focused world view of the human being than procedural programming.
- While it is necessary to learn the “mindset” behind object-oriented programming, the concepts are relatively easy to understand and bring enormous benefits to the programmer.

2.2.8 Disadvantages of OOP

- Design errors in the base class affect all other classes in the inheritance chain. Retrospective corrections require extensive reworking of derived classes.
- When OOP is used, the decision as to which function must be called is often made during runtime. As a result of this dynamic binding, the programs take longer to execute. This increased runtime should be taken into account when program concepts are prepared. It is easy to compensate for runtime increases by appropriate use of existing programming mechanisms (references to these mechanisms can be found at various places in chapter 3). Since it is not usual for objects to be dynamically generated and destroyed in control systems during runtime, however, the programs take less time to execute than they do in other systems.

2.3 Tips about defining classes

Once you have embarked on an object-oriented programming project and you have analyzed your existing programs, you will ultimately be faced with the question: How can I define the right classes from these programs? It is a question to which there is no easy answer nor is there any standard recipe that will show you a clear way forward. That is to say, there are various possible solutions, all of which can be correct. As often happens in life, your skills increase with your experience, and as you spend more time working with OOP, you will find it easier to see the right way forward. These comforting gems of wisdom unfortunately do little to help the beginner. For this reason, we have decided to give you some tips about defining classes at this early stage and so help you to get started. We have used "The C++ Programming Language" written by Bjarne Stroustrup as a source of information for the following description.

There are essentially two types of class within a system:

1. Classes that are application-oriented.
   They are directly deployed by the user to describe solutions.
   a) These classes describe the real objects of the actual application. These are objects that represent elements or assemblies in the machine including, for example, axes, valves, valve-cylinder combinations or other machine elements, but also software modules that act as a direct link to operator interfaces.
2. Classes which represent artificial implementation constructs.
   These are classes used by designers and programmers to express their implementation techniques.
   a) This type of class describes the fundamental implementation method and focuses on the design of the software environment.
   b) This category also includes classes for defining the method of connection of I/O components or classes which utilize hardware resources of the system. As a general rule, these classes are not directly used by the programmer, but are used to adapt the software to various system responses.
   c) Last but not least are the defined interfaces that need to be implemented in classes. Interfaces define communication relationships and support the independent development of software modules. Interfaces can embody definitions for transferring information between different programs, for example, or can function as neutralized interfaces to and from I/O components of the same type (see chapter 3.5).

It is entirely possible that some readers of this section will still have a few question marks in their minds and find some descriptions difficult to understand. But take our word for it – once you have worked through the coming chapters, the fog will lift and everything become clearer. Before you can gain a comprehensive understanding of the subject, you need a detailed explanation of the mechanisms used in object-oriented programming.

As you start to learn more, you will still have a lot of open questions that need to be answered. Using knowledge gained from various experiences, we have found

that the following method has proven to be a practical solution. First of all, you need to focus on the specific objects of a machine and identify a suitable object for implementing the first class. Once the relevant objects with this class are proven to function reliably in a test environment, you can functionally extend the class using inheritance mechanisms. Once you have advanced to this stage, you will need to find some method of displaying potential errors in the object. You will need to develop a solution for passing on error messages. You will need to ensure that the functional level in which you are working can be implemented independently of the display function. It therefore makes sense to program a class specifically for the display function. OOP techniques must be used to link this class in such a way that both classes (display and machine objects) can be developed independently.

To successfully complete these tasks, you will need to apply and understand all the OOP mechanisms implemented in the system. By using these techniques, you will improve your understanding of the underlying principles and answer some of the questions you had at the beginning. As you work through this process, you will need to resist the temptation to resort to the "procedural programming" mindset. As your understanding of object-oriented programming increases, you will start to see software in a different light. As you think about software solutions, you will find that your mind can work on a much more abstract level. You will develop a completely new approach to software design and forget the procedural way of thinking. You will need to allow your brain a little time to adapt to this new way of thinking. It is going to take a certain amount of patience, energy and practice. The learning path described here is supported by relevant examples. If you practice using the programs in this book, you will find it easier to understand and use object-oriented mechanisms.

Other approaches to learning the principles of class definition in object-oriented programming certainly already exist, but we feel that it is generally advantageous to work from the simple to the complex and from the specific to the abstract. Every programmer and designer should trust in their own common sense and above all find the energy and courage to give up on a model they have created if it proves to be unfit for the purpose. Because even unworkable models can provide us with useful experience.

The chapters below are structured so that we start with an explanation of the principles of a range of interrelated issues. The theory is then applied to specific software examples. This principle of an explanation followed by a relevant example is applied consistently. But we should mention right at the outset that you will still not be a specialist in object-oriented programming even after you have worked through all the chapters. But you will have a solid foundation for becoming a specialist in the future.

References

There is a wide range of books available on the subject of OOP software design. However, since OOP is still not as widely used in automation as it is for PC applications, fewer references exist in relation to object-oriented programming according to IEC. Nevertheless, the migration problems that arose when C++ was introduced are precisely the same as the problems caused by introduction of OOP to automation engineering. Furthermore, software planning methods are not dependent on the selected programming language.

For this reason, we will mention here two books that contain a vast amount of information and guidance on software planning:

- Bjarne Stroustrup: The C++ Programming Language. 4th edition, Addison-Wesley 2013
- Alan Dennis, Barbara Haley Wixom, David Tegarden: Systems Analysis and Design: An Object-Oriented Approach with UML. 5th edition, Wiley 2015

3 Object-Oriented Programming

Object-oriented programming is a programming paradigm that has been reliably applied in various PC programming languages for very many years. The term "programming paradigm"[2] refers in this context to a fundamental style of programming, i.e. to a programming method. A well-defined programming method makes it easier to read and understand the program codes written by programmers.

Programming methods have changed several times in the past and these changes were referred to as "paradigm shifts". A paradigm shift can naturally only take place if the relevant programming language supports a particular programming method.

As with all technical constructs, programming languages generally undergo continual development. What makes life easier for the user is that a programming language can be functionally expanded without losing its existing scope of functions. This means that existing programs do not need to be modified. Gradual changeover to new methods can be smoothly planned and the experience gained can be put to good use to improve the quality of programs.

With the advent of object-oriented programming, the field of automation engineering is facing another paradigm shift. Over time, this change will have a huge impact on the methods used to program control systems. OOP makes it simpler to increase the security of programs by encapsulation, to adapt software more easily to new requirements and to achieve greater modularity. Programs can be defined on a more abstract level and the software design can be planned. The programs are thus easier to read and the software simpler to maintain and develop. Since the existing programming methods can comfortably coexist with object-oriented programming, users will have time to plan the transition between them. To achieve a smooth changeover, however, users must understand the available mechanisms.

3.1 Implementation of OOP with SIMOTION

A large number of extensions have been added to IEC 61131-3 ED3 as compared to previous editions. A significant new feature is the definition of the scope of language for object-oriented programming. These definitions for OOP form the basis for implementation of OOP in the SIMOTION control system.

The IEC permits two basic concepts for the implementation of object-oriented programming:

- The first concept describes the use of OOP with object-oriented function blocks. With this form of implementation, all functions are provided by appropriately defined and programmed OO function blocks which therefore substitute classes.
- The second concept describes OOP implementation based on classes.

[2] "Programming paradigm" In: Wikipedia – The Free Encyclopedia. Revision level: March 17, 2016, 16:54 UTC. URL: https://en.wikipedia.org/wiki/Programming_paradigm (viewed on: March 22, 2016, 16:22 UTC)

The decision has been made to implement class-based object orientation for the SIMOTION control system. This corresponds to the way in which OOP is usually implemented in other programming languages. The extensions defined in the IEC standard have been applied to SIMOTION in a two-stage process.

- The first stage includes the implementation of classes with methods and the inheritance mechanisms associated with the classes. It also covers implementation of interfaces and the mechanisms (e.g. interface variables) including inheritance that are normally required for use of interfaces. The user therefore gains the use of all the important OOP mechanisms in this first stage.
- The second stage involves full implementation of the extensions described in the IEC standard. The scope of software design options will be expanded significantly for the user, particularly by the introduction of references.

This staged concept has been designed to ensure that SIMOTION users can use object-oriented programming effectively during the first stage. Interfaces and abstract classes increase the level of abstraction for developing programs and give users the option of developing completely separate applications.

The second stage completes implementation of object-oriented programming and gives the user an enhanced scope of design options. The stages have been conceived to ensure that the user has full compatibility between both stages. This means that the second stage simply expands the range of available functions and that any programs already developed during the first stage can be accepted without any changes. The first stage essentially covers the following functions:

1. Methods in function blocks
2. Introduction of classes (CLASS) with their attributes (variables)
3. Introduction of methods (METHOD) to classes
4. Inheritance of classes and their methods
5. Implementation of interfaces
6. Handling of interface variables

In the second stage, the following are added to the functional scope:

1. General references
2. I/O references
3. Namespaces

3.2 Function blocks with methods

The IEC provides for the use of object-oriented function blocks (including derivation and methods) as an object-oriented model. It has been decided not to use this form of object-oriented programming in SIMOTION.

But the option of programming methods in function blocks (FBs) has been implemented, though without inheritance (derivation) or overriding of methods. By using methods in function blocks, therefore, it is possible to take a step towards object-oriented programming without needing to switch completely over to OOP. The

extensions associated with this approach also work on earlier versions of SIMOTION Runtime.

Another advantage of programming methods in FBs is that the content of the blocks can be structured much better than previously possible. The programmer can control access to variables. As a result, these FBs are better encapsulated than their predecessors. The extended scope of options for programming function blocks is described in the following chapters. In order to illustrate the improvements more clearly, we will first take a look at the old method and temporarily ignore the OOP extensions implemented in SIMOTION V4.5.

3.2.1 Modularization without OOP extensions

Modular programming was essentially based on the use of function blocks. An element used several times in the same form in one machine was represented in the software by programming a function block.

Hydraulic or pneumatic cylinders are frequently deployed in machines to perform simple motions. The motion is controlled by a valve suitable for the purpose (Figure 12).

The movements of the cylinder are triggered by energization of the valve solenoids. The limit switches “StartPos” and “EndPos” output a signal when the cylinder reaches the relevant limit position. The 4/3-way valve used in this example has the advantage that the cylinder stops in its current position when the valve is in mid-position (no solenoid energized). The cylinder behaves differently depending on whether it is

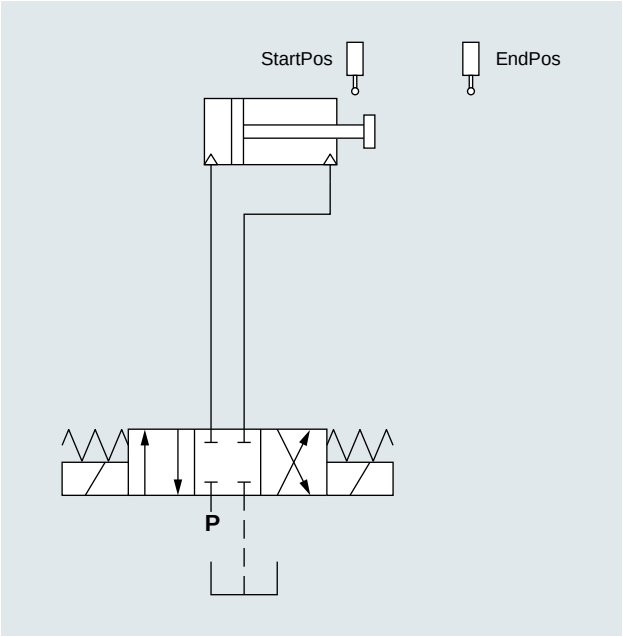


Figure 12 Valve-cylinder combination

hydraulic or pneumatic. If a hydraulic cylinder is used, it will actually stop (unless there is leakage of hydraulic fluid). If a pneumatic cylinder is used, the cylinder might continue moving (due to compression of gases) even when the valve is in mid-position. The difference in the behavior of these cylinders can certainly necessitate variations in the programming in a function block. For the sake of simplicity, oil is used as the medium in this example.

To provide the functionality of a valve-cylinder combination of the kind illustrated above, a function block named “FB_Valve” is programmed with the required input and output interfaces.

- Inputs
  - Forw – forward command
  - Backw – backward command
  - EndPos – front limit switch
  - StartPos – rear limit switch
- Outputs:
  - QForw – output to valve forward
  - QBackw – output to valve backward
- IN_OUT
  - State

In order to execute program code for different valves in the same machine, an instance of the type FB_Valve is created in the program for each valve. The FB is called via the relevant instance. The appropriate signals are combined via the FB interfaces so as to activate the functions of the FB. But it is not possible to recognize the selected function directly from the instance call. This can only be worked out

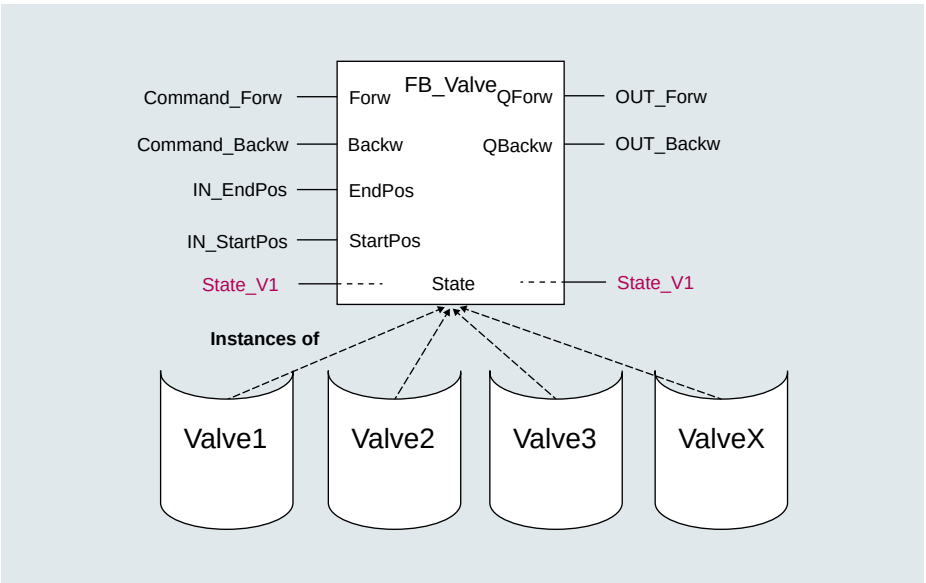


Figure 13 FB_Valve

by monitoring the signal status as the program executes. Furthermore, FBs of this kind can perform a large number of functions by the combination of signals. But too many signals can lead to confusion.

Data can be transferred between the function block and other software components via the IN_OUT parameter “State”. It is generally by this method that read and write structures are passed on. The valve deposits its status in this structure. This status can be used, for example, to display the valve status in an HMI system for diagnostic purposes.

If the FB needs only to write its valve status to the structure, but not have read access to it, it is possible in this case to change the IN_OUT “status” to an OUTPUT parameter.

3.2.2 Program and data are separate

Definition of function blocks and declaration of their instances are treated as separate programming tasks. For a new valve (which requires an additional FB call), for instance, the programmer defines a variable with the data type of the FB. The instance is generated when the program is compiled.

The FB data are stored in the instance. The definitions of the input and output interfaces need to be public of course. All other variables are local and thus cannot be accessed externally. However, since it is often necessary for certain data to be propagated outwards when the FB is processed, the programmer needs to take corresponding measures in the FB program. The normal method of doing this is to transfer a reference at the IN_OUT parameter to the external data. This reference is the “State_V1” in the example below. Thus, these data are declared outside the FB and completely separate from the FB although they actually belong to the FB program.

According to the requirements defined in IEC 61131-3, the user has two options of implementing an FB call.

The first option is to use the so-called complete “formal call”. With this call type, all parameters of the FB are assigned in parentheses after the instance name. The call from our example then looks like this:

```
//complete formal call
Valve1    (Forw:=Command_Forw
          ,Backw:=Command_Backw
          ,EndPos:=IN_EndPos
          ,StartPos:=IN_StartPos
          ,State:=State_V1
          ,QForw=>OUT_Forw
          ,QBackw=>OUT_Backw
          ,State=>State_V1);
//incomplete formal call
Valve1    (Forw:=Command_Forw
          ,Backw:=Command_Backw
          ,State:=State_V1
          ,State=>State_V1);
```

With a complete formal call, all parameters with their assignments are programmed.

If the call contains only some parameters and their assignments, it is referred to as an “incomplete formal call”. This form is the other type of FB call permitted accord-

ing to IEC. Incomplete formal calls are often used for blocks with a large number of parameters. This call variant is especially used in cases where individual parameters have no relevance in certain operating modes of the block. As a consequence, a separate call has to be programmed for each operating mode, resulting in more complicated program sequences.

Since the instance of the FB provides a memory and the INPUT parameters always form part of the instance data and have a default value, the FB can also function without any parameter transfer at all. The user of the FB does not therefore need to define any parameter transfers at the call interface. To ensure that the block functions can be executed, it is still of course necessary to program its parameter assignments. Parameters can be assigned at a different location (e.g. before the call). By specifying *<Instance name>.<Variable name>* and assignment, parameters can also be transferred (Valve2.Forw:=Command_Forw). Read access to the outputs then logically takes place after the call. However, this solution also makes the program more difficult to read.

The call option "non-formal call" is also available for functions and methods. In this instance, parameter names with assignment are not written in parentheses, but an argument list is specified instead. This list contains values or unassigned variables that are evaluated in the sequence in which they are declared.

```
myfunc(Command_Forw, Command_Backw, 0, 0, State_V1)
```

This call method is used only in rare cases and is generally only meaningful for functions with one parameter or for certain standard functions. The list becomes far more difficult to read for blocks with a large number of parameters, and the call ceases to function altogether if the call interface is changed.

It is possible to deduce the following conclusions on the basis of the possible block call options:

- Anyone can access (even unintentionally) the public data of an instance. When a complete block call is used, errors caused by unintentional modification of parameter settings at a different code position can be prevented.
- In the case of incomplete block calls, the functional relationship is not easily identifiable at one clear point in the program. Data transfer and call are partially, or in some cases, completely separate.

The programmer must implement links to other software components using a data area (e.g. array status list) that is located outside the FB (Figure 14). This is the only method by which data can be exchanged between different software components. All participating software components read and write in this data area. These data are transferred to an IN_OUT parameter by means of a reference and are necessarily public. The FB processes and changes the values (e.g. State_V1). But the data are still completely separate from the FB. When the program and data are completely separate, however, there is a risk that data will be changed unintentionally, resulting in errors that are extremely difficult to find.

The complexity of programming required when the function block and data are separate leads to programs that are not easy to read.

Since all functions are activated via the call interface of the FB, it is not easy to achieve an FB implementation that is protected against parameterization errors and comprehensive testing will therefore be required.

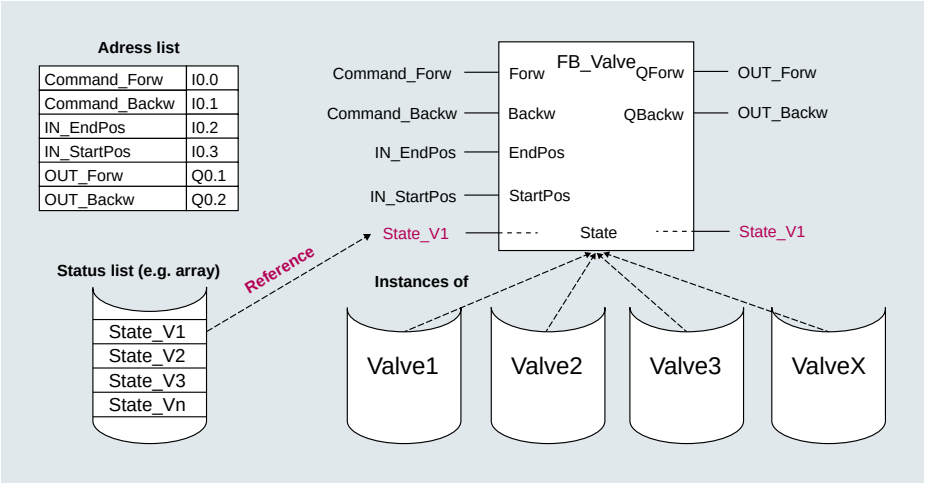


Figure 14 Program and data are separate in function blocks

We need to state quite clearly here, however, that this method of programming is completely normal today and is not in any sense wrong. The systems function exactly as prescribed by the IEC standard ED 2. As the IEC is developed further into ED 3, however, it will become possible to design better, more secure programs.

3.2.3 Advances in the life cycle of software

Since all software is subject to continuous changes over its life cycle, the same is also true of function blocks. Technical modifications to the mechanical engineering design generally necessitate software modifications as well. In this case, the usual procedure is to copy the existing FB and adapt the copy (Figure 15). This "copy and adapt" solution is a continuous process, resulting in different function block versions with modified functions.

The problem with this approach is that it can be difficult to maintain a clear overview because various derivations of software solutions are created, all of which need to be managed.

When machines in the field are controlled by a particular variant of the FB, it is unlikely that the FB will need to be replaced while the machines are in operation on condition that they operate reliably with the FB programmed for them. When the machines themselves are modernized, the software developer needs to take care when deciding which base software will be used for the new equipment. Any measures taken to debug or improve the function block must be properly documented and all the relevant persons (commissioning and service engineers) must be notified. The software can otherwise proliferate chaotically. Debugging may need to be repeated several times when older blocks are used because errors that have already been corrected will reemerge.

A large number of functions of mechanical systems are implemented by various function blocks. In other words, the problem doesn't just exist once but in multiple

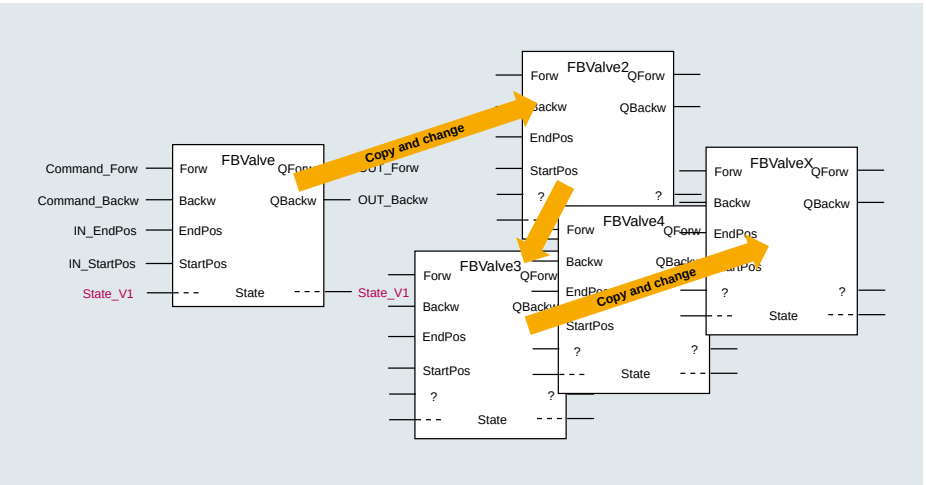


Figure 15 Function blocks need to be copied and adapted

locations. An effective solution can be achieved only by disciplined working, carefully planned software development and an efficient system of release management.

With object-oriented programming, the corresponding process for adapting and upgrading software is simpler. OOP mechanisms are designed to support the adaptation and modification of software so that it is no longer necessary to make random copies of software in the way described above.

With object-oriented programming, careful planning of the software development process is of course essential. The best approach is to analyze the software and then convert the more complex software components to object-oriented programming. Simpler programs or those that are essentially based on pure combinational logic are not generally converted. OOP should only be used where it brings real benefits. It is highly likely that a large number of programs still in use for PC applications were originally created by the procedural programming method.

3.2.4 Disadvantages of programming without OOP extensions

Conventional programming methods have several disadvantages that can be overcome by changing to object-oriented programming. But even with OOP, it is possible to create poor program code if the software design process is unstructured. The objective is not to create poor program code, but to use OOP to improve the software.

- The “modular programming” approach makes it easier to modularize software, but it is not completely satisfactory when it comes to further developing the software.
- Modularization within an FB is not supported. An FB cannot be broken down further into independent software sections because the requisite structuring mechanisms do not exist.
- As function blocks grow in size, it follows that development of the software will become an increasingly complex and difficult process.

- Reusability of the software can be guaranteed only if all of the programmers involved adhere to strict rules.
- If various software areas (e.g. communication or HMI link) need to be coupled with machine functions, the software may need to be modified, resulting in delays in the work schedules of the various development departments involved in the process. Independent development of the software then becomes impossible.
- Procedural programming is not a method that gives adequate support to the software design process, particularly when it comes to the definition and use of software interfaces.
- Data and programs are separate, unrelated entities and therefore carry a high error risk.
- The degree of abstraction when developing software is limited with the procedural programming method.
- It is virtually impossible to test software modules without adjusting the program code.

With the programming of methods in function blocks, a first step towards object-oriented programming has been made. It allows better structuring and modularization of FBs. By controlling access rights to the FB data, generally accessible data no longer need to be explicitly moved outside the function block. The fact that local types or constants can be assigned different access rights is a further advantage that significantly improves the software design. Finally, the use of methods affords, on the one hand, a more clearly organized FB design because the code is divided into methods. On the other hand, the call interface of the FB can be simplified because it is no longer necessary to map the entire scope of functions at this interface. Special methods for placing commands, for example, need to be developed for this purpose.

The extended scope of programming options for function blocks is explained in the following chapter.

3.2.5 Extensions to FBs and their access specification

The option of specifying the access to variables and methods has been developed for function blocks. Even the declarations for types and constants that are permitted in FBs in the SIMOTION system can be assigned an access specification. These modifications make it easier to handle variables or methods. By contrast with the old solution, the programmer can now decide which variables/methods may be accessed from outside the function block. Various keywords have been defined for this purpose:

PRIVATE

This keyword identifies constants, data types, variables or methods which may be called only from inside the FB. PRIVATE is the default key and can be omitted. In other words, all methods with METHOD *<MethodName>* in the method definition are automatically PRIVATE.

PUBLIC

A method to which this access identifier is assigned can be called from anywhere. Variables with the PUBLIC identifier also permit access from outside the function block. If, for example, the variables VAR1 and VAR2 are identified by the keyword

PUBLIC in a VAR block, it is possible to gain read and write access to variable VAR1 using <*InstanceName*>.VAR1 (VAR2). Data types and constants identified by the keyword PUBLIC can also be utilized outside the FB. The data type TYPE1 with <fb_name>.TYPE1 can be used, for example, to declare variables outside the FB.

It is permissible to include multiple VAR blocks with different access identifiers in one FB. In this way, the programmer can simply assign different access rights to variables. The use of FBs with methods thus increases the security of programs while also enhancing the scope for structuring FBs so that programs are easier to maintain.

The following declaration subsections can be used in the definition of an FB:

- TYPE
- VAR
- VAR CONSTANT
- VAR_INPUT
- VAR_IN_OUT
- VAR_OUTPUT
- VAR_TEMP

None of the input and output variables such as VAR_INPUT, VAR_IN_OUT, VAR_OUTPUT or VAR_TEMP may be declared as PRIVATE or PUBLIC (as this would obviously be senseless).

As in the past, each FB has its own scope, i.e. the names of variables, data types and methods within the scope must be unique.

Methods can be integrated in an FB in order to enhance the software structure. Each method has its own scope. There must be no overlap between names within the scope. Overlap between method names and variables is also not allowed because it is the method name by which a potentially defined return value is assigned. The following keywords are used to program methods in FBs.

METHOD <*MethodName*>

This is the keyword which denotes the start of a method. It can be a PUBLIC or PRIVATE method. In this case, a method can (like the FB) have an interface with variables. If a method is declared as PRIVATE (default), all variables will also be PRIVATE including the interface variables (by contrast with the FB). In other words, the variables of a method are not allowed to have a separate access identifier. The following variables may be programmed for methods:

- VAR or VAR_TEMP
- VAR_INPUT
- VAR_IN_OUT
- VAR_OUTPUT
- The method ends with keyword END_METHOD.

THIS.<*MethodName*>() is the method call within the FB. Methods may be called within the body of the FB as well as in methods of the FB. As an additional option to those specified in the IEC standard, it is permissible to use the alternative call with <*MethodName*>(). This variant is generally used to call PRIVATE methods. If a variable of a method hides a variable declared in the FB, the access conflict can be

resolved easily by specifying <*FBName*>.<*VariableName*> in the method. This is an extension of the IEC standard.

3.2.6 Use of methods to improve program structuring

The following example of a function block with methods is based on the scope of functions required to control the 4/3-way valve of a valve-cylinder combination. The example has been intentionally kept simple. The methods in this case have been selected to provide an internal structure for the FB, but further methods could be added to extend its functionality.

For the sake of clarity, the variable names at the inputs and outputs of the FB have been labeled with meaningful names. As a general rule, these names are defined differently in actual programming.

The FB contains the methods mForw() and mBackw() which are responsible for controlling the cylinder (Figure 16). These methods are then called using THIS.<MethodName()> in the program body of the FB. By contrast with classes, however, this method call is static rather than dynamic. The method can also be called via the method name alone (i.e. without the additional THIS keyword).

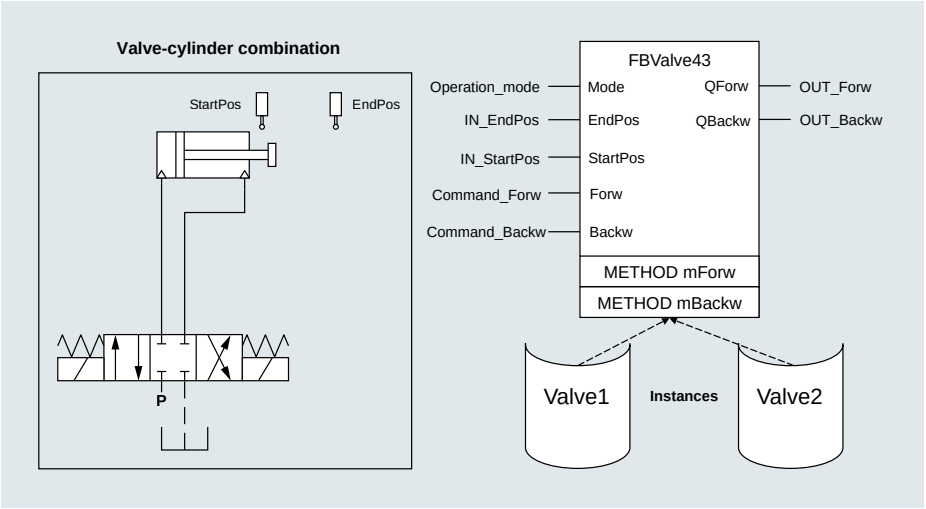


Figure 16 Programming FB Valve43 with methods

According to the IEC standard, special additional restrictions apply to the variables at the interface of the block (VAR_INPUT, VAR_OUTPUT, VAR_IN_OUT (not used in this example)). This therefore means, for example, that variables from VAR_OUTPUT may not be utilized outside the FB implementation. The keyword PRIVATE or PUBLIC may not be assigned to any of these declaration subsections of the FB.

No keyword has been specified in the VAR block. The default keyword is PRIVATE which means that these variables are not accessible from outside the FB.

3.2.6.1 Example of FB with methods

```
FUNCTION_BLOCK FBValve43
    VAR_INPUT
        Mode         : BOOL;
        EndPos       : BOOL;
        StartPos     : BOOL;
        Forw         : BOOL;
        Backw        : BOOL;
    END_VAR

    VAR_OUTPUT
        QForw        : BOOL;
        QBackw       : BOOL;
    END_VAR

    VAR
        boMode           : BOOL;
        boEndPos         : BOOL;
        boStartPos       : BOOL;
        boForward        : BOOL;
        boBackward       : BOOL;
        boMoveForward    : BOOL;
        boMoveBackward   : BOOL;
    END_VAR

    METHOD mForw // Method move forward
        IF NOT boMode THEN // Jog mode
            IF boForward AND NOT boBackward THEN
                boMoveForward    := TRUE;
                boMoveBackward   := FALSE;
            ELSE
                boMoveForward    := FALSE;
            END_IF;
        ELSE    // Automatic mode
            IF (boForward OR boEndPos) AND NOT boBackward THEN
                boMoveForward    := TRUE;
                boMoveBackward   := FALSE;
            END_IF;
        END_IF;
    END_METHOD

    METHOD mBackw // Method move backward
        IF NOT boMode THEN // Jog mode
            IF boBackward AND NOT boForward THEN
                boMoveForward    := FALSE;
                boMoveBackward   := TRUE;
            ELSE
                boMoveBackward   := FALSE;
            END_IF;
        ELSE // Automatic mode
            IF (boBackward OR boStartPos) AND NOT boForward THEN
                boMoveForward    := FALSE;
                boMoveBackward   := TRUE;
            END_IF;
        END_IF;
    END_METHOD

    boMode           := Mode;
    boEndPos         := EndPos;
    boStartPos       := StartPos;
```

```
    boForward       := Forw;
    boBackward      := Backw;
    THIS.mForw(); // Internal call mForward (static)
    THIS.mBackw(); // Internal call mBackward (static)
    QForw     := boMoveForward;
    QBackw    := boMoveBackward;
END_FUNCTION_BLOCK
```

This FB has an input/output interface and 2 methods, i.e. mForw() and mBackw(). The methods are called in the body of the function block by means of the static call for a method with THIS. The call may be alternatively programmed without THIS (Forw(); or Backw()) in SIMOTION programs. It is important to ensure that methods are programmed before the function block body. By defining instances, it is now possible to reuse the function block.

There are no further options for function blocks in SIMOTION other than the functions described above. There is no possibility of FB derivation or method overriding. These extended mechanisms of OOP are available to SIMOTION users in the form of classes with derivation and method overriding.

3.2.6.2 Example of a function block call

```
PROGRAM pValve
    VAR
        Valve1       : FBValve43;
        Valve2       : FBValve43;
        iboEAMode    : BOOL;
        iboComFor1   : BOOL; // from here are these normally I/Os
        iboComFor2   : BOOL;
        iboComBack1  : BOOL;
        iboComBack2  : BOOL;
        iboEndPos1   : BOOL;
        iboEndPos2   : BOOL;
        iboStartPos1 : BOOL;
        iboStartPos2 : BOOL;
        qboOutFor1   : BOOL;
        qboOutFor2   : BOOL;
        qboOutBack1  : BOOL;
        qboOutBack2  : BOOL;
    END_VAR

    // first call FBValve43 (Valve1) complete call
    Valve1(Mode       := iboEAMode
          ,EndPos     := iboEndPos1
          ,StartPos   := iboStartPos1
          ,Forw       := iboComFor1
          ,Backw      := iboComBack1
          ,QForw      => qboOutFor1
          ,QBackw     => qboOutBack1
          );

    // second call FBValve43 (Valve2) incomplete call
    Valve2.Mode     := iboEAMode;
    Valve2.EndPos   := iboEndPos2;
    Valve2.StartPos := iboStartPos2;
    Valve2.Forw     := iboComFor2;
    Valve2.Backw    := iboComBack2;
```

```
    Valve2();

    qboOutFor2  := Valve2.QForw;
    qboOutBack2 := Valve2.QBackw;
END_PROGRAM
```

Note: To ensure that this example is executable, the program must be assigned to the BackgroundTask in the execution system of the SIMOTION CPU. For instructions on how to assign programs in the execution system, see chapter 8.9.13.

With respect to the FB call, a function block programmed with methods in order to improve the structure of the FB body is not used in any different way to a function block programmed by the conventional method. Since the methods and variables can be declared as PUBLIC or PRIVATE, however, this kind of programming ensures more secure programs because it is easier to control access to the blocks.

In cases where PRIVATE variables need to be accessed from outside the function block (for reading or setting), the programmer can implement various methods known as Setter or Getter methods. For example, a PUBLIC method SetPrivValue(Value) (set variable) or GetPrivValue(Value) (read variable) can be programmed in the FB. These methods can be called from anywhere. A specific value can be transferred with the parameter (Value). Whether the transfer value "Value" is actually used to set the PRIVATE variable depends on the admissibility check implemented in the method program. Thus, it is easy to program a check for admissible values in Set methods of this kind. This further increases the security of programs.

3.2.7 Function block with methods for placing commands

In the past, function blocks have been programmed for execution in a cyclic context. In other words, the program and sequences in the function block are designed to work with precisely one cyclic FB call. All signals are transferred at the FB interface and are read and written cyclically. By using methods, however, it is possible in some respects to depart from the cyclic context and thus to influence the FB program in a different, more flexible manner. The commands at the inputs of the FB now take the form of method calls. These methods may be called asynchronously to the cyclic processing sequence.

When implementing an FB of this kind, the programmer needs to give sufficient consideration to the commands as these can obviously be issued to the FB at any time. Since command transfer is now performed with methods, the signals that previously carried out this task are no longer required at the FB interface.

It must of course be possible to transfer the binary signals for the limit switches and the outputs for controlling the valve. A method named "mExecute" is implemented in the FB for this purpose. mExecute must then be called once per valve instance during cyclic program execution.

Commands are transferred using the methods "mForw", "mBackw" and "mStop". These methods transfer the commands once and therefore need not be called cyclically.

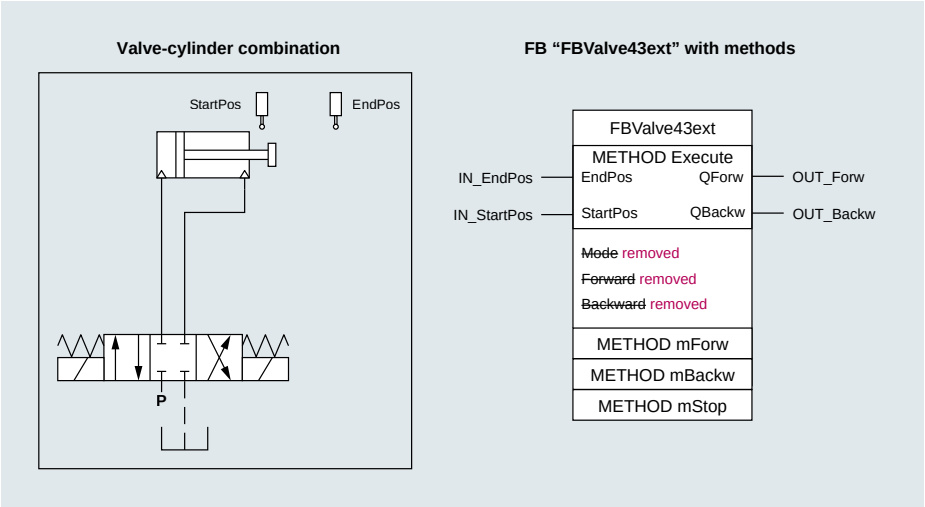


Figure 17 Further development FB Valve43 extended

Since all of the FB functions are implemented in methods in this model, the FB does not have a body. The FB itself merely forms the framework for programming methods and is responsible for supplying global variables.

Alternatively, you could of course omit the method mExecute() altogether and implement the relevant functions in the body of the FB.

The following example shows the program code for the modified function block.

3.2.7.1 Example of the FB with command methods

```
FUNCTION_BLOCK FBValve43ext

    VAR
        boEndPos         : BOOL;
        boStartPos       : BOOL;
        boForward        : BOOL;
        boBackward       : BOOL;
        boMoveForward    : BOOL;
        boMoveBackward   : BOOL;
    END_VAR

    METHOD PUBLIC mExecute : VOID // Method Execute for cyclic call
        VAR_INPUT
            EndPos       : BOOL;
            StartPos     : BOOL;
        END_VAR

        VAR_OUTPUT
            QForw        :BOOL;
            QBackw       :BOOL;
        END_VAR
```

```
        boEndPos    := EndPos;
        boStartPos  := StartPos;

        IF boForward AND NOT boEndPos THEN
            boMoveForward   := TRUE;
            boMoveBackward  := FALSE;
        ELSE
            boMoveForward   := FALSE;
            boForward       := FALSE;
        END_IF;

        IF boBackward AND NOT boStartPos THEN
            boMoveBackward  := TRUE;
            boMoveForward   := FALSE;
        ELSE
            boMoveBackward  := FALSE;
            boBackward      := FALSE;
        END_IF;

        QForw   := boMoveForward;
        QBackw  := boMoveBackward;
    END_METHOD

    METHOD PUBLIC mForw : VOID // Method command move forward
        IF NOT boForward AND NOT boEndPos THEN
            boForward := TRUE;
        END_IF;
    END_METHOD

    METHOD PUBLIC mBackw : VOID // Method command move backward
        IF NOT boBackward AND NOT boStartPos THEN
            boBackward := TRUE;
        END_IF;
    END_METHOD

    METHOD PUBLIC mStop : VOID // Method command Stop
        boBackward := FALSE;
        boForward  := FALSE;
    END_METHOD
    ;
END_FUNCTION_BLOCK
```

The FB now has a total of four methods. The method mExecute is called cyclically in the execution system and is responsible for controlling operation of the valve. It is also used to interconnect the limit switches and the output signals for the valve.

The command methods mForw(), mBackw() and mStop() have no interface parameters. They transfer commands to the mExecute() method via variables defined internally in the FB. Since a PUBLIC identifier has been assigned to the methods, they can be called from anywhere. The relevant command is passed once to the method mExecute. It is therefore also an ideal solution to call the commands from sequences.

Within the function block, all functions are now programmed in methods which means that the body of the FB does not need a program. Since every FB must contain a program, however, a blank program line (semicolon) is inserted at the end of the FB. Implementing all functions in methods is the same as programming classes, a technique that we will learn more about later on.

3.2.7.2 Example of an FB call with command methods

```
PROGRAM pValve
    VAR
        iboEndPos    : BOOL;
        iboStartPos  : BOOL;
        iboForward   : BOOL;
        iboBackward  : BOOL;
        qboForward   : BOOL;
        qboBackward  : BOOL;
        Valve1       : FBValve43ext;
    END_VAR

    IF iboForward THEN
        Valve1.mForw();
    END_IF;

    IF iboBackward THEN
        Valve1.mBackw();
    END_IF;

    IF (NOT iboForward AND NOT iboBackward) THEN
        Valve1.mStop();
    END_IF;

    Valve1.mExecute(EndPos := iboEndPos
                   ,StartPos := iboStartPos
                   ,QForw    => qboForward
                   ,QBackw   => qboBackward);
END_PROGRAM
```

Note: To ensure that this example is executable, the program must be assigned to the BackgroundTask in the execution system of the SIMOTION CPU. For instructions on how to assign programs in the execution system, see chapter 8.9.13.

An instance “Valve1” of the FB Valve43ext is created in the program pValve. The method mExecute with the combined limit switch and output signals is then called. To perform a simple command test, the call for commands mForw(), mBackw() and mStop() via signal combinations is also stored in the cyclic section. As mentioned above, methods can be called from anywhere.

With respect to the call of the cyclic section, the use of a function block with command methods differs in the fact that only the input and output variables are transferred.

We have of course kept the example FB program very simple for the purpose of illustrating the principle of command methods. The program for an actual valve control system would almost certainly be larger and more elaborate.

The two operating modes “automatic” and “manual” are not necessary for this example. It is extremely easy to make the distinction between these two operating modes outside the FB. The method mStop() is called in manual mode when a manually-operated key (iboForward, iboBackward) is not actuated. In automatic mode, the method is generally called from a sequence in which the stop command is issued via corresponding sequence steps. To ensure that this functions properly, the operating mode must also be linked into the queries of the signals iboForward and iboBack-

ward. But designing software is all about considering aspects of this kind and it is up to the programmer to decide how a program should best be implemented. It may well be meaningful to retain an operating mode evaluation function in the valve FB.

3.3 Classes (CLASS)

Classes will be introduced with SIMOTION version 4.5. A class[3] forms the blueprint for objects of the same kind with its attributes (properties/variables) and methods (modes of behavior). To use a more general formulation, a class corresponds to the data type of an object. A class thus has some resemblance with a complex data type (such as a structure, TYPE), but goes even further. The programmer defines additional algorithms (methods) in the class which work with these data.

A class is programmed in an ST source in the SIMOTION engineering system. The ST source is a user-defined container in which programs can be written using a text editor. The program for a class starts with the keyword CLASS<*ClassName*> and ends with END_CLASS.

A description of how an ST source is generated in the engineering system can be found in chapter 8.

Once a class has been fully programmed in the editor, it is also displayed in the project navigator (PNV). The project navigator is similar to Windows Explorer and displays all the elements of a SIMOTION project in a tree structure. The icon for a class is labeled with a C (Figure 18). The methods are displayed underneath the class.

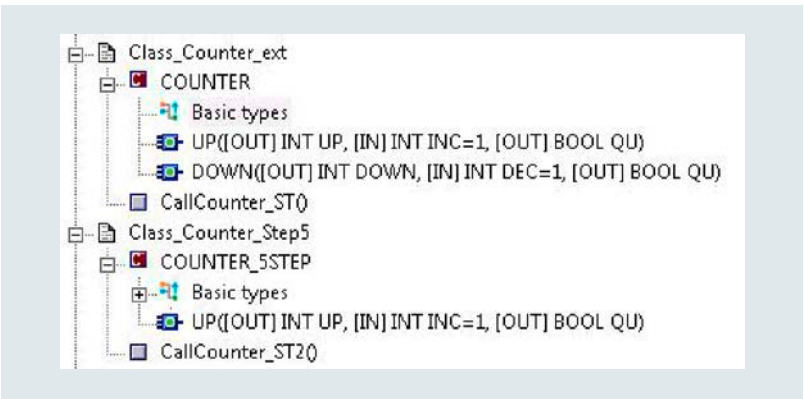


Figure 18 CLASS in the PNV

Unlike function blocks, a class may contain only variable declarations, in other words, it is not permissible to use VAR_INPUT, VAR_OUTPUT, VAR_IN_OUT or VAR_TEMP in the declaration part of a program.

[3] "Object-Oriented Programming". In: Wikipedia – The Free Encyclopedia. Revision level: January 12, 2016, 08:24 UTC. URL: https://en.wikipedia.org/wiki/Object-oriented_programming (viewed on: March 22, 2016, 17:28 UTC)

The methods belonging to the class are specified within the class. Underneath the class, the methods implemented within the class with their interfaces (signature) are displayed in the project navigator. The icons for the methods are color-coded to match the defined access identifier.

If a class is derived from another class (base class), the PNV shows the base class under “Basic types”. The methods of the base class are not displayed here. The user can jump to the base class in the PNV via the context menu in order to gain easy access to the inherited methods. The methods of the base class are then displayed.

3.3.1 Keywords supported for a class

Table 1 shows the features defined according to IEC 61131-3 ED3 Table 48 (page 120 ff.) that are supported by SIMOTION.

Table 1 Keywords for classes

IEC No.	Keyword	Description
1)	CLASS ...END_CLASS	The declaration starts with the keyword CLASS followed by the class identifier and ends with END_CLASS.
1a)	FINAL	When this keyword is used in a defined class, no further subclasses can be derived from the class. In other words, it prohibits inheritance.
2a)	VAR...END_VAR	Declaration and initialization of the variables of a class.
3a)	RETAIN	Not specified in conjunction with PUBLIC.
3b)-4b)		Omitted because NON RETAIN and VAR EXTERNAL are not supported.
	Methods and their identifiers	
5)	METHOD ... END_METHOD	The definition of a method starts with the keyword METHOD followed by the method name and ends with END_METHOD.
5a)	PUBLIC	Identifier indicating that the method can be called from anywhere (by any other class). The identifier is inserted directly after the keyword METHOD.
5b)	PRIVATE	Identifier indicating that the method can be used only within the class.
5c)	INTERNAL	Available only when supported by namespaces.
5d)	PROTECTED	Method can be used within the class and its subclasses. This identifier is the default and is effective even if no identifier has been assigned.
5e)	FINAL	Method cannot be overridden.
	Inheritance	
6)	EXTENDS	This class is the extension (derivation) of a base class. The keyword is inserted after the class name followed by the base class. For example: CLASS *derivedClass* EXTENDS *baseClass*
7)	OVERRIDE	Method overrides method of base class. The signature and access identifier must be the same.

IEC No.	Keyword	Description
8)	ABSTRACT	ABSTRACT CLASS – cannot be instantiated. ABSTRACT METHOD – method is ABSTRACT and does not contain any code. Class cannot be instantiated.
	Access reference	
9a)	THIS	Call of a method by dynamic binding within class (or a subclass) in which method is defined.
9b)	SUPER	Call of a method of the base class. Can only be used in subclasses (derived classes).
	Variable access identifiers	
10a)	PUBLIC	Variable can be used anywhere.
10b)	PRIVATE	Variable can be used only within the class in which it is defined.
10c)	INTERNAL	Available only when supported by namespaces.
10d)	PROTECTED	Variable can be used within the class in which it is defined and its subclasses (derivations). This identifier is the default and is effective even if no identifier has been assigned.
	Polymorphism	
11a)	VAR_IN_OUT	An instance of a derived class may be transferred to VAR_IN_OUT of a (base) class.
11b)	Mit Referenz	The address of an instance of a derived class may be transferred to a reference to a (base) class (see chapter 6.3.2).

Methods or variables to which an access identifier has not been assigned have PROTECTED status per default.

This kind of method is called within an implemented class via dynamic binding and the keyword THIS. The term “dynamic binding” refers to the resolution of the actual method call during runtime. The opposite of dynamic binding is “static binding”. With static binding, the method call is clearly defined at the time the program is compiled.

The keyword SUPER is used to call a method defined in the context of the base class (one level higher) from a derived class.

The access identifier (e.g. PUBLIC, etc.) for variable declarations of the class is specified at the relevant VAR declaration block of the variables (VAR ...) and not individually for each variable. This means that the VAR CONSTANT block within CLASS can also be used to define private or public constants.

It is also permissible to use the TYPE declaration block within CLASS to define private/public data types (with access identifier if necessary). Thus, it is possible to define class-specific data types (structures) within the class.

If a method is to override a method implemented in the base class, it must be assigned the OVERRIDE identifier. Methods identified with the keyword FINAL may

not be overridden. A compiler error message will be issued if an attempt is made to override a FINAL method. If the identifier FINAL is assigned to a class, it is not permissible to derive any subclasses from this class.

Classes identified with the keyword ABSTRACT cannot be instantiated. They merely serve as a base class for deriving subclasses. Abstract methods may not contain any program code and are only formulated in a derived class. The essential purpose of abstract classes and methods is thus to assist the software designer with the definition of classes.

3.3.1.1 Example of a CLASS declaration

```
CLASS ABSTRACT name

    VAR (*vars*); END_VAR

    METHOD PUBLIC name_1    // Method anywhere usable
        VAR_INPUT (*inputs*); END_VAR
        VAR_OUTPUT (*outputs*); END_VAR
        ;
    END_METHOD

    METHOD FINAL name_i     // Method is final; OVERRIDE not possible
        VAR_INPUT (*inputs*); END_VAR
        VAR_OUTPUT (*outputs*); END_VAR
        ;
    END_METHOD

END_CLASS
```

Note: This example is only a condensed presentation designed to improve your understanding of class definition. It is not an executable program.

3.3.2 Methods (METHOD)

The executable program code of a class is implemented in methods by the programmer. A method roughly corresponds to a function and includes a declaration part and the executable program body. The definition of a method starts with the keyword METHOD <*MethodName*> and ends with END_METHOD.

Methods resemble normal functions in terms of their behavior. In other words, the method receives its data when it is called via the specified interface (VAR_INPUT, VAR_IN_OUT) and writes its results to the output interface (VAR_OUTPUT, return value). The data of the method are stored in the CPU stack and are deleted on exit (the same as with functions).

Variables of the type VAR_INPUT, VAR_OUTPUT, VAR_IN_OUT and VAR (VAR_TEMP) can be specified in the declaration part of a method. The method icon with interface information does not become visible in the PNV until the complete method interface (signature) has been entered. In this case, all the code for the method (END_METHOD) as well as the class (END_CLASS) must be entered or else the tree cannot show the information.

3.3.3 Methods and their access specification

The user can define for each method in a class the program or program area from which the method can be called. To do this, the programmer uses special keywords that are typed before the name of the method definition (METHOD *<KEYWORD>* *<MethodName>*). The following specification keywords are available:

PROTECTED

- The keyword PROTECTED can be used if an inheritance has been implemented. It indicates that the method can be used only within the class in which it is defined and all subclasses derived from this class. PROTECTED is the default key and may be omitted. In other words, all methods with METHOD *<MethodName>* in the method definition are automatically PROTECTED.

  Note: If no inheritance has been implemented, this access definition has the same meaning as PRIVATE. It must be noted, however, that the PROTECTED method (in base classes without derived classes) is called with dynamic binding.

INTERNAL

- If namespaces have been implemented, the keyword INTERNAL can be used (supported in versions later than V4.5). It indicates that the method can be called within the namespace in which the class has been declared. The namespace thus defines that access area. If namespace A comprises program X and classes B and C, for example, then INTERNAL methods can be called in program X and in classes B and C.

PUBLIC

- This access identifier permits the method to be called from anywhere that the relevant class is used. The place in which a class is used are the program sections in which the instance of the class can be accessed.

PRIVATE

- The identifier PRIVATE indicates that the method can be called only within the class in which it is defined. By contrast with a PROTECTED method, a method identified as PRIVATE may not be accessed from a derived class.

Figure 19 shows an example for access to methods defined in class C. Here follows the explanation:

a) Access definition: PUBLIC, PRIVATE, INTERNAL, PROTECTED
   - **PUBLIC** M1 can be accessed from anywhere with the method call M1
   - **PRIVATE** M2 can be accessed with the method call M2 only within class C
   - **INTERNAL** M3 can be accessed with method call M3 within NAMESPACE A (as well as classes B and C)
   - **PROTECTED** M4 can be accessed with method call M4 within class C_derived (as well as class C)

b) Method calls inside/outside classes:
   - M2 can be accessed within class C with **THIS**.
   - M1, M3 and M4 are called outside class C – using keyword **SUPER** for M4.

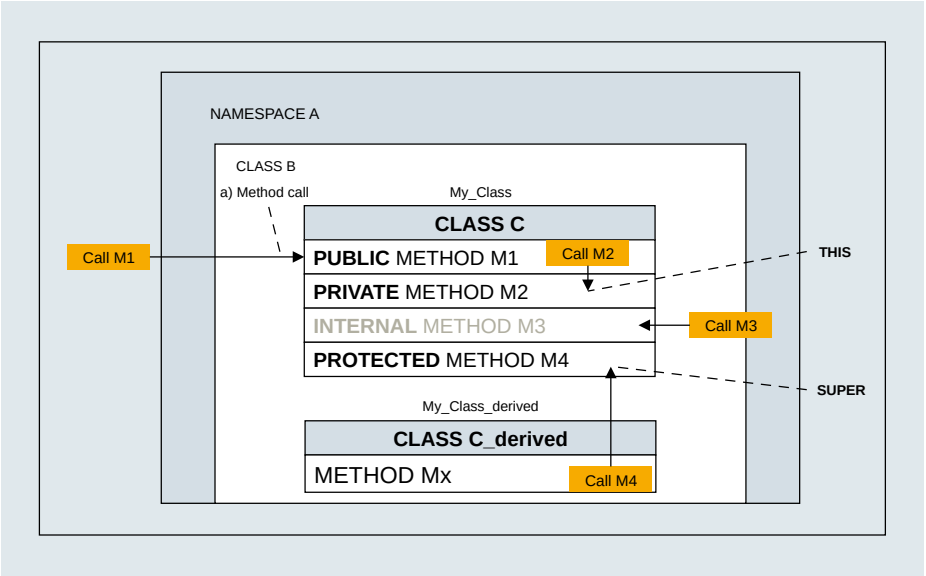


Figure 19 Access definition for methods (source: IEC 61131-3 ED3)

3.3.4 Declaration of instances of a class

The instances of classes are declared by the same method as structured variables. When they are specified as a data type, the class name is typed after the colon. When the object is created, the variables are initialized according to the initialization data specified in the class.

If the variables are to be initialized with different values for a specific object, the programmer can specify the initialization values via the expressions in parentheses. The IEC standard states that this is only permitted for variables declared as PUBLIC.

Table 2 shows the features defined according to IEC 61131-3 ED3 Table 49 (page 123) that are supported by SIMOTION.

Table 2 Declaration of instances of a class

IEC No.	Description	Example

1	Declaration of instances of a class with default initialization	`VAR`
		`  MyCounter1:Counter;`
		`END_VAR`

2	Declaration of instances of a class with initialization of values	`VAR`
		`  MyCounter2:Counter:=(MAX_Val:=1000,`
		`  MIN_Val:=-1);`
		`END_VAR`

Since PUBLIC variables can be accessed from anywhere, the value of the variables can be changed from outside the class at any time. For example, the programmer can easily change the value in the program by specifying “MyCounter1.Max_Val:=2000;”.

If the variables are declared as PROTECTED or PRIVATE, the programmer must provide a setter method in order to change a value.

In the interests of simplicity, the programmer may also permit initialization for PROTECTED and PRIVATE variables with SIMOTION (this is an extension to the IEC standard). By specifying the keyword OVERRIDE at the variable declaration of the class, the programmer also enables initialization of PROTECTED/PRIVATE variables at the instance declaration. In other words, the example in Table 2 in which values may also be initialized for PROTECTED/PRIVATE variables will work if the programmer has declared this as permissible in the class (see chapter 7.3.1). The definitions in the access specification apply to all other access operations.

3.3.5 Rules for identifiers in a class

Each class forms its own scope. Names must be uniquely defined within this scope. For this reason, the names of methods, variables and data types defined within a class must not be duplicated. This rule also applies to derived classes. The only exceptions to this rule are variables and methods identified as PRIVATE in the base class because these are not visible in the derived class. Furthermore, the name of the class and the base classes may not be used as the name of a variable or method.

The IEC standard stipulates that parameter names declared at methods may not hide the names of variables or methods in the class itself.

In order to avoid various problems, SIMOTION permits (in deviation from the IEC standard) an overlap between the identifiers of instance data/method names and method parameters. Each method is given its own scope that is subordinate to the scope of the class.

If an overlap between an identifier of parameter names and identifiers in the scope of the class itself is detected in the program code for a method, the compiler outputs a warning message. Moreover, to solve conflicts of this kind, it is permissible to use the keyword THIS.<*MethodName*> to access methods within the class itself and methods that are declared (and public) in the base class. It is permissible to access variables of the class with <*ClassName*>.<*VariableName*> in addition to the simple specification <*VariableName*>. (This is an extension of or modification to the IEC standard, see chapter 7.2)

3.3.6 Use of class methods

Methods are used in a program in the same way as functions. The rules defined for functions (formal call or non-formal call) with respect to parameter transfer also apply to methods.

The following variants of method call in textual notation are available:

<*InstanceName*>.<*MethodName*>();	External call with specification of the instance (static binding) or reference transferred via VAR_IN_OUT (dynamic binding)
THIS.<*MethodName*>();	Call of a method within class in which method is defined; internal call (dynamic binding)
SUPER.<*MethodName*>();	Explicit call of a method of the base class (static connection)

As an extension to the IEC standard, the following additional call variants are also permissible:

<MethodName>();	Call of a method within class in which method is defined; Internal call (static binding)
<ClassName>.<MethodName>();	Call of a method from class in which it is defined, or a method of a base class (static binding)

Method calls in graphical programming languages are used in conjunction with the dialog boxes for function block calls that are already familiar to programmers. The instance name of the class and the *<ClassName>.<MethodeName>* must be specified in this case. The interfaces suitably formulated for assigning parameters are thus represented.

With instance calls, the instance name of the class is typed in the position in which the instance name of the function block is specified today. The call box for graphic programming languages contains the class name and method name (separated by a full stop) instead of the FB name. Since it is not possible to declare classes in graphical programming languages with SIMOTION, only an external call with specification of the instance name can be programmed here.

In order to explain the principles of a class, a simple programming example is given below. This example is based on information from IEC 61131-3, but has been modified in order to demonstrate various mechanisms. The example is moreover limited to only those program sections that are absolutely necessary. Checks for correct parameter transfer, etc. have been omitted in order to more clearly demonstrate the principle.

A class “COUNTER” with a method “UP” (for count up) has been selected as the example. In other words, the class has only one method UP() which counts up a counter value (CV). You would not of course normally need to implement a class for a counter, but our sole intention here is to illustrate certain principles of object-oriented programming.

3.3.6.1 Example of a CLASS COUNTER

```
CLASS COUNTER
    VAR
        CV:INT; // Current value of counter
    END_VAR

    VAR OVERRIDE
        MAX_Val:INT := 100;
        MIN_Val:INT := 0;
    END_VAR

    METHOD PUBLIC UP: INT // Method for count up by inc
        VAR_INPUT
            INC:INT:=1;
        END_VAR
        VAR_OUTPUT
            QU:BOOL;
        END_VAR
        // Upper limit detection
        IF CV <= (MAX_Val - INC) THEN
```

```
                CV := CV + INC; // Count up of current value
              QU := FALSE;
          ELSE
                QU := TRUE;     // upper limit reached
          END_IF;
          UP := CV; // Result of method
      END_METHOD
  END_CLASS
```

The CLASS COUNTER has one method UP (with a defined return value INT) that can count up a value (CV) for as long as the programmed maximum value (MAX_Val) is not reached. When the maximum value is reached, the method sets the output variable QU to TRUE. This output of QU (MAX_Val reached) can be used to skip another call of the same method in the calling program.

If the method is still called, the COUNTER ceases to count and the counts are lost. Although this is one of the weaknesses of this example, our aim here is to explain certain principles using the smallest possible programs.

The following example illustrates use of the method.

3.3.6.2 Use of the method of CLASS COUNTER

```
PROGRAM CallCounter_ST
    VAR
        C1:COUNTER:=(MAX_Val:=1000,MIN_VAL:=0);
        CountOut: INT;
        Locking: BOOL;
    END_VAR
    // Call COUNTER UP for increment
    IF Locking = FALSE THEN
        CountOut:=C1.UP(INC:= 1, QU=>Locking); // increment
    END_IF;
END_PROGRAM
```

Note: To ensure that this example is executable, the program must be assigned to the BackgroundTask in the execution system of the SIMOTION CPU. For instructions on how to assign programs in the execution system, see chapter 8.9.13.

The program creates instance C1 for the class COUNTER. According to the definition of the instance, initialization of variable values is permissible (MAX_Val:=1000, MIN_VAL:=0). The IEC standard states that this is only permitted for PUBLIC variables. With SIMOTION, however, the values of PROTECTED variables can also be initialized if this is enabled by the programmer at the variable block with keyword OVERRIDE (see chapter 3.3.6.1). To ensure that this applies to only selected variables, the programmer can define multiple variable blocks in the class.

The current count value is transferred in the method's return value to variable "CountOut". When the maximum value is reached, the Boolean variable "Locking" is set to TRUE and further calls of the method are skipped in the program. If the method UP() were to be called again, the counter would cease to count.

In order to describe the use of extended mechanisms, we are now going to add another method for counting down (decrementing) to the class COUNTER.

3.3.6.3 Extension of the CLASS COUNTER and use of THIS

```
CLASS COUNTER
    VAR
        CV  :INT; // Current value of counter
    END_VAR
    VAR OVERRIDE
        MAX_Val:INT := 100;
        MIN_Val:INT := 0;
    END_VAR

    METHOD PUBLIC UP: INT // Method for count up by inc
        VAR_INPUT
            INC:INT:=1;
        END_VAR
        VAR_OUTPUT
            QU:BOOL;
        END_VAR
        // Upper limit detection
        IF CV <= (MAX_Val - INC) THEN
            CV := CV + INC; // Count up of current value
            QU := FALSE;
        ELSE
            QU := TRUE;     // upper limit reached
        END_IF;
        UP := CV; // Result of method
    END_METHOD

    METHOD PUBLIC DOWN : INT
        VAR_INPUT
            DEC:INT:=1;  // decrement
        END_VAR

        IF CV > MIN_Val THEN
            DOWN := THIS.UP(-DEC); // Internal method call
        END_IF;
    END_METHOD
END_CLASS
```

The method DOWN has now been added to the base class COUNTER. This method DOWN() uses the internal method call THIS and the method UP() to count down (decrement). In this case, however, the method UP has been preset to the increment value “-1”. In other words, the method UP is counting up with -1, i.e. it is counting down. The method DOWN can now be used in a program in exactly the same way as the method UP.

The methods UP() and DOWN() are PUBLIC methods and can be called from anywhere. When UP is called followed by DOWN in a program run, the relevant count value is first incremented and then decremented. This behavior must be taken into account in the calling program so as to ensure that the counter functions as required.

In other words, the subsequent method calls in PROGRAM are programmed such that only UP() or DOWN() is called in each case.

3.3.6.4 Use of the methods UP and DOWN

```
PROGRAM CallCounter_ST
    VAR
        C1:COUNTER:=(MAX_Val:=1000,MIN_Val:=0);
        CountOut: INT;
        Locking: BOOL;
        UpValreached: BOOL;
    END_VAR
    // Call COUNTER for increment/decrement
    IF UpValreached = TRUE THEN Locking := TRUE; END_IF;
    IF Locking = FALSE THEN
        CountOut:=C1.UP(QU=>UpValreached); // increment
    END_IF;
    IF Locking = TRUE THEN
        CountOut:=C1.DOWN(); // decrement
    END_IF;
    IF CountOut <= 0 THEN
        Locking := FALSE;
        UpValreached := FALSE;
    END_IF;
END_PROGRAM
```

Note: To ensure that this example is executable, the program must be assigned to the BackgroundTask in the execution system of the SIMOTION CPU. For instructions on how to assign programs in the execution system, see chapter 8.9.13.

In this program, the count value is counted up until the maximum value is reached. The new method DOWN is then called and this counts the value down until it has reached zero. When the count value has reached zero, it is counted up again. The switchover between counting up and counting down is controlled by the variable “Locking”.

The calls of methods UP() and DOWN() utilize the specified default of the values at the INPUT variables of the methods. Transferring values to these variables would change the increment of the counter. If the values specified were not divisible by the MAX_Val, the counter might exceed the maximum value or drop below the minimum value. This is another weakness of this example. But we need this potential change in the transferred value in order to progress to the next stage of the explanation.

The examples we have used until now have only demonstrated use of a class with methods that could be implemented just as well as a function block. The fact that data and methods defined in classes can be inherited constitutes the major difference between classes and function blocks. The principle of inheritance – an essential building block of object-oriented programming – is explained in the following section. We will then show you an example of the class COUNTER extended by these mechanisms.

3.3.7 Classes and inheritance

In object-oriented programming, it is possible to derive subclasses from an existing class. When a subclass is derived, it inherits all the properties of the original class. This means that all the variables and methods defined in the original (base) class

are available in the derived class although the source code is not visible in the derived class. The new class created by this method can be used in turn to derive further classes.

From base class X with methods a and b, we can derive class X1 (Figure 20). The keyword for the derive process is EXTENDS and is specified in the definition of the new class X1: CLASS X1 EXTENDS X. Method b from the base class is overridden in class X1, i.e. it is implemented differently. Method c is also made available. Using this new class X1 as a base, we can derive further subclasses such as X11 and X12.

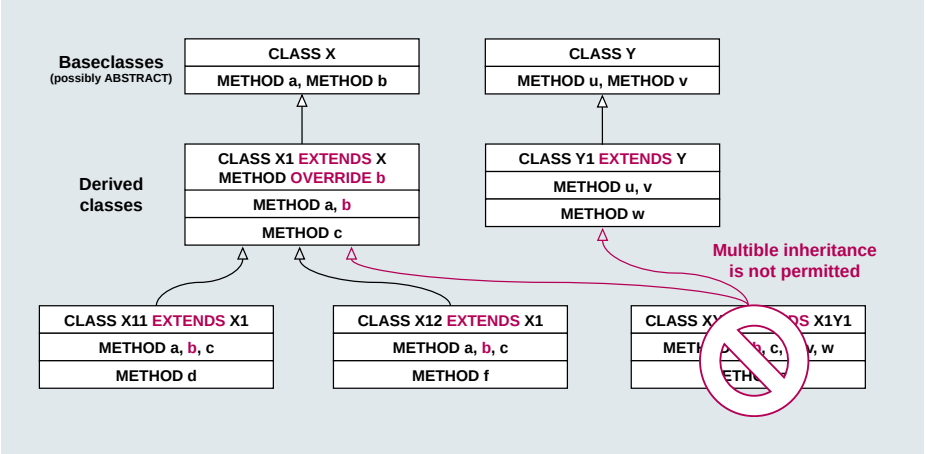


Figure 20 Classes and their derivations

IEC 61131-3 ED3 does not make any provision for multiple inheritance.

A derived class inherits all the properties (variables and methods) of the original class. To derive a class only makes sense, however, when it is necessary to adapt the inherited programs to fulfill different requirements. This is why provision is made for overriding the methods from the base class in the new derived class in order to program them with new functions. The overriding of methods must be programmed by keywords in the new class. For example: `METHOD PUBLIC OVERRIDE b`.

By this means it is possible to integrate an adapted code for method b in the new class.

As regards overriding, it is important to note that while the method can be implemented with new functionality, the signature (interface definition and name) may not be changed. Programmers can also use their own temporary variables (VAR, VAR_TEMP) that are defined differently to the variables in the method of the base class.

To illustrate how this all works, we have again used our class COUNTER in the following examples.

3.3.7.1 Example of derivation of a class

```
CLASS COUNTER_5STEP EXTENDS COUNTER
    METHOD PUBLIC OVERRIDE UP: INT // Method override
        VAR_INPUT
            INC:INT:=1;
        END_VAR
        VAR_OUTPUT
            QU:BOOL;
        END_VAR
        UP:=SUPER.UP(INC:=INC*5,QU=>QU);
    END_METHOD
END_CLASS
```

In this example we have created a class COUNTER_5STEP that we have derived from the class COUNTER. This derived class inherits all the variables and methods from the base class (COUNTER). The derive operation is implemented with the statement: CLASS COUNTER_5STEP EXTENDS COUNTER. Figure 21 shows the structure of this example.

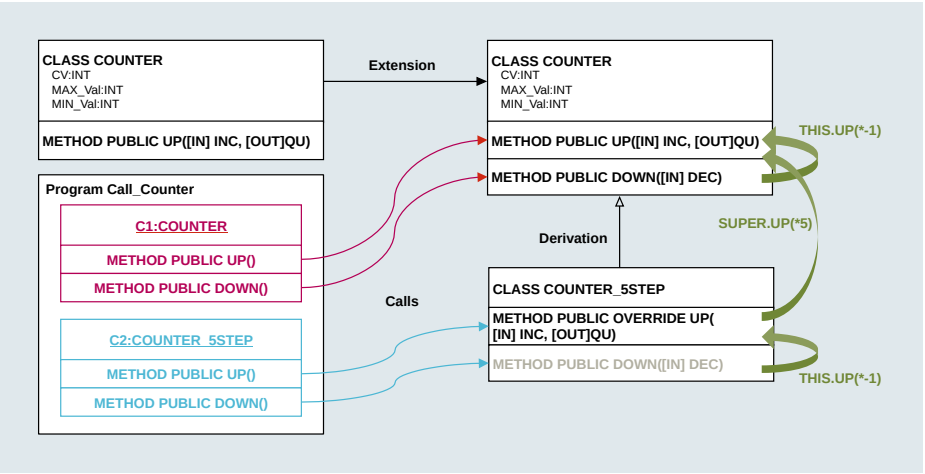


Figure 21 Derivation and counter call principle

We now need to alter the functionality of the new class to that the counter can count up and down in increments of five. In order to do this, we need to override the method UP() inherited from the base class. The program code for method UP is therefore changed and programmed with the new functionality “Count in increments of five”.

Method UP() is overridden with the statement: METHOD PUBLIC OVERRIDE UP. Since we are not permitted to change the method signature, the name, the return value and the definition for the INPUT and OUTPUT parameters must remain the same as they are in the base class.

To be able to count in increments of five, the method UP() in the derived class calls the method UP() in the base class using keyword SUPER, but also transfers the

increment value (INC) multiplied by 5. This causes the method UP in the base class to count in increments of five.

In response to the method call with SUPER, the base class passes the output variable QU to the output variable QU of the derived class. Just to clarify once again: these two variables QU are variables in different scopes that must have the same name because they have the same signature. In other words, the programming of QU prior to parameter assignment belongs to the scope of the base class method, while its programming thereafter belongs to the scope of the current method.

Using this simple override mechanism, it is easy to alter the functionality of the base class. Method UP() of COUNTER_5STEP counts in increments of five. Now you will certainly ask yourself: What happens to the count down function?

When we derived class COUNTER_5STEP, it also inherited the method DOWN() from the base class. The program code for this method has not been changed and is therefore visible only in the base class, but it is still operative in the derived class. In other words, COUNTER_5STEP has, like the base class, a method UP() and an inherited method DOWN().

This means that the method DOWN() can also be called for an object of COUNTER_5STEP. A reminder: The method UP() is called with THIS(DOWN := THIS.UP(-DEC)) in the method DOWN(). This is a dynamic call which means that the overridden method UP() of COUNTER_5STEP is called in this case, but (as in the base class) with a negated increment value (-DEC). This negated value migrates via the SUPER call multiplied by 5 to the base class. The result is a count down operation in increments of five.

We have to admit here that this solution probably looks pretty tricky to a procedural programmer. Furthermore, the example works with the use of THIS and the derivation only because the increment can be specified at the INPUT parameters.

The transfer of values via the INPUT parameters INC and DEC is a weak point of this example because transferring a value other than “1” or “5” would lead to undesirable results. If you are asking yourself why we cannot use an example without weak points, you are quite justified to do so. The answer to the question is simple: This example uses minimum program code (one code line change) to illustrate the effect of dynamic calls and the mechanism of inheritance – and this was precisely our goal. and we therefore decided to accept the weak points of the example. As we mentioned right at the beginning, nobody would define a class just for a counter function, but would use the standard FBs of the system.

3.3.7.2 Example of how to use base and derived classes

```
PROGRAM CallCounter_ST2
   VAR
        C1:COUNTER;
        C2:COUNTER_5STEP;
        CountOut: INT;
        CountOut2: INT;
        Locking: BOOL;
        Locking2: BOOL;
        UpValreached: BOOL;
        UpValreached2: BOOL;
    END_VAR
```

```
        // Call COUNTER (C1) for increment/decrement
        IF UpValreached = TRUE THEN Locking := TRUE; END_IF;
        IF Locking = FALSE THEN
            CountOut:=C1.UP(QU=>UpValreached); // increment
        END_IF;

        IF Locking = TRUE THEN
            CountOut:=C1.DOWN(); // decrement
        END_IF;
        IF CountOut <= 0 THEN
            Locking := FALSE;
            UpValreached := FALSE;
        END_IF;
        // Call COUNTER_5STEP (C2) for increment/decrement
        IF UpValreached2 = TRUE THEN Locking2 := TRUE;
        END_IF;
        IF Locking2 = FALSE THEN
            CountOut2:=C2.UP(QU=>UpValreached2); // increment
        END_IF;
        IF Locking2 = TRUE THEN
            CountOut2:=C2.DOWN(); // decrement
        END_IF;
        IF CountOut2 <= 0 THEN
            Locking2 := FALSE;
            UpValreached2 := FALSE;
        END_IF;
    END_PROGRAM
```

Note: To ensure that this example is executable, the program must be assigned to the BackgroundTask in the execution system of the SIMOTION CPU. For instructions on how to assign programs in the execution system, see chapter 8.9.13.

The program code in this example illustrates the simultaneous use of the methods in classes COUNTER and COUNTER_5STEP. In this case, the methods UP and DOWN are being used as described above. The objects C1 and C2 continue counting up until the maximum value is reached. The counter value is then decremented again until the counter reading is 0.

When the methods UP and DOWN from class COUNTER are used, the count value is incremented and decremented by one in each case. The methods defined in class COUNTER_5STEP increment and decrement the count value by 5 in each case.

If we allow this program to execute, we can clearly observe that it is ultimately always the method UP from the base class that is applied in the program, but with different parameters. The restriction that either the UP or DOWN method from each class must be called in order to achieve full count up or count down still applies.

3.3.7.3 Other aspects of the method call

In the COUNTER_5STEP example, we have only overridden the method UP. Since this subclass inherited the DOWN method() and the call with THIS implemented within the method, it will call the overridden method UP(). In this method, the keyword SUPER has been used in turn to call the method UP() from the base class. Thanks to this chaining of calls, the method DOWN() also works in the derived class without having to be overridden.

If this indirect influence is not desired, the programmer can prevent it by changing the call for the method UP() in the base class. This is the call implemented with THIS:

```
DOWN := THIS.UP(-DEC); // method call (dynamic)
```

If the programmer replaces THIS by the name of the class, the method is called statically. In other words, with methods inherited in derived classes, it is always the method from the base class that is called. The call then looks like this:

```
DOWN := COUNTER.UP(-DEC); // method call (static)
```

By making this change to our example, the counter in the derived class COUNTER_5STEP would only count up in increments of five. But when the method DOWN() in the derived class is called, the counter still counts down in increments of one.

When implementing the base class, therefore, the programmer needs to decide how method override mechanisms will or should work. While this makes the programming process more complex and requires the programmer to understand the use of methods, it ensures an extremely high degree of flexibility in program application. It is often necessary for a programmer to precisely define whether or not an override will have an effect on other methods. We just want to point out once again that programming this change of behavior in the implementation of a counter does not, of course, make any technical sense. But what makes this example so useful is that it demonstrates very neatly the difference between dynamic and static bindings and their effects.

The IEC standard describes another option for prohibiting overrides. The keyword FINAL can be programmed for a class or method in order to protect it against being overridden.

CLASS FINAL <*ClassName*> can no longer be used as a base class from which other classes can be derived. If a method is identified accordingly (METHOD FINAL <*MethodName*>) within a class (that is not itself identified as FINAL), the class can still be used as a base class but the method cannot be overridden and the compiler will output an error message if any attempt is made to do so.

3.3.7.4 Example of base and derived classes in a function

The following example demonstrates the use of counter functionality programmed in a function. In this case, the necessary signals including the object reference to the COUNTER are transferred to the function via VAR_IN_OUT.

The switchover between counting up and counting down is thus implemented in the function for both scenarios – count in increments of one and count in increments of five.

The relevant object instance of the class (C1 and C2) is transferred with a dynamic function binding when the function is called in the program at parameter C. This must be done via VAR_IN_OUT because a reference of the class instance is transferred via this interface. Any other declaration would lead to a compilation error. With the implicit reference creation at VAR_IN_OUT, the underlying polymorphism is also clearly demonstrated here. Because an object of the base class (C1) is transferred to the variable C of the function at one location, and an object of the derived class (C2) at another.

The program call is thus made simpler.

```
FUNCTION CallSingleCounter:VOID
    VAR_IN_OUT
        CountOut: INT;
        Locking: BOOL;
        UpValReached: BOOL;
        C : COUNTER;
    END_VAR
    // Call COUNTER ( C ) for increment/decrement
    IF UpValReached = TRUE THEN Locking := TRUE;
    END_IF;
    IF Locking = FALSE THEN
        CountOut:=C.UP(QU=>UpValReached); // increment
    END_IF;
    IF Locking = TRUE THEN
        CountOut:=C.DOWN(); // decrement
    END_IF;
    IF CountOut <= 0 THEN
        Locking := FALSE;
        UpValReached := FALSE;
    END_IF;
END_FUNCTION

PROGRAM CallCounter_ST3
    VAR
        C1:COUNTER;
        C2:COUNTER_5STEP;
        CountOut: INT;
        CountOut2: INT;
        Locking: BOOL;
        Locking2: BOOL;
        UpValReached: BOOL;
        UpValReached2: BOOL;
    END_VAR
    // Call COUNTER (C1) for increment/decrement
    CallSingleCounter( C := C1, CountOut := CountOut,
        Locking := Locking, UpValReached := UpValReached);
    // Call COUNTER_5STEP (C2) for increment/decrement
    CallSingleCounter( C := C2, CountOut := CountOut2,
        Locking := Locking2, UpValReached := UpValReached2);
END_PROGRAM
```

Note: To ensure that this example is executable, the program must be assigned to the BackgroundTask in the execution system of the SIMOTION CPU. For instructions on how to assign programs in the execution system, see chapter 8.9.13.

3.3.8 Abstract classes

According to IEC, a class may be defined as an “abstract class“. Abstract classes are used in object-oriented programming as a means of structuring the software. The software designer can use abstract classes to define the structure of classes. The commonalities between objects still to be created are summarized in the abstract class. In addition, an abstract class might contain the entire program code for individual methods that can be used by classes derived from the abstract class.

It is not possible to create instances or objects from an abstract class. We have already discussed the topic of objects using “bicycle”, “car” or “horse-drawn carriage” as

examples in chapter 2.1.3, i.e. objects that can be assigned to the higher-level class “vehicles”. In this context, the class “car” represents a class for the formation of real objects. The class “car” is derived from the class “vehicles”. But it is not possible to create any objects from the class “vehicles” because it represents a generalization (abstraction). In other words, the class “vehicles” is an abstract class which predefines the properties and operations for all vehicles. In order to create actual objects, we need to derive subclasses from the class “vehicles” and enrich them with additional properties and operations.

Abstract classes are identified by the keyword CLASS ABSTRACT <*ClassName*>. If at least one method is identified as ABSTRACT (METHOD ABSTRACT <*MethodName*>) in a class without the keyword ABSTRACT, then this class also becomes an abstract class. If a method is defined as ABSTRACT, it contains only the interface definition but no program code. It is not possible to create instances or objects from an abstract class.

A subclass is derived from the abstract class. This derived class inherits all the methods and must (so that it can be instantiated) also override the methods identified as ABSTRACT and fill them with the requisite program code.

If an abstract class already contains fully programmed methods, subclasses derived from it will also inherit these. Programmers can thus save themselves the time and effort of programming these methods again.

Let’s use a concrete example to illustrate the relevant principles: A machine is equipped with a variety of different motors, e.g. induction motors with contactor-type starters, motors with star-delta starters, or speed-controlled drive systems. It must be possible to start up and shut down all motors or drive systems. However, the startup and shutdown procedure varies according to the motor or drive type and these variations need to be reflected in the programming. The methods Switch_on() and Switch_off() could now be defined in an abstract class together with the input and output parameters that they may need. In addition, it should be possible to query the status of the relevant motor or drive. To make this possible, a method Status() is defined in the abstract class. This method is capable of supplying the information “motor/drive is “running”, “stationary”, “starting”, “stopped”, “actual speed” or “error”. This method must be used by all the derived classes and may be programmed in full in the abstract class. Once these decisions have been reached, the individual development teams can start the program development work assigned to them.

The programmer responsible for programming the class for direct-on-line starting drives derives a subclass named e.g., “DirectDrive”, from the abstract class and fills the defined methods Switch_on() and Switch_off() with the program code actually required for the direct-on-line starting drive application. This programmer tests the program section until it is completed.

The other programmers work in the same way to complete the programs for the other types of motor. At the end of this process, three different classes have been created, each containing the appropriate program code for Switch_on() and Switch_off() of the relevant motor type. But all these have one thing in common: the methods all have the same name and the same input/output interface (signature).

This approach allows different development teams to work independently. The structure of the software has been defined and the time/effort required to create it can be reduced because programming can progress independently and in parallel.

You can find an example illustrating the use of abstract classes in chapter 3.7.

3.4 Examples of valve applications with OOP

In previous chapters, we have explained the principle of using classes with inheritance and also discussed method override mechanisms, but these examples merely served to explain the principle. So we will now turn our attention to an example with a more practical orientation and so improve your understanding of classes and their use.

Valves are often deployed in combination with cylinders in mechanical engineering. We therefore see this scenario as the perfect example for implementing a control system for machine elements with valves and cylinders using object-oriented programming mechanisms.

Valve designs are remarkably diverse and the way in which valves operate also varies depending on the application. Programming valve applications using OOP offers many advantages. By using OOP with inheritance, it is possible to extend the scope of control functions easily without needing to adapt executable programs that have already been tested. It thus becomes much easier to maintain the software or adapt it as required. The following examples illustrate this.

Note: We have limited the content of the examples so as to illustrate only the essential elements. The diagrams and programs contain only those elements and depictions that are needed to improve understanding of the functions. Other essential components (such as hydraulic pumps, throttles, etc.) have been omitted in order to focus your attention on the principles of OOP.

3.4.1 Example with 4/3-way valve

A cylinder is to be controlled by means of a 4/3-way valve. In this example, the valve for the forwards and backwards movements is controlled by means of the corresponding solenoids. If the solenoid is not energized, the valve moves to mid-position and the cylinder stops.

Two cylinder operating modes must be implemented in the plant. In setup mode, the cylinder can be moved forwards by means of a pushbutton (Forward). The pushbutton (Backward) is actuated to reverse the cylinder. When the pushbutton is released, neither of the solenoids is energized (i.e. the valve moves to mid-position). In automatic mode, the cylinder must move forward in a self-holding motion in response to a Forward command. The cylinder remains in this state until a Backward command is issued.

Two limit switches, one to signal the start position and the other to signal the end position, are provided. The valve must be held in the end positions by the corresponding control signal. This is to prevent the cylinder from moving in the event of leakage.

The following signals are therefore required:

- Inputs:
  - Mode – automatic mode = TRUE, setup = FALSE
  - Forward – forward command
  - Backward – backward command

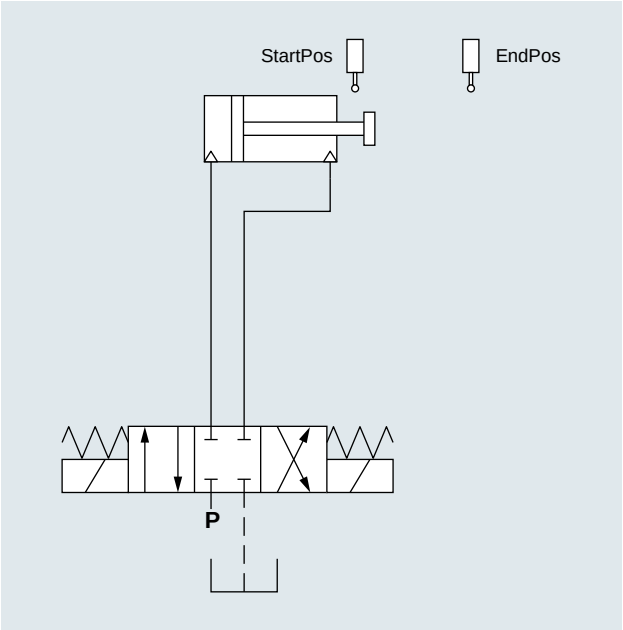


Figure 22 Plant with a 4/3-way valve

  - EndPos – end position reached
  - StartPos – start position reached
- Outputs:
  - Out_Forward – output to valve
  - Out_Backward – output to valve

3.4.1.1 Example of a class for 4/3-way valves

```
CLASS ValveControl43
    VAR
        cboMode:BOOL;
        cboEndPos:BOOL;
        cboStartPos:BOOL;
        cboForward:BOOL;
        cboBackward:BOOL;
        cboOut_Forward:BOOL;
        cboOut_Backward:BOOL;
    END_VAR

    METHOD PUBLIC mExecute:VOID // Method for cyclic call
        VAR_INPUT
            Mode:BOOL;
            EndPos:BOOL;
            StartPos:BOOL;
            Forward:BOOL;
            Backward:BOOL;
        END_VAR
```

```
        VAR_OUTPUT
            Out_Forward:BOOL;
            Out_Backward:BOOL;
        END_VAR

        cboMode:=Mode;
        cboEndPos:=EndPos;
        cboStartPos:=StartPos;
        cboForward:=Forward;
        cboBackward:=Backward;

        THIS.mForw(); // Internal call Forward
        THIS.mBackw(); // Internal call Backward

        Out_Forward:=cboOut_Forward;
        Out_Backward:=cboOut_Backward;
    END_METHOD

    METHOD mForw:VOID // Method move forward
        IF cboMode = FALSE THEN // Jog mode
            IF cboForward AND NOT cboBackward THEN
                cboOut_Forward:=TRUE;
                cboOut_Backward:=FALSE;
            ELSE
                cboOut_Forward:=FALSE;
            END_IF;
        ELSE    // Automatic mode
            IF (cboForward OR cboEndPos) AND NOT cboBackward THEN
                cboOut_Forward:=TRUE;
                cboOut_Backward:=FALSE;
            END_IF;
        END_IF;
    END_METHOD

    METHOD mBackw:VOID // Method move backward
        IF cboMode = FALSE THEN // Jog mode
            IF cboBackward AND NOT cboForward THEN
                cboOut_Forward:=FALSE;
                cboOut_Backward:=TRUE;
            ELSE
                cboOut_Backward:=FALSE;
            END_IF;
        ELSE // Automatic mode
            IF (cboBackward OR cboStartPos) AND NOT cboForward THEN
                cboOut_Forward:=FALSE;
                cboOut_Backward:=TRUE;
            END_IF;
        END_IF;
    END_METHOD
END_CLASS
```

The class ValveControl43 contains 3 methods. The method mExecute passes the input and output signals to the class. The methods mForw() and mBackw() perform the actual movement. These methods are not visible from outside the class and are called in the method mExecute by means of internal method call with THIS.

The class ValveControl43 can therefore be called and executed with various instances and as many times as required for various machine elements.

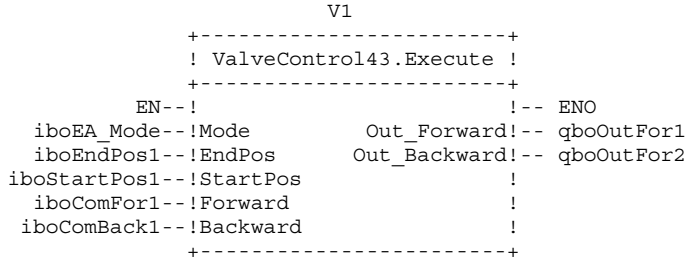
Remark: From now onwards, we will be using so-called "prefixes" for the variables in our examples. Prefixes are typed in lower case letters and digits before the actual variable names and designate first the location and then the type. For example, variable "cboEndPos" belongs to a class (c), has data type BOOL (bo) and has the name EndPos. These prefixes make the program significantly easier to read. By contrast, the interface variables of the blocks have no prefixes.

3.4.1.2 Example of a valve call

```
PROGRAM CallValveControl
    VAR
        V1:ValveControl43;
        V2:ValveControl43;
        iboEA_Mode: BOOL;
        iboComFor1: BOOL; // from here are these normally I/Os
        iboComFor2: BOOL;
        iboComBack1: BOOL;
        iboComBack2: BOOL;
        iboEndPos1: BOOL;
        iboEndPos2: BOOL;
        iboStartPos1: BOOL;
        iboStartPos2: BOOL;
        qboOut_For1: BOOL;
        qboOut_For2: BOOL;
        qboOut_Back1: BOOL;
        qboOut_Back2: BOOL;
    END_VAR
    // first call ValveControl43 (V1)
    V1.mExecute(
        Mode:=iboEA_Mode
        ,EndPos:=iboEndPos1
        ,StartPos:=iboStartPos1
        ,Forward:=iboComFor1
        ,Backward:=iboComBack1
        ,Out_Forward =>qboOut_For1
        ,Out_Backward =>qboOut_Back1
        );
    // second call ValveControl43 (V2)
    V2.mExecute(
        Mode:=iboEA_Mode
        ,EndPos:=iboEndPos2
        ,StartPos:=iboStartPos2
        ,Forward:=iboComFor2
        ,Backward:=iboComBack2
        ,Out_Forward =>qboOut_For2
        ,Out_Backward =>qboOut_Back2
        );
END_PROGRAM
```

Note: To ensure that this example is executable, the program must be assigned to the BackgroundTask in the execution system of the SIMOTION CPU. For instructions on how to assign programs in the execution system, see chapter 8.9.13.

The call in a graphic programming language is as follows:



There is no difference between using the object-oriented program in class ValveControl43 and using a function block that has been programmed by the conventional procedural method. The question we therefore need to answer is: What exactly is the advantage of using object-oriented programming?

When a class is created, it is possible to exert some influence over access rights by specifying the data (variables) and the methods using keywords PUBLIC, PRIVATE, (INTERNAL), PROTECTED (with PROTECTED being the default). The variables in class ValveControl43 are PROTECTED which means that they cannot be influenced from outside the class. In other words, the data are encapsulated.

In the event of an interruption (e.g. as a result of an interrupt) during execution of a program, this protection prevents the data from being changed unintentionally because the program called in the interrupt has no access to the data in the interrupted program. Furthermore, all the data required to permit interaction between objects can also be incorporated in the class and made public if necessary. Transfer of external data is made impossible. Programs can be executed more securely.

The other major advantages of using this programming method are revealed when it becomes necessary to adapt existing programs to meet new requirements. In this case, the existing functionality can be inherited by a derived class and the necessary adjustments made in the derived class. The original functions that have already been tested remain intact.

This is illustrated by the following example.

3.4.1.3 Example with 4/3-way valve with fast/slow speed

An additional fast/slow speed switchover mechanism has now been added to the plant. We can now derive a subclass from the existing class and implement the new functionality in the subclass.

A reminder: The following inputs and outputs were required.

- Inputs:
  - Mode – automatic mode = 1, setup = 0
  - Forward – forward command
  - Backward – backward command
  - EndPos – end position reached
  - StartPos – start position reached

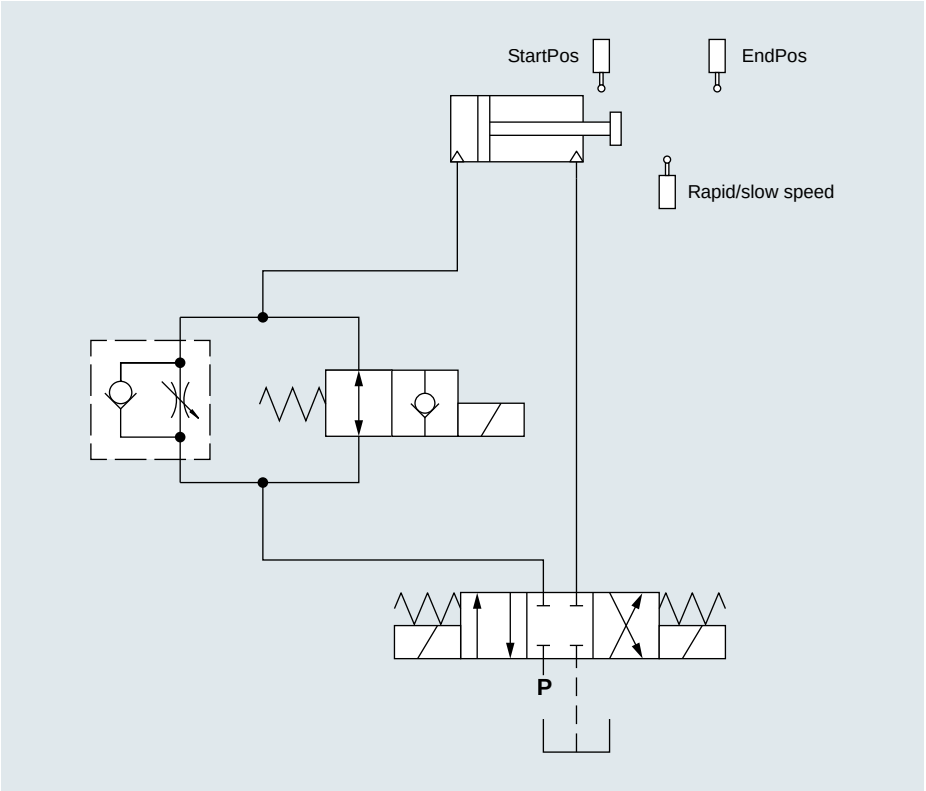


Figure 23 4/3-way valve with fast/slow speed

- Outputs:
  - Out_Forward – output to valve
  - Out_Backward – output to valve

We have now added an additional valve for fast/slow speed switchover and another limit switch to activate it. One additional input FastSlow and one additional output Out_Slow are thus required in each case.

- Additional input:
  - FastSlow – fast/slow speed switchover
- Additional output:
  - Out_Slow – output to valve slow speed

3.4.1.4 Example of a derived class ValveControl43FS

```
CLASS ValveControl43FS EXTENDS ValveControl43
    VAR
        cboFastSlow:BOOL;
        cboOut_Slow:BOOL;
    END_VAR
```

```
    METHOD PUBLIC mExecute1 // Method for cyclic call

        VAR_INPUT
            Mode:BOOL;
            EndPos:BOOL;
            StartPos:BOOL;
            Forward:BOOL;
            Backward:BOOL;
            FastSlow:BOOL;
        END_VAR

        VAR_OUTPUT
            Out_Forward:BOOL;
            Out_Backward:BOOL;
            Out_Slow:BOOL;
        END_VAR

        SUPER.mExecute(
             Mode:=Mode
            ,EndPos:=EndPos
            ,StartPos:=StartPos
            ,Forward:=Forward
            ,Backward:=Backward
            ,Out_Forward=>Out_Forward
            ,Out_Backward=>Out_Backward);

        cboFastSlow:=FastSlow;
        Out_Slow:=cboOut_Slow;

        cboOut_Slow:=cboFastSlow; // switch Fast to Slow
    END_METHOD
END_CLASS
```

In this class ValveControl43FS (derived from ValveControl43), method Execute is replaced by method Execute1. The reason it has been programmed like this in the example is to ensure that all the necessary I/Os can be transferred at the call interface. Execute1 uses the method Execute (SUPER.Execute(...)) from the base class. which means that the program code does not need to be written again.

3.4.1.5 Example of calls of base class and extended class

```
PROGRAM CallValveControl_2
    VAR
        V1:ValveControl43;
        V2:ValveControl43FS;
        iboEA_Mode: BOOL;
        iboComFor1: BOOL; // from here are these normally I/Os
        iboComFor2: BOOL;
        iboComBack1: BOOL;
        iboComBack2: BOOL;
        iboEndPos1: BOOL;
        iboEndPos2: BOOL;
        iboStartPos1: BOOL;
        iboStartPos2: BOOL;
        iboInFastSlow: BOOL;
        qboOut_For1: BOOL;
        qboOut_For2: BOOL;
        qboOut_Back1: BOOL;
        qboOut_Back2: BOOL;
        qboOut_Slowm: BOOL;
```

```
    END_VAR
    // first call ValveControl43 (V1)
    V1.mExecute(
         Mode:=iboEA_Mode
        ,EndPos:=iboEndPos1
        ,StartPos:=iboStartPos1
        ,Forward:=iboComFor1
        ,Backward:=iboComBack1
        ,Out_Forward =>qboOut_For1
        ,Out_Backward =>qboOut_Back1);
    // second call ValveControl43FS (V2)
    V2.mExecute1(
         Mode:=iboEA_Mode
        ,EndPos:=iboEndPos2
        ,StartPos:=iboStartPos2
        ,Forward:=iboComFor2
        ,Backward:=iboComBack2
        ,FastSlow:=iboInFastSlow
        ,Out_Forward =>qboOut_For2
        ,Out_Backward =>qboOut_Back2
        ,Out_Slow =>qboOut_Slowm);
END_PROGRAM
```

Note: To ensure that this example is executable, the program must be assigned to the BackgroundTask in the execution system of the SIMOTION CPU. For instructions on how to assign programs in the execution system, see chapter 8.9.13.

We are now using both classes in this call example. Each class is instantiated once (V1 from ValveControl43 and V2 from ValveControl43FS) and the functions are executed in cyclical operation by means of the relevant method call.

3.4.1.6 Example of call of extended class with basic function

```
PROGRAM CallValveControl_3
    VAR
        V1:ValveControl43FS;
        V2:ValveControl43FS;
        iboEA_Mode: BOOL;
        iboComFor1: BOOL; // from here are these normally I/Os
        iboComFor2: BOOL;
        iboComBack1: BOOL;
        iboComBack2: BOOL;
        iboEndPos1: BOOL;
        iboEndPos2: BOOL;
        iboStartPos1: BOOL;
        iboStartPos2: BOOL;
        iboInFastSlow: BOOL;
        qboOut_For1: BOOL;
        qboOut_For2: BOOL;
        qboOut_Back1: BOOL;
        qboOut_Back2: BOOL;
        qboOut_Slowm: BOOL;
    END_VAR
    // first call ValveControl43 with inherited Execute (V1)
    V1.mExecute(
        Mode:=iboEA_Mode
        ,EndPos:=iboEndPos1
```

```
            ,StartPos:=iboStartPos1
            ,Forward:=iboComFor1
            ,Backward:=iboComBack1
            ,Out_Forward =>qboOut_For1
            ,Out_Backward =>qboOut_Back1);
        // second call ValveControl43FS (V2)
        V2.mExecute1(
            Mode:=iboEA_Mode
            ,EndPos:=iboEndPos2
            ,StartPos:=iboStartPos2
            ,Forward:=iboComFor2
            ,Backward:=iboComBack2
            ,FastSlow:=iboInFastSlow
            ,Out_Forward =>qboOut_For2
            ,Out_Backward =>qboOut_Back2
            ,Out_Slow =>qboOut_Slowm);
    END_PROGRAM
END_IMPLEMENTATION
```

Note: To ensure that this example is executable, the program must be assigned to the BackgroundTask in the execution system of the SIMOTION CPU. For instructions on how to assign programs in the execution system, see chapter 8.9.13.

In this call example, the extended class is used for V1 and V2. Since ValveControlFS inherits all methods and properties of the base class, the method Execute can naturally be used in the call. This also means that only the variables declared in Execute may be used for the class in the call interface. The call thus functions in the same way as in the previous example.

Implementing the method mExecute1 for the extension of the I/O interface is compulsory because the method signature of the method mExecute may not be changed in the derived class. While this solution is not particularly elegant or sound, this is the only way it can be implemented using the knowledge we have gained so far. When we have learned about other object-oriented programming mechanisms, we will be able to find better solutions. For example, we will be able to neutralize the connection of I/O devices using interfaces (see chapter 3.5.9). Another, less flexible option would be to use I/O references, but this mechanism will not become available until after version V4.5 (see chapter 6.1).

3.5 Interfaces

Understanding the concept of interfaces probably presents the greatest hurdle to first-time users of object-oriented programming. Programmers who have previously used the procedural programming method need to embark along a certain learning curve in order to grasp the uses of this construct. It is important to state, however, that understanding the functional scope of interfaces is absolutely crucial to anyone who wishes to use object-oriented programming. It is only when classes are combined with interfaces that the immense potential of OOP is unleashed. The concept of interfaces was originally conceived as a feature of the JAVA or C# programming language and has therefore been included in the IEC.

In the context of object orientation, the purpose of the INTERFACE is to separate the function specification from its actual implementation in classes. In other words: An INTERFACE defines the call-interface of functions without actual program code. The classes that implement an interface make the functions available and operate the interfaces in exactly the way defined in the INTERFACE. A class can implement multiple interfaces. By using this construct, therefore, it is in a sense possible to achieve multiple inheritance at functional level.

Interfaces have the ability to provide a general description of functions (methods) when the program code for the functions does not actually exist. The interface thus represents a declaration for methods that are yet to be programmed, i.e. the methods of an interface are “abstract methods”. It is not however necessary to write the keyword ABSTRACT for the methods of an interface because “abstract” is an inherent property of the interface. An interface thus strongly resembles an abstract class that contains only abstract methods. You can find further information about this topic in chapter 3.8.

Another property of interfaces is that the methods defined within them are automatically PUBLIC which means that they can be called from anywhere. When these methods are later implemented in classes, this property must not be changed. The methods of an interface remain PUBLIC, even in all the subclasses derived from the class that has implemented the interface.

Interfaces define the relationships between different types of object and thus provide a neutral platform for information exchange between objects. Interfaces allow objects of different types to interact while retaining their independence.

Interfaces can be used to create programs which will use the methods defined in the interface before the program code of the methods has even been developed. The fact that this is so does not need to concern programmers who want to use methods defined in the interface. When the program sections developed by different programmers are later joined together and compiled, the compiler will monitor the program for any declaration violations (changes to interfaces). The warnings output by the compiler can be used to correct the problem immediately. The program will work provided that the declaration has not been violated.

3.5.1 Supported features

Table 3 shows the features defined according to IEC 61131-3 ED3 Table 51 (page 137) that are supported by SIMOTION.

Table 3 Keywords for interfaces

IEC No.	Keyword	Description
1)	INTERFACE... END_INTERFACE	The interface declaration begins with the keyword INTERFACE followed by the interface identifier, and ends with END_INTERFACE.
2)	METHOD... END_METHOD	Prototype methods are defined within the interface. Prototype methods are ones that have only their names and interface definition, but no program code. All methods are PUBLIC.

IEC No.	Keyword	Description
	Inheritance	
3)	EXTENDS	The interface is derived from a base interface (only one derivation).
	Use of interface	
4a)	IMPLEMENTS	Implements an interface in the class definition.
4b)	IMPLEMENTS	Implements multiple interfaces in the class definition.
4c)	Interface as variables	A variable of the type Interface supports referencing of functions.

Using these options, the programmer can define interfaces with prototype methods. Interfaces can use inheritance mechanisms to pass their methods on to other interfaces. The interface methods are then implemented in classes.

An interface definition begins with the keyword INTERFACE <*InterfaceName*> and ends with END_INTERFACE. Prototype methods (methods without program body) and their interfaces are defined within this declaration.

It is permissible to derive an interface from another (base) interface. This is done using the keyword EXTENDS (i.e. it extends the base interface). The methods inherited from the base interface must not be changed in the derived interface. Only additional methods can be added to a derived interface.

A class can implement one or multiple interfaces. This is programmed by typing the keyword IMPLEMENTS and the name of the interface to be implemented after the class name. If the class is to implement multiple interfaces, the interface names must be separated by commas. The function, i.e. the program code, of the interface methods is defined in the class in which the interface is implemented. These classes can pass their functions on to other classes and override the methods as and when required.

Interface variables are another useful feature of interfaces. The programmer can use interface variables to handle the references of class instances (classes which implement interfaces) and transfer them during runtime.

We have explained these principles in more detail in the following chapters.

3.5.2 Principles of interfaces

In the example shown in Figure 24, a base interface “A” with method “ma” has been programmed. Interface A passes method “ma” on to interfaces A1 and A2 and these are extended by methods mb and mc respectively.

Class B implements the interfaces A1 and A2 and now needs to define (provide the program code for) the functions of methods ma, mb and mc.

Further subclasses can subsequently be derived from class B by means of the usual mechanisms (see chapter 3.3.7 “Classes and inheritance”). The possibility of implementing multiple interfaces in a single class increases the scope of potential for inheritance via derivation of classes. Since no provision has been made (and deliberately so) for implementing multiple inheritance with classes, interface inheritance with multiple implementation in classes offers an appropriate means

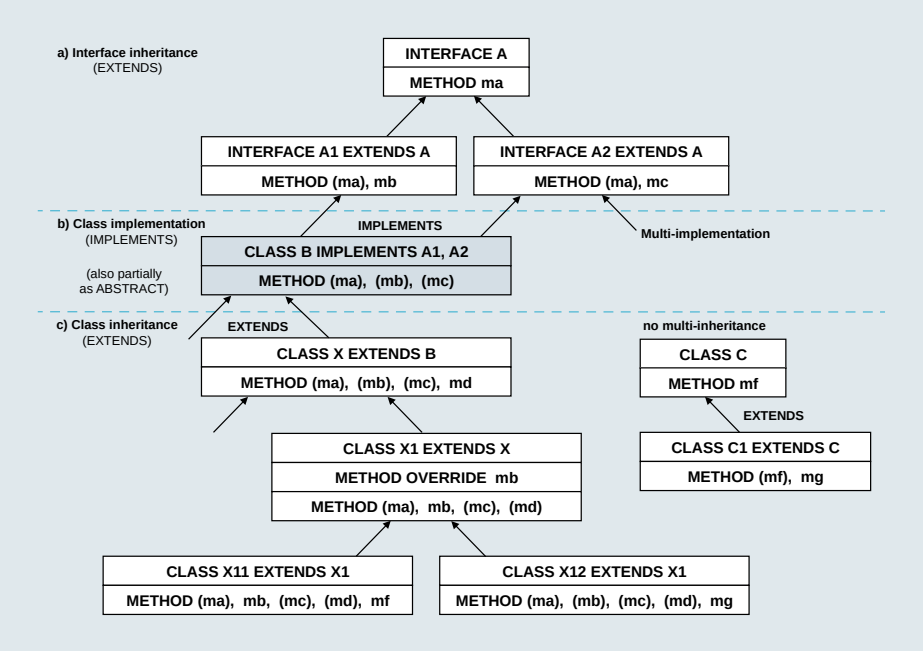


Figure 24 Interfaces (source: IEC 61131-3 ED3)

of compensation. This means that even derived classes can implement additional interfaces for expanding the scope of object functions. A derivation of derivedClass EXTENDS baseClass IMPLEMENTS IF_additional1, IF_additional2 (these interfaces are not included in Figure 24) is permissible and very widely used.

3.5.2.1 Example of an interface declaration

```
INTERFACE
    INTERFACE A
        METHOD ma
            VAR_INPUT
                IN1_ma:BOOL;
            END_VAR
            VAR_OUTPUT
                OUT1_ma:BOOL;
            END_VAR
        END_METHOD
    END_INTERFACE

    INTERFACE A1 EXTENDS A
        METHOD mb
            VAR_INPUT
                IN1_mb:INT;
            END_VAR
            VAR_OUTPUT
                OUT1_mb:INT;
            END_VAR
        END_METHOD
    END_INTERFACE
```

```
    INTERFACE A2 EXTENDS A
        METHOD mc
            VAR_INPUT
                IN1_mc:REAL;
            END_VAR
            VAR_OUTPUT
                OUT1_mc:BOOL;
            END_VAR
            VAR_IN_OUT
                INOUT1_mc:DWORD;
            END_VAR
        END_METHOD
    END_INTERFACE
    PROGRAM Usage;
END_INTERFACE

IMPLEMENTATION
    // Class implementation
    CLASS B IMPLEMENTS A1, A2
        VAR (*<vars>*); END_VAR
        METHOD PUBLIC ma
            VAR_INPUT
                IN1_ma:BOOL:=FALSE;
            END_VAR
            VAR_OUTPUT
                OUT1_ma:BOOL;
            END_VAR
            //<PROGRAM-Code FOR ma>
            ;
        END_METHOD

        METHOD PUBLIC mb
            VAR_INPUT
                IN1_mb:INT:=0;
            END_VAR
            VAR_OUTPUT
                OUT1_mb:INT;
            END_VAR
            //<PROGRAM-Code FOR mb>
            ;
        END_METHOD

        METHOD PUBLIC mc
            VAR_INPUT
                IN1_mc:REAL;
            END_VAR
            VAR_OUTPUT
                OUT1_mc:BOOL;
            END_VAR
            VAR_IN_OUT
                INOUT1_mc:DWORD;
            END_VAR
            //<PROGRAM-Code FOR mc>
            ;
        END_METHOD

        METHOD md:VOID
            VAR_INPUT
                IN1_md:DINT:=0;
            END_VAR
            ;
        END_METHOD
    END_CLASS
```

```
    CLASS X EXTENDS B
        METHOD PUBLIC OVERRIDE mb
            VAR_INPUT
                IN1_mb : INT := 0;
            END_VAR
            VAR_OUTPUT
                OUT1_mb : INT;
            END_VAR
        END_METHOD


    PROGRAM Usage
        VAR
            Inst_B: B;
            (*vars;*)
        END_VAR

        // Call method ma
        Inst_B.ma()//(IN1_ma:=<VAR>, IN2_ma:=<VAR>…)
        ;
        // Call method mb
        Inst_B.mb()//(IN1_mb:=<VAR>,…)
        ;
        // Call method mc
        Inst_B.ma()//(IN1_mc:=<VAR>,…)
        ;
    END_PROGRAM
END_IMPLEMENTATION
```

Note: This example is only a condensed presentation designed to improve your understanding of class definition with interface. It is not an executable program.

3.5.3 Representation of interfaces in the PNV of SCOUT

The project navigator (PNV) is functionally expanded to display interfaces. Interfaces are represented in the same way as classes, but with a different icon. The methods defined in the interface are displayed underneath the interface in the PNV (Figure 25). Since an interface can be derived from a base interface, the PNV has a tree node labeled “Basic types” (as it does for classes). The base interface is visible underneath the node. The programmer can use the “Go to…” command in the context menu to jump to the base interface.

An interface is created in exactly the same way as normal units (inserting an ST source). The interface definition with the corresponding keywords is stored in the source. When completed in the PNV, this definition will be displayed with the prototype methods it contains. After the code has been compiled, the interface definition will also be available in other sources.

In SIMOTION, a unit represents a container for data and programs. This container is divided into an interface section and an implementation section. The programmer defines the data that are to be “externally visible” in the interface section. In the implementation section, the programmer creates programs, function blocks, functions and classes. The interface section of the unit begins with INTERFACE and

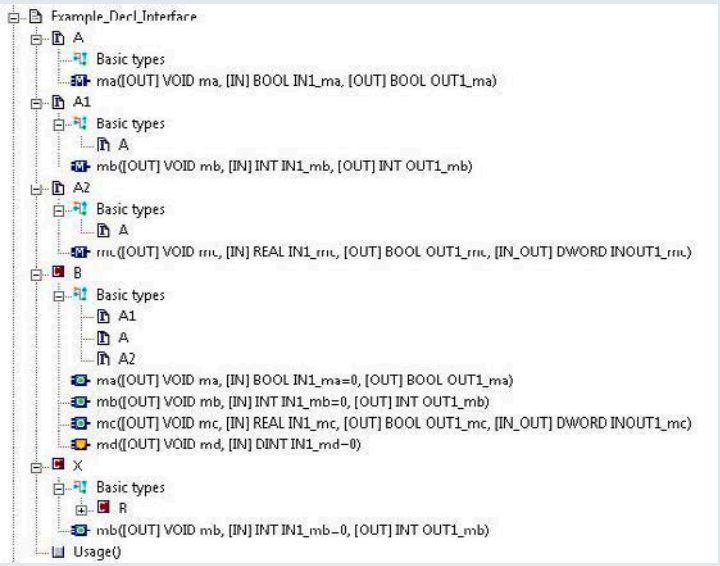


Figure 25 Interface representation in PNV

ends with END_INTERFACE. Within this interface section, the programmer can now define one or more interfaces for OOP.

The interface is represented by a corresponding icon with an “I” for “Interface”. The prototypes of methods with an “M” icon are visible beneath. The icon symbols used to represent interface methods are different to those used to represent class methods. This is because interface methods are always PUBLIC and ABSTRACT.

Note: Readers who have a working knowledge of SIMOTION are already familiar with the keywords INTERFACE and END_INTERFACE from the interface section of a unit. With the requirements for interfaces defined in IEC 61131-3 ED3, these keywords have now unfortunately acquired a double meaning in SIMOTION as these were previously used to identify the exported elements of a source.

Explanation: The IEC standard specifies these as the keywords for defining an OOP interface. With version V4.5 of SIMOTION, it is thus now permissible to program the keyword INTERFACE, followed by InterfaceName and ending in END_INTERFACE, in the interface section of a unit (and, of course, in the IMPLEMENTATION section of a unit as well) in order to define an OOP interface. This OOP interface with name can thus be used for other programs (units). This correlation is clearly illustrated in the example given in chapter 3.5.7.2.

Readers without any previous knowledge of SIMOTION can find further information about units in the section “Introduction to SIMOTION”, chapter 8.7.1.

3.5.4 Benefits of interfaces

While IEC 61131-3 ED3 has made no provision for implementing multiple inheritance in classes, the interface construct can be used instead. Interfaces can be implemented multiple times in a class.

Methods are designed as prototypes in an interface. The interfaces required by these methods are also programmed. A class that implements the interfaces now needs to provide the program code for the functions of the prototype methods. Each class that has implemented interfaces defines the methods with the specific functionality required by each method. Thus, as illustrated in Figure 26, the methods c and d in CLASS X and CLASS Y may well be programmed with completely different functions, but they both possess exactly the same interface (signature). This rule is particularly important because a call for a method from a class will only function if the interfaces of the methods exactly correspond to the specifications in the interface. If this rule is ignored, the compiler will issue an error message.

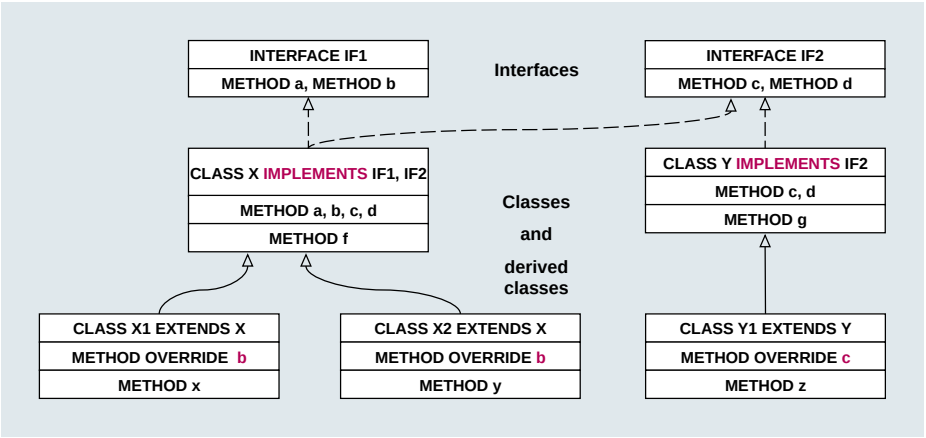


Figure 26 Interfaces in classes

Thus, by defining interfaces and knowing how they have been defined, programmers can develop program code independently although these other software components that are vital to the overall functioning of the program are still missing.

If you decide to use this programming mechanism, you should give serious consideration to the data, data models and functions before you generate any program code. Without careful planning, you may find that you have to extend the interfaces subsequently because you forgot something. This would mean that all the classes that implement or use these interfaces would also have to be amended. No automatic adjustment function is available.

The essential benefit of the interface is that the interface definition provides a neutral platform for information exchange between different objects. The interface definition constitutes a binding agreement (contract) between different software modules. A programmer familiar with the interface can be confident that interface method calls will work. All classes that implement interfaces must treat the interfaces

in accordance with the interface definition and provide the program code for the method functions.

So that objects can now use interface methods to communicate regardless of whether or not the implementing class is known, the IEC standard has made provision for specification of interface variables. Interface variables support the exchange of references to classes between different objects. References may be transferred during runtime, so allowing a variable call of the interface methods of the different objects. This is an extremely convenient and elegant programming method.

We will illustrate use of the corresponding mechanisms in the following explanations and examples.

3.5.5 Interfaces as a reference to classes

A reference is a variable that does not contain any value itself, but refers instead to a memory area in the system in which the value can be found. General references are typically used to refer to data (values). In addition to this general form of reference, the IEC standard also defines interfaces and their use.

An interface specification may be used in two different ways.

1. For class declaration.
   Implementation of the interface in a class declaration specifies the methods that need to be programmed in the class.
2. As a type of variable.
   Variables of the interface type represent references to instances of classes that implement this interface. Interface variables must be assigned before they are used. An interface specification may not be used as an IN_OUT variable.

(Source: IEC 61131-3 ED3)

Thus, the interface is a special form (definition) of a reference to an implemented function, i.e. it should be regarded as a collection of operations.

An interface constitutes the definition of a number of methods regardless of how these methods have actually been implemented. If classes implement one or more interfaces, it is the classes that provide the program code for the functions of the interface methods. In this case, the program code for the interface methods may vary widely from class to class. But none of the classes is allowed to change the interfaces of the methods.

Since classes are the blueprints for objects, the functions of a class do not come alive until objects (instances of a class) have been created. As a result, these interface methods may be implemented in various ways at different objects. A variable of this interface type is therefore used to ensure that these method variations can be addressed independently in a uniform way or even without any knowledge of how the object has been implemented. Using this type of interface variable, it is possible to call all of the methods that are defined in the interface.

By specifying interface types, it is possible to ensure that an interface variable gives either full access to a referenced function (the interface variable references a class instance) or none at all (the interface variable contains NULL).

Since interface variables ultimately represent object references, they can naturally only be used for method calls in cases where the variable actually refers to a class instance (object). This reference is normally acquired by the assignment operator ":=".

Interface variables thus help the programmer to establish the required connections between objects. Object references are exchanged by means of variables of this kind even when the programmer does not know how the object is implemented.

To achieve even greater flexibility, object references can be transferred to interface variables or between them with the "?=" operator. In this case, the type is not tested when the program is compiled but during runtime. If type-safe conversion is possible, the relevant interface variable contains a valid object reference; if not, it contains NULL.

The following example illustrates the possible transfer options.

```
INTERFACE ITF1
    METHOD m1 END_METHOD
END_INTERFACE
INTERFACE ITF2
    METHOD m2 END_METHOD
END_INTERFACE
CLASS A IMPLEMENTS ITF1
. . .;
END_CLASS
CLASS B IMPLEMENTS ITF2
. . .;
END_CLASS
CLASS C IMPLEMENTS ITF1, ITF2
. . .;
END_CLASS

FUNCTION func_if1 : VOID
    VAR_INPUT i : ITF1; END_VAR
    IF i <> NULL THEN
        i.m1();
    END_IF;
END_FUNCTION

FUNCTION func_if2 : VOID
    VAR_INPUT i   : ITF2; END_VAR
    VAR       tmp : ITF1; END_VAR
    IF i <> NULL THEN
        i.m2();
    END_IF;
    tmp ?= i; // try a dynamic type conversion to type ITF1
    IF tmp <> NULL THEN
        tmp.m1();
    END_IF;
END_FUNCTION

PROGRAM P
    VAR
        inst_a : A; // instance from CLASS A
        inst_b : B; // instance from CLASS B
        inst_c : C; // instance from CLASS C
        interf1: ITF1; // interf1 has NULL
        interf2: ITF2; // interf2 has NULL
    END_VAR
```

```
        interf1 := inst_a; // interf1 contains a valid reference to
                           // inst_a
        func_if1(interf1); // within the function inst_a.m1() is called
        func_if1(inst_a);  // the same call as the a line above

        interf2 := inst_b; // interf2 contains a valid reference to
                           // inst_b
        func_if2(interf2); // within the function inst_b.m2 is called
                           // the call TO m1 is NOT executed because tmp
                           // is NULL executing ?= Operator

        interf2 := inst_c; // interf2 contains a valid reference to
                           // inst_b
        func_if2(interf2); // within the function inst_c.m2 is called
                           // also inst_c.m1 is called; tmp contains
                           // the reference to inst_c executing
                           // ?= Operator
    END_PROGRAM
```

Note: This example is only a condensed presentation for the purposes of illustration, but it is not an executable program.

Two interfaces (ITF1 and ITF2) are defined in the example shown. Each interface contains one method, i.e. m1() in ITF1 and m2() in ITF2.

Interface ITF1 is implemented in class A, ITF2 in class B and both interfaces in class C. In a real program, these classes would of course contain the program code for the relevant functions of the interface methods, But this code is not relevant to the understanding of this example and has therefore been omitted.

The two functions func_if1 and func_if2 each have the option of accepting an interface variable from an input variable i with the relevant interface type. The functions use the query " i<>NULL" to check whether i refers to a class implementation, i.e. whether it contains a valid reference. If the reference is valid, the functions call the methods of the interfaces.

Function func_if2 has the additional option of transferring the reference of i to a tmp variable of type ITF1 (tmp ?= i). If i contains a reference that matches type ITF1, the reference is transferred by the "?=" operator. If the reference does not match, NULL is entered in tmp. If the reference in tmp is valid, method m1() is called.

The instances of classes A, B and C are generated and the interface variables interf1 and interf2 also set up in program P.

In the program code sequence, the class instances are assigned to the relevant variables interf1 and interf2. The subsequent call of the functions with transfer of the interface variables results in various interface methods being called within the functions.

When func_if1 is first called, interf1 contains the reference to the instance of class A (inst_a). The function thus calls the method m1() because the reference is valid.

The second call takes place with the transfer of inst_a. Since class A has implemented the interface ITF1, this call is functionally identical to the first call.

func_if2 is called for the first time when interf2 is transferred. interf2 contains the reference to class B. func_if2 thus calls the method m2() because class B has only implemented ITF2.

The second call of func_if2 is implemented with the reference to class C in interf2. In this case, func_if2 calls methods m1() and m2() because both interfaces are implemented in class C.

This example demonstrates perfectly how interface variables with corresponding check functions can be handled and the degree of programming flexibility offered by this approach.

This mechanism of course works in exactly the same way in methods, but we have used functions here instead of methods because the code example was simpler.

With this option for creating object references and interface variables, it is possible to specifically query (even during runtime) the types of interfaces possessed by the object type. This allows the programmer to prevent non-existent methods from being called. Newly developed object types implement existing interfaces and can thus be integrated with ease into existing programs without necessitating any further adaptations.

Thanks to this neutrality, it is possible to independently develop and test program modules. Interfaces are a description of the interfaces shared by the program modules containing the program code of the methods to be implemented. This data description can then be used, for example, to perform a test in a program area with dummy methods and data supplied by reference transfer. When the formulated program sections are created later on, the tested sections will function automatically. Of course, this arrangement will work successfully only if the interfaces are not changed retrospectively. But the compiler monitors interfaces carefully for any changes.

We will continue our explanation below by describing the use of an interface in the valve class that you will be familiar with from a previous chapter.

3.5.6 Valve classes with interfaces

In order to further illustrate the advantage of using interfaces, we will use the valve classes that we have already created. For this purpose, we need to expand the functionality of class ValveControl43 beforehand. In its current form, the class ValveControl43 has purely control functionality. As a general rule, control functions of this kind are programmed with additional error diagnosis to allow the detection of malfunctions. In its simplest form, this is a limit switch monitoring function capable of detecting the following errors:

1. Limit switch error front/rear
   (both limit switches are activated simultaneously)
2. Limit switch error rear
   (the cylinder must move away from the rear limit switch within 0.5 s when moving forwards)
3. Limit switch error front
   (the cylinder must move away from the front limit switch within 0.5 s when moving backwards)

This functionality is implemented by an additional method “mLSMon” (Limit Switch Monitoring) in the class ValveControl43 (Figure 27).

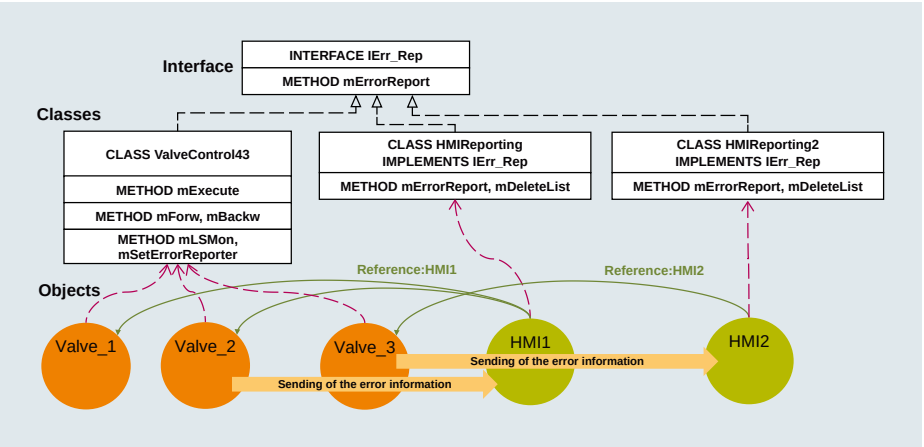


Figure 27 Overview of valve and HMI development

The relevant errors are detected by the method and need to be signaled. Errors of this kind should normally be displayed on an HMI system in order to provide the operator with information about the error locations. Since more than one valve is normally installed in a machine, it is important that the operator can clearly identify the affected valve. The error message therefore needs to contain various items of information:

1. Component ID (component affected by the error)
2. Error number
3. Error text

To be able to signal error messages to an HMI system, it is generally necessary to develop the fundamental program code for the signaling mechanism at the same time as the actual error detection function. With the procedural programming method, continuous coordination of these two programming activities is unavoidable.

But it would be an advantage if these two development processes (function development and HMI development) could take place separately and the time and work involved in coordinating them reduced to a minimum. It is precisely for scenarios of this kind that the “interfaces” construct in object-oriented programming was conceived. Different development teams cooperate and coordinate with one another to define and specify interfaces. The individual teams can then start developing software at their own pace and are not held up by waiting for deliveries of software from other teams. Delays or idle times can be eliminated altogether or occur much less frequently.

3.5.7 Declaration of the valve interface

An interface “IErr_Rep (Error Reporting) is used to allow a valve application with limit switch monitoring to signal errors to an HMI system (Figure 28). The interface contains a prototype method named “mErrorReport”. Programmed in this method is a structure for signaling errors to the HMI, a structure which comprises the elements TimeStamp, KompID, ErrorNo and ErrorText. The method mErrorReport does not include any program code.

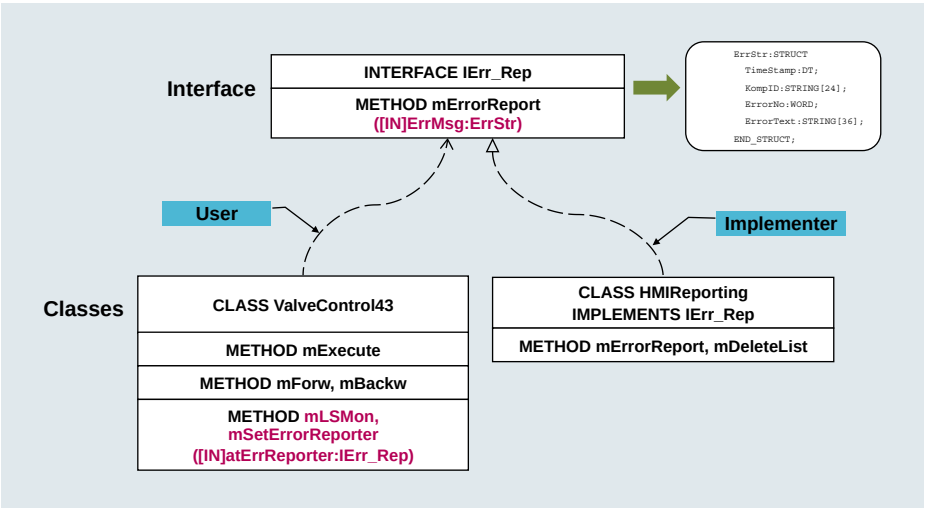


Figure 28 Interface for error reporting

The error reporting structure must be known in the class ValveControl43. The error message can then be issued in the class ValveControl43 if the instance of HMIReporting has been transferred via the method mSetErrorReport. The concept used ensures that this method simply sends the corresponding error message when a limit switch error occurs. What then happens with this error message is not taken into consideration when the valve functions are programmed. The processing and display of the error messages generated by valves are handled in a class “HMIReporting” that has still to be programmed.

Because it is possible to use and define interfaces in this way, those responsible for programming the software section containing the valve functionality do not need to have any detailed knowledge about error processing in the HMI. Both these software sections can be developed and tested individually.

We have used the following examples to demonstrate the extended program code for the valves.

3.5.7.1 Example of ValveControl43 with limit switch monitoring

In order to illustrate the advantage of interfaces, we first need to add a limit switch monitoring function to the class ValveControl43. This monitoring function generates error messages to be displayed on an HMI system. Since we do not yet know how the valve control will be linked to the HMI, we need to design the programs in such a

way that we will not have to make any further changes to the valve control software when the link to the HMI is later established.

```
CLASS ValveControl43
    VAR
        cboMode:BOOL;
        cboEndPos:BOOL;
        cboStartPos:BOOL;
        cboForward:BOOL;
        cboBackward:BOOL;
        cboOut_Forward:BOOL;
        cboOut_Backward:BOOL;
        cb16Error_LS:WORD;
        ctLSTimer:TON;
        cboReset:BOOL;
    END_VAR

    METHOD PUBLIC mExecute // Method for cyclic call
        VAR_INPUT
            Mode:BOOL;
            EndPos:BOOL;
            StartPos:BOOL;
            Forward:BOOL;
            Backward:BOOL;
            Reset:BOOL;
        END_VAR
        VAR_OUTPUT
            Out_Forward:BOOL;
            Out_Backward:BOOL;
            Error_LS:WORD;
        END_VAR

        cboMode:=Mode;
        cboEndPos:=EndPos;
        cboStartPos:=StartPos;
        cboForward:=Forward;
        cboBackward:=Backward;
        cboReset:=Reset;

        THIS.mForw(); // Internal call Forward
        THIS.mBackw(); // Internal call Backward
        THIS.mLSMon(); // Internal call LimitSwitchMonitoring

        IF cb16Error_LS <> 0 THEN // on error stop movement
            cboOut_Forward:=FALSE;
            cboOut_Backward:=FALSE;
        END_IF;

        Out_Forward:=cboOut_Forward;
        Out_Backward:=cboOut_Backward;
        Error_LS:=cb16Error_LS;
    END_METHOD

    METHOD mForw // Method move forward
        IF cboMode = FALSE THEN // Jog mode
            IF cboForward AND NOT cboBackward THEN
                cboOut_Forward:=TRUE;
                cboOut_Backward:=FALSE;
            ELSE
                cboOut_Forward:=FALSE;
            END_IF;
        ELSE    // Automatic mode
```

```
            IF (cboForward OR cboEndPos) AND NOT cboBackward THEN
                cboOut_Forward:=TRUE;
                cboOut_Backward:=FALSE;
            END_IF;
        END_IF;
    END_METHOD

    METHOD mBackw // Method move backward
        IF cboMode = FALSE THEN // Jog mode
            IF cboBackward AND NOT cboForward THEN
                cboOut_Forward:=FALSE;
                cboOut_Backward:=TRUE;
            ELSE
                cboOut_Backward:=FALSE;
            END_IF;
        ELSE // Automatic mode
            IF (cboBackward OR cboStartPos) AND NOT cboForward THEN
                cboOut_Forward:=FALSE;
                cboOut_Backward:=TRUE;
            END_IF;
        END_IF;
    END_METHOD

    METHOD mLSMon // Method Limit Switch Monitoring
        // Fault LS StartPos&EndPos
        IF (cboStartPos AND cboEndPos) AND
            NOT cb16Error_LS.15=TRUE THEN
            cb16Error_LS:=16#8001;
        // Fault StartPos
        ELSIF (cboStartPos AND cboOut_Forward) AND
               NOT cb16Error_LS.15=TRUE THEN
                ctLSTimer (pt:=T#500ms, IN:=TRUE);
            IF (ctLSTimer.Q) = TRUE THEN
                cb16Error_LS:=16#8002;
            END_IF;
            // Fault EndPos
        ELSIF (cboEndPos AND cboOut_Backward) AND
               NOT cb16Error_LS.15=TRUE THEN
               ctLSTimer (pt:=T#500ms, IN:=TRUE);
           IF (ctLSTimer.Q) = TRUE THEN
               cb16Error_LS:=16#8003;
           END_IF;
        END_IF;

        // Reset
        IF cboReset = TRUE THEN
            cb16Error_LS:=0;
            ctLSTimer (IN:=FALSE);
        END_IF;
    END_METHOD
END_CLASS
```

The method Limit Switch Monitoring (mLSMon) has been added to the class ValveControl43. This extension has been programmed directly in the base class and is not implemented through overriding by a derived class. This solution has been chosen because all classes are to have this basic function.

The method mLSMon checks the limit switches StartPos and EndPos. If signals StartPos and EndPos are set simultaneously, error 8001 will be issued. If the cylinder is at the start position and begins to move towards EndPos, the cylinder must exit from StartPos within 500 ms. If the cylinder fails to do so, error 8002 is output. The

same check is performed at the end position. In this case as well, the cylinder must exit from EndPos within 500 ms or else error message 8003 will be output. It would be meaningful to integrate a few other checks but these have been omitted for the sake of clarity.

The error messages now need to be propagated to an HMI system in some meaningful way. To this end, it must be possible to transfer a time stamp, an error text and a component ID with the error message. This information will be processed later by the HMI, but it is not necessary to know at this point how the information will be processed. All the class ValveControl43 needs to know is the definition of the interface.

3.5.7.2 Example of ValveControl43 with error reporting

Unit HMI_IF

```
INTERFACE
    TYPE
        ErrStr:STRUCT
            TimeStamp:DT;
            KompID:STRING[24];
            ErrorNo:WORD;
            ErrorText:STRING[36];
        END_STRUCT;
    END_TYPE

    INTERFACE IErr_Rep

        METHOD mErrorReport:VOID
            VAR_INPUT
                ErrMsg:ErrStr;
            END_VAR
        END_METHOD

    END_INTERFACE
END_INTERFACE
```

Note: The unit HMI_IF defines the interface for later implementation in HMIReporting. The prototype methods with their interfaces are defined here. Only one method (ErrorReport) is declared in this case.

Unit TestHMI

```
INTERFACE
    USES HMI_IF;
    CLASS HMIReporting; // only for test
END_INTERFACE

IMPLEMENTATION
    // this class is a dummy for testing only, connection must be
    // removed if real class HMIReporting exists
    CLASS HMIReporting IMPLEMENTS IErr_Rep
        VAR
            TestErrorList:ErrStr;
        END_VAR
```

```
        METHOD PUBLIC mErrorReport:VOID
            VAR_INPUT
                ErrMsg:ErrStr;
            END_VAR
            TestErrorList:=ErrMsg;
        END_METHOD
    END_CLASS
END_IMPLEMENTATION
```

A dummy class HMIReporting has been programmed in the unit TestHMI. This will temporarily perform the functions of the real HMIReporting class that will be finished at a later time. The dummy class makes it possible to generate an HMI object for testing the valve classes. Since this class has been implemented in its own unit, it will be easier later on to switch over from TestHMI to the correct unit with HMI by swapping the unit connection. As a result, it will be possible to avoid the need to change other program sections.

Unit Valve_Class

```
INTERFACE
    USES HMI_IF;
    CLASS ValveControl43;
END_INTERFACE

IMPLEMENTATION
    CLASS ValveControl43
        VAR PRIVATE
            ciValveErrorRep:IErr_Rep;
            cboLock:BOOL;
            cboMode:BOOL;
            cboEndPos:BOOL;
            cboStartPos:BOOL;
            cboForward:BOOL;
            cboBackward:BOOL;
            cboOut_Forward:BOOL;
            cboOut_Backward:BOOL;
            cb16Error_LS:WORD;
            csgID_No:STRING[24];
            ctLSTimer:TON;
            cdtSysTime:DT;
            myRTC:RTC;
            cboReset:BOOL;
            csgF0000:STRING[36]:='';
            csgF8001:STRING[36]:='Limit switch error START-/END-POS';
            csgF8002:STRING[36]:='Limit switch error START-POS     ';
            csgF8003:STRING[36]:='Limit switch error END-POS       ';
        END_VAR

        METHOD PUBLIC mSetErrorReporter // setter method
            VAR_INPUT
                atErrReporter:IErr_Rep;
            END_VAR
            IF (atErrReporter=NULL) THEN
            ; // Dummy ErrorReporter or fault handling
            ELSE
                ciValveErrorRep:=atErrReporter;
            END_IF;
        END_METHOD

        METHOD PUBLIC mExecute // Method for cyclic call
```

```
VAR_INPUT
    ID_No:STRING[24];
    Mode:BOOL;
    EndPos:BOOL;
    StartPos:BOOL;
    Forward:BOOL;
    Backward:BOOL;
    Reset:BOOL;
END_VAR
VAR_OUTPUT
    Out_Forward:BOOL;
    Out_Backward:BOOL;
    Error_LS:WORD;
END_VAR
VAR
    myErrStr:gsErrStr;
END_VAR

cboMode:=Mode;
cboEndPos:=EndPos;
cboStartPos:=StartPos;
cboForward:=Forward;
cboBackward:=Backward;
cboReset:=Reset;
csgID_No:=ID_No;

THIS.mForw(); // Internal call Forward
THIS.mBackw(); // Internal call Backward
THIS.mLSMon(); // Internal call LimitSwitchMonitoring

IF cb16Error_LS <> 0 THEN // on error stop movement
    cboOut_Forward:=FALSE;
    cboOut_Backward:=FALSE;
END_IF;

Out_Forward:=cboOut_Forward;
Out_Backward:=cboOut_Backward;
Error_LS:=cb16Error_LS;

// Error Reporting
// no error
IF cb16Error_LS=0 THEN
    myErrStr.ErrorNo:=0;
    myErrStr.ErrorText:=csgF0000;
    cboLock:=FALSE;
ELSE

    myRTC(read:=TRUE,cdt=>cdtSysTime);

    IF cb16Error_LS=16#8001 THEN // first error
        myErrStr.TimeStamp:=cdtSysTime;
        myErrStr.KompID:=csgID_No;
        myErrStr.ErrorNo:=cb16Error_LS;
        myErrStr.ErrorText:=csgF8001;
    END_IF;
    IF cb16Error_LS=16#8002 THEN // second error
        myErrStr.TimeStamp:=cdtSysTime;
        myErrStr.KompID:=csgID_No;
        myErrStr.ErrorNo:=cb16Error_LS;
        myErrStr.ErrorText:=csgF8002;
    END_IF;
    IF cb16Error_LS=16#8003 THEN // third error
        myErrStr.TimeStamp:=cdtSysTime;
```

```
                myErrStr.KompID:=csgID_No;
                myErrStr.ErrorNo:=cb16Error_LS;
                myErrStr.ErrorText:=csgF8003;
            END_IF;
            // call of Method ErrorReport if an error occurs
            IF (cboLock=FALSE AND cb16Error_LS.15=TRUE) THEN
                ciValveErrorRep.mErrorReport(ErrMsg:=myErrStr);
                cboLock:=TRUE;
            END_IF;
        END_IF;
    END_METHOD

    METHOD PRIVATE mForw // Method move forward
        IF cboMode = FALSE THEN // Jog mode
            IF cboForward AND NOT cboBackward THEN
                cboOut_Forward:=TRUE;
                cboOut_Backward:=FALSE;
            ELSE
                cboOut_Forward:=FALSE;
            END_IF;
        ELSE    // Automatic mode
            IF (cboForward OR cboEndPos) AND NOT cboBackward THEN
                cboOut_Forward:=TRUE;
                cboOut_Backward:=FALSE;
            END_IF;
        END_IF;
    END_METHOD

    METHOD PRIVATE mBackw // Method move backward
        IF cboMode = FALSE THEN // Jog mode
            IF cboBackward AND NOT cboForward THEN
                cboOut_Forward:=FALSE;
                cboOut_Backward:=TRUE;
            ELSE
                cboOut_Backward:=FALSE;
            END_IF;
        ELSE // Automatic mode
            IF (cboBackward OR cboStartPos) AND NOT cboForward
                THEN
                cboOut_Forward:=FALSE;
                cboOut_Backward:=TRUE;
            END_IF;
        END_IF;
    END_METHOD

    METHOD PRIVATE mLSMon // Method Limit Switch Monitoring
        // Fault LS StartPos&EndPos
        IF (cboStartPos AND cboEndPos) AND NOT cb16Error_LS.15=TRUE
            THEN
            cb16Error_LS:=16#8001;
        // Fault StartPos
        ELSIF (cboStartPos AND cboOut_Forward) AND NOT
            cb16Error_LS.15=TRUE THEN
            ctLSTimer (pt:=T#500ms, IN:=TRUE);
            IF (ctLSTimer.Q) = TRUE THEN
                cb16Error_LS:=16#8002;
            END_IF;
            // Fault EndPos
        ELSIF (cboEndPos AND cboOut_Backward) AND NOT
            cb16Error_LS.15=TRUE THEN
            ctLSTimer (pt:=T#500ms, IN:=TRUE);
```

```
                IF (ctLSTimer.Q) = TRUE THEN
                    cb16Error_LS:=16#8003;
                END_IF;
            END_IF;
            // Reset
            IF cboReset = TRUE THEN
                cb16Error_LS:=0;
                ctLSTimer (IN:=FALSE);
            END_IF;
        END_METHOD
    END_CLASS
END_IMPLEMENTATION
```

The ErrorReporting method has also been integrated into the class ValveControl43. Since the class HMIReporting does not yet exist, however, there is a class with the same name that allows transmission of the error message to be tested. Transmission of the error message is programmed in the method mExecute.

The following example illustrates the application of the class in the call environment for two valves.

3.5.7.3 Example of ValveControl43 with test error reporting

Unit Valve_Program

```
INTERFACE
    USES Valve_Class, TestHMI;
    PROGRAM CallValveControl_3;
END_INTERFACE

IMPLEMENTATION
    PROGRAM CallValveControl_3
        VAR
            HMI1:HMIReporting;
            V1:ValveControl43;
            V2:ValveControl43;
            CPUStart:BOOL:=0;
            EA_Mode: BOOL;
            ComFor1: BOOL; // from here are these normally I/Os
            ComFor2: BOOL;
            ComBack1: BOOL;
            ComBack2: BOOL;
            EndPos1: BOOL;
            EndPos2: BOOL;
            StartPos1: BOOL;
            StartPos2: BOOL;
            in_Reset:BOOL;
            Out_For1: BOOL;
            Out_For2: BOOL;
            Out_Back1: BOOL;
            Out_Back2: BOOL;
        END_VAR

        // set reference for ErrorReporting
        IF CPUStart = FALSE THEN
            V1.SetErrorReporter(atErrReporter:=HMI1);
            V2.SetErrorReporter(atErrReporter:=HMI1);
            CPUStart:=TRUE;
        END_IF;
```

```
        // first call ValveControl43 (V1)
        V1.Execute(
            ID_No :='Ventil 1'
            ,Mode:=EA_Mode
            ,EndPos:=EndPos1
            ,StartPos:=StartPos1
            ,Forward:=ComFor1
            ,Backward:=ComBack1
            ,Reset:=in_Reset
            ,Out_Forward =>Out_For1
            ,Out_Backward =>Out_Back1);

        // second call ValveControl43 (V2)
        V2.Execute(
            ID_No :='Ventil 2'
            ,Mode:=EA_Mode
            ,EndPos:=EndPos2
            ,StartPos:=StartPos2
            ,Forward:=ComFor2
            ,Backward:=ComBack2
            ,Reset:=in_Reset
            ,Out_Forward =>Out_For2
            ,Out_Backward =>Out_Back2);

    END_PROGRAM
END_IMPLEMENTATION
```

Note: To ensure that this example is executable, the program must be assigned to the BackgroundTask in the execution system of the SIMOTION CPU. For instructions on how to assign programs in the execution system, see chapter 8.9.13.

The call of 2 valves is illustrated in this application. The reference to HMIReporting is transferred by means of the method mSetErrorReporter. Until the class HMIReporting has been developed, a class of the same name will be used to perform tests. This class contains only one structural element ErrMsg of the type gsErrStr in which errors are entered. As a result, this class can only hold one error entry and this must be taken into account during testing.

Once the final version of class HMIReporting has been developed, the test class HMIReporting must be deleted from the program.

The following is an example program for the class HMIReporting.

3.5.7.4 Example of class HMIReporting

In the example above, any errors pertaining to the classes ValveControl43 are simply entered in a test class. However, the valve application does not make any provision for the actual processing of errors. This will now be done using the class HMIReporting.

Unit HMI_Class

```
INTERFACE
    USES HMI_IF;
    CLASS HMIReporting;
END_INTERFACE
```

```
IMPLEMENTATION

    CLASS HMIReporting IMPLEMENTS IErr_Rep
        VAR CONSTANT
            MAXNO :INT:=100;
            EMPTYSTRUCT:gsErrStr:=
                (tTimeStamp:=DT#0001-01-01-0:0:0
                ,sgKompID:=''
                ,b16ErrorNo:=16#0000,
                sgErrorText:='');
        END_VAR

        VAR
            caErrorList :ARRAY[0..MAXNO] OF gsErrStr;
            ci32Index   :DINT;
            cb16ErrorNo :WORD;
            ci32MAXNO   :DINT:=MAXNO;
        END_VAR

        METHOD PUBLIC mErrorReport:VOID
            VAR_INPUT
                ErrMsg:gsErrStr;
            END_VAR

            IF (ErrMsg.b16ErrorNo<>0 AND caErrorList[ci32Index]<>ErrMsg)
            THEN
                cb16ErrorNo:=ErrMsg.b16ErrorNo;
                IF ci32Index < MAXNO THEN
                    ci32Index:=ci32Index+1;
                    caErrorList[ci32Index]:=ErrMsg;
                ELSE
                    ci32Index:=0;
                END_IF;
            END_IF;
        END_METHOD

        METHOD PUBLIC DeleteList:VOID
            VAR
                tmpIndex:DINT;
            END_VAR

            tmpIndex:=0;

            FOR tmpIndex:=0 TO ci32MAXNO DO
                caErrorList[tmpIndex]:=EMPTYSTRUCT;
            END_FOR;

        END_METHOD

    END_CLASS
END_IMPLEMENTATION
```

The method mErrorReport in this class enters the error in an error list (caErrorList) which consists of an array of the structure gsErrStr. The method mDeleteList has been implemented to delete the list. The class constant EMPTYSTRUCT is used for deletion.

The following example now shows the application for two valves, including entry of errors in the list followed by deletion of the entire list.

3.5.7.5 Example of ValveControl43 with error reporting

```
INTERFACE
    USES Valve_Class, HMI_Class;
    PROGRAM CallValveControl_4;
END_INTERFACE
IMPLEMENTATION
    PROGRAM CallValveControl_4
        VAR
            V1:ValveControl43;
            V2:ValveControl43;
            HMI1:HMIReporting;
            CPUStart:BOOL; // for first run
            myTrig:R_TRIG;
            Flag:BOOL;
            EA_Mode: BOOL; // from here are these normally I/Os
            ComFor1: BOOL;
            ComFor2: BOOL;
            ComBack1: BOOL;
            ComBack2: BOOL;
            EndPos1: BOOL;
            EndPos2: BOOL;
            StartPos1: BOOL;
            StartPos2: BOOL;
            in_Reset:BOOL;
            Out_For1: BOOL;
            Out_For2: BOOL;
            Out_Back1: BOOL;
            Out_Back2: BOOL;
        END_VAR

        IF CPUStart = FALSE THEN
            V1.SetErrorReporter(atErrReporter:=HMI1);
            V2.SetErrorReporter(atErrReporter:=HMI1);
            CPUStart:=TRUE;
        END_IF;

        // first call ValveControl43 (V1)
        V1.Execute(
            ID_No:='Ventil 1'
            ,Mode:=EA_Mode
            ,EndPos:=EndPos1
            ,StartPos:=StartPos1
            ,Forward:=ComFor1
            ,Backward:=ComBack1
            ,Reset:=in_Reset
            ,Out_Forward =>Out_For1
            ,Out_Backward =>Out_Back1);

        // second call ValveControl43 (V2)
        V2.Execute(
            ID_No:='Ventil 2'
            ,Mode:=EA_Mode
            ,EndPos:=EndPos2
            ,StartPos:=StartPos2
            ,Forward:=ComFor2
            ,Backward:=ComBack2
            ,Reset:=in_Reset
            ,Out_Forward =>Out_For2
            ,Out_Backward =>Out_Back2);
```

```
        myTrig(CLK:=in_Reset,Q=>Flag); // Trigger for deleting list

        IF Flag = TRUE THEN
            HMI1.DeleteList();
        END_IF;
    END_PROGRAM
END_IMPLEMENTATION
```

Note: To ensure that this example is executable, the program must be assigned to the BackgroundTask in the execution system of the SIMOTION CPU. For instructions on how to assign programs in the execution system, see chapter 8.9.13.

This example program illustrates use of the valves with the final version of the class for HMIReporting. In this case, the only change made was to swap the connection to the class TestHMI over to the class HMI_CLASS. With this minor amendment, it has been easy to switch over from the test environment to the real environment. None of the other program sections need to be altered.

The mechanisms demonstrated here also help programmers to implement or plan software test environments for testing software modules. Provision can be made for suitable test scenarios at the class planning stage. These can be prepared through the implementation of test methods and Setter or Getter methods. Using preprocessor statements, it is easy to remove the test environment code before the software is delivered (see chapter 8.9.11). The use of methods means that the program code remains easy to read.

3.5.8 Interface for neutralizing I/O components

Another purpose for which interfaces are ideally suited is to provide a neutral platform for data exchange between various complex I/O components. Complex I/O devices with different characteristics are often deployed to perform specific functions in plants. However, the application itself is always negatively affected if I/O components have the same scope of functions but a different data interface. In such cases, the application needs to be adapted every time an I/O component of a particular kind is used on one machine, but the next machine uses an I/O component of a different kind.

To get a better idea of what this actually means, let's take a look at a concrete example. The range of interrelated technical issues that arise when camera systems are connected to handling systems are ideal for this purpose. Let's start with a few basics.

3.5.8.1 Connection of cameras to the control system

Camera systems connected to control systems are complex I/O components. Cameras are deployed to perform the following tasks in machines or plants:

- **Monitoring**
  Cameras can be used as monitors when it is necessary to check internal machine processes that are not visible from the outside. Where monitoring is a manual process, information is transmitted to screens for the monitoring personnel. In this case, the camera does not influence the process directly.

If the monitoring process is automatic, the camera system must be capable of detecting process anomalies and supplying relevant information to the control system. The control system must then stabilize the process by applying correction factors (open-loop or closed-loop control) or react in order to prevent damage (shutdown).

Automatic camera-based monitoring solutions of this kind can be used, for example, to check fill levels in liquid-filling applications or to monitor the position of stackable parts.

- **Part identification**
  Production processes often need to be supplied with parts. As a general rule, the parts need to be sorted and correctly positioned. It is often worthwhile to use robots or handling equipment to perform these tasks. Mechanical sorting by vibratory feeders and appropriate chicanes is too inflexible and time-consuming during production changes. The parts are supplied randomly positioned in a random sequence on conveyor belts. The camera system detects the position of each part and transmits this information to the controller. The controller processes the data and alters the approach positions of the robot or handling system so that it can pick up the part and place it down again for further processing.

Camera systems are frequently used in combination with handling equipment to identify parts in production machines. With its system function “Path interpolation”, SIMOTION provides a tool for utilizing standardized kinematics for handling systems. In this case, the handling system is responsible for fetching parts from a pickup position and setting them down at the target position without causing collisions.

This task can be extremely complex since the parts are not normally located at a fixed pickup position, but are moving instead along a conveyor belt. This means that the handling system must be synchronized with the part movement so that it can pick up the part. After the part has been picked up, the handling system is desynchronized and then moved to the setdown position. If the setdown position is fixed, the process of setting down the part is simple. If the setdown position is also moving, however, the handling equipment must be synchronized again.

The handling system thus has to perform the following part handling tasks:

- Pick up
- Sort (if parts are different)
- Transport
- Set down

The outstanding capabilities of the delta picker robot (Figure 29) makes it the system of choice for many handling applications. Thanks to its delicate mechanical system, this robot has very low moments of inertia, but is also designed for outstanding stability, making it capable of very fast acceleration rates and velocities.

The diagram shows the kinematics of a delta picker robot, with pickup positions (P1, P3) and setdown position (P2). The pickup position is not actually two positions, but one programmed position P1 and a dynamic offset to position P3. Since the belt is moving, the part can be picked up without any problems until it reaches position

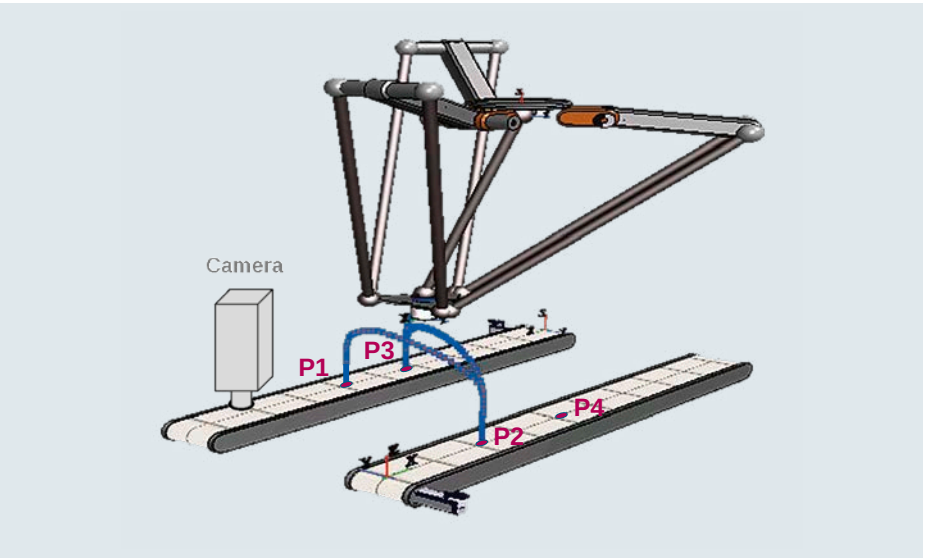


Figure 29 Delta picker with two belts

P3. Another pickup cycle for the part is not started again once it has moved past position P3.

A camera is used to identify the parts on the belt. It must detect the position of the parts and transfer this information to the SIMOTION system. Since the parts are arranged randomly on the belt, the following data are required:

- Time stamp of the image recording
- Number of parts and the following information for each part
  - X position
  - Y position
  - Angle of rotation
  - Label (if available)

Using the time stamp of the image recording, the position for synchronizing the robot can be calculated for each part. If a recorded image contains multiple parts, the camera system supplies the position, angle of rotation and label (if available) for each part. The area captured by the camera supplies overlapping image data. In other words, the same part will appear in two consecutive images. This arrangement ensures that each part is completely captured at least once. However, the images must be subsequently analyzed by software to eliminate any duplication of parts.

Conveyor belt

The parts are placed in a random sequence and at various angles of rotation on the conveyor belt. If the parts are placed too close to the edge of the belt, pickup by the robot could become very problematic if the edge is not stable enough (e.g. parts P3 and P7 in Figure 30). In this case, parts placed too close to the edge of the belt must not be picked up.

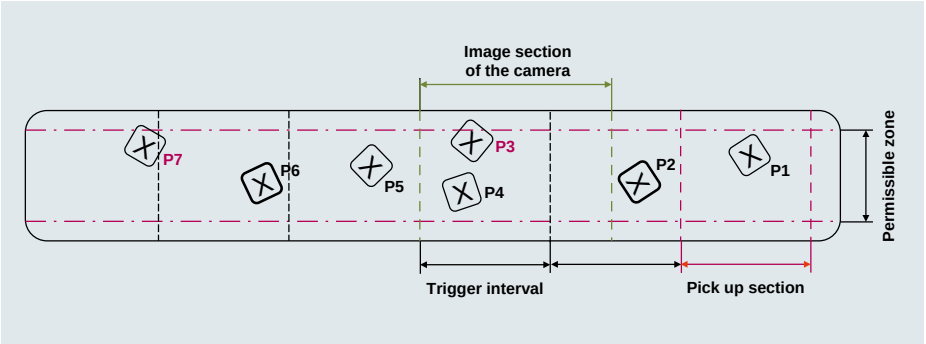


Figure 30 Conveyor belt with parts

The distance between the parts must also be detected. If the parts are positioned too close to one another, it might also be very difficult for the robot to pick them up. Analysis by software then also eliminates these parts from the pickup strategy.

The diagram shows how parts might be positioned on a conveyor belt. The camera system starts recording images in response to a trigger signal. It evaluates the image information and supplies the data in a TCP/IP telegram, for example, to the control system. The data are then entered in an intermediate buffer. This kind of register is required because of the delay between the moment the camera captures an image and the moment the part is picked up by the robot. In our example, the camera is triggered three times until the parts enter the pickup range of the robot.

- If the camera system itself is capable of detecting parts that are too close to one another or the belt edge and then eliminating them from the selection, the data are transferred from the intermediate buffer to a product register.
- If the camera system does not have this capability, the intermediate buffer must be analyzed by software in the control system.
- If a part has already been entered in the product register because it has been captured in successive images, it must be deleted from the intermediate buffer. Parts that are positioned too close to one another or to the edge of the belt are also deleted.
- The parts remaining following the analysis are then entered in order of priority at the end of the product register for pickup by the robot.
- The data of the parts with the highest priority are passed to the picker so that it picks them up; the data are then deleted.
- If the camera transmits a new data telegram, these data are transferred to the (now empty) intermediate buffer and subjected to analysis by software if necessary.

The parts to be picked up by the robot are listed in order of descending priority in the product register. The product register could be arranged as illustrated in Figure 31. This product register contains a a very broad range of information, including the time stamp data of the image recording. On the basis of the time stamp, the software can calculate the current position values to allow the robot to pick up the part. The traversing time of the robot to the pickup position might also need to be included in the calculation.

Product register (sLPRhProductRegisterType)												
Number of products in register												
Actual accessing command to register												
	Status	Prio.	Client	Product data						Trigger reference		
				Type	X pos.	Y pos.	Z pos.	Angle	Details	Pos.	Counter	Time
0												
1												
2												
3												
4												

Figure 31 Product register of SIMOTION handling

The Z position is relevant only if the camera is capable of detecting the height of the part to be picked up. For parts that are identical in height, a 2D camera system (X and Y) is generally sufficient. For parts with identical diameter, the angle of rotation is not relevant.

The parts in the product register are sorted according to position (e.g. X position) and are thus assigned a priority. The part in position "0" of the product register is then the next product to be picked up by the robot.

Since the belt is moving, according to the set belt speed, the positions in the product register must be converted in the interpolation cycle of the control system for all parts.

This brief exploration of the technical complexities of connecting a camera to a handling system clearly demonstrates the level of sophistication of the software required to process and supply data. It is for systems of this kind in particular that a highly modularized, carefully planned software design is needed. Without modularization, software derivations need to be created and repeatedly adapted for each application. The objective of effective software design is to ensure that no changes need to be made to the picker robot application, even if camera systems of different kinds are used.

Data telegram of the camera

Modern camera systems are extremely intelligent, sophisticated I/O components. They generally communicate with controllers via existing bus systems, in other words, the camera transfers the data in a telegram to the controller. The method by which data are stored in the telegram is specified by the camera manufacturer or the programmer of the camera software. Since camera systems can possess a very broad scope of functions, there is no set standard governing data storage in telegrams. The data structures supplied by camera systems can thus vary. It is always necessary to adapt the camera data when camera systems of different types are used.

Depending on the application, a distinction is made between cameras that can record images in two dimensions (2D), two and a half dimensions (2½D) or three dimensions (3D). They are also distinguished according to whether they can read text (OCR) or barcode/data matrix code. Considerable variations in the telegram structure are possible depending on the capabilities of the camera. Moreover, it is often necessary to define several telegrams for specific information. This always depends on the scope of options of the camera software and the programming of the camera.

We are not going to discuss supplementary initialization routines and camera performance settings at this point, but these also need to be programmed in the camera software section.

If a standard were to be drawn up for the useful data telegram of cameras, it could look something like the one illustrated in Figure 32. The telegram comprises a header with a standardized structure and which receives its information from the camera. The number of products is a crucial item of information. The length of the useful data area can be calculated according to the number of products, i.e. the useful data area is multiplied according to the number of products recorded by the camera.

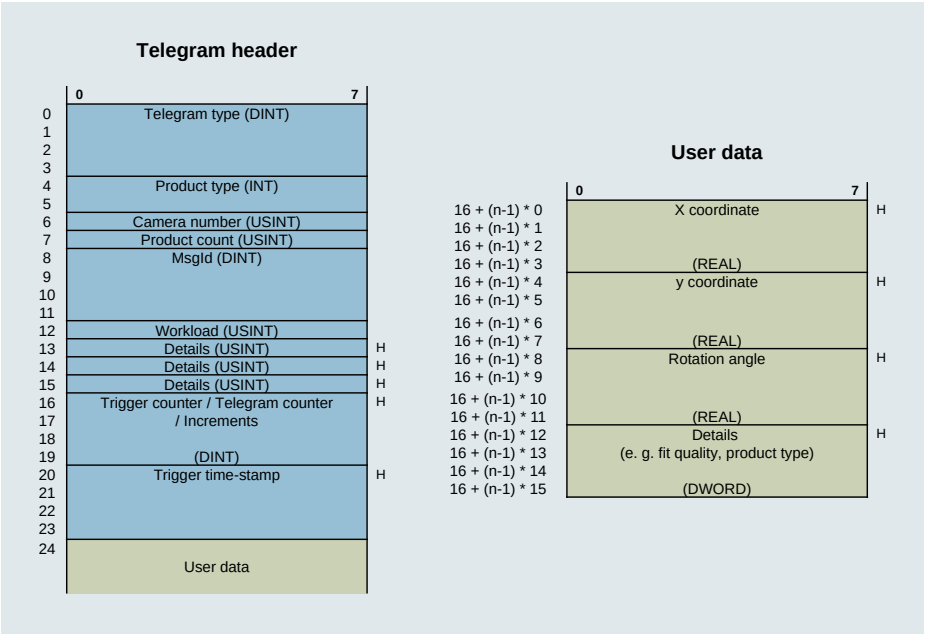


Figure 32 Proposal for a standard telegram for cameras

The useful data are entered below the header in the telegram. The number of products in byte 7 determines the telegram length. The telegram length can thus be calculated as follows:

Telegram length = telegram header + product data
Telegram length = 24 bytes + number of products * n bytes

The useful data shown above indicate that this camera is a 2D device. If it were a camera capable of recording parts in 3 dimensions, the useful data would also include the Z coordinate of the part.

This data exchange with the camera is normally defined by appropriate design kits for the relevant camera. These kits can vary widely for each camera type and for each camera manufacturer. It should be understood, therefore, that this description is merely an example designed to illustrate the potential structure of a camera telegram. An example of how to program a data exchange efficiently can be found in chapter 7.4.5.

To create a more independent software for the handling system, we are now going to use object-oriented programming mechanisms and design an interface for connecting a camera.

3.5.8.2 Interface definition for a camera connection

In order to connect a camera, we are definitely going to need a suitable structure and we will call it gsCameraData (Figure 33). This structure will contain all the information that we need for the handling system. We are now going to define an interface ICamera with methods mTrigger and mData. The method mTrigger can be called in order to activate the camera to record new images. The method mData will be used to obtain product information from the camera.

By defining the interface, we have also programmed the methods and their interfaces. How the data acquired by the camera are actually processed is not relevant for our program.

By specifying methods in the interface, we are now able to call the interface methods in our class DeltaPicker2D for the handling system. In other words, the class can

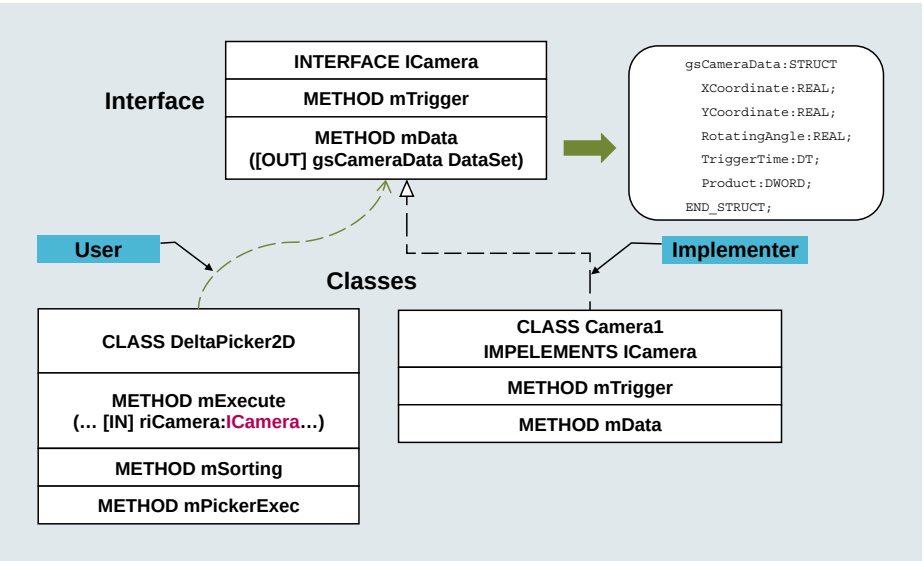


Figure 33 Interface for camera

activate the camera to record an image. Once the image has been recorded, we use the method mData to transfer the data into the product register (which we described above). To ensure that methods can be accessed, we need the reference to a camera class that is transferred, for example, via an input variable at the method mExecute. The programmer also needs to be aware that the task of taking images might be transferred to another camera during runtime. If the programmer does not want transfer of the task to a different camera during runtime, riCamera can be defined as a class variable so that specification of a particular camera is forced when the object is initialized (see chapter 7.3.1).

The methods from the interface must be implemented of course in a class for the camera (Camera1). Only when the program code for these methods has actually been written will it be possible to connect a camera to our handling system. The handling software can nevertheless be tested in a test class. For this purpose, a camera class for testing is created which the methods fill with a test program. For example, data are supplied to a structure (gsCameraData) when the trigger method is called. These data are then transferred to the handling system by the method mData. A simple test program of this kind can be used to test the entire handling system even when no real camera is connected to it. Once the real camera class actually exists, the connection can be made fully functional simply by swapping over the reference.

A range of other methods for specific functions can be implemented in the class for the handling system. A sorting routine (mSorting) can be added to ensure that products are properly sorted in the register according to the time stamp of the image recording and the relevant picking strategy.

We have decided not to include any example programs in this description because the SIMOTION handling package comprises tens of thousands of program lines. Another reason is that those reading this book will almost certainly not have access to a camera component and will not therefore be able to test the programs. But to give you the opportunity to try out the same principle in programs that can be tested, we have developed another, easily testable example.

In view of the large size of the program, we could of course have omitted this chapter altogether. However, it is a very useful indication of the scope and complexity of the programs required for modern automation systems. These programs are extremely difficult to manage if the software structure and design are not well organized or planned.

3.5.9 Interface for neutral I/O connection (condensed example)

The connection of I/O components via neutral interfaces is an ideal solution for complex peripheral devices, as described above in the camera connection example.

Most readers will probably not have the use of complex equipment of this kind in order to test the principle. For this reason, we are going to demonstrate the same principle using a much simpler example. We are again going to turn to the valve application already used in previous chapters.

When programming software modules, it is absolutely essential to ensure that no direct access to hardware resources is programmed in the module itself. If you do not follow this rule, it will be more difficult to reuse the software because when you port a program to a modified hardware variant, you will always need to adapt the

program. In order for a module to function properly, however, it must of course be linked to the real hardware, but this link is created outside the module. Moreover, it is always advisable to set up this link at one dedicated point in the program. If this mapping is programmed in different places, there is a risk that the programmer will forget or overlook something.

Large modules are normally built from several smaller modules. An outer shell often encapsulates internally integrated functional units (control modules). In such instances, it is more important than ever to observe the above mentioned rule. Essential input signals are transferred through the individual layers by means of variables. The signals for actuators must also be passed upwards again by the same principle. This approach ensures that the modules can easily be reassembled because each module can function autonomously. The module has been independently tested and is simple to reuse.

A reset signal is often required, for example, to restore modules to their initial state (Figure 34). This signal could be provided by a reset button at an input of a plant control system. The reset signal is not directly transmitted to the individual functional units, however, but is transferred via corresponding variables (e.g. Reset_Modul1, Reset_Modul2, etc.). Within the modules themselves, the signal is transferred by Reset_Modulx to the resets of the control modules. It would not be expedient to connect the Reset input directly to the control modules. The diagram shows just a few modules, but if a plant were to include 100 axes or several hundred valves, it would be necessary to modify the plant software in several hundred places.

Looking at it from this point of view, it becomes clear why software design is such a crucial factor. Without well planned software and clear rules governing the programming of modules, it is impossible to achieve a successful modular concept.

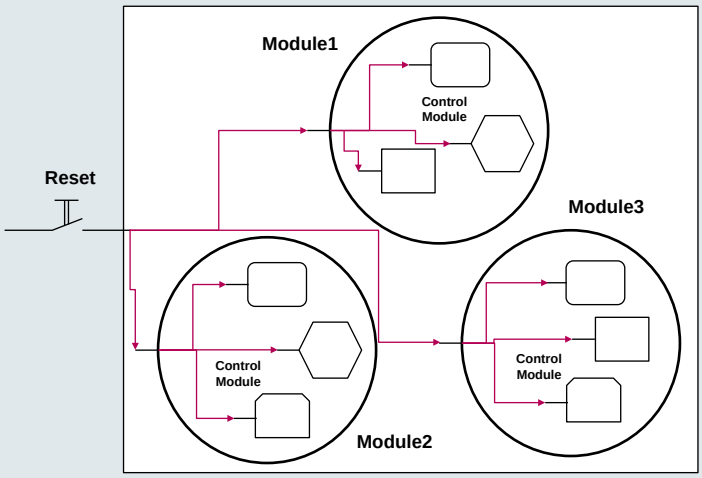


Figure 34 Principle of signal transfer in layers

3.5.9.1 Interface definition for neutral I/O connection

Now let's get back to reality. We are going to use the interface IValveIO to create a neutral I/O interface for our valve example in chapter 3.4.1.1. This interface describes a function for transferring limit switch signals via a method. A second method is used to transmit motion signals to the actual outputs for the valve. The interface mechanism is neutrally defined and possesses only the requisite interfaces. The connection is established in the classes in which the interface is implemented (Figure 35).

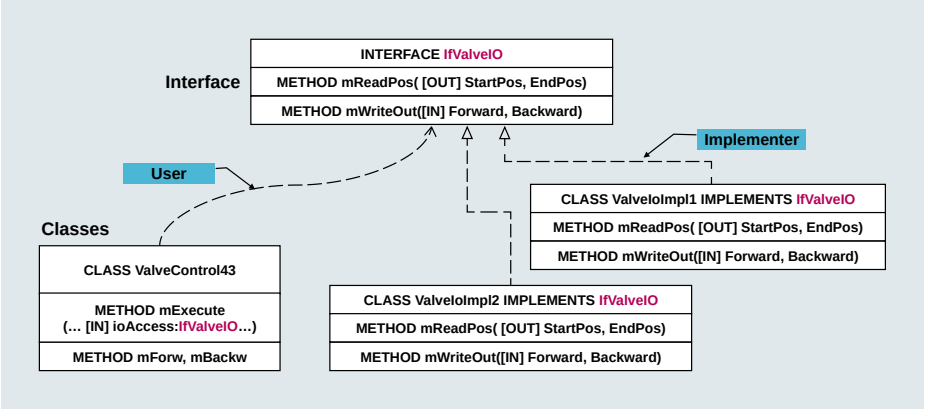


Figure 35 Neutral interface

This means that limit switch signals are no longer transferred to the class for the valve and the outputs for controlling the valve are also no longer implemented in the method mExecute(). Instead, the method mExecute now has an input parameter at which an interface can be transferred that is implemented by a class that contains access to the signals. This interface cannot be transferred of course until an object with the relevant implementation actually exists. For this purpose, we need to create two additional classes.

3.5.9.2 Implementation in classes

In order to connect the signals, let's use classes ValveIoImpl1 and ValveIoImpl2. The I/O signals are connected in these classes, but not directly – they are linked by means of intermediate variables. The intermediate variables in turn link the signals using a global variable table at the beginning of the unit (Figure 36). No provision is thus made within the class or the objects for direct access to the I/O components.

Allowing these classes direct access to I/O components would mean that they could no longer be programmed independently of the hardware.

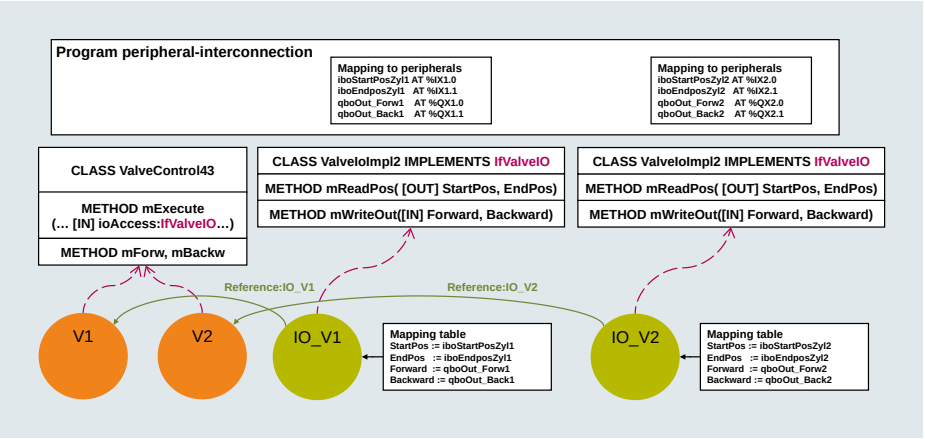


Figure 36 Valve with neutral I/O connection

3.5.9.3 Interface definition and mapping table program

```
INTERFACE
    INTERFACE IfValveIO
      METHOD mReadPos : VOID
        VAR_OUTPUT
          StartPos : BOOL;
          EndPos : BOOL;
        END_VAR
      END_METHOD
      METHOD mWriteOut : VOID
        VAR_INPUT
          Forward : BOOL;
          Backward: BOOL;
        END_VAR
      END_METHOD
    END_INTERFACE

    VAR_GLOBAL
        iboStartPosZyl1 AT %IX1.0:BOOL;
        iboEndPosZyl1   AT %IX1.1:BOOL;
        iboComFor1      AT %IX1.2:BOOL;
        iboComBack1     AT %IX1.3:BOOL;
        qboOut_For1     AT %QX1.0:BOOL;
        qboOut_Back1    AT %QX1.1:BOOL;
        iboStartPosZyl2 AT %IX2.0:BOOL;
        iboEndPosZyl2   AT %IX2.1:BOOL;
        iboComFor2      AT %IX2.2:BOOL;
        iboComBack2     AT %IX2.3:BOOL;
        qboOut_For2     AT %QX2.0:BOOL;
        qboOut_Back2    AT %QX2.1:BOOL;
    END_VAR

    CLASS ValveControl43;

    PROGRAM CallValveControl;

END_INTERFACE
```

The interface IfValveIo has two prototype methods; one for the limit switches and the other for the output signals to the valve. The interface definition is followed by the table which maps intermediate variables onto the actual I/O components.

3.5.9.4 Program for implementation and use of classes

```
IMPLEMENTATION
    CLASS ValveControl43
        VAR
            cboMode:BOOL;
            cboEndPos:BOOL;
            cboStartPos:BOOL;
            cboForward:BOOL;
            cboBackward:BOOL;
            cboOut_Forward:BOOL;
            cboOut_Backward:BOOL;
        END_VAR

        METHOD PUBLIC mExecute:VOID // Method for cyclic call
            VAR_INPUT
                Mode:BOOL;
                ioAccess : IfValveIO;
                Forward:BOOL;
                Backward:BOOL;
            END_VAR
            ioAccess.mReadPos(StartPos => cboStartPos,
            EndPos=> cboEndPos );
            cboMode:=Mode;
            cboForward:=Forward;
            cboBackward:=Backward;

            THIS.mForw(); // Internal call Forward
            THIS.mBackw(); // Internal call Backward
            ioAccess.mWriteOut(Forward:=cboOut_Forward,
            Backward:=cboOut_Backward);
        END_METHOD

        METHOD mForw:VOID // Method move forward
            IF cboMode = FALSE THEN // Jog mode
                IF cboForward AND NOT cboBackward THEN
                    cboOut_Forward:=TRUE;
                    cboOut_Backward:=FALSE;
                ELSE
                    cboOut_Forward:=FALSE;
                END_IF;
            ELSE    // Automatic mode
                IF (cboForward OR cboEndPos) AND NOT cboBackward THEN
                    cboOut_Forward:=TRUE;
                    cboOut_Backward:=FALSE;
                END_IF;
            END_IF;
        END_METHOD

        METHOD mBackw:VOID // Method move backward
            IF cboMode = FALSE THEN // Jog mode
                IF cboBackward AND NOT cboForward THEN
                    cboOut_Forward:=FALSE;
                    cboOut_Backward:=TRUE;
                ELSE
                    cboOut_Backward:=FALSE;
                END_IF;
```

```
            ELSE // Automatic mode
                IF (cboBackward OR cboStartPos) AND NOT cboForward THEN
                    cboOut_Forward:=FALSE;
                    cboOut_Backward:=TRUE;
                END_IF;
            END_IF;
        END_METHOD
    END_CLASS

// Classes to implement I/O Mapping;
    CLASS ValveIoImpl1 IMPLEMENTS IfValveIO
      METHOD PUBLIC mReadPos : VOID
        VAR_OUTPUT
          StartPos : BOOL;
          EndPos : BOOL;
        END_VAR
// here we can implement the real Input Mapping for the first instance
         StartPos:=iboStartPosZyl1;
         EndPos:=iboEndPosZyl1;
      END_METHOD
      METHOD PUBLIC mWriteOut : VOID
        VAR_INPUT
          Forward : BOOL;
          Backward: BOOL;
        END_VAR
// here we can implement the real Output Mapping for the first instance
         qboOut_For1:=Forward;
         qboOut_Back1:=Backward;
      END_METHOD
    END_CLASS

    CLASS ValveIoImpl2 IMPLEMENTS IfValveIO
      METHOD PUBLIC mReadPos : VOID
        VAR_OUTPUT
          StartPos : BOOL;
          EndPos : BOOL;
        END_VAR
// here we can implement the real Input Mapping for the second instance
        StartPos:=iboStartPosZyl2;
        EndPos:=iboEndPosZyl2;
      END_METHOD
      METHOD PUBLIC mWriteOut : VOID
        VAR_INPUT
          Forward : BOOL;
          Backward: BOOL;
        END_VAR
// here we can implement the real Output Mapping for the second instance
         qboOut_For2:=Forward;
         qboOut_Back2:=Backward;
      END_METHOD
    END_CLASS

    PROGRAM CallValveControl
        VAR
            V1:ValveControl43;
            V2:ValveControl43;
            // instances for I/O Mapping
            IO_V1 : ValveIoImpl1;
            IO_V2 : ValveIoImpl2;
            iboEA_Mode: BOOL;
        END_VAR
```

```
        // first call ValveControl43 (V1)
        V1.mExecute(
                    Mode:=iboEA_Mode
                    ,ioAccess := IO_V1
                    ,Forward:=iboComFor1
                    ,Backward:=iboComBack1
                    );

        // second call ValveControl43 (V2)
        V2.mExecute(
                    Mode:=iboEA_Mode
                    ,ioAccess := IO_V2
                    ,Forward  :=iboComFor2
                    ,Backward :=iboComBack2
                    );
    END_PROGRAM
END_IMPLEMENTATION
```

The valve class (of the kind already used in a previous chapter) is implemented first in this program, but this time we have left out the limit switch monitoring and error reporting functions. The two classes for I/O implementation then follow and finally their use in a program. It can be seen clearly that the call interface of the method mExecute is now considerably simpler.

3.5.9.5 Interface for fast/slow speed switchover

We now want to take a closer look at how we can use interfaces to integrate a fast/slow speed switchover function. First of all, we will define an INTERFACE extension of IfValveIO with which the additional sensor and the actuator can be controlled.

```
INTERFACE
    USES valve_control;
    INTERFACE IfValveIOFS EXTENDS IfValveIO
      METHOD mReadFsSwitch : VOID
        VAR_OUTPUT
          FastSlow : BOOL;
        END_VAR
      END_METHOD
      METHOD mWriteSpeed : VOID
        VAR_INPUT
          Out_Slow : BOOL;
        END_VAR
      END_METHOD
    END_INTERFACE

    VAR_GLOBAL
        iboFSSwitch     AT %IX2.4:BOOL;
        qboFS           AT %QX2.2:BOOL;
    END_VAR

    CLASS ValveControl43FS;
    PROGRAM CallValveControlFS;
END_INTERFACE
```

3.5.9.6 Implementation of classes for fast/slow speed

We now need to implement our classes for the valve and provide implementation of an interface to supply the additional IOs. To do this, we need to extend the existing implementation of ValveIOImpl2 because we are intending to operate the second valve instance with fast/slow speed switchover. Our extended valve class overrides the method mExecute and adapts itself dynamically by means of the operator "?=" to the fast/slow speed switchover (see chapter 3.5.5 for "?=").

```
IMPLEMENTATION
    CLASS ValveControl43FS EXTENDS ValveControl43
        VAR
            m_FastSlow:BOOL;
        END_VAR
        METHOD PUBLIC OVERRIDE mExecute:VOID // Method for cyclic call
            VAR_INPUT
                Mode:BOOL;
                ioAccess : IfValveIO;
                Forward:BOOL;
                Backward:BOOL;
            END_VAR
            VAR
                ioAccessFS : IfValveIOFS;
            END_VAR

            // switch Fast to Slow
            ioAccessFS ?= ioAccess;
            IF (NULL <> ioAccessFS) THEN
                ioAccessFS.mReadFsSwitch(FastSlow => m_FastSlow);
                ioAccessFS.mWriteSpeed(Out_Slow := m_FastSlow);
            END_IF;

            SUPER.mExecute(
                Mode:=Mode
                ,ioAccess:=ioAccess
                ,Forward:=Forward
                ,Backward:=Backward
                );
        END_METHOD
    END_CLASS

    CLASS ValveIoImpl2FS EXTENDS ValveIoImpl2 IMPLEMENTS IfValveIOFS
        METHOD PUBLIC mReadFsSwitch : VOID
            VAR_OUTPUT
                FastSlow : BOOL;
            END_VAR
  // implement the additional Input Mapping for the second instance
            FastSlow:=iboFSSwitch;
        END_METHOD
        METHOD PUBLIC mWriteSpeed : VOID
            VAR_INPUT
                Out_Slow : BOOL;
            END_VAR
  // implement the additional Output Mapping for the second instance
            qboFS := Out_Slow;
        END_METHOD
    END_CLASS

    PROGRAM CallValveControlFS
        VAR
            V1:ValveControl43;
            V2:ValveControl43FS;
```

```
            // instances for I/O Mapping
            IO_V1 : ValveIoImpl1;
            IO_V2 : ValveIoImpl2FS;

            iboEA_Mode: BOOL;
        END_VAR

        // first call ValveControl43 (V1)
        V1.mExecute(
                    Mode:=iboEA_Mode
                    ,ioAccess := IO_V1
                    ,Forward:=iboComFor1
                    ,Backward:=iboComBack1
                    );

        // second call ValveControl43FS (V2)
        V2.mExecute(
                    Mode:=iboEA_Mode
                    ,ioAccess := IO_V2
                    ,Forward:=iboComFor2
                    ,Backward:=iboComBack2
                    );
    END_PROGRAM
END_IMPLEMENTATION
```

In the implementation of our program, we only need to use other instances in order to operate the second instance of our valve from the previous example with fast/slow speed switchover. V2 is created as an instance of the extended valve class. To ensure that this is also supplied with the additional sensors, the relevant I/O implementation must be set up as an instance of the extended class ValveIOImpl2FS. We only need to make these changes in the program in order to perform the set task.

In our example, the extended interface IfValveIOFS is derived from IFValveIO. We have decided that the I/O connection represents an extension of the I/O connection of the valve. There is no technical justification for doing this, however. The program would work just as well if we were to remove this derivation between the interfaces.

It is important to remember that we are using this example to demonstrate the principle of neutralizing I/O components and a simple valve of this kind would not normally be programmed in this way in practice. This construct makes a lot of sense whenever the function of a class is clear, but the actual interface to the I/O equipment is unknown. It is possible to finish programming the classes and then adapt the I/O interface to the hardware of the particular application. This example thus belongs to the second kind of class, i.e. an artificial structure for the use of hardware resources (see chapter 2.3).

3.6 Further optimization of the valve class

3.6.1 Existing implementation of ValveControl

The way in which we implemented the class ValveControl43 above had a close resemblance to procedural programming. The signals in the method Execute were essentially interconnected on a level-controlled basis, i.e. the signals were processed

continuously in the methods and the system responded if the signal level changed. This method of programming is very widely used, but has various disadvantages:

- The implemented program code is generally processed – it is not therefore a time-optimized solution. When the number of calls is multiplied because the plant contains a large number of valves, the program runtime can become a significant issue.
- It is often necessary to issue commands from different program areas. The commands in automatic processes are called from sequences, for example. Manual operation is implemented in a different program area. These various commands need to be interconnected at the call interface of the method Execute (Figure 37). The sections of program for the valve with all operating modes are thus distributed among different programs and the construct can become overcomplicated.
- In controllers with a multi-tasking system, it is more difficult to achieve asynchronous programming of functions from different tasks when the class ValveControl43 is implemented in the way shown here.

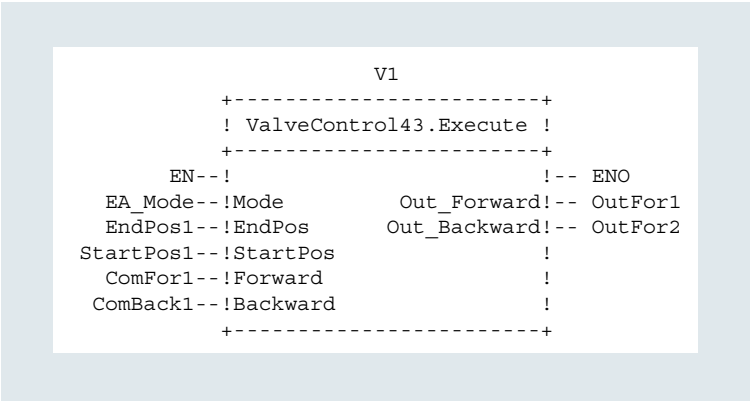


Figure 37 Valve with signal interconnection

3.6.2 Design of a state machine

A more time-efficient version of the valve function can be achieved by programming a state machine. This model defines the corresponding states for the valve-cylinder combination. Changeover from one state to the next is initiated by conditions (transitions). The valve-cylinder combination can only be in one state at any given point in time and it can change over to the next state only when changeover is initiated by a specific event or condition (referred to as a transition). With this model, a distinction is made between dynamic and static states. As a result, the valve-cylinder combination can be defined in full with a total of five states (Figure 38):

- *Stop*
  The system is in this state when none of the system elements is moving. This is also the initial state of the system at switch on. In response to the "Stop" command (ComStop), the system can also switch to the stop state from the dynamic states Forw and Backw.

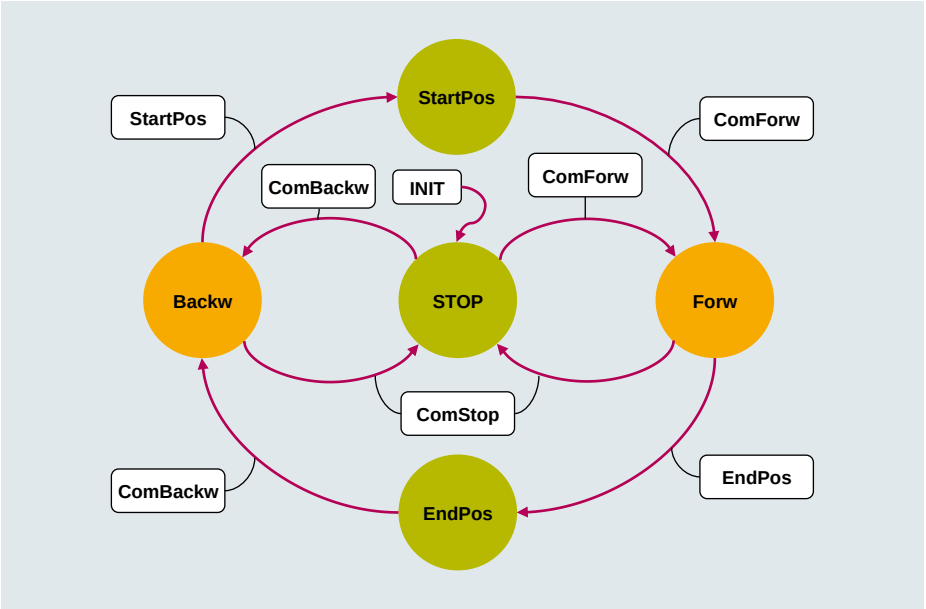


Figure 38 Valve state machine

- *StartPos*
  The system switches from the Backw state to the StartPos state when the limit switch StartPos is reached. From this state it can switch to the Forw state only when the ComForw command is issued.
- *EndPos*
  The state EndPos is the counterpart of StartPos. The system switches to this state when the limit switch EndPos is reached.
- *Forw*
  The system always switches to the dynamic state Forw during a forwards movement. It can exit from this state in response to ComStop or EndPos. It switches over to this state when command ComForw is issued.
- *Backw*
  The system switches to this state during a backwards movement. It can exit the state when StartPos is reached or the command ComStop is issued.

This state control mechanism only ever executes the program section containing the code for the state itself or the sections containing the code for exit from the state. No other parts of the program are executed. In other words, the program runtime is minimized. Even if new states have to be added to the state machine, the increase in the processing time is still lower than it would be with a level-controlled program. This advantage can be highly relevant if the state machine is to be processed as part of a large program. By reducing the cycle time by up to 10 to 20 % of the originally required cycle time, it is possible to significantly raise the machine output (throughput) without increasing the load on the machine.

This type of state machine doesn't have anything directly to do with object-oriented programming, but it is a type of programming that combines well with OOP. With

OOP, methods are called to enable communication between objects. Thus, we could implement corresponding methods for the commands ComForw, ComBackw and ComStop and these would then transfer the commands to the state machine. It would also be easy to integrate appropriate mechanisms in the methods to check the admissibility of command transfers.

With a multi-tasking operating system like SIMOTION, the commands can simply be called from other tasks, allowing the control to be implemented across several tasks without any difficulties. To allow information about the status of the state machine to be queried, we need to implement a method GetState(). The program issuing the command can then query the current state of the machine beforehand so that the commands it subsequently issues are successful.

In addition to the various advantages described above, however, we do not want to conceal the fact that this model also has disadvantages. Since commands are issued directly by calling methods, debugging is made more difficult, for example, if commands are output concurrently by mistake. Because in our example, it is always the command last issued prior to execution of the Execute method that has priority.

3.6.2.1 Example of ValveControl43ST – state machine using CASE

In this example, the class ValveControl has been converted to a state machine according to chapter 3.4.1.1 “Example of a class for 4/3-way valves”.

```
INTERFACE
    USES HMI_Class;
    TYPE
        eValveStTYPE : (STOP
                       ,FORWARD
                       ,BACKWARD
                       ,ENDPOS
                       ,STARTPOS
                       ,ERROR);
    END_TYPE

    CLASS cValveControl43ST;
    PROGRAM pCallValveControl6;
    PROGRAM pCallValveControl7;
END_INTERFACE

IMPLEMENTATION
CLASS CValveControl43ST
    VAR CONSTANT
        ERR_F0000 : STRING[36] := '';
        ERR_F8001 : STRING[36] := 'LIMIT switch error START-/END-POS';
        ERR_F8002 : STRING[36] := 'LIMIT switch error START-POS     ';
        ERR_F8003 : STRING[36] := 'LIMIT switch error END-POS       ';
    END_VAR

    VAR OVERRIDE
        // Reference of type Interface ErrorReport
        crefiValveErrorRep  : IErrRep := *;
    END_VAR
    VAR
        cboEndPos           : BOOL;
        cboStartPos         : BOOL;
        cboMoveForward      : BOOL;
        cboMoveBackward     : BOOL;
```

```
    cb16ErrorLS          : WORD;
    cFBLSTimer           : TON;
    cboReset             : BOOL;
    cboLock              : BOOL;
    cboComForward        : BOOL;
    cboComBackward       : BOOL;
    cboComStop           : BOOL;
    cdtSysTime           : DT;
    cFBRTC               : RTC;
    ceVC43State          : eValveStType;
    csgIdNo              : STRING[24];
    csErrStr             : sErrStrType;
END_VAR

METHOD PUBLIC mExecute // Method for cyclic call
    VAR_INPUT
        idNo       : STRING[24];
        endPos     : BOOL;
        startPos   : BOOL;
        reset      : BOOL;
    END_VAR
    VAR_OUTPUT
        moveForward     : BOOL;
        moveBackward    : BOOL;
        errorLS     : WORD;
    END_VAR

    IF cb16ErrorLS <> 0 THEN
        THIS.mErrState(TRUE);
    END_IF;
    // Error Reporting
    // no error
    IF cb16ErrorLS = 0 THEN
        csErrStr.b16ErrorNo  := 0;
        csErrStr.sgErrorText := ERR_F0000;
        cboLock              := FALSE;
        cFBRTC(read := FALSE
              ,cdt  => cdtSysTime);
    ELSE
        cFBRTC(read := TRUE
              ,cdt  => cdtSysTime);
        IF cb16ErrorLS = 16#8001 THEN // first error
            csErrStr.dtTimeStamp := cdtSysTime;
            csErrStr.sgKompID    := csgIdNo;
            csErrStr.b16ErrorNo  := cb16ErrorLS;
            csErrStr.sgErrorText := ERR_F8001;
        ELSIF cb16ErrorLS = 16#8002 THEN // second error
            csErrStr.dtTimeStamp := cdtSysTime;
            csErrStr.sgKompID    := csgIdNo;
            csErrStr.b16ErrorNo  := cb16ErrorLS;
            csErrStr.sgErrorText := ERR_F8002;
        ELSIF cb16ErrorLS = 16#8003 THEN // third error
            csErrStr.dtTimeStamp := cdtSysTime;
            csErrStr.sgKompID    := csgIdNo;
            csErrStr.b16ErrorNo  := cb16ErrorLS;
            csErrStr.sgErrorText := ERR_F8003;
        END_IF;
        // call of Method ErrorReport if an error occurs
        IF cb16ErrorLS.15 AND NOT cboLock THEN
            crefiValveErrorRep.mErrorReport(errMsg := csErrStr);
            cboLock := TRUE;
        END_IF;
    END_IF;
```

```
// State machine for ValveControl
CASE ceVC43State OF
    eValveStType#STOP:
        IF cboComForward THEN
            ceVC43State := eValveStType#FORWARD;
        ELSIF cboComBackward THEN
            ceVC43State := eValveStType#BACKWARD;
        END_IF;
        cboMoveForward   := FALSE;
        cboMoveBackward  := FALSE;

    eValveStType#FORWARD:
        IF cboComStop THEN
            ceVC43State     := eValveStType#STOP;
            cboComForward   := FALSE;
        ELSIF cboEndPos THEN
            ceVC43State     := eValveStType#ENDPOS;
            cboComForward   := FALSE;
        END_IF;

        cboMoveForward   := TRUE;
        cboMoveBackward  := FALSE;

    eValveStType#BACKWARD:
        IF cboComStop THEN
            ceVC43State     := eValveStType#STOP;
            cboComBackward  := FALSE;
        ELSIF cboStartPos THEN
            ceVC43State     := eValveStType#STARTPOS;
            cboComBackward  := FALSE;
        END_IF;

        cboMoveForward   := FALSE;
        cboMoveBackward  := TRUE;

    eValveStType#ENDPOS:
        IF cboComBackward THEN
            ceVC43State     := eValveStType#BACKWARD;
            cboComForward   := FALSE;
        END_IF;

        cboMoveForward   := FALSE;
        cboMoveBackward  := FALSE;

    eValveStType#STARTPOS:
        IF cboComForward THEN
            ceVC43State     := eValveStType#FORWARD;
            cboComBackward  := FALSE;
        END_IF;

        cboMoveForward   := FALSE;
        cboMoveBackward  := FALSE;

    eValveStType#ERROR:
        IF cboReset THEN
            ceVC43State := eValveStType#STOP;
        END_IF;

        cboMoveForward   := FALSE;
        cboMoveBackward  := FALSE;
```

```
    ELSE
        ceVC43State := eValveStType#ERROR;
    END_CASE;

    cboEndPos   := endPos;
    cboStartPos := startPos;
    cboReset    := reset;
    csgIdNo     := idNo;
    THIS.mLSMon(); // Internal call LimitSwitchMonitoring
    moveForward     := cboMoveForward;
    moveBackward    := cboMoveBackward;
    errorLS         := cb16ErrorLS;
END_METHOD

METHOD mLSMon // Method Limit Switch Monitoring
    // Fault LS StartPos&EndPos
    IF(cboStartPos AND cboEndPos) AND (NOT cb16ErrorLS.15) THEN
        cb16ErrorLS := 16#8001;
    // Fault StartPos
    ELSIF (cboStartPos AND cboMoveForward) AND
            (NOT cb16ErrorLS.15) THEN
        cFBLSTimer(pt := T#500ms
                   ,IN := TRUE);
        IF (cFBLSTimer.Q) THEN
            cb16ErrorLS := 16#8002;
        END_IF;
    // Fault EndPos
    ELSIF (cboEndPos AND cboMoveBackward) AND
            (NOT cb16ErrorLS.15 = TRUE) THEN
        cFBLSTimer(pt := T#500ms
                   ,IN := TRUE);
        IF (cFBLSTimer.Q) THEN
            cb16ErrorLS := 16#8003;
        END_IF;
    ELSE
        cFBLSTimer(IN := FALSE);
    END_IF;
    // Reset
    IF cboReset THEN
        cb16ErrorLS := 0;
        cFBLSTimer(IN := FALSE);
    END_IF;
END_METHOD

METHOD PUBLIC mGetState // asking for actual state
   VAR_OUTPUT
       actState : eValveStType;
   END_VAR
   actState := ceVC43State;
END_METHOD

METHOD PUBLIC mForward : eValveStType // Command Forward
    VAR_INPUT
        condition : BOOL;
    END_VAR

    IF NOT condition THEN // only output of actual state
       mForward := ceVC43State;
    ELSE // if bo condition=true method(command) is executed
        IF (ceVC43State = eValveStType#STOP) OR
              (ceVC43State = eValveStType#STARTPOS) THEN
```

```
                ceVC43State       := eValveStType#FORWARD;
                mForward          := ceVC43State;
                cboComForward     := TRUE;
                cboComStop        := FALSE;
            END_IF;
        END_IF;
    END_METHOD

    METHOD PUBLIC mBackward : eValveStType // Command Backward
        VAR_INPUT
          condition : BOOL;
        END_VAR

        IF NOT condition THEN // only output of actual state
            mBackward := ceVC43State;
        ELSE // if bo condition=true method(command) is executed
            IF (ceVC43State = eValveStType#STOP) OR
                 (ceVC43State = eValveStType#ENDPOS) THEN
                ceVC43State       := eValveStType#Backward;
                mBackward         := ceVC43State;
                cboComBackward    := TRUE;
                cboComStop        := FALSE;
            END_IF;
        END_IF;
    END_METHOD

    METHOD PUBLIC mStop : eValveStType // Command Stop
        VAR_INPUT
            condition : BOOL;
        END_VAR

        IF NOT condition THEN // only output of actual state
            mStop := ceVC43State;
        ELSE // if bo condition=true method(command) is executed
            ceVC43State       := eValveStType#STOP;
            mStop             := ceVC43State;
            cboComStop        := TRUE;
            cboComForward     := FALSE;
            cboComBackward    := FALSE;
        END_IF;
    END_METHOD

    METHOD mErrState // Internal Method
        VAR_INPUT
            condition    : BOOL;
        END_VAR

        IF condition THEN
            ceVC43State := eValveStType#ERROR;
        END_IF;
    END_METHOD
END_CLASS

PROGRAM pCallValveControl6
    VAR
        HMI1      : cHMIReporting;
        Valve1    : cValveControl43ST := (crefiValveErrorRep := HMI1);
        sMyState        : eValveStType := eValveStType#STOP;
        boEndPos        : BOOL;
        boStartPos      : BOOL;
        boReset         : BOOL;
        boMoveForward   : BOOL;
```

```
        boMoveBackward  : BOOL;
        b16ErrorLS      : WORD;
    END_VAR

    Valve1.mGetState(actState => sMyState); // Actual State

    IF (NOT boMoveForward) AND boMoveBackward AND
       (sMyState = eValveStType#STOP OR
        sMyState = eValveStType#ENDPOS) THEN
        sMyState := Valve1.mBackward(TRUE); // Call of Method Backw
    END_IF;

    IF boMoveForward AND (NOT boMoveBackward) AND
       (sMyState = eValveStType#STOP OR
        sMyState = eValveStType#STARTPOS) THEN
        sMyState := Valve1.mForward(TRUE); // Call of Method Forw
    END_IF;

    sMyState := Valve1.mStop(FALSE);

    IF (NOT boMoveForward) AND (NOT boMoveBackward) AND
       (sMyState = eValveStType#FORWARD OR
        sMyState = eValveStType#BACKWARD) THEN
        sMyState := Valve1.mStop(TRUE); // Call of Method Stop
    END_IF;

    Valve1.mExecute(iDNo            := 'Valve1'
                   ,endPos          := boEndPos
                   ,startPos        := boStartPos
                   ,reset           := boReset
                   ,moveForward     => boMoveForward
                   ,moveBackward    => boMoveBackward
                   ,errorLS         => b16ErrorLS
                   );
END_PROGRAM
END_IMPLEMENTATION
```

Note: To ensure that this example is executable, the program must be assigned to the BackgroundTask in the execution system of the SIMOTION CPU. For instructions on how to assign programs in the execution system, see chapter 8.9.13.

The class ValveControl43ST in which the valve functions are defined has now been converted to a state machine. The states are processed by means of a central case branch. The states no longer change in response to signals, but to method calls. The methods Forw, Backw and Stop have been created for this purpose. The system may switch to another state only if this is admissible according to the state model. A method GetState has been defined in the class to allow the current state of the machine to be queried. The example has been further enhanced by the addition of a means of querying the condition in the form of a signal "Condition" in the command methods. When the method Forw, Backw or Stop is called with Condition=False, the current status is supplied in the return value. The call with Condition=True initiates switchover to another state. The methods do check, however, whether state switchover is permissible. If not, the method is simply terminated.

The command methods can now be called from any task and therefore allow a higher degree of programming flexibility. pCallValveControl6 shows examples of how com-

mand methods are called. The options available for querying the current status of the state machine are also shown.

The valve functions must be processed in cyclic operation using the method mExecute. In this version, only the limit switches and output signals need to be interconnected, but interconnection of the previous signals for motions is no longer necessary.

This example of a state machine for the valve represents a very simple implementation. It has no command queue which can hold more than one command so that no information functions relating to queuing commands have been implemented. Our primary goal was to explain the principle of the state machine and thus make the logical connection to the Technology Objects that are a feature of SIMOTION. The TOs of SIMOTION work in principle like this valve state machine, but have a significantly broader scope of functions (such as a command queue, for example). SIMOTION TOs use a function call interface as a command interface instead of methods.

3.6.2.2 Example of ValveControl43ST – state machine with classes

Implementing a state machine using the Case mechanism is rather a classic programming solution. While it is easier to read and understand, it also has the disadvantage that the program code needs to be altered every time new states are added to the machine. A solution with which the addition of new states to the machine would not affect the code for state transition would be better. The following example shows this kind of solution.

```
INTERFACE
    CLASS ValveControlST;
    PROGRAM SMB;
END_INTERFACE
IMPLEMENTATION
CLASS cStDiagr
    TYPE PUBLIC
 // the states
 eStType    : (Stop, Forw, Backw, StartPos, EndPos);
 // the transitions
 eTransType : (CmdStop, CmdComBackwd, CmdComFwd, CmdEndPos,
               CmdStartPos);
    END_TYPE
    TYPE PRIVATE
        sTransEntryType : STRUCT
            eAct      : eStType;
            eTrans    : eTransType;
            eNext     : eStType;
        END_STRUCT
    END_TYPE
    VAR CONSTANT PRIVATE
// The entries in this table have to be sorted in ascending order
// by the member eAct -> according to eStType! That means
// first all Stop, then all Forw, then all Backw and so on.
// Because the state diagram is execution time optimized
        casStateTable : ARRAY [0..7] OF sTransEntryType
:= [(eAct := eStType#Stop, eTrans := eTransType#CmdComBackwd,
     eNext := eStType#Backw),
    (eAct := eStType#Stop, eTrans := eTransType#CmdComFwd,
     eNext := eStType#Forw),
```

```
(eAct := eStType#Forw, eTrans := eTransType#CmdEndPos,
 eNext := eStType#EndPos),
(eAct := eStType#Forw, eTrans := eTransType#CmdStop,
 eNext := eStType#Stop),
(eAct := eStType#Backw, eTrans := eTransType#CmdStartPos,
 eNext := eStType#StartPos),
(eAct := eStType#Backw, eTrans := eTransType#CmdStop,
 eNext := eStType#Stop),
(eAct := eStType#StartPos, eTrans := eTransType#CmdComFwd,
 eNext := eStType#Forw),
(eAct := eStType#EndPos, eTrans := eTransType#CmdComBackwd,
 eNext := eStType#Backw)];
END_VAR
VAR PRIVATE
    ceActState       : eStType := eStType#Stop;  // the actual state
    cboInitialized   : BOOL;
    // to see if optimization table is initialized
    cai32LookupTable : ARRAY [ENUM_TO_DINT(eStType#MIN)..
                       ENUM_TO_DINT(eStType#MAX)] OF DINT;
END_VAR

METHOD PRIVATE mDoInit
    VAR
        i          : DINT := LOWER_BOUND(casStateTable);
        eLastState : eStType := eStType#MIN;
    END_VAR
    cai32LookupTable[ENUM_TO_DINT(eLastState)] := i;
    WHILE (i <= UPPER_BOUND(casStateTable)) DO
        IF (casStateTable[i].eAct <> eLastState) THEN
            eLastState  := casStateTable[i].eAct;
            cai32LookupTable[ENUM_TO_DINT(eLastState)] := i;
        END_IF;
        i := i + 1;
    END_WHILE;
    cboInitialized := TRUE;
END_METHOD

METHOD PUBLIC FINAL mDoCommand : eStType
    VAR_INPUT
        cmd : eTransType;
    END_VAR
    VAR
        i : DINT:= LOWER_BOUND(casStateTable);
    END_VAR
    IF NOT cboInitialized THEN
        mDoInit();
    END_IF;
    i := cai32LookupTable[ENUM_TO_DINT(ceActState)];
    WHILE (i <= UPPER_BOUND(casStateTable)) DO
        IF (casStateTable[i].eAct = ceActState AND
            casStateTable[i].eTrans = cmd) THEN
            ceActState := casStateTable[i].eNext;
            EXIT;
        END_IF;
        IF ceActState < casStateTable[i].eAct THEN
            EXIT;
        END_IF;
        i := i + 1;
    END_WHILE;
    mDoCommand := ceActState;
END_METHOD
```

```
    METHOD PUBLIC FINAL mGetState : eStType
        mGetState := ceActState;
    END_METHOD
END_CLASS


CLASS cValveControlST
    VAR PRIVATE
        cMyState : cStDiagr;
    END_VAR

    METHOD PUBLIC mStopp
        cMyState.mDoCommand(cStDiagr.eTransType#CmdStop);
    END_METHOD
    METHOD PUBLIC mForward
        cMyState.mDoCommand(cStDiagr.eTransType#CmdComFwd);
    END_METHOD
    METHOD PUBLIC mBackward
        cMyState.mDoCommand(cStDiagr.eTransType#CmdComBackwd);
    END_METHOD

    METHOD PUBLIC mExecute
        VAR_INPUT
            endPos      : BOOL;
            startPos    : BOOL;
        END_VAR
        VAR_OUTPUT
            moveForward     : BOOL;
            moveBackward    : BOOL;
        END_VAR
        VAR
            eTmpState   : cStDiagr.eStType;
        END_VAR

        // make the transitions
        IF endPos THEN
      eTmpState := cMyState.mDoCommand(cStDiagr.eTransType#CmdEndPos);
        ELSIF startPos THEN
      eTmpState := cMyState.mDoCommand(cStDiagr.eTransType#CmdStartPos);
        ELSE
            eTmpState := cMyState.mGetState();
        END_IF;

        // do state dependend outputs;
        moveForward  := FALSE;
        moveBackward := FALSE;

        IF eTmpState = cStDiagr.eStType#Forw THEN
            moveForward := TRUE;
        ELSIF eTmpState = cStDiagr.eStType#Backw THEN
            moveBackward := TRUE;
        END_IF;

    END_METHOD
END_CLASS

PROGRAM pCallValveControl7
    VAR
        Valve1          : cValveControlST;
        boForward       : BOOL;
        boBackward      : BOOL;
        boEndPos        : BOOL;
        boStartPos      : BOOL;
```

```
        boMoveForward    : BOOL;
        boMoveBackward   : BOOL;
    END_VAR

    IF boForward THEN
        Valve1.mForward();
    END_IF;

    IF boBackward THEN
        Valve1.mBackward();
    END_IF;

    IF NOT boForward AND NOT boBackward THEN
        Valve1.mStopp();
    END_IF;

    Valve1.mExecute (endPos          := boEndPos
                    ,startPos        := boStartPos
                    ,moveForward     => boMoveForward
                    ,moveBackward    => boMoveBackward
                    );

END_PROGRAM

END_IMPLEMENTATION
```

The state machine is implemented on the basis of separate definition of states (eStType), transitions between states (eTransType) and the transition table (casStateTable). Implementation of the two methods doCommand and getState would be sufficient to provide the actual functions of the state machine. This solution can be used to implement state machines of any kind irrespective of the specifically defined states and transitions.

The function doInit is a special feature. After this function has been executed once, significantly fewer entries in the transition table need to be evaluated in the method doCommand in order to determine whether or not a transition must be made. This is true only on condition that entries are sorted in ascending sequence in the transition table casStateTable according to the current state (eAct) in the order in which the states are specified in eStType.

3.7 Abstract class for different drives

In a similar way to interfaces, abstract classes allow the independent development of different software sections and minimize the dependencies between them. We have already briefly presented an example with different drive components in the same plant in chapter “Abstract classes”. In this instance, the fact that we defined the functions for switching drives on and off in an abstract class made it possible to use drives neutrally irrespective of the drive type actually installed in the plant. Various drive types including direct-on-line starting drives, motors with star-delta starters, or speed-controlled drive systems might be installed in a plant (Figure 39).

All these drive types are switched on and off by different methods and these variations need to be reflected in the programming. The methods mOn() for switch on and mOff() for switch off could now be defined in an abstract class together with

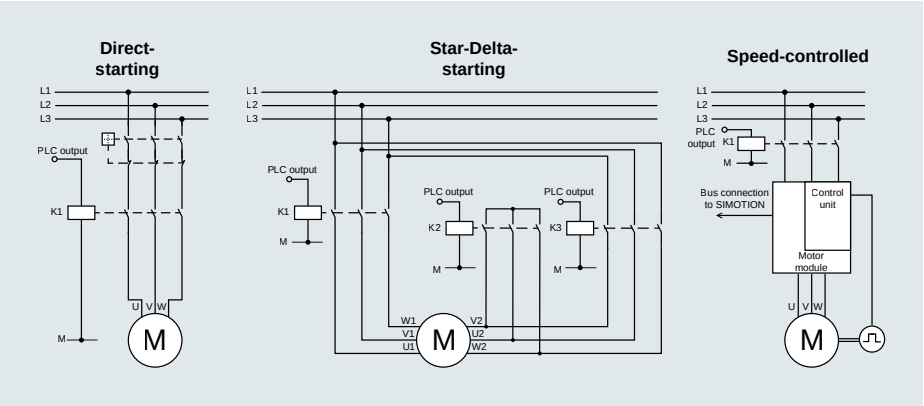


Figure 39 Different drive types in one plant

their input and output parameters. To allow interrogation of the drive status, a method mStatus() is implemented with the necessary information that should be the same for all drive components. Once the various development teams have reached agreement about interface requirements and then defined the interfaces, they can begin their programming tasks and work independently of one another.

3.7.1 Functional differences between various drive solutions

The different software development teams must first of all come to an agreement about the definition of interfaces and the data required. Each developer draws on his or her expertise to assist in the definition of appropriate interfaces.

With *direct-starting drives*, the drive is switched on via an output that must be provided on the control system. This output must be wired to the relevant contactor K1 and set to switch on the drive. The output for the contactor is reset again in order to switch off the drive.

Drives with *star-delta starters* are always deployed in cases where the starting current would be too high if a drive were to be started directly on line. The motor is started first via K1 and simultaneously with K2 in a star connection. Starting in a star connection reduces the starting current by a factor of 3. A delay timer runs down and the motor windings are then reconfigured to a delta connection by K2 dropping out and K3 picking up simultaneously. Generally speaking, an interlock implemented via the auxiliary contacts of contactors K2 and K3 ensures that K2 and K3 can never be "on" at the same time (the consequence of that would be a short circuit). The program code in the control system must ensure that the three outputs for the contactors are controlled accordingly to switch on the drive. The state "switch on" is not reached until the drive is running in a delta connection.

From the point of view of electronic circuitry requirements, *speed-controlled drives* are significantly more complex than drives started by contactors. Frequency inverters or servo converters are generally used for such applications. Programming the switch on and switch off procedures for these drives is a more complicated process than it is for contactor-controlled drives. So that the devices can function properly,

the load voltage must always be connected via a suitable power contactor. A separate load voltage connection is normally used in plants for this purpose. This connection process is implemented separately and is not therefore included in the method “Switch on” in the class definition. These drives are often connected to the controller via bus systems and operation of the drive components is also controlled via the bus system. SIMOTION is a motion controller and features standard functions designed to control drive systems of this kind. The SIMOTION system uses configurable axis objects that can perform this task. The example utilizes the Technology Objects “speed-controlled axis” for this purpose. Further information about axis objects can be found in the section “Introduction to SIMOTION” in chapter 8.9.6.

A method mStatus() will be used to query the status of the drives. This method must be capable of delivering all the information about the different drives, but, depending on the drive type, some of this information will not be available. Nonetheless, the method must be universally applicable to all drive types. That is why the programmers involved need to agree a structure that can be used by all drives.

It is always useful to document the information that can be supplied for each drive type.

The information that can be supplied by individual drive types is as follows:

- Direct-on-line starting drives
  - Drive is off
  - Drive is on
  - Drive error (available only if motor starter protector is installed)
- Drive with star-delta starter
  - Drive is off
  - Drive is on
  - Drive is accelerating
  - Drive error (available only if motor starter protector is installed)
- Speed-controlled drive
  - Drive is off
  - Drive is on
  - Drive is accelerating (positive speed change)
  - Drive is decelerating (negative speed change)
  - Setpoint speed reached
  - Actual speed value
  - Drive error
  - Error number/error code

This information should be used to define a suitable structure that can be used by all drives and in which they can deposit their data. Programmers normally agree on a maximum structure that is only partially supplied with information by the more simple drives. It is certainly reasonable to consider whether this status information should be made available subsequently for display on an HMI system. If it does, it can make sense to include the HMI data in the structure of the drive component. The definition of an interface would also be suitable for this purpose.

When considering whether information should be made available for other purposes, the programmer may deem it necessary to prevent overriding of the method mStatus(). It is for this purpose precisely that the keyword FINAL is provided, and the programmer can use it to protect the method mStatus() (e.g. METHOD FINAL mStatus()) from being overridden.

As regards the status of the drives, the programmers must agree that the information obtained from the feedback above must be stored in a status word (Status:WORD). A bit in the word is assigned to each individual status. Based on the maximum possible feedback, the following is thus defined:

- Drive is off – status bit 0
- Drive is on – status bit 1
- Drive is accelerating (positive speed change) – status bit 2
- Drive is decelerating (negative speed change) – status bit 3
- Setpoint speed reached – status bit 4
- Drive error – status bit 15
- Actual speed value – separate word
- Error number/error code – separate word

3.7.2 Class model for connecting different drives

With all the information that we have now collected, we can develop a class model with the abstract class CDrive and the derivations CDriveDirect, CDriveStarDelta and CDriveSpeedControl (Figure 40).

The basic structure in the abstract class CDrive is defined by the methods mOn(), mOff() and mStatus(). From this class it is possible to derive the three subclasses for the different drive components and the relevant programmers can then develop the program code for the derived classes. While this model prescribes a specific structure, it leaves the programmers sufficient freedom to implement their program code as they deem fit.

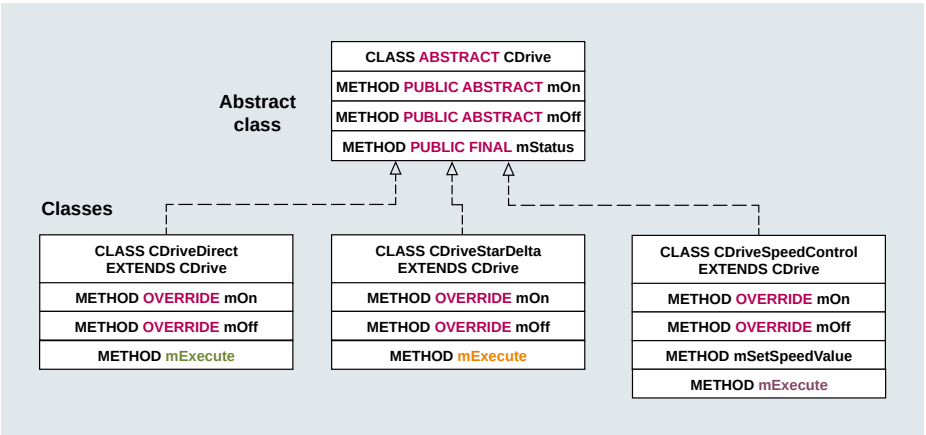


Figure 40 Class model CDrive

A method mExecute() is also assigned to each of the three classes. This allows connection of the drives with a different interface, but also ensures that superfluous interfaces are not programmed for any drive.

The implementation of the class model described here including the necessary classes and the actual program code of the methods mOn(), mOff(), mStatus() and mExecute() can be found in the program examples below.

3.7.2.1 Example of abstract class “CDrive”

The software designer supplies an abstract class “CDrive” for general use in each of the classes for the drives. The methods mOn(), mOff() and mStatus() with their interfaces are defined in this class. The methods mOn() and mOff() are defined as abstract methods in the class “CDrive”. The persons charged with programming the classes must write the appropriate program code for these two methods to suit their drive component. The method mStatus() is universally applicable to all drives and is fully programmed in the abstract class. It must be utilized, but not changed by the class users. For this reason, this method is identified with the keyword FINAL so that it cannot be overridden by other programmers.

```
// UNIT UDrive
INTERFACE
    CLASS CDrive;
END_INTERFACE

IMPLEMENTATION
    CLASS ABSTRACT CDrive
        VAR
            cboOff           : BOOL; // Status-WORD BIT0
            cboOn            : BOOL; // Status-WORD BIT1
            cboRampup        : BOOL; // Status-WORD BIT2
            cboRampdown      : BOOL; // Status-WORD BIT3
            cboSpeed_reached : BOOL; // Status-WORD BIT4
            cr64Speed_actual : LREAL;
            cr64Speed_value  : LREAL;
            cboFault         : BOOL; // Status-WORD BIT15
            cb16Errno        : WORD;
        END_VAR

        METHOD PUBLIC ABSTRACT mOn : VOID
        END_METHOD

        METHOD PUBLIC ABSTRACT mOff : VOID
        END_METHOD

        METHOD PUBLIC FINAL mStatus : WORD
            VAR_OUTPUT
                ActualSpeed : LREAL;
                Errno       : WORD;
            END_VAR

            IF cboOff = TRUE THEN
                mStatus.0 := TRUE;
            END_IF;
            IF cboOn = TRUE THEN
                mStatus.1 := TRUE;
            END_IF;
```

```
            IF cboRampup = TRUE THEN
                mStatus.2 := TRUE;
                mStatus.3 := FALSE;
                mStatus.4 := FALSE;
            END_IF;
            IF cboRampdown = TRUE THEN
                mStatus.3 := TRUE;
                mStatus.2 := FALSE;
                mStatus.4 := FALSE;
            END_IF;
            IF cr64Speed_actual = cr64Speed_value THEN
                mStatus.4 := TRUE;
                mStatus.2 := FALSE;
                mStatus.3 := FALSE;
            END_IF;
            IF cboFault = TRUE THEN
                mStatus.15 := TRUE;
                Errno      := cb16Errno;
            ELSE
                mStatus.15 := FALSE;
                Errno      := 0;
            END_IF;
            ActualSpeed := cr64Speed_Value;
        END_METHOD

    END_CLASS

END_IMPLEMENTATION
```

Variables for storing values have been created in the abstract class CDrive. All methods can access these variables which means that the data are not lost. If this option is not utilized, the data required for processing must be transferred to the relevant method but this involves too much work within a class.

3.7.2.2 Example of class for direct-on-line starting drives

The class for direct-on-line starting drives is now derived from the abstract class. This is the simplest implementation of all classes. To ensure that the necessary input and output signals for the relevant drive can be transferred, an additional method mExecute() with the appropriate interface is defined for the class. The method mExecute() is not defined in the abstract class CDrive because it would otherwise be necessary to specify the interface for the inputs and outputs for all derived classes and it would no longer be necessary to extend the method. The signature of methods must not be changed when they are overridden.

```
// UNIT UDriveDirect
INTERFACE
    USES UDrive;
    CLASS CDriveDirect;
END_INTERFACE

IMPLEMENTATION
    CLASS CDriveDirect EXTENDS CDrive

        METHOD PUBLIC OVERRIDE mOn : VOID // Override method mOn
            IF cboFault = TRUE THEN
                cboOn  := FALSE;
                cboOff := TRUE;
```

```
            RETURN;
            END_IF;
            IF cboOn = FALSE THEN
                cboOn  := TRUE;
                cboOff := FALSE;
            END_IF;
        END_METHOD

        METHOD PUBLIC OVERRIDE mOff:VOID // Override method mOff
            cboOn  := FALSE;
            cboOff := TRUE;
        END_METHOD

        METHOD PUBLIC mExecute : VOID // New method mExecute
            VAR_INPUT
                MotProtSwitch : BOOL;
            END_VAR
            VAR_OUTPUT
                Q_Kx : BOOL;
            END_VAR

            IF cboOn = TRUE THEN
                Q_Kx := TRUE;
            END_IF;

            IF cboOff = TRUE THEN
                Q_Kx := FALSE;
            END_IF;

            IF MotProtSwitch = FALSE THEN
                Q_Kx     := FALSE;
                cboOn    := FALSE;
                cboOff   := TRUE;
                cboFault := TRUE;
            ELSE
                cboFault := FALSE;
            END_IF;

            cboRampup   := FALSE;
            cboRampdown := FALSE;
            cb16Errno   := 0;

        END_METHOD
    END_CLASS
END_IMPLEMENTATION
```

3.7.2.3 Example of class for drives with star-delta starters

We are now going to derive subclass CDriveStarDelta from class CDrive in the same way as we derived subclass CDriveDirect. We will also implement a method named mExecute in this class, but in an extended form for this kind of drive. We can do this because mExecute() has not been defined in the abstract class CDrive.

```
// UNIT UDriveStarDelta
INTERFACE
    USES UDrive;
    CLASS CDriveStarDelta;
END_INTERFACE

IMPLEMENTATION
    CLASS CDriveStarDelta EXTENDS CDrive
```

```
VAR
    cFBDSDTimer : TON;
END_VAR

METHOD PUBLIC OVERRIDE mOn:VOID // Override method mOn
    IF cboFault = TRUE THEN
        cboOn  := FALSE;
        cboOff := TRUE;
    RETURN;
    END_IF;

    IF cboOn = FALSE THEN
        cboOn  := TRUE;
        cboOff := FALSE;
    END_IF;
END_METHOD

METHOD PUBLIC OVERRIDE mOff : VOID // Override method mOff
    cboOn  := FALSE;
    cboOff := TRUE;
END_METHOD

METHOD PUBLIC mExecute : VOID // New method mExecute
    VAR_INPUT
        MotProtSwitch : BOOL;
    END_VAR
    VAR_OUTPUT
        Q_Kx : BOOL; // Contactor main K1
        Q_Ky : BOOL; // Contactor star K2
        Q_Kz : BOOL; // Contactor delta K3
    END_VAR

    IF (cboOn = TRUE AND cboRampup = FALSE) THEN
        cboRampup := TRUE;
    END_IF;
    // start timer for switch to delta
    cFBDSDTimer (PT := T#900ms, IN := cboRampup);
    // switch star
    IF (cboOn = TRUE AND cFBDSDTimer.Q = FALSE) THEN
        Q_Kx := TRUE;
        Q_Ky := TRUE;
    END_IF;

    IF MotProtSwitch = FALSE THEN
        cboOn    := FALSE;
        cboOff   := TRUE;
        cboFault := TRUE;
    ELSE
        cboFault := FALSE;
    END_IF;
    // switch to delta if timer is expired
    IF cFBDSDTimer.Q = TRUE THEN
        Q_Kx := TRUE;
        Q_Ky := FALSE;
        Q_Kz := TRUE;
    END_IF;
    // switch off
    IF cboOff = TRUE THEN
        Q_Kx      := FALSE;
        Q_Ky      := FALSE;
        Q_Kz      := FALSE;
        cboRampup := FALSE;
    END_IF;
```

```
            // set non needed values
            cboRampdown := FALSE;
            cb16Errno   := 0;
        END_METHOD
    END_CLASS
END_IMPLEMENTATION
```

Important note!

When implementing methods in classes, it is vital to remember that the data of the method itself are stored in the SIMOTION CPU stack when the method is called. This means that the variables at the interface (VAR_INPUT, VAR_IN_OUT, VAR_OUTPUT, VAR) are transferred again or initialized with each method call. When the variables are reinitialized, they are effectively deleted and thus need to be set again in the method. If a memory that retains its status over several calls is required for a variable, a variable with memory must be set up in the class. Variables that retain their value between two method calls cannot be set up in a method.

3.7.2.4 Example of class for speed-controlled drives

The class for speed-controlled drives involves a slightly more complicated program than the two previous classes. This is due to the very nature of this kind of drive. A speed-controlled drive has a significantly broader range of functions than the other two simpler drive types.

There are many different variants of speed-controlled drive. They all possess different characteristics and thus need to be treated differently with regard to programming. The program should reflect the type of drive system used, its functional capabilities and the method by which it is connected to the control system. It is the drive manufacturer who determines the functional scope of a drive and the means by which it must be coupled with a controller. All the relevant information can be found in the documentation supplied by the drive system manufacturer.

A machine manufacturer will select the drives most suitable for the application in question and will wish to keep the number of different variants used within manageable limits. Because each variant requires different software and this software will need to be maintained accordingly over the entire service life of the drive. It makes economic sense to minimize maintenance requirements and this can best be achieved by limiting the number of drive variants.

SIMOTION is a motion control system with integral motion control and drive functions. For this reason, our example uses the functions for speed-controlled axes integrated in SIMOTION in the class CDriveSpeedControl that we are going to create. To help you understand the example program better, we are first going to explain some of the features of SIMOTION speed-controlled axes. You can find further information about handling the axes of the SIMOTION system from chapter 8.9.6 onwards.

Axis and drive functions are integrated in the SIMOTION Motion Control System. Using wizards in the engineering system, the user can create and configure “Technology Objects” (TO) with specific properties (Figure 41). A Technology Object “speed-controlled axis” represents, for example, a drive connected to the

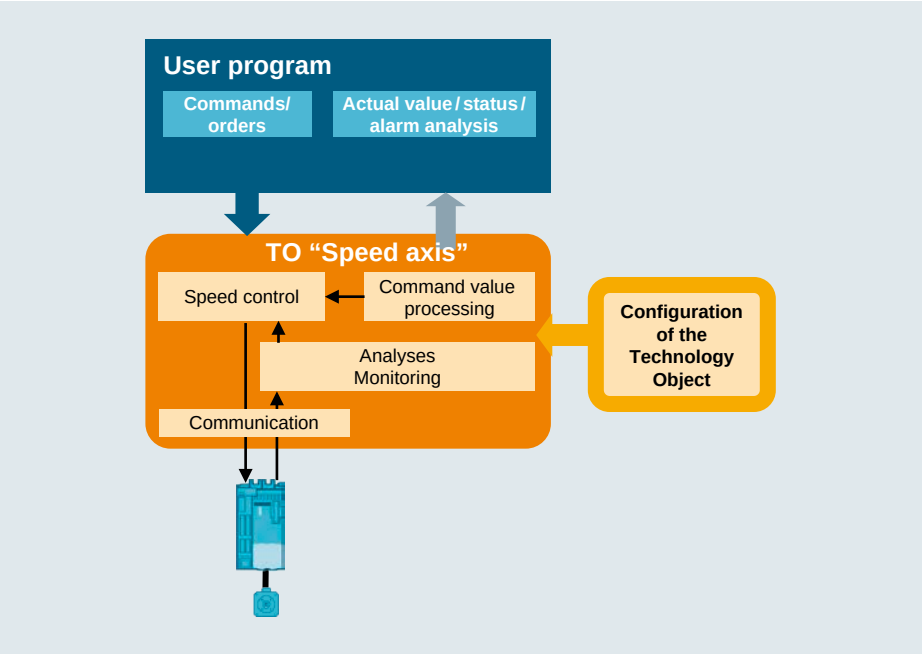


Figure 41 SIMOTION Technology Objects

SIMOTION system. The object contains all the relevant data including the specified means of communication to the drive. The object also contains an integrated closed-loop control system with additional monitoring functions. As well as issuing commands to the object, the user can also interrogate the current processing activity of the object via a function call interface.

For the purpose of controlling a speed-controlled axis, this kind of “driveAxis” (speed-controlled axis) Technology Object is set up in the SIMOTION system. One TO “driveAxis” must be defined for each drive installed in the machine. The user assigns a unique name to each TO by which it can be identified. The engineering system downloads the configured objects to the SIMOTION controller. Appropriate programs make the functions of the Technology Objects available to the user. To ensure that the class CDriveSpeedControl can make use of objects, the class first needs to be made aware of the object’s existence. This is achieved by transferring a reference to the TO. Even in our example of a valve program, we used a reference to transfer data to the HMI.

A variable `ctoRefAxis:DRIVEAXIS` is set up in the class CDriveSpeedAxis. The reference is actually transferred at the INPUT variable “RefAxis” in the method Execute(). It would also have been possible to implement the transfer in a separate method, e.g. SetRefAxis(). We are likewise assuming that the speed-controlled axis is continuously connected to our class. In order to change the axis during operation, we would need to implement additional actions in the program, but we have decided not to do this in the interest of keeping the example programs as simple as possible.

As a result, it is only necessary to call the method Execute() in order to make use of a speed-controlled axis. The process of switching on the speed-controlled axis is

programmed in two steps in the method Execute(). The axis enable signal must first be transmitted to the drive. This is done with function call _enableAxis(). Once the speed-controlled axis is enabled, it can be started with a speed setpoint by issuing the command _move() and passing suitable values to the parameters.

The user will wish to be able to input an appropriate drive speed for speed-controlled axes. For this reason, we have implemented a method mSetSpeedValue() in the class CDriveSpeedControl. By calling this method, the user can specify a speed setpoint at the interface. A new speed setpoint can be passed to the drive at any time, but it will not take effect until the axis is restarted.

The method Execute calls the function _stop() in order to stop the speed-controlled axis. Once it has been stopped, it can be restarted with a new speed setpoint (if one has been specified). As with the classes for the simpler drives, the speed-controlled drive is started and stopped by the methods mOn() and mOff(). Only movement with a positive speed has been programmed in this example.

```
// UNIT UDriveSpeedControl
INTERFACE
    USEPACKAGE Cam;
    USES UDrive;
    CLASS CDriveSpeedControl;
END_INTERFACE

IMPLEMENTATION
    CLASS CDriveSpeedControl EXTENDS CDrive
        VAR
            ctoRefAxis         : DRIVEAXIS; // Reference of TO
            cCommandID         : CommandIDType;
            csRetCommandState  : StructRetCommandState;
            cboLock            : BOOL;
        END_VAR
        // Method for setting a new speed value
        METHOD PUBLIC mSetSpeedValue:VOID
            VAR_INPUT
                Speed_value : LREAL;
            END_VAR

            cr64Speed_value := Speed_value;

        END_METHOD

        METHOD PUBLIC OVERRIDE mOn : VOID
            IF cboFault = TRUE THEN
                cboOn  := FALSE;
                cboOff := TRUE;
                RETURN;
            END_IF;

            IF cboOn = FALSE THEN
                cboOn  := TRUE;
                cboOff := FALSE;
            END_IF;
        END_METHOD

        METHOD PUBLIC OVERRIDE mOff : VOID
            cboOn  := FALSE;
            cboOff := TRUE;
        END_METHOD
```

```
METHOD PUBLIC mExecute : VOID
    VAR_INPUT
        MotProtSwitch:BOOL;
        RefAxis:DRIVEAXIS; // Reference for TO axis
    END_VAR

    VAR_OUTPUT
        Q_Kx : BOOL; // Contactor main K1
    END_VAR

    IF RefAxis = TO#NIL THEN
        RETURN;
    END_IF;

    IF RefAxis <> ctoRefAxis THEN
        ctoRefAxis := RefAxis;
    END_IF;

    IF cboOn = TRUE THEN
        Q_Kx := TRUE;
        IF RefAxis.control = INACTIVE THEN
            _enableAxis(
                axis          := ctoRefAxis
                ,enableMode   := ALL
                ,nextCommand  := IMMEDIATELY
                ,commandId    := _getCommandID()
                );
        END_IF;
    END_IF;
    IF (RefAxis.control = ACTIVE AND
        cboLock = FALSE AND
        cboOn = TRUE)
    THEN
        cCommandId := _getCommandId();
        _bufferAxisCommandId (axis := ctoRefAxis
                              ,commandId := cCommandId);
        cboLock := TRUE;
        _move(
            axis          := ctoRefAxis
            ,direction    := POSITIVE
            ,velocityType := DIRECT
            ,velocity     := cr64Speed_value
            ,nextCommand  := IMMEDIATELY
            ,commandId    := cCommandId);
    END_IF;
    IF MotProtSwitch = FALSE THEN
        IF cboFault = FALSE THEN
            _disableAxis(axis := ctoRefAxis
                         ,disableMode := ALL);
        END_IF;
        cboOn    := FALSE;
        cboOff   := TRUE;
        cboFault := TRUE;
    ELSE
        cboFault := FALSE;
    END_IF;

    IF (cboOff = TRUE AND cboLock = TRUE)
    THEN
        csRetCommandState := _getStateOfAxisCommand(
                             axis := ctoRefAxis
                             ,commandId := cCommandId);
```

```
                IF csRetCommandState.functionResult = 0 THEN
                    IF csRetCommandState.commandIdState = ACTIVE THEN
                        // RegisterdCommandId remove at TO
                        _removeBufferedAxisCommandId(
                        axis := ctoRefAxis
                        ,commandId := cCommandId);
                    END_IF;
                END_IF;
                _stop(axis              := ctoRefAxis
                    ,stopMode           := STOP_AND_ABORT
                    ,stopSpecification  := ALL_AXIS_MOTION
                    ,mergeMode          := IMMEDIATELY
                    ,nextCommand        := IMMEDIATELY
                    ,commandId          := _GetCommandId()
                    ,movingMode         := CURRENT_MODE);
                cboLock := FALSE;
            END_IF;

            IF cr64Speed_value = RefAxis.motionStateData.actualVelocity
            THEN
                cboSpeed_reached := TRUE;
                cboRampup        := FALSE;
                cboRampdown      := FALSE;
            ELSE
                cboSpeed_reached := FALSE;
            END_IF;

            IF cr64Speed_actual < cr64Speed_value THEN
                cboRampup := TRUE;
            END_IF;

            IF cr64Speed_actual > cr64Speed_value THEN
                cboRampdown := TRUE;
            END_IF;

            cr64Speed_actual := RefAxis.motionStateData.actualVelocity;
            cb16Errno := 0;
        END_METHOD
    END_CLASS
END_IMPLEMENTATION
```

3.7.2.5 Example program for controlling drives of different types

Now we have developed all the classes we need for the different types of drive, we can create and use objects based on the classes. To illustrate the use of classes, we have created and used an object for each class in this example.

```
// UNIT UProgramCallDrives
INTERFACE
    USEPACKAGE Cam;
    USES UDriveDirect, UDriveStarDelta, UDriveSpeedControl;
    PROGRAM pCallDriveObjects;
END_INTERFACE

IMPLEMENTATION
    PROGRAM pCallDriveObjects
        VAR
            DD1          : CDriveDirect;
            DSD1         : CDriveStarDelta;
            DSC1         : CDriveSpeedControl;
```

```
    StatusDD1     : WORD;
    StatusDSD1    : WORD;
    StatusDSC1    : WORD;
    DD1ActSpeed   : LREAL;
    DSD1ActSpeed  : LREAL;
    DSC1ActSpeed  : LREAL;
    DD1Errno      : WORD;
    DSD1Errno     : WORD;
    DSC1Errno     : WORD;
    iboIN1        : BOOL; // from here these are normally I/Os
    iboIN2        : BOOL;
    iboIN3        : BOOL;
    iboIN4        : BOOL;
    iboIN5        : BOOL;
    iboIN6        : BOOL;
    iboIN7        : BOOL;
    iboIN8        : BOOL;
    iboMS1        : BOOL;
    iboMS2        : BOOL;
    iboMS3        : BOOL;
    qboK1         : BOOL;
    qboKx1        : BOOL;
    qboKy1        : BOOL;
    qboKz1        : BOOL;
    qboK2         : BOOL;
END_VAR

// Switch on DriveDirect
IF iboIN1 = TRUE THEN
    DD1.mOn();
END_IF;
// Switch off DriveDirect
IF iboIN2 = TRUE THEN
    DD1.mOff();
END_IF;
// Call Execute of DriveDirect
    DD1.mExecute(MotProtSwitch := iboMS1, Q_Kx => qboK1);

// Switch on DriveStarDelta
IF iboIN3 = TRUE THEN
    DSD1.mOn();
END_IF;
// Switch off DriveStarDelta
IF iboIN4 = TRUE THEN
    DSD1.mOff();
END_IF;
// Call Execute of DriveStarDelta
DSD1.mExecute(MotProtSwitch := iboMS2
            ,Q_Kx => qboKx1
            ,Q_Ky => qboKy1
            ,Q_Kz => qboKz1);

// Switch on DriveSpeedControl
IF iboIN5 = TRUE THEN
    DSC1.mOn();
END_IF;
// Switch off DriveSpeedControl
IF iboIN6 = TRUE THEN
    DSC1.mOff();
END_IF;
// Set different speed value
IF iboIN7 = TRUE THEN
    DSC1.mSetSpeedValue(200.0);
```

```
        ELSE
            DSC1.mSetSpeedValue(100.0);
        END_IF;
        // Call Execute of DriveSpeedControl
        DSC1.mExecute(MotProtSwitch := iboMS2
                     ,RefAxis := SpeedAxis_1
                     ,Q_Kx => qboK2);

        // Read status of all drives
        IF iboIN8 = TRUE THEN
            StatusDD1:=DD1.mStatus(ActualSpeed => DD1ActSpeed
                                  ,Errno => DD1Errno);
            StatusDSD1:=DSD1.mStatus(ActualSpeed=>DSD1ActSpeed
                                    ,Errno=>DSD1Errno);
            StatusDSC1:=DSC1.mStatus(ActualSpeed => DSC1ActSpeed
                                    ,Errno=>DSC1Errno);
        END_IF;
    END_PROGRAM
END_IMPLEMENTATION
```

Note: To ensure that this example is executable, the program must be assigned to the BackgroundTask in the execution system of the SIMOTION CPU. For instructions on how to assign programs in the execution system, see chapter 8.9.13.

If the classes are programmed in different units, a link to the units of the classes must be set up in the interface section of the unit. Keyword USES <*UnitName*> is used to create this connection.

When the drive status is queried using the method Status, various values such as ActualSpeed, for example, are output as “0” for the simpler drives. These drives operate at the speed that corresponds to the number of pole pairs in the drive motor. Since the object is not aware of this information, the output value “0” has been programmed. This information is available as a variable for speed-controlled drives. The program transfers this information to the variable cr64Speed_actual.

3.8 Abstract class versus interface

If we look at the definition of an abstract class and compare it to the definition of an interface, we will find that both constructs are very similar. Both constructs contain prototype methods for which the appropriate functions must be programmed in a class.

Programmers therefore ask themselves with some justification: what is the difference between these constructs and, more importantly: which of them should I use for my specific application?

To be able to answer this question more simply, the relevant properties are listed in Table 4.

By defining prototypical methods, both constructs (abstract class and interface) represent a declaration guaranteeing that the interfaces defined at the methods will be used exactly as defined. A programmer can thus feel confident that the use (call)

Table 4 Comparison between abstract class and interface

Abstract class	Interface
Abstract classes can contain abstract methods as well as fully programmed methods, i.e. real methods.	An interface may contain only prototype methods (keyword ABSTRACT does not need to be specified).
Abstract classes can possess defined properties (attributes).	An interface cannot possess any properties (attributes).
The properties of methods can be selected (PUBLIC, PRIVATE, PROTECTED).	All methods in the interface are PUBLIC and cannot be changed retrospectively.
An abstract class cannot be instantiated.	Interface variables can be created.
The methods of an abstract class must be fully programmed when they are derived unless the derivation itself is also ABSTRACT.	All the methods of an interface must be programmed in the class in which the method is implemented, or identified as abstract.
A class can be derived from an (abstract) class.	An interface can be derived from a base interface.
An (abstract) class can implement multiple interfaces.	_

of prototype methods will function reliably when the entire program is finished and the individual program sections are joined together.

An experienced programmer once gave us an answer to the question above. I liked it very much and think it could serve as a useful guide for use of these two constructs. This is what he said:

"An abstract class constitutes a contract for use in the derivation chain, i.e. within the class hierarchy, while the interfaces constitute a contract that is valid externally between different program sections (see valve: example with interface). As a programmer, however, you must remember that this clear distinction does not always exist in every programming scenario and you might be able to use either construct in a given situation. As so often in life, nothing is ever one hundred percent clear."

From the information we have given you, you should find it easier to decide which of these solutions is best suited for a specific task.

- Whenever a more generalized definition needs to exist between different programs and the methods may/must be accessible to anyone, you should use interfaces.
- An abstract class is the better option in cases where you need to define a structure for a program area and/or methods must be concealed (i.e. must not be PUBLIC).

If you find yourself in a situation where you cannot work out or predict the right choice, you still have to make a decision as to which construct to use. Once you have made a decision and start programming your application, you may discover later on that you made the wrong choice. By this time, you will have generated a load of program code and possibly created many derived classes. In such situations, you might not be able to avoid refactoring, i.e. the process of converting interfaces to abstract classes or vice versa. You will not always find it easy to make this decision and you will need to weigh up the pros and cons of both options. But it should comfort you to remember that it is significantly easier to refactor object-oriented programs than procedural programs.

Abstract classes can be combined very effectively with interfaces in object-oriented software and thus provide a wonderful tool for planning and designing the software structure. The option of being able to write all the program code for some methods in abstract classes reduces the time and effort involved in writing program sections that can be used by everyone in the same way. The methods are simply inherited when real classes are derived. The specification of access rights to methods enhances the security of programs.

Interfaces allow different program sections to be connected in a highly flexible manner and make it possible for software development teams to work more independently of one another.

3.9 OOP opens up the world of design patterns

By using the object-oriented programming method, programmers and in particular software designers will discover new opportunities for "optimizing" the solutions that they are trying to create. Since OOP has been long established as a method for programming PCs, many developers have devised meaningful, pre-defined solutions for a variety of approaches. The authors Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides have collected and published 23 such solutions in their book *Design Patterns. Elements of Reusable Object-Oriented Software*. This work by the so-called "Gang of Four" is a highly influential work in the field of software engineering.

These design patterns[4] provide tried-and-tested templates for solving recurrent design problems that are encountered by software architects and developers. Design patterns describe solutions for a variety of repetitive software development tasks.

Design patterns are thus reusable templates for resolving problems in various contexts. The descriptions are formulated neutrally, i.e. without reference to a specific programming language, but are based on the mechanisms available with object-oriented programming. In the interests of clarity, some templates contain sample code for specific programming languages, But this sample code can easily be converted to other languages. How a solution is actually implemented in the software is of secondary importance. What matters with design patterns is that they offer an identical potential solution irrespective of the language.

The use of design patterns is meaningful only if the software meets certain conditions:

- The problem to be solved must not be trivial in nature, i.e. it must not be a problem that is simple to deal with.
- The problem must recur frequently enough that it is worth taking the time to learn the use of design patterns.
- It must be a problem that can be solved by design patterns.
- The purpose of design patterns is to solve specific problems. In other words, use of a design pattern must ultimately result in executable, usable

[4] "Software design pattern". In: Wikipedia – The Free Encyclopedia. Revision level: December 21, 2015. 16:50 UTC. URL: https://en.wikipedia.org/wiki/Software_design_pattern (viewed on: March 22, 2016, 17:25 UTC)

program code. Design patterns do not address general software design problems.

- It must be possible to adapt the design pattern for the automation engineering solution. There is no point, for example, in basing a solution on design patterns which necessitate object generation in the control system during runtime.

As OOP becomes more widely established in the field of automation engineering, programmers will have the opportunity to use design patterns, although their use is not restricted to object-oriented programming. Patterns can be implemented in any high-level language. The target language largely determines how convenient the conversion process will be. Owing to the fact that the generation or destruction of objects during runtime does not exist in the OOP implemented in SIMOTION, patterns that are based on mechanisms of this kind (creation patterns) cannot be transferred to SIMOTION.

If somebody else has already found an elegant solution to a problem, it makes complete sense to simply adopt this solution to solve your own problem. It takes, of course, a certain level of expertise to evaluate design patterns and their application. Since many design patterns are based on OO mechanisms, it is important that programmers are familiar with the object-oriented programming tool kit so that they can understand and transfer code examples.

After this short discussion about design patterns, we are going to close this chapter. We just wanted to draw your attention to another useful mechanism for creating reusable software. Anyone who wishes to delve deeper into the subject of design patterns will find that a large number of books has been written about the topic.

4 OOP Supports Modular Software Concepts

One of the big advantages of object-oriented programming is that it allows programmers to create consistently modular software. Thanks to the "object concept", it is possible to implement fully encapsulated modules which can be influenced externally only in the ways expressly permitted by the programmer. The software developer implements a set of base objects as well as a series of more complex objects. The complex objects obtain their functionality from the finished base objects, in other words, they are aggregations of these base objects. Aggregating different combinations of base objects makes it possible to program exactly the right module for any task. The underlying software principles are always the same, and the time and effort required to test newly developed modules can therefore be reduced.

The modules must be capable of functioning independently. It is thus vital that their functional capability is tested so as to ensure that they can be used in different settings (machines). This is done by integrating the modules into specific test environments that are designed to simulate the typical conditions in the relevant environment and deliver appropriate results. Object-oriented programming makes this whole process easier by providing encapsulated modules capable of independent functioning which can be tested in relatively simple test environments.

After development and testing, modules can be combined easily to implement all the tasks required to operate a plant. As a consequence, software for machines can be developed much more quickly.

So far, so good – it all sounds pretty straightforward. But if we look at the software that we designed in the past and attempt to achieve this level of simplicity with these designs, we are suddenly confronted with a myriad of different questions.

- How does the software need to be designed so that it is easy to combine different modules?
- How is it possible to aggregate modules, i.e. pre-existing class objects, without violating their independence?
- How can different modules be combined without necessitating the addition of further program code for the connection?
- Is it possible to use engineering tools to automate the process of combining existing modules?

These different interrelated issues – module design, interconnection of modules or automated combination of modules – are not directly related to object-oriented programming. But one of the major advantages of this programming method is that it gives valuable help to programmers who want to achieve these goals. For this reason, we will explain these issues in more detail in the following chapters.

4.1 Assembling projects for real machines

A major challenge faced by any company is to maintain its own competitive position. As a general rule, there are precisely two options for doing this:

1. If a company is building unique machines that are not produced by any other manufacturer, it can regard itself as a technology leader and does not have the immediate problem of market price pressure. This kind of manufacturer can naturally only remain successful if there is a market demand for the special machine functions they are offering. But the laws of the market also dictate that the company cannot maintain this leadership role for long. Their competitors will attempt to reproduce the functional capabilities of the machine or even copy the entire machine. If the company's competitors manage to break its role as market leader, the machine manufacturer will attach more importance to the second option. As a general rule, machine manufacturers can maintain their market leadership only through a process of sustained innovation.
2. If there are various machine manufacturers on the market offering comparable machines, each of them will be subject to a certain price pressure. In other words, they must each offer their machines at a competitive price in order to be able to sell them at all. To do this, each manufacturer must keep costs under control and will also attempt to minimize them.

These two scenarios admittedly represent the extreme ends of a range of possibilities. Generally speaking, machine manufacturers and their machines will find themselves somewhere in-between. But each will be working to achieve the position described under 1. above. They will attempt to differentiate themselves from their competitors by integrating innovative features into their products. They can also set themselves apart by offering additional benefits such as, for example, special quality products, excellent service, spare parts supply, etc. But machine manufacturers also make continuous efforts to keep their own costs as low as possible. Thus, all machine manufacturers aim to achieve the targets described under 1. and 2. above. Designing innovative product features requires new developments and therefore costs money, But these costs also need to be reduced or minimized at the same time.

That is why many companies analyze their processes. Developing software for a fully operational machine is a time-consuming, complex and thus cost-intensive process. The software component in machinery is steadily increasing and the cost of software development is thus also rising as a proportion of overall costs. It is therefore a natural step to attempt to find cost savings in the software design budget. The quickest possible way of achieving cost savings would of course be to fire some (or even all) of the software developers. But any company that did this would deprive itself of the means of innovating its products and would thus enjoy only short-term success. After a brief interlude, it would disappear from the market altogether. Analyzing software development processes is certainly worthwhile, however, and will generally reveal various ways in which costs can be cut.

If a software design is modular in structure and the modules can be handled as independent entities, they can be used multiple times for a single machine in a similar way to mechanical components. Machine components are technically designed so that they can be combined in different ways with other components, i.e. adapted for specific purposes. This is exactly what we need to do with software modules as

well. They need to be designed with interfaces that can be freely assembled and freely combined. The software design itself thus makes an important contribution to reducing the cost of software development.

4.1.1 Module design

Unsuitable software designs increase the labor and costs involved in compiling a machine application because the work involved in adapting software to each individual machine is considerable. Even duplicate machines are rarely one hundred percent identical. As the number of machines increases, so too does the diversity of software versions adapted to match the features of each individual unit. The time and labor involved in maintaining the software steadily increase over its lifetime and thus tie up an ever growing share of development capacity.

The only way to avoid these problems is to keep the number of software variants to within manageable limits. To achieve this, the machine software must be modular in structure and the software modules designed such that they can be reused as often as necessary in their original form. At first glance, this objective might seem to conflict with the stated desire to integrate innovative machine features because this will obviously involve modification of the software. But it is precisely in this respect that object-oriented programming is such a useful concept because it allows us to retain fully functional software components while allowing us to adapt the software using the inheritance mechanism (see also chapter 5.4.2 “Software needs to be planned”). The module interfaces should be designed such that parts of modules or even complete modules can be exchanged.

4.1.2 The role of the software developer

Software planning and design of a modular software concept are the essential prerequisites for compiling a fully functioning machine application easily and without the need for complex programming. In an ideal situation, appropriate tools can even be used to assemble the application automatically. Creating a machine application automatically offers a number of different benefits that can lead to significant cost savings.

- If the automatic compilation tool has a front end designed to facilitate operation, the application does not necessarily need to be compiled by a software programmer. Appropriately trained personnel can carry out this work independently and so ease the burden on the software development department.
- Automatic compilation of the machine application is easy to repeat and always delivers the same results. Careless mistakes and operating errors do not generally occur in the way they do with applications assembled manually from different projects.
- Compiling an application with an automatic tool takes very little time. Even if the compilation work does need to be carried out by software development personnel, the amount of time they need to spend on this important, if rather dull, task is still significantly reduced. The most important task of a programmer is to create high-quality, error-free and reusable software.

Good software developers are not cheap and they are sometimes headstrong. It is also not true that these people can simply be replaced, even if this fallacy still seems to be deeply entrenched in the minds of today's management professionals.

A study carried out by Sackman, Erikson and Grant[6] in 1968 analyzed the productivity of software developers. The results revealed a large difference in productivity (10:1) between the best and worst test persons. The fastest developers achieved the shortest execution times with the least amount of program code. Another result worthy of consideration was the fact that productivity levels were not necessarily linked to professional experience.

In short, you could say that a good developer is worth more than ten poor ones. Every possible measure should therefore be taken to ensure that good developers are used as effectively as possible. Anything that gets in their way of doing their actual job (such as invitations to attend unnecessary meetings or burdening them with superfluous tasks) must be prevented at all costs.

If a good software developer is creating effective program code, then he or she will also be capable of developing a useful software concept. He or she will construct the programs in such a way that they contain only a few copied program sections or none at all. A specific task will be performed by only one program module (rather than by a number of similar modules). This approach makes the programs easier to maintain and test (in this case, they are modular in structure anyway).

But let's get back to our original point. Our main objective is to compile the software for a machine in the most effective manner so that we can also cut costs. To achieve this goal, our software must have a modular structure. This work is done by software developers who are capable of designing and programming reusable software.

But demands on the mechanical engineering design may also arise from the software development process. While mechanical engineering is significantly more advanced than software development with respect to standardization, selection of the wrong gear ratio, failure to install limit switches or use of overly cheap encoders can make complicated software solutions necessary. When mechanical engineering changes are easy to implement, the associated software programs will also be significantly simpler. This approach can help to avoid, or at least minimize, future problems. Only when the hardware and software in a machine are mutually coordinated can an optimum outcome be achieved. Mechanical engineering and software development personnel must therefore cooperate closely and so achieve their common objective.

4.1.3 Modularizing software

All software ultimately executes on a control system, i.e. it needs to be incorporated into a runtime environment so that the control system can execute the programs. Since the control systems available on the market behave differently with respect to program execution, the controllers to be used need to be specifically selected. Even if machine manufacturers would like a situation where they could easily replace the control system but keep the same software, the fact that control systems operate in different ways forces them to make a selection.

[6] http://cartoon.iguw.tuwien.ac.at/fit/fit02/CPT_Motivation_Bsp_details.html (Viewed on: March 22, 2016, 16:34 UTC)

Moreover, the software also requires an interface to the I/O devices connected to the control system (such as inputs and outputs or bus-connected components (e.g. servo converters)). A software module therefore has an interface via which it can be connected to the necessary I/O components. It is a capital offence to implement software modules that have direct access to I/O components (e.g. inputs). The software programming MUST be independent of the hardware. If the module needs to be used more than once in the control system, the I/O mapping concept takes on a vital role (see also chapters 3.5.9 and 6.1). This may very well have repercussions as regards planning the control system I/O components and thus the design of the control cabinet.

There is normally a section in the software that coordinates the execution of individual modules. This software section generally needs to be adapted individually for each machine. The extent to which it requires adaptation depends on its generic design. If message exchange between software modules is appropriately modeled and implemented, it can minimize the work involved in adapting the software. Since the existing machine software may already be modularized to a certain degree, thought has to be given to ways in which it can be integrated into an overall concept without necessitating adaptation of the entire software application. This could be done effectively by incorporating a neutralizing software layer into the existing modules that allows all of them to communicate in the same way with higher-level software components. The interfaces described above therefore represent the best means of design to achieve this end. The individual modules can then be gradually optimized and improved as separate modules. The standard ISA-88 offers an excellent definition of suitable models[7] for structuring processes in a machine control system (Figure 42).

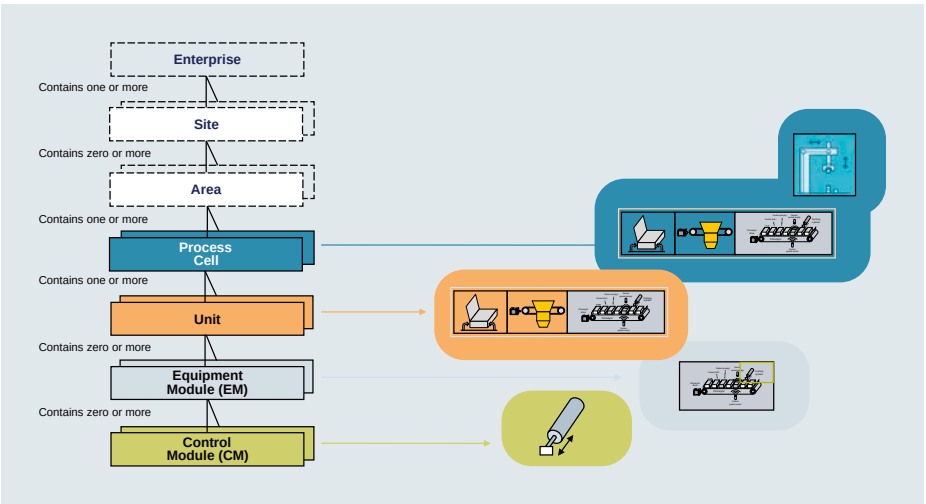


Figure 42 Hierarchy as defined by ISA-88-01

[7] "ISA-88". In: Wikipedia – The Free Encyclopedia. Revision level: January 9, 2016, 21:54 UTC. URL: https://de.wikipedia.org/w/index.php?title=ISA-88&oldid=150007795 (viewed on: March 22, 2016, 16:34 UTC)

This standard defines a design philosophy for software, equipment and procedures. The proposed models break manufacturing processes down into process stages and then further down into individual functions. The standard defines “equipment modules” (among others) that are fully functional as independent modules and capable of reacting to and answering messages. Thus, an equipment module must be capable of carrying out minor processing activities. Equipment modules are in turn comprised of smaller functional units referred to as “control modules”. We could regard a control module as being, for example, objects for the valve-cylinder combination or drive objects (see previous chapter). Several control modules of this kind can be combined in an equipment module to perform a specific function within a machine. Other important functions such as error diagnosis software or an operating mode manager could also take the form of an equipment module.

A “process cell” consists of machines or “units”. In Figure 42, the process cell comprises the units “packaging machine” and “stacking unit”. The packaging machine contains various equipment modules such as box erector, filler and ejector. The equipment module “ejector” is in turn comprised of various control modules.

This kind of modular design concept is also reflected in the software and can be combined by simple mechanisms to create different machine variants. If the engineering environment provided for a control system also offers suitable configuring tools, the operation of combining modules can even be automated and so help to make the process of creating plant software very efficient. There are some machine manufacturers who use this method to automatically compile more than 80 percent of their machine software. Depending on the amount invested, this level of automation could be increased up to 100 percent. As always, a cost/benefit analysis needs to be carried out in order to determine the best approach.

4.1.3.1 Creating equipment modules

We have now reached the point at which enough theoretical considerations and opinions have been presented. Let’s now take a look at how we can actually create these kinds of reusable module. We are of course going to use the same examples as those we worked through earlier in this book. In previous chapters, we created classes for valve-cylinder combinations and drives. These are suitable for combining as modules for use in a machine.

We want to highlight all the relevant principles here and, for this reason, we have only developed the most important elements in the following example. For a real machine application it would be necessary to program other elements such as, for example, interruptions to the process or detailed error diagnosis in the case of component failure. As we have already said, this example is provided to illustrate the principle of modularization.

A conveyor belt with an ejector function for unfilled packages will be used as a machine module in a plant. The machine module must ensure that only full packages arrive at the next section of the machine. The packages are transported through this section of the machine on a conveyor belt. A simple three-phase asynchronous motor will be used to drive the conveyor belt. Each package is detected by a sensor as it arrives. A second sensor determines whether the package is full or empty. Empty packages are ejected to the side by a cylinder.

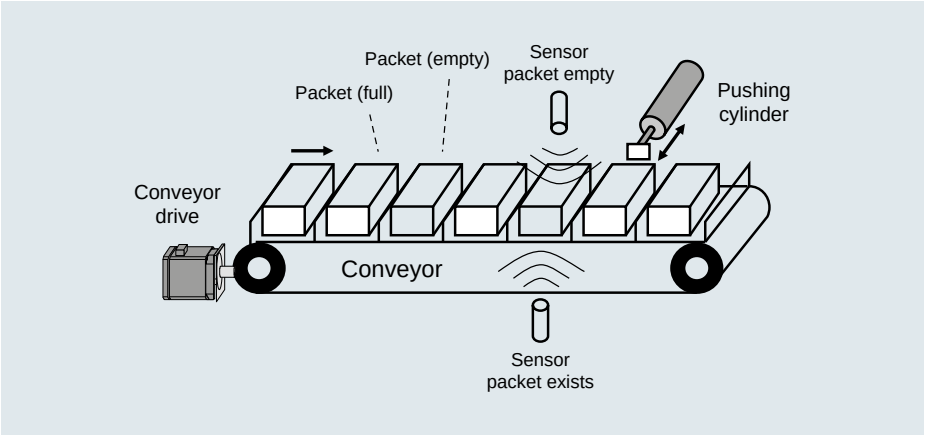


Figure 43 Equipment module for conveyor belt with ejector

Since it is important that the software is reusable, we can also use an object of the CDriveDirect class (chapter 3.7.2.2) as the conveyor belt drive. An object of the ValveControl43ST class will perform the ejector functions (chapter 3.6.2.2).

Another important component is still missing. One characteristic of the equipment module is that it is capable of receiving requests or commands and processing them independently. In other words, we need to create a software module that represents the equipment module and in which the two control modules “drive” and “ejector” are integrated. The module also has the interface function to at least one higher-level module and must execute the internal sequence independently.

It must be possible to start the equipment module in automatic mode. The start command is supplied by a higher-level instance and is not implemented as a hardware button. The conveyor belt must start up automatically in response to the start command. If the sensors detect an empty package, it must be ejected automatically. When the machine is switched from automatic mode to manual mode, any action that has already started (e.g. eject) will be completed and the conveyor belt then stopps. If a package is not ejected, the conveyor belt will stop when the sensor “package present” outputs a signal. This is to prevent intermediate positioning of packages that could hinder the ejection process when the conveyor belt restarts.

In manual mode, the conveyor belt is started by means of a pushbutton and stops again when this button is released. In this operating mode, the ejection cylinder can also be moved forwards and backwards by means of pushbuttons. It must not be possible to move the ejection cylinder by the pushbuttons if the package position is ambiguous (intermediate position) irrespective of whether or not the belt is moving.

4.1.3.2 Software design of the equipment module

With the information available to us, we can now develop a software design for the equipment module.

The task of managing the equipment module (EM) will be carried out by an object which is defined in a new class “CEMPusher” (Figure 44). This management module

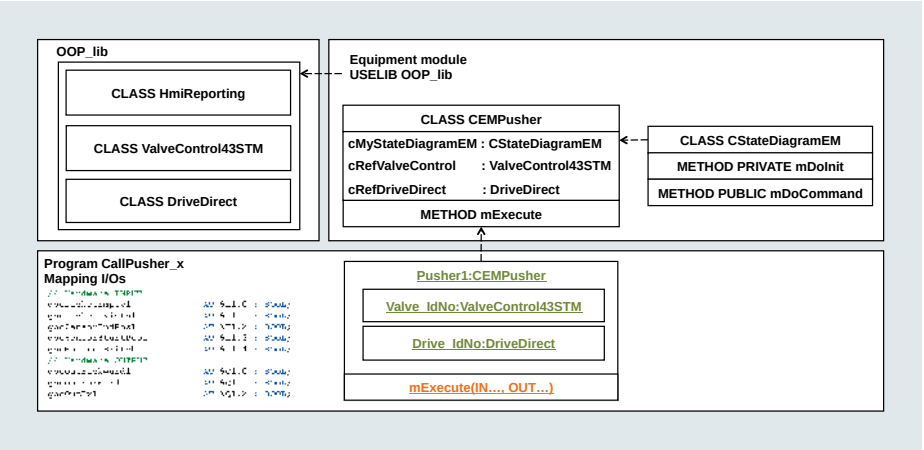


Figure 44 Software design of the equipment module

contains a state machine, but also has the task of organizing changeover between automatic and manual. All commands are transferred via the CEMPusher. It thus acts as the interface to higher-level modules and also delivers the status information required by these modules.

The objects for a valve and a direct-on-line starting drive are integrated as control modules in the equipment module. Since these control modules are based on their own classes, these classes have been swapped to a library (OOP-lib). These classes are therefore easy to reuse in different programs and thus also in our equipment module. We have made use of this option to integrate objects from these classes directly into our management module (CEMPusher). As a result, the objects for valve and drive are now integral components of the class CEMPusher. The higher-level management module transfers all information to the lower-level control modules and passes feedback upwards.

With this aggregation, however, it is important to note that a dependency exists between the class CEMPusher and the classes CDriveDirect and ValveControl-43STM. This fact must be taken into account when these two classes are modified or extended. Changes to the interfaces of the methods of the classes CDriveDirect and ValveControl43STM will necessitate a change to the class CEMPusher. But CEMPusher will automatically inherit any compatible changes or debugs. It is precisely this flexibility that is the main benefit of object-oriented programming. By deriving subclasses from these classes, they can be modified or extended without impairing the functionality that they currently possess.

The class CEMPusher is stored in its own unit “Equipment module” and utilizes the OOP_lib. The actual use of the class CEMPusher is programmed in turn in the unit UEMPusher, including generation of the object for the equipment module, definition of the required references and setup of the connection to the relevant I/O devices. Now that this structure is in place, the equipment module is ready for testing.

To ensure that the programs of the equipment module can be executed in a CPU, the main program of the module must be integrated in the execution system of the SIMOTION CPU. This is normally done by a program (e.g. pCallPusher1) that con-

tains the required variable declarations and the calls for all other software modules. It is logical to map the I/Os at the variable declaration of this program. The unit can now be reused by copying it with the program. It needs to be clarified once again here that this copy of the PROGRAM pCallPusher1 merely serves to make another object of the equipment module and the required I/O connection possible. The program code is not copied because it is precisely this that OOP is designed to prevent. In order to maintain the unique identity of each module, the program name (e.g. pCallPusher2) must be changed and the declarations adjusted accordingly in the copies. The program thus assumes the function of a template (see chapter 4.1.3.4).

4.1.3.3 Example of the class "CEMPusher"

The class CEMPusher organizes every aspect of the equipment module's behavior. The required valve-cylinder combination and the drive are integrated. This class also contains an error diagnostic function with reporting to a display system. The objects integrated in the class also need a reference to this object in the form of the interface to the HMI report. Interface variables are generally used for this purpose. The current implementation is transferred to the interface variables in the class during initialization of the class CEMPusher. The class CEMPusher transfers these class references to its internal objects.

The management module CEMPusher uses a state machine CStateDiagramEM to process sequences. The states required (Figure 45) for the equipment module are defined in this machine.

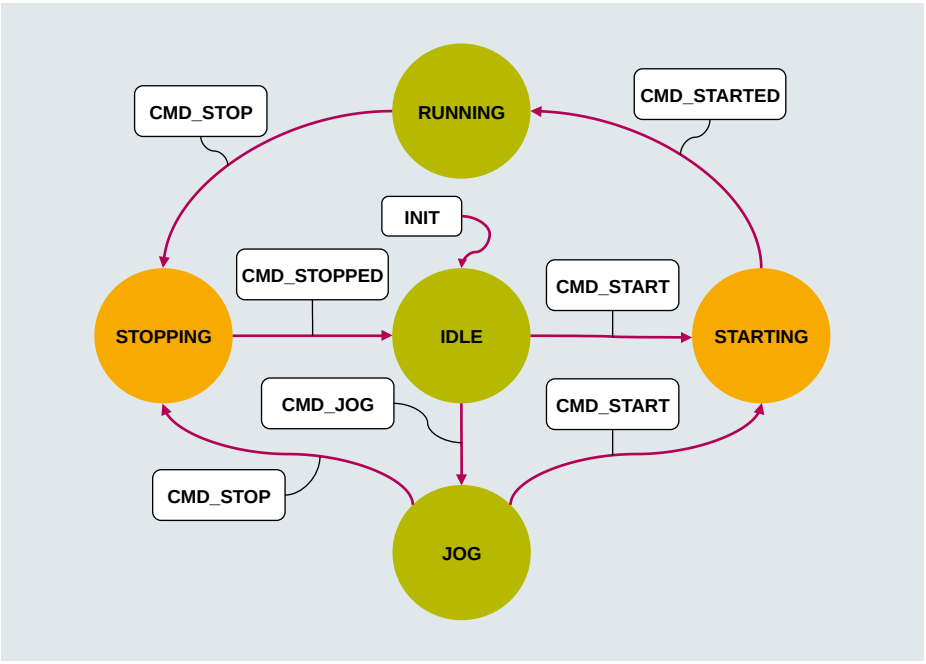


Figure 45 States of the equipment module

```
INTERFACE
    USELIB OOP_lib;
    CLASS CEMPusher;

END_INTERFACE
IMPLEMENTATION
    CLASS FINAL CStateDiagramEM
        TYPE PUBLIC
            // states equipmentmodule
            eStateType      : (IDLE
                               ,STARTING
                               ,RUNNING
                               ,STOPPING
                               ,JOG);

            // transitions equipmentmodule
            eTransitionType : (CMD_START
                               ,CMD_STARTED // only for internal
                                            // transition
                               ,CMD_STOP
                               ,CMD_STOPPED // only for internal
                                            // transition
                               ,CMD_JOG);
        END_TYPE
        TYPE PRIVATE
            sTransitionEntryType : STRUCT
                    eActState      : eStateType;
                    eTransition    : eTransitionType;
                    eNextState     : eStateType;
                END_STRUCT;
        END_TYPE
        VAR CONSTANT PRIVATE
            // the entries in which table must be
            // sorted by member act -> state_type!
            AS_STATE_TABLE : ARRAY [0..6] OF sTransitionEntryType := [
                (eActState := eStateType#IDLE,
                    eTransition := eTransitionType#CMD_START,
                    eNextState := eStateType#STARTING),
                (eActState := eStateType#IDLE,
                    eTransition := eTransitionType#CMD_JOG,
                    eNextState := eStateType#JOG),
                (eActState := eStateType#JOG,
                    eTransition := eTransitionType#CMD_START,
                    eNextState := eStateType#STARTING),
                (eActState := eStateType#JOG,
                    eTransition := eTransitionType#CMD_STOP,
                    eNextState := eStateType#STOPPING),
                (eActState := eStateType#STARTING,
                    eTransition := eTransitionType#CMD_STARTED,
                    eNextState := eStateType#RUNNING),
                (eActState := eStateType#RUNNING,
                    eTransition := eTransitionType#CMD_STOP,
                    eNextState := eStateType#STOPPING),
                (eActState := eStateType#STOPPING,
                    eTransition := eTransitionType#CMD_STOPPED,
                    eNextState := eStateType#IDLE) ];
        END_VAR
        VAR PRIVATE
            ceActState       : eStateType := eStateType#IDLE;
            // actual state
            cboInitialized   : BOOL := FALSE; // to see if optimization
                                              // table is initialized
```

```
        cai32LookupTable : ARRAY [ENUM_TO_DINT(eStateType#MIN)..
                                  ENUM_TO_DINT(eStateType#MAX)] OF DINT;
    END_VAR

    METHOD PRIVATE mDoInit
        VAR
            i          : DINT := LOWER_BOUND(AS_STATE_TABLE);
            eLastState : eStateType := eStateType#MIN;
        END_VAR
        cai32LookupTable[ENUM_TO_DINT(eLastState)] := i;
        WHILE (i <= UPPER_BOUND(AS_STATE_TABLE)) DO
            IF (AS_STATE_TABLE[i].eActState <> eLastState) THEN
                eLastState  := AS_STATE_TABLE[i].eActState;
                cai32LookupTable[ENUM_TO_DINT(eLastState)] := i;
            END_IF;
            i := i + 1;
        END_WHILE;
        cboInitialized := TRUE;
    END_METHOD

    METHOD PUBLIC FINAL mDoCommand : eStateType
        VAR_INPUT
            cmd : eTransitionType;
        END_VAR
        VAR
            i : DINT:= LOWER_BOUND(AS_STATE_TABLE);
        END_VAR

        IF NOT cboInitialized THEN
            mDoInit();
        END_IF;

        i := cai32LookupTable[ENUM_TO_DINT(ceActState)];
        WHILE (i <= UPPER_BOUND(AS_STATE_TABLE)) DO
            IF (AS_STATE_TABLE[i].eActState = ceActState AND
                    AS_STATE_TABLE[i].eTransition = cmd) THEN
                ceActState := AS_STATE_TABLE[i].eNextState;
                EXIT;
            END_IF;
            IF ceActState < AS_STATE_TABLE[i].eActState THEN
                EXIT;
            END_IF;
            i := i + 1;
        END_WHILE;
        mDoCommand := ceActState;
    END_METHOD

    METHOD PUBLIC FINAL mGetState : eStateType
        mGetState := ceActState;
    END_METHOD
END_CLASS

CLASS CEMPusher
    VAR PRIVATE OVERRIDE
        cMyDriveDirect      : CDriveDirect;
        cMyValve43          : CValveControl43STM;
    END_VAR
    VAR PRIVATE
        cMyStateDiagramEM   : CStateDiagramEM;
        cboStartOld         : BOOL;
        cboModeOld          : BOOL;
    END_VAR
```

```
        METHOD PUBLIC mSetErrorReporter // setter method
            VAR_INPUT
                errReporter    : IErrReport; // Reference of Interface
            END_VAR
            cMyValve43.mSetErrorReporter(errReporter := errReporter);
            cMyDriveDirect.mSetErrorReporter(errReporter :=
errReporter);
        END_METHOD

        METHOD PUBLIC mExecute
            VAR_INPUT
                idNo             : STRING[25];
                mode             : BOOL;
                start            : BOOL;
                packetEmpty      : BOOL;
                packetExists     : BOOL;
                sensorEndPos     : BOOL;
                sensorStartPos   : BOOL;
                motProtSwitch    : BOOL;
            END_VAR
            VAR_OUTPUT
                forward  : BOOL;
                backward : BOOL;
                kx       : BOOL;
            END_VAR

            VAR
                eActState        : cStateDiagramEM.eStateType;
                eVC43State       : eValveStateType;
            END_VAR
            // make the transitions
            eActState := cMyStateDiagramEM.mGetState();
            IF start AND NOT cboStartOld AND NOT mode THEN
            // rising edge start and mode to AUTOMATIC
                eActState := cMyStateDiagramEM.mDoCommand
                 (cStateDiagramEM.eTransitionType#CMD_START);
            ELSIF cboStartOld AND NOT start AND
                 (eActState <> cStateDiagramEM.eStateType#JOG) THEN
            // falling edge start and actual state unequal JOG
                 eActState := cMyStateDiagramEM.mDoCommand
                  (cStateDiagramEM.eTransitionType#CMD_STOP);
            ELSIF start AND NOT cboStartOld AND mode THEN
            // rising edge start and mode to JOG
                eActState := cMyStateDiagramEM.mDoCommand
                  (cStateDiagramEM.eTransitionType#CMD_JOG);
            ELSIF cboModeOld AND NOT mode AND NOT start THEN
            // falling edge mode and start = false transition from
               JOG to STOPPING
                eActState := cMyStateDiagramEM.mDoCommand
                  (cStateDiagramEM.eTransitionType#CMD_STOP);
            ELSIF cboModeOld AND NOT mode AND start THEN
            // falling edge mode and start = true transition from
               JOG to STARTING
                eActState := cMyStateDiagramEM.mDoCommand
                  (cStateDiagramEM.eTransitionType#CMD_START);
            ELSIF eActState = cStateDiagramEM.eStateType#STARTING THEN
            // automatic transition from STARTING to RUNNING
                eActState := cMyStateDiagramEM.mDoCommand
                  (cStateDiagramEM.eTransitionType#CMD_STARTED);
            ELSIF eActState = cStateDiagramEM.eStateType#STOPPING THEN
            // automatic transition from STOPPING to IDLE
                eActState := cMyStateDiagramEM.mDoCommand
                  (cStateDiagramEM.eTransitionType#CMD_STOPPED);
```

```
            END_IF;
            // do state dependend outputs / method calls
            cMyValve43.mGetState(actState => eVC43State);
            CASE eActState OF
                cStateDiagramEM.eStateType#STARTING:
                    cMyDriveDirect.mOn();

                cStateDiagramEM.eStateType#RUNNING:
                    IF packetEmpty AND packetExists AND
                        ((eVC43State = eValveStateType#STARTPOS) OR
                          (eVC43State = eValveStateType#STOP)) THEN
                        eVC43State := cMyValve43.mForward(TRUE);
                    ELSIF eVC43State = eValveStateType#ENDPOS THEN
                        eVC43State := cMyValve43.mBackward(TRUE);
                    END_IF;

                cStateDiagramEM.eStateType#STOPPING:
                    eVC43State := cMyValve43.mBackward(TRUE);
                    cMyDriveDirect.mOff();

                cStateDiagramEM.eStateType#JOG:
                    IF start THEN
                        cMyDriveDirect.mOn();
                        IF packetEmpty AND packetExists AND
                            ((eVC43State = eValveStateType#STARTPOS) OR
                              (eVC43State = eValveStateType#STOP)) THEN
                            eVC43State := cMyValve43.mForward(TRUE);
                        ELSIF (eVC43State = eValveStateType#ENDPOS) THEN
                            eVC43State := cMyValve43.mBackward(TRUE);
                        END_IF;
                    ELSE
                        cMyDriveDirect.mOff();
                        eVC43State := cMyValve43.mStop(TRUE);
                    END_IF;
            ELSE
                ;
            END_CASE;

            //-----------------------------------------------------------
            // call execute-method of internal used classes
            //-----------------------------------------------------------
            cMyValve43.mExecute(idNo             := CONCAT('Valve_',idNo)
                               ,endPos           := sensorEndPos
                               ,startPos         := sensorStartPos
                               ,reset            := FALSE
                               ,forward          => forward
                               ,backward         => backward);

           cMyDriveDirect.mExecute(idNo          := CONCAT('Drive_',idNo)
                                  ,motProtSwitch := motProtSwitch
                                  ,QKx           => kx);

            cboStartOld := start;
            cboModeOld  := mode;
        END_METHOD
    END_CLASS
END_IMPLEMENTATION
```

In the final section of the class CEMPusher it is clearly visible that the objects for the valve and drive are integral components of this class. The methods mExecute() are called here. The internal class variables cMyValve43 and cMyDriveDirect are

instances of these objects. The class type “CValveControl43STM” of the variable cMyValve43 is exactly the same as the type “ValveControl43ST” described in chapter 3.6.2.2. We have moved the aggregated classes to a library OOP_lib. The variables of the aggregated classes are defined in the VAR section of the class CEMPusher and it is therefore essential that they are specified with suitable references when the class is initialized (see chapter 7.3.1).

The integral state machine functions according to the same principle as the one described in chapter 3.6.2.2. We naturally needed to adapt the transition table and the processing sequence to suit the characteristics of the equipment module.

4.1.3.4 Example of an equipment module call

We will describe the example program pCallPusher1 for calling the equipment module in this section. The generation of the object cPusher1 and the object for establishing a link to an error reporting display system (HMI1) are defined in the variable declaration of the program. The information to be transferred to essential I/O components is also specified here.

The I/O signals are transferred to the object cPusher1 in the call of the method mExecute() of the object. The object identification is stored at the parameter idNo of mExecute. This idNo is passed to the aggregated objects for the valve and drive and linked to the idNo in these objects. When an error occurs, this information can be displayed in HMI1 to clearly identify the error location. The string Valve_Pusher1 or Drive_Pusher1 is then displayed in the HMI.

```
INTERFACE
    USES Equipmentmodul;
    PROGRAM pCallPusher1;
END_INTERFACE

IMPLEMENTATION
    PROGRAM pCallPusher1
       VAR
           HMI1       : CHmiReporting;
           cPusher1   : CEMPusher     := (cMyDriveDirect   := (CDrive :=
                                          (criDriveErrorRep := HMI1 )),
                                           cMyValve43       :=
                                          (criValveErrorRep := HMI1 ));
           gboMode1             : BOOL; // Jog or Automatic
           gboStart1            : BOOL; // Start

           // Hardware INPUT
           gboPacketEmpty1     AT %I1.0 : BOOL; // Sensor packet empty
           gboPacketexists1    AT %I1.1 : BOOL; // Sensor packet exists
           gboSensorEndPos1    AT %I1.2 : BOOL; // Sensor end position
           gboSensorStartPos1  AT %I1.3 : BOOL; // Sensor start position
           gboMotProtSwitch1   AT %I1.4 : BOOL; // motor prot. switch
           // Hardware OUTPUT
           gboOutBackward1     AT %Q1.0 : BOOL; // output valve backward
           gboOutForward1      AT %Q1.1 : BOOL; // output valve forward
           gboOutKx1           AT %Q1.2 : BOOL; // output motor on
       END_VAR

       cPusher1.mExecute         (idNo            := 'Pusher<1>'
                                 ,start           := gboStart1
                                 ,mode            := gboMode1
                                 ,packetEmpty     := gboPacketEmpty1
```

```
                                    ,packetExists     := gboPacketexists1
                                    ,sensorEndPos     := gboSensorEndPos1
                                    ,sensorStartPos   := gboSensorStartPos1
                                    ,motProtSwitch    := gboMotProtSwitch1
                                    ,forward          => gboOutForward1
                                    ,backward         => gboOutBackward1
                                    ,kx               => gboOutKx1);

    END_PROGRAM
END_IMPLEMENTATION
```

With this implementation we have finished the equipment module – it is now ready for testing in a SIMOTION system. If necessary, we could also neutralize the HMI link as programmed in instance HMI1 so that the module could be connected to different HMI devices.

Note: To ensure that this example is executable, the program must be assigned to the BackgroundTask in the execution system of the SIMOTION CPU. For instructions on how to assign programs in the execution system, see chapter 8.9.13.

4.1.4 Preparations for multiple reuse

We have now prepared the program code for the example equipment module such that we can use the module once in a CPU. But it can also be reused in different CPUs. If the I/O connection changes, the user must manually adjust the connection accordingly in the program pCallPusher1.

The module can also be reused multiple times in the same CPU. To do this, the unit UEMPusher must be copied and inserted in the project as UEMPusher2. The program name, variable names and object names must then be adjusted as well. In other words, all information has to be adjusted in order to prevent duplication of names within the CPU. The new program created in this way (let's call it CallPusher2) must be integrated in the execution system of the CPU in the same way as pCallPusher1 above. As mentioned above, this procedure should not be regarded in any way as a "copy operation"; instead, the unit UEMPusher will be used as a template for multiple reuse of modules. By copying and adapting the unit, new programs with different I/O connections for the relevant equipment module will be created.

It would be very useful, however, if we could also carry out these adjustments automatically in the engineering environment rather than doing it by hand. SIMOTION has an external program for this purpose. This program is the project generator "easyProject" that is supplied with the engineering system (see chapter 4.2). This project generator is based on the scripting functions implemented in the engineering system. Project data can be manipulated and altered in many different ways with these functions.

A program can be reused multiple times only in cases where it is possible to neutralize the connection to essential I/O components in a suitable way. If, for example, inputs and outputs are directly programmed in the program code, these will also be present in every single copy made of the program. These places in the copies will need to be adjusted later on. The project generator features a solution for working around this problem. This involves attaching "labels" to names and variables. The

project generator can swap the labels for the actual connection. This applies to all uniquely identifiable elements such as object names, externally visible variables, program names, etc.

4.1.4.1 Example of the neutralized equipment module

The program names, object names and variables are now neutralized in the example below. To do this, we have used the labels <Unitname> and <ByteAddress>. The project generator recognizes these labels and replaces them with the current data. Duplication of names can be prevented in this way because all names are automatically changed when the project is generated. For greater clarity, the labels have been highlighted in red in the program text.

```
INTERFACE
    USES Equipmentmodul;
    PROGRAM pCall<Unitname>;
END_INTERFACE

IMPLEMENTATION
    PROGRAM pCall<Unitname>
        VAR
            HMI1         : CHmiReporting;
            c<Unitname> : CEMPusher := (cMyDriveDirect := (CDrive :=
                                        (criDriveErrorRep := HMI1 )),
                                         cMyValve43:=
                                        (criValveErrorRep := HMI1 ));
            gboMode             : BOOL; // Jog or Automatic
            gboStart            : BOOL; // Start

            // Hardware INPUT
            gbo<Unitname>PacketEmpty    AT %I<ByteAddress>.0 : BOOL;
            // Sensor packet empty
            gbo<Unitname>PacketExists   AT %I<ByteAddress>.1 : BOOL;
            // Sensor packet exists
            gbo<Unitname>SensorEndPos   AT %I<ByteAddress>.2 : BOOL;
            // Sensor end position
            gbo<Unitname>SensorStartPos AT %I<ByteAddress>.3 : BOOL;
            // Sensor start position
            gbo<Unitname>MotProtSwitch  AT %I<ByteAddress>.4 : BOOL;
            // motor prot. switch
            // Hardware OUTPUT
            gbo<Unitname>OutBackward    AT %Q<ByteAddress>.0 : BOOL;
            // output valve backward
            gbo<Unitname>OutForward     AT %Q<ByteAddress>.1 : BOOL;
            // output valve forward
            gbo<Unitname>OutKx          AT %Q<ByteAddress>.2 : BOOL;
            // output motor on
        END_VAR

        c<Unitname>.mExecute (idNo           := '<Unitname>'
                             ,start          := gboStart
                             ,mode           := gboMode
                             ,packetEmpty    := gbo<Unitname>PacketEmpty
                             ,packetExists   := gbo<Unitname>PacketExists
                             ,sensorEndPos   := gbo<Unitname>SensorEndPos
                             ,sensorStartPos :=
                             gbo<Unitname>SensorStartPos
                             ,motProtSwitch  :=
                             gbo<Unitname>MotProtSwitch
```

```
                                ,forward        => gbo<Unitname>OutForward
                                ,backward       => gbo<Unitname>OutBackward
                                ,kx             => gbo<Unitname>OutKx);

    END_PROGRAM
END_IMPLEMENTATION
```

The numerical designations that previously existed at various variables have also been removed. They are no longer needed because the swapping of labels has eliminated any risk of ambiguity and the variable name no longer needs to include a number.

In the control files for the project generator, the labels are linked to input fields of the user interface so that the user can supply them with current data. It is also possible of course to integrate an automatic data assignment function in the project generator.

But our sole objective is to explain the principle and we have therefore chosen the option which involves entering data via the user interface of the project generator. A detailed explanation of how to work with the project generator is given below.

The project generator shouldn't just be viewed in the context of object-oriented programming, but rather as a convenient tool that supports the reuse of modular software. Since it is precisely this modular approach that is supported by OOP, it is natural that we have decided to explain how this tool can be deployed for the purpose of reusing object-oriented programs.

4.2 SIMOTION easyProject project generator

The "SIMOTION easyProject" project generator is an external software application designed to support the automatic compilation of projects. It is supplied free of charge to users with every SCOUT engineering software package. This software utilizes the scripting interface integrated in SCOUT. The project generator itself is controlled by an integral XML descriptive interface with a large number of commands and functions. The project generator can be customized to meet user requirements via this XML front end.

The project generator provides you with an extensive library of preprogrammed functions and equipment modules (Figure 46), which are easy to use in SIMOTION.

A major advantage of this tool, however, is the ease with which you can adapt it and thus add your own modules. In other words, you can add your own equipment modules to the project generator data. It is therefore a convenient tool for creating your own reusable software.

Each SCOUT engineering software package delivered includes a directory named "Utilities_Applications" which has an HTML interface. You can easily access and browse the contents of the Utilities&Applications directory using any standard web browser. Numerous tools, supplementary information, sample projects and documentation all make it easier to access the SIMOTION system. You start the web browser by double clicking on "index.html" in directory "Utilities_Applications".

A direct link to the project generator is displayed on the overview page of the directory Utilities&Applications. By following this link (Figure 47), you can reach the next page "SIMOTION easyProject project generator". Apart from extensive documenta-

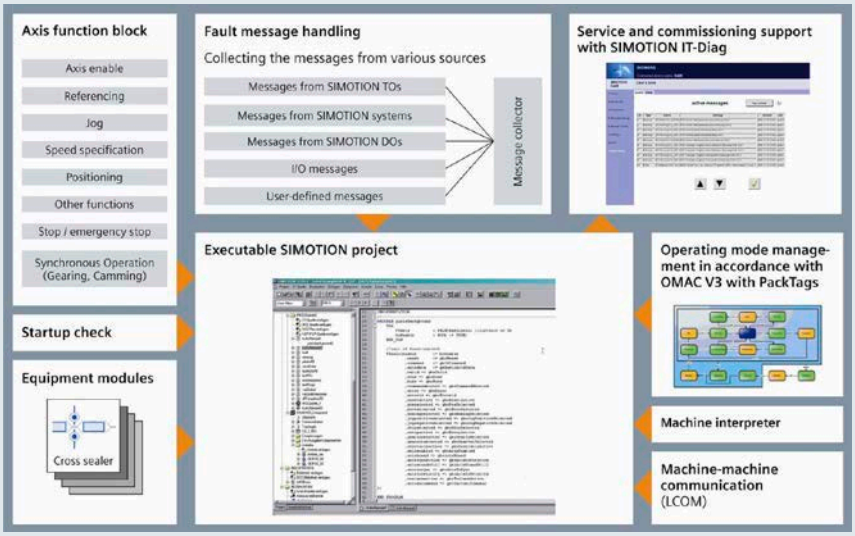
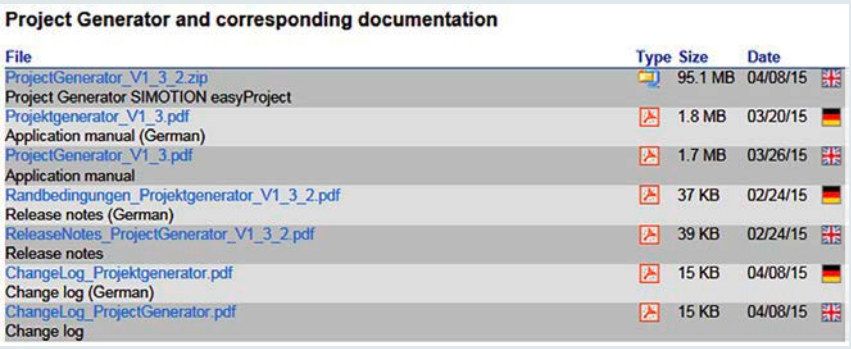


Figure 46 Functions of the easyProject project generator

tion, you will also find the project generator software on this page. You can unpack this ZIP file to any drive on your computer.

After the software has been unpacked, the project generator is immediately ready for use. It doesn't need any installation routine, but the SCOUT engineering software must have been installed on the computer beforehand. The unzipped main directory contains the file "ProjectGenerator.exe". Double-click on this file to start the project generator.



Project Generator and corresponding documentation

File	Type	Size	Date
ProjectGenerator_V1_3_2.zip Project Generator SIMOTION easyProject		95.1 MB	04/08/15
Projektgenerator_V1_3.pdf Application manual (German)		1.8 MB	03/20/15
ProjectGenerator_V1_3.pdf Application manual		1.7 MB	03/26/15
Randbedingungen_Projektgenerator_V1_3_2.pdf Release notes (German)		37 KB	02/24/15
ReleaseNotes_ProjectGenerator_V1_3_2.pdf Release notes		39 KB	02/24/15
ChangeLog_Projektgenerator.pdf Change log (German)		15 KB	04/08/15
ChangeLog_ProjectGenerator.pdf Change log		15 KB	04/08/15

Figure 47 "easyProject" project generator

The same main directory also includes the instructions for use and operation of the project generator in German and English (PDF format), but you can of course also access this information via the web interface. A program for opening PDF files (e.g. Acrobat Reader) must be installed on your computer.

When you start the project generator, the user interface opens and the generator displays screens to guide you through the procedure. All the actions you take are recorded in an XML file by the project generator. This file is stored in the TEMP directory on your computer. The recorded file can be passed to the generator as start information, in which case the project generator starts without a user interface and automatically performs the steps in “silent” mode. You do not therefore need to repeat any inputs provided that they are stored in this file.

Files that are added to the expandable project generator might not always work correctly straight away. In this case, the user interface displays appropriate error messages and indicates the error location. This interface does not appear in “silent” mode, however, so that errors cannot be displayed. For this reason, the project generator stores any errors in log files as it processes the file data. Errors can be analyzed easily (even retrospectively) using the information in these logs.

In addition to generating a new project, you can also add additional CPUs or modules to existing projects. The project generator is easily capable of integrating axes and drives into a project according to your instructions. The capabilities of this project generator are so outstanding that the task of creating a project to develop an application could hardly be any easier.

The project generator screen in which you can select a control system for the project is shown in Figure 48. You can insert one or more CPUs into the project and freely

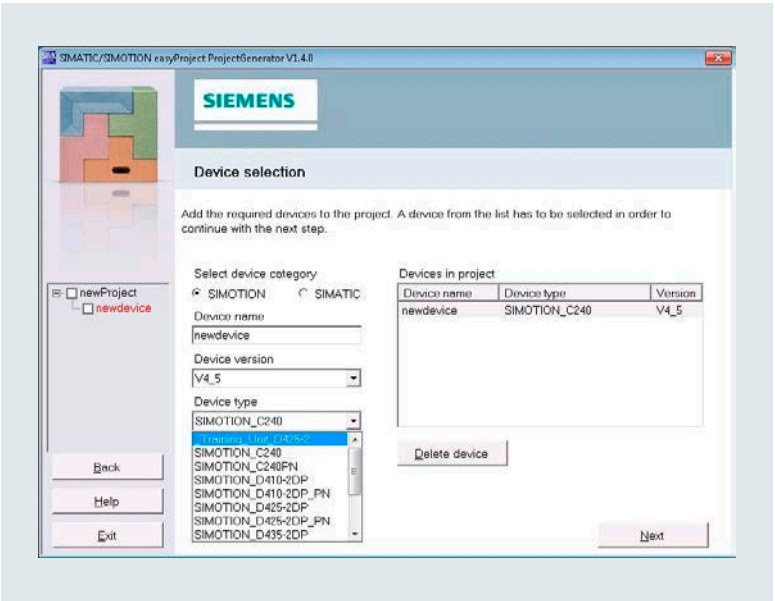


Figure 48 User interface of the project generator

select the CPU name. Once you have inserted the CPUs, you can advance to the next screen by selecting “Next”.

The project generator features as standard an extensive library containing useful equipment modules (Figure 49). You can select these by checking the appropriate

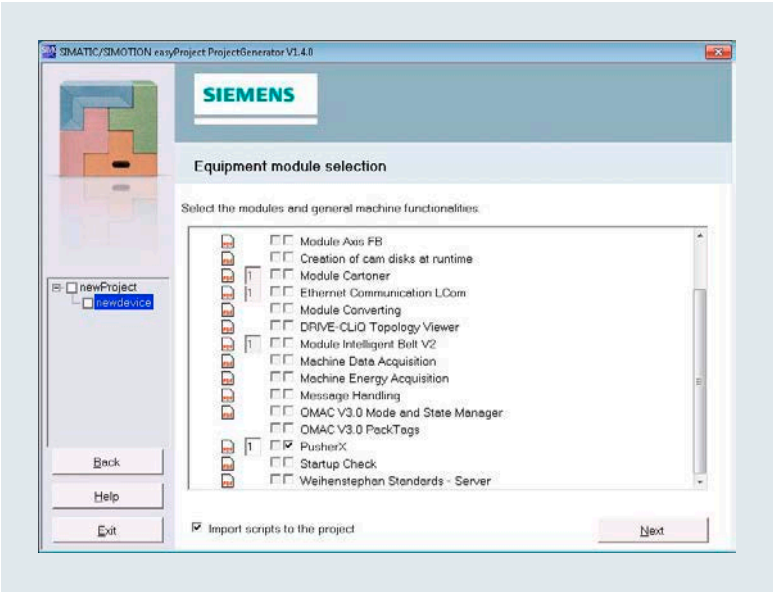


Figure 49 Equipment modules of the project generator

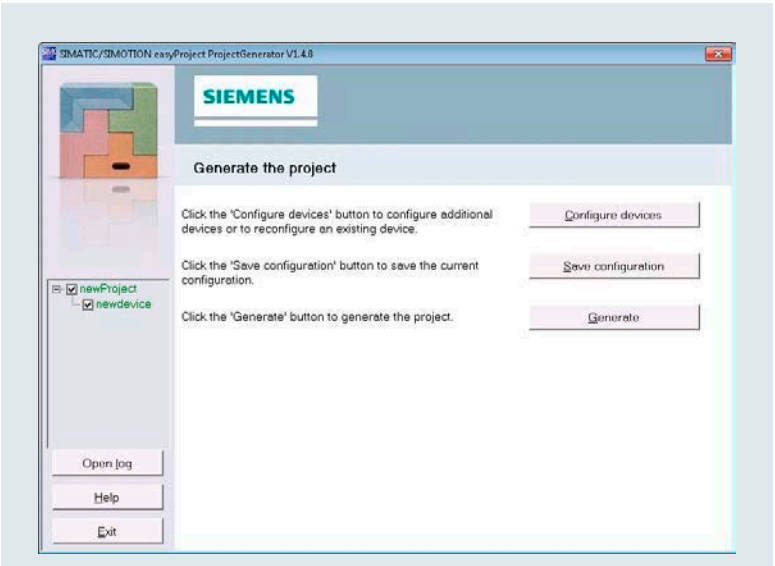


Figure 50 Generating a project

boxes so that they will be integrated in the project. Depending on which modules you select, other screens will appear in which the module parameters can be set.

Once you have entered all the data for the selected equipment modules, the project generator displays the generation screen (Figure 50). You can return to the control system selection screen by selecting the button “Configure devices”. You can save the data you have already entered for use in “silent mode” with the button “Save configuration”. The last step is the actual generation process that is initiated with the button “Generate”.

When the generation process is complete, the project generator outputs a message to notify you that the project is available and asks you whether the SCOUT engineering system should be opened with this project.

Now we have come to the end of our brief explanation about the project generator, we will turn our attention to describing how you can add your own modules to it.

4.2.1 Adding your own modules to the project generator

As mentioned above, you can add your own equipment modules to the project generator. The associated procedure is described in the documentation. We will therefore simply present an abbreviated summary of the process relevant to the equipment module that we discussed in the previous chapter.

But we would still like to draw your attention to the web page www.siemens.com/simotion where you will find a link to “SIMOTION Tutorials” that will give you access to a range of different tutorials. One of these explains the functions of the SIMOTION easyProject project generator.

The database of the project generator comprises a completely standard Windows directory structure in which you will find all the data you need (Figure 51). These data are organized according to various equipment families and devices as well as the equipment modules available for them. Since various module options have

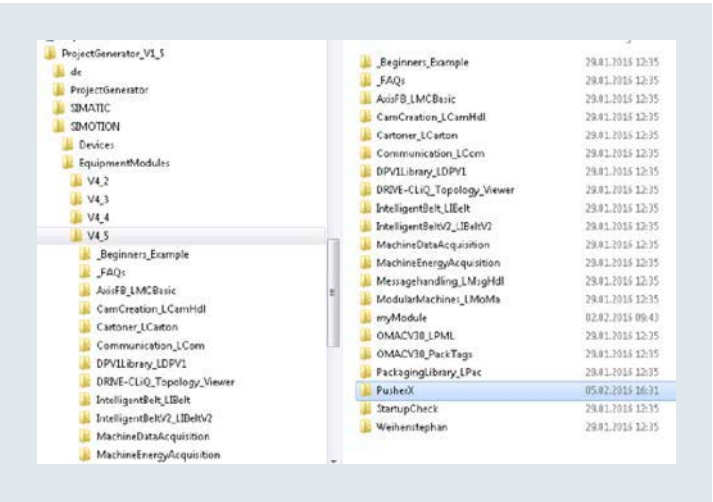


Figure 51 Structure of the project generator data

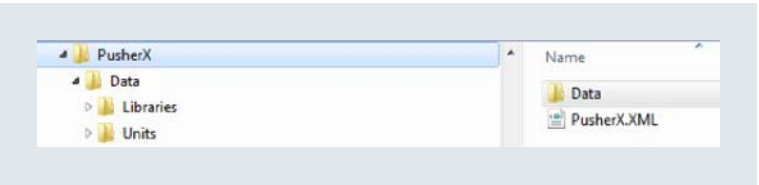


Figure 52 Equipment module PusherX

been implemented for SIMOTION depending on the software version, the individual modules have been divided up again according to the relevant version (V4.2, V4.3, V4.4,...). After you have selected the appropriate CPU version, the project generator displays the functions available for this version. Further information about use and adaptation of equipment modules can be found in the folder "_FAQs" contained in the folder of the relevant version under "EquipmentModules".

The equipment modules available for version 4.5 are displayed in Figure 51. The data for the equipment module are stored in the relevant folder. The folder name is the same as the equipment module name on the project generator interface.

We have already inserted a folder for our new equipment module named PusherX (Figure 52). Each equipment module has an XML file of the same name that contains the user interface and control logic of the module for the project generator.

Other folders containing additional data are stored under the directory "Data". In our case, the directory "Libraries" contains the data of the library that we need (OOP_lib). The program sources of the equipment module are stored in the directory "Units". The data required by the project generator are very easy to generate with the SCOUT engineering environment. Data stored in the engineering system can be exported in XML format using an integral export function. Data can be exported from various levels of the engineering system. It is therefore possible to export individual components such as program sources or libraries, as well as nested elements such as CPUs with the programs stored under them, or even the entire project.

The import function can be used to reimport these data exports into new or existing projects. This import mechanism makes it simple to reuse existing programs. The project generator accesses this engineering functionality via the scripting interface of the SCOUT system.

The generic of the project generator imports data from the lower-level folders as it generates a project and uses them to compile the project. At the end of the generation process, the project is available to the user for further development or testing.

To make the equipment module convenient to handle as a multiple-use module, we will need a user interface and various commands for generation of the module by the project generator.

4.2.2 Creating a user interface for the project generator

As we have mentioned above, each equipment module has an XML control file with the same name as the module. This control file contains the commands for displaying the user interface and the commands to the project generator.

In chapter 4.1.4 and the following example, we explained the principle of neutralizing the name, the variables and their connection to I/O devices. It is only when the equipment module is generated that this information needs to be specified, but to do this we need a means of entering it in the project generator.

When the equipment module is generated, you have to specify the name of the module and the address of the connected I/O devices. The interface displays the two input fields for this purpose (Figure 53). The values can be preset for greater clarity.

Tooltips are integrated to assist the parameterization process. We have shown the inputs required for our equipment module in the simplest possible form so that the control file content is easy to comprehend.

If the module is integrated multiple times in the same project, the name and I/O address information must be entered again for each module in the project manager. If you choose to integrate the module three times into the project, for example, the generator will display the screen three times. A number in the top right-hand corner of the screen will indicate the number of the module currently being parameterized. The next parameterization process is always initiated by the button “Next”.

Once all the equipment modules have been parameterized, you progress to the project generation screen. When you click on button “Generate”, the project generator generates the project and integrates the modules as parameterized. When the generation operation is complete, you can open and compile the project. You can then download it to the control system for testing.

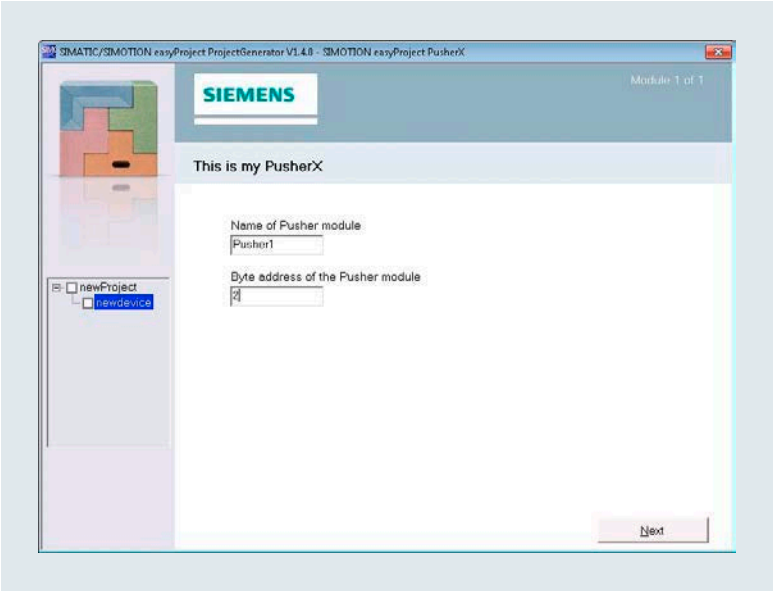


Figure 53 User interface of the equipment module

To finish off our discussion about the equipment module, we will now demonstrate how we created the user interface by explaining a few details about the XML description file for the equipment module.

4.2.3 XML description of the equipment module

Each equipment module has a description file of the same name containing XML code. Our module is called PusherX and the name of its XML file is thus "PusherX.XML". The file consists of several sections and each of these performs different tasks. We will restrict our explanation to the most important aspects, i.e. the XML descriptions shown below are only incomplete extracts from the XML file. The complete file is available to download. Further details about commands and parameters can be found in the list manual of the project generator.

Section 1

The first section comprises the specifications that determine how the module will be displayed in the selection list for all equipment modules.

```
<!-- ================================================================ -->
<!-- The following section describes how the Equipment Module will  -->
<!-- be displayed in the Equipment Module selection dialog          -->
<!-- ================================================================ -->
<!-- The XML element CommandList must be the root element of the    -->
<!-- XML file. The attribute DisplayText specifies the text that    -->
<!-- will be displayed in the selection of the Equipment Modules.   -->
<CommandList Name="PusherX"
             DisplayText="PusherX"
             MaxNumberOfModules=""
             Mode="UnitOnly"
             IsPTemplNecessary="True"
             ToolTip="This is a tooltip for PusherX."
             FlyerFile=""
             ModulInfoFile=
             "SIMOTION\EquipmentModules\V4_5\PusherX\PusherXHelp.txt">
```

The parameter "Name" defines the name of the module. Parameter "DisplayText" determines the descriptive text for the module in the list because there may well be modules whose name is not a clear indication of their function. If the module is to be integrated multiple times, the maximum number can be specified in parameter "MaxNumberOfModules". Where no number is specified (as in our example), the module can be integrated an unlimited number of times.

Section 2

All control elements for the module are created in this section. The command "Command ID="1"" is the first command to be executed and instructs the project generator to change the display.

All subsequent commands allow you to specify a wide variety of design features for the user interface. Labels with texts can be added, for example. Text boxes for value inputs or buttons for activating other functions can be inserted in the interface. The options include radio buttons, check boxes, list boxes or combo boxes, to name but a few. We have not used the latter in our example.

```
<!-- ============================================================ -->
<!-- The following section contains the description of the user   -->
<!-- interface of the Equipment Module                            -->
<!-- ============================================================ -->
<!-- The command with the ID 1 will be executed first             -->
<!-- The attribute Name specifies the function of the Project     -->
<!-- Generator that should be called                              -->
<Command ID="1" Name="ChangeForm" Caption="SIMOTION easyProject
PusherX">
```

In order to identify the module to be integrated, we need to give it a name and specify the I/O address. We do this using controls and their parameters. A control begins with the keyword <Control, includes parameters and ends with </Control>.

```
<!-- The XML element Control is used to add or remove controls    -->
...
<Control Action="add"
         Type="TextBox"
         Name="TB_Unit_Name"
         Text="Pusher1"
         Location="220, 200"
         Enabled="True"
         Autosize="false"
         Size="110, 25"
         ToolTip="Enter the name of the Pusher.">
</Control>
...
<Control Action="add"
         Type="TextBox"
         Name="TB_Byte_Address"
         Text="Address"
         Location="220, 260"
         Enabled="True"
         Autosize="false"
         Size="110, 25"
         ToolTip="Enter the byte address of the I/O.">
</Control>
```

The parameter “Action” can assume two values, i.e. “add” and “remove”. This can be set to add or remove elements. These two controls create two text boxes in which you can enter the name and the byte address of the equipment module. The parameter “Text” is provided so that the entries in the text boxes can be preset. You need to enter appropriate text for these boxes. Parameters “Location” and “Size” determine the position and size of the text box respectively.

```
<Control Action="add"
         Type="Button"
         Name="BT_Next"
         Text="&amp;Next"
         Location="650, 531"
         Size="130, 30"
         Enabled="true"
         Visible="true"
         ToolTip="Complete the configuration">
<Events>
    <Click code="If @TB_Byte_Address@.Text ='' OR
                 @TB_Unit_Name@.Text =''
                 THEN
                      Return
                 End If
```

```
                    @TB_Unit_Name_prefix@.Text = 'pCall'+@TB_Unit_Name@.Text
                      MyApp.NextCommand(11)
                      REM Next Equipment Module
                      MyApp.NextCommand(1000)"/>
</Events>
</Control>
```

This program sequence creates a button labeled "Next" which, when actuated, calls the embedded Visual VBScript.Net. This script processes the input data and replaces the defined labels in the source data with the input data. The program continues with NextCommand(11) in section 3. Once section 3 has been executed with all commands, MyApp.NextCommand(1000) is called and the program branches to <Command ID="1000".

Section 3

The third section of the XML file performs the actual import into the engineering environment. The section is divided up by commands, each of which has its own ID. NextCommand(11) was specified in the previous section, and so the program continues at ID="11".

```
<!--================================================================ -->
<!--= The following section contains the commands of the Project  = -->
<!--= Generator, which are used to apply the configured inputs    =  -->
<!--================================================================ -->
<!-- The attribute Name specifies the function of the Project        -->
<!-- Generator that should be called                                 -->
<!-- If a command has the attribute NCID, the command with the ID,  -->
<!-- which is specified in NCID, will be called next.                -->
<Command
       ID="11"
       Name="ImportUnit"
       NCID="12">
       <Parameter
       Name="Equipmentmodul"
       Path="SIMOTION\EquipmentModules\V4_5\PusherX\Data\Units\
       Equipmentmodul.xml" ForceImport="true"/>
</Command>
<Command
       ID="12"
       Name="ImportUnit"
       NCID="13">
       <Parameter
       Name="__ref_TB_Unit_Name.Text"
       Path="SIMOTION\EquipmentModules\V4_5\PusherX\Data\Units\
       UEMPusher.xml" ForceImport="true"/>
</Command>
...
```

ID11 now effects import of the unit "Equipment module" that is stored in the specified path. The parameter "NCID" defines the next step in the sequence. The next unit is imported in step 12. All of the required units and libraries are imported into the project by this method.

```
...
<Command
    ID="15"
    Name="SetProgram" NCID="16">
    <Parameter
        Name="__ref_TB_Unit_Name_prefix.Text"
```

```
        Unit="__ref_TB_Unit_Name.Text"
        TargetTask="BackgroundTask"
        Position="First"/>
</Command>
...
```

Other commands can be used to integrate imported programs into the execution system of the SIMOTION CPU. The command "SetProgram" is used for this purpose. "Name" identifies the program to be integrated, while "Unit" tells the generator where the program is stored. The parameter "TargetTask" specifies the task and "Position" determines the position in the task system at which the program must be called. This can be important where interdependencies exist and programs need to be executed in a specific sequence.

Once all commands in the XML file have been executed, the module configuration is ended by the code sequence for calling the next equipment module. If there are no further modules to parameterize, the project generator displays the screen for generating the project.

```
...
<!-- The function ReadNextEquipmentModuleConfig will end the
     configuration of the current Equipment Module. If any
     other Equipment Modules were selected, the configuration
     of the next Module would be started. -->
<Command ID="1000" Name="ReadNextEquipmentModuleConfig"/>
</CommandList>
...
```

We will now bring our tour through the SIMOTION easyProject generator to an end so that we can turn our attention back to object-oriented programming. There are many different ways in which the data for automatic compilation of projects can be manipulated. We would therefore advise you to carefully read through the documentation supplied with the project generator so that you work with it more effectively. Like all the other example programs, the example we have given here is of course available for downloading by our readers.

5 Guide to Designing and Developing Software

We have made repeated reference to the design of software in previous chapters. But what does this actually mean? To answer this question – in rudimentary terms at least – we have assembled in this chapter various principles for the development of easily readable and reusable software using object-oriented mechanisms.

Developing software is a complex process – a path strewn with many obstacles that need to be overcome. We don't unfortunately have a standardized procedure or, if you like, a cooking recipe that shows us a simple way forward. Because intrinsic to cooking recipes is their encapsulation of a process that is repeated exactly, i.e. they don't typically involve exploration of new paths. But developing software tends to require the creation of something new.

Designing something completely new is a creative process that demands a certain level of experience. As a general rule, a big task needs to be analyzed and broken down into smaller subtasks. The results of the analysis can be used as a basis for defining a structure for the software. The smaller subtasks obtained from breaking down the main task can then be implemented as programmable units. Before we can start developing any software, however, we need to have a clear picture in our minds of what it is that we want to develop.

5.1 Establishing requirements

If we analyze the challenges of implementing a software project, we will often encounter a common problem, i.e. that new requirements are specified for the project when work on it is already underway. More minor demands are easy to manage and resolve. But if the new requirement cannot be satisfied within the framework of the existing design, the software will often need to be changed extensively. The work involved can endanger the scheduled delivery deadline.

This scenario can be avoided only if the developer or software designer asks the right questions about the requirements to be fulfilled by the solution. It is often helpful if the developer/designer asks questions in the negative such as "which functions is the solution not required to have or perform?", for example. By adopting this approach, everyone involved will gain a clear picture of the task. A single piece of information received after a software design has been defined can render the whole design unusable.

Working out requirements as accurately as possible and achieving clarity about the proposed solution are thus essential if problems of this kind are to be avoided later. As the developer, however, you will not receive any guidance from the end user as to how the inner software mechanisms should work. End users can only describe their expectations of the software and how they want it to behave in certain situations. Based on this behavior in given situations, the experienced developer can reach conclusions about which mechanisms will be required. The ultimate objective in

defining requirements, therefore, is to determine the overall scope of functions that a solution must have. This functional description should be as detailed and accurate as possible so that it can be used as a basis for choosing the architecture and structure of the software. Any issue that is not clarified or discussed will be open to interpretation and become a source of dissatisfaction to the end user. Any software design should be a source of delight and satisfaction to the end user!

5.1.1 Starting point – user interfaces

A very good way to start the process of establishing requirements would be a discussion about the interface required to operate the software, i.e. the user interface. By talking through the control elements and displays required, it is very easy to identify requirements, including those of functions accessed via the interface. When user interface plans depicted by precise sketches are discussed, the user gives the developers information about the relevant sequences when a control element is actuated or a value entered. The user generally expresses how he or she expects the system to react to specific actions.

5.1.2 Starting point – process operations

Automation technology for machine manufacturing applications is normally characterized by its need to fulfill the requirements of a specific production process. In other words, the machines are producing a product. As a result of this production process, it is necessary to precisely define process steps within the machine and its subcomponents. Machine designers normally have a clear idea as to how the process must flow in order to manufacture a particular product.

It is exactly these definitions that are a valuable source of information for software planners. Time-distance diagrams or flow diagrams provide an extremely detailed description of the required process sequence. In many cases, however, this information only describes the process without any interruptions. In order to find a complete solution for the task, the developer needs more information. In discussions with the people involved in the project, the software developer often has to ask how the software should respond in the event of planned and unplanned process interruptions. There are basically two ways in which a process is interrupted:

- *External events*
  Interruptions caused by intervention of the machine operator. These can include, for example, “stop” instructions or opening of protective equipment.
- *Internal events*
  Interruptions triggered by internal plant events such as, for example, error messages issued by components (defects), feed parts that are incorrect or missing, but also incorrect or errored data.
- *Unforeseeable events*
  These are events that cannot be controlled by the software. A sudden power failure, for example, that completely shuts down the control system. Movements are stopped without intervention by the control system. Another scenario that belongs to this category is the movement of machine components when the control system is shut down.

Working out how a system will behave when the process is interrupted is a vital task because the implementation of the relevant functions often takes up far more development time than programming the functions for normal process operation. The first two cases listed above are generally easier to manage because the software remains operational in these instances and can initiate defined reactions.

The third category is the very hardest task for the developer. Because these events require the software to perform an analysis to determine the precise position of the machine when the system is powered up again. It can then identify possible points of entry into the process flow.

Working out the precise position of machine elements is often a very complicated task, Particularly if the machine is designed such that motions are mutually independent and a high risk of collision exists. These dependencies can always be resolved provided that the positions of individual actuators can be clearly identified. Costs might have been streamlined to such an extent that the sensors required to do this are missing, or incremental encoders are used as position sensors rather than absolute encoders.

Cutting costs by eliminating sensors may make the cost of developing the software very much more expensive. It is often this point in particular that is disregarded or underestimated. To avoid or resolve problems of this kind, the software developer has to make some demands of the mechanical engineer and these may well result in the redesign of some machine elements. An early information exchange between the different disciplines will make later changes to the machine design or software unnecessary.

5.1.3 Starting point – mechanical engineering elements

The mechanical design department plans and implements the mechanical components for a machine. The electrical engineering department uses this information to determine electrical equipment requirements such as motors and their drives, switching elements, valves, etc. The control cabinet design department is responsible for planning the control cabinets and the cabling. Using all this information, the software developer can identify those components that need to be controlled by the software to be developed. This same information can be used to analyze the elements of the system and the ways in which they interact.

An abstract needs to be made of the description of elements so that it contains only the properties/functions that are relevant for the analysis. All unnecessary details are eliminated. Each element must then be analyzed individually and the following questions asked:

- What does the element do?
- What are the relevant operations?
- Where does the element need to be moved to?
- What are the different scenarios/movements that need to be realized?
- Which process interruptions can potentially occur?
- How will the element react to interruptions?
- How does the element relate to/interact with other elements?

- What is the hierarchical position of the element?
- How does the element relate to/interact with higher-level elements?

The results of these analyses can be transferred to an object-oriented design (OOD) which represents an essential preliminary stage to the process of object-oriented programming.

The software developer should specify the relevant requirements and conduct the appropriate analyses at the earliest possible opportunity. In too many cases, this planning phase is delayed because information is still missing. But waiting too long can very quickly lead to deadline problems because the time it takes to develop software is frequently underestimated.

There are only two possible ways that the software developer can resolve this problem:

1. To actively demand any missing information from the relevant persons is the best way to gain fast access to the data required. To be firm but friendly often helps.
2. If despite these efforts the information cannot be obtained in the short term, the software should be planned in the most flexible possible way so that it can be adapted later without necessitating any changes to the overall software concept. It's sometimes useful to have a plan B.

If neither of these solutions works out, then the software developer is confronted with the problem of being unable to give a sufficiently accurate estimate of the development time required. A rough estimate almost always results in failure to adhere to the planned delivery deadline.

5.1.4 Existing solutions

The following procedure is normally applied to develop object-oriented software solutions for existing plants. Plants of this kind are already controlled by a software design that works and it is thus unnecessary to ask the customer to specify requirements.

The existing software needs to be analyzed and processed in an implementation plan. The more modular the structure of the existing software, the more easily it can be converted to an object-oriented programming design. In all fairness, we need to point out here that the more modular the existing solution, the fewer advantages OOP has to offer. The developer is then faced with the dilemma that using OOP will not be particularly profitable.

The greatest benefits are to be had when software components that are overcomplicated and difficult to manage are changed to OOP. It is exactly in such cases that the need for change is most urgent. It will not be possible to resolve this more difficult task, however, until the developer really understands the mechanisms involved. As a result, the procedure described below should be regarded as a learning path that will enable the developer to find solutions to difficult problems.

The best software modules for conversion are those components that have a very low dependency (ideally none at all) on other modules. "Dependency" in this context means the internal dependency in the actual program and dependency on external

data. This kind of module naturally works within, and is dependent on, an overall context within the plant. But the module software must be programmed such that there are no direct dependencies in the program code. If these dependencies are neutralized by the use of appropriate input/output interfaces or transfer mechanisms, the module can simply be replaced with another improved or newly programmed module.

Another criterion for replacing the module is the frequency with which it is used. The more often the module is used throughout the plant, the greater will be the benefit of converting it. Software components that are used only once are not generally good candidates for conversion to OOP. When the ultimate objective is to convert all the software, modules of this kind would of course be included in the process. But it is important to ask here whether a unique module would be a suitable starting point.

The greatest advantages are naturally to be gained by converting overcomplicated, cumbersome modules. It is precisely software components of this kind that are just calling out for improvement. They are difficult to upgrade and awkward to maintain. But programmers need a certain degree of experience before they can use OOP mechanisms to successfully improve modules of this kind. They need to gain this experience by starting with simple cases.

Should the analysis of an existing plant reveal that its software does not include any suitable software modules, either because the interdependencies between modules are too great or the data structures are too complicated, the programmer will reach the unavoidable conclusion that the software needs to be completely refactored. Once this becomes clear, it is highly probable that refactoring of the software will be a feasible option only if the plant itself is redesigned or modernized. Because refactoring is more likely to succeed if the software developer can learn and gather experience by studying aspects of the mechanical engineering design process.

5.2 Object-oriented design

Whether object-oriented programming can be used profitably or not depends entirely on the correct design of the software. If you are an inexperienced OO programmer, you will find it particularly difficult at the beginning to define classes in the right way. But you will overcome this problem relatively quickly, particularly when it comes to defining the classes for a real application. To help you get over this hurdle faster, we have given you various tips and explanations of a variety of principles in this chapter.

5.2.1 Encapsulation

One of the fundamental principles of OOP is the fact that objects always have control over their methods and properties. An object is thus intrinsically encapsulated. Direct access to the object data must not be allowed and is normally prevented by the programmer. Where the object properties need to be variable, the programmer will provide appropriate PUBLIC methods that will subject values to suitable checks before passing them to the protected data areas. It can thus be ensured that the

values will always remain within a defined range and can only be changed at suitable times. Interference with the inner mechanisms of the software is no longer possible.

Newcomers to the world of object-oriented programming sometimes violate this principle because they are still clinging to the procedural mindset. They use the access identifier PUBLIC to make the class data public and then program the class according to the procedural principle. Because this is what they learned to do in the past. While they might initially think that this is an easier solution, they are merely paving the way for problems in the future. We want to expressly warn everybody against taking this approach. The end result is merely a procedural program wrapped up in a class structure. It has little to do with object-oriented programming. The reason that programmers choose this option is because they have selected a software module that is too large for conversion to OOP. When choosing modules for conversion, start with the small ones.

5.2.2 Responsibility of a class

A class is the blueprint for objects. An object is thus the smallest software unit (module) in a system. The programmer has the responsibility of defining the functions of a class. This merely involves implementation of those functions that are essential to the class so that it can fulfill its task (responsibility). The programmer could decide, for example, to program a class for valves that covers all variants of valve installed in the machine. Or, if you like, a universal valve class capable of controlling all types of valve.

By pursuing this idea, however, the programmer can get into all sorts of trouble. The functional behavior of valves varies according to the valve type, and the code needs to be programmed in different ways to reflect these variations. In its final programmed form, the class will present a variety of problems:

1. The program code for the universal class will be relatively extensive and thus more difficult to read than a simple class.
2. Owing to their differences in behavior, individual valves will require different program sections (methods). In other words, only the relevant section of the program will be executed for each valve. All the other sections are dead code.
3. The I/O interface of the method(s) is larger than required for the relevant valve.
4. The class is significantly more difficult to test than a simple class.
5. Developing the class further is much more difficult and results in an even greater quantity of dead program code.

To avoid these problems, the class definition should be as lean as possible. A class should possess only one responsibility and thus really be a small unit. The software will then remain manageable, easy to read and simple to continue developing.

Where different responsibilities exist, different classes must also be developed. Now you might want to argue that commonalities do exist between these different types of valve and that no one will want to copy these each time. To resolve this issue, you need to analyze it and find a solution of the kind described below.

5.2.3 Commonalities and differences between objects

When planning classes, it is absolutely essential to analyze the objects. The object-oriented programming concept should help the programmer to develop a specific piece of program code only once and then use it for various purposes. The code exists only once and is not copied multiple times.

This can be done only by identifying the commonalities and differences between objects of the same kind. This applies to functionalities as well as properties. This identification process is essential to the successful design of classes. The process of identifying commonalities and differences is relatively easy. The individual objects are written in the column headers of a table. Functions and properties are listed in the rows. The programmer analyzes each object to determine whether it possesses each of the listed functions and properties. From the result of this evaluation it is very easy to deduce which functions or properties are shared by all the objects.

Commonalities are all included in the base class. The base class is then defined accordingly. In some cases there may not be enough commonalities to create a functioning object. In this case, the base class is defined as an abstract class. Objects cannot be derived from an abstract class, but it is possible to map the common basic functionality (see abstract class CDrive) and program it just once.

Differences are dealt with by means of derivation, i.e. by deriving classes from the base class. Methods and properties are added to the derived classes in order to implement the actual differences between objects.

The method override mechanism can also be used to further adapt objects to different modes of behavior. However, if a large number of the methods in a base class need to be overridden, something has either gone wrong with the class design of the base class, or the base class is not a suitable match for the derived class. A potential indication of the latter problem is when the base class method is not called or cannot be called in a method override mechanism. In this case, the class design has to be revised. It can also be useful to carry out a test implementation. The programmer will then notice pretty quickly whether or not the derived class has been planned correctly. When conducting the test implementation, however, the programmer should be aware that it may need to be discarded again. But this insight will help the programmer to make progress and prevent the need for extensive changes further down the line.

If it becomes clear as development continues that functions are missing from the base class, then these can simply be added. All classes derived from the base class will then inherit the new basic functions.

5.2.4 Principle of replaceability with derived classes

This principle demands of the class programmer that comparable operations of a derived class must behave in exactly the same way as the operations of the base class. In other words, operations of a derived class with identical names to operations in the base class must behave such that their behavior does not change in any way if they are replaced by an object of the base class. This principle protects the users of objects against any unpleasant surprises.

The possibility that this principle will be violated cannot be excluded owing to the polymorphism associated with inheritance. This is often not immediately evident.

The following example is provided to help you understand this better: We have already explained the classes for the valve-cylinder combination in a previous chapter. In this class, the cylinder moves towards “EndPos” in response to the command “Forward” and towards “StartPos” in response to the command “Backward”. If we now need to swap the directions of movement for technical reasons in a derived class, this change would exactly exemplify how the principle of replaceability can be violated. An object of a derived class cannot now be replaced by a base object. Commanding or coordinating program sections cannot cope with this change because, as a general rule, these program sections do not know whether they are commanding a base object or a derived object.

This example also illustrates that an interface is not solely defined by methods and their signatures. The underlying semantics also form part of the interface although they cannot be found directly in the declaration.

That this principle is also valid for various implementations of interfaces goes without saying. It is especially here that different object implementations must be mutually interchangeable.

5.2.5 Determining relationships

Objects in an environment do not stand alone but must interact with each other within an overall context in order to represent a complete functional unit (normally a machine). It is for this reason that provision for interaction between objects must be made in object-oriented software. This involves the definition and design of suitable interfaces that will allow interaction between objects.

The object-oriented programming concept uses the definition of interfaces as a means of modelling interfaces and thus mapping the interaction between objects of different types. By defining interfaces, the programmer can determine how an object will exchange information with another object. To be able to define interfaces, it is essential to determine the relationships that exist between different objects. Analysis will reveal the relationships between objects and these relationships can then be represented. By adding appropriate mechanisms to these relationships, it is possible to define the interface and specify how it must be used.

We have shown the relationship between “valve” and “error reporting” in Figure 54. At this stage, the programmer is still working at an abstract level where the representation still does not contain any details about the programming language to be used later.

It should be mentioned here that the analysis must take into account data relationships as well as runtime relationships. UML diagrams are useful design tools specially developed for this type of analysis.

The definition of an interface is limited in this case to the necessities of the objects involved (classes). If an interface is implemented by a class, the interface addresses only the requirements of the individual module and nothing else. If the object has further relationships with other objects, the experienced designer will define additional interfaces with the requisite functionality. A class with multiple relationships then implements the different interfaces. The option of implementing multiple relationships between classes should be used very sparingly. In this case, the pro-

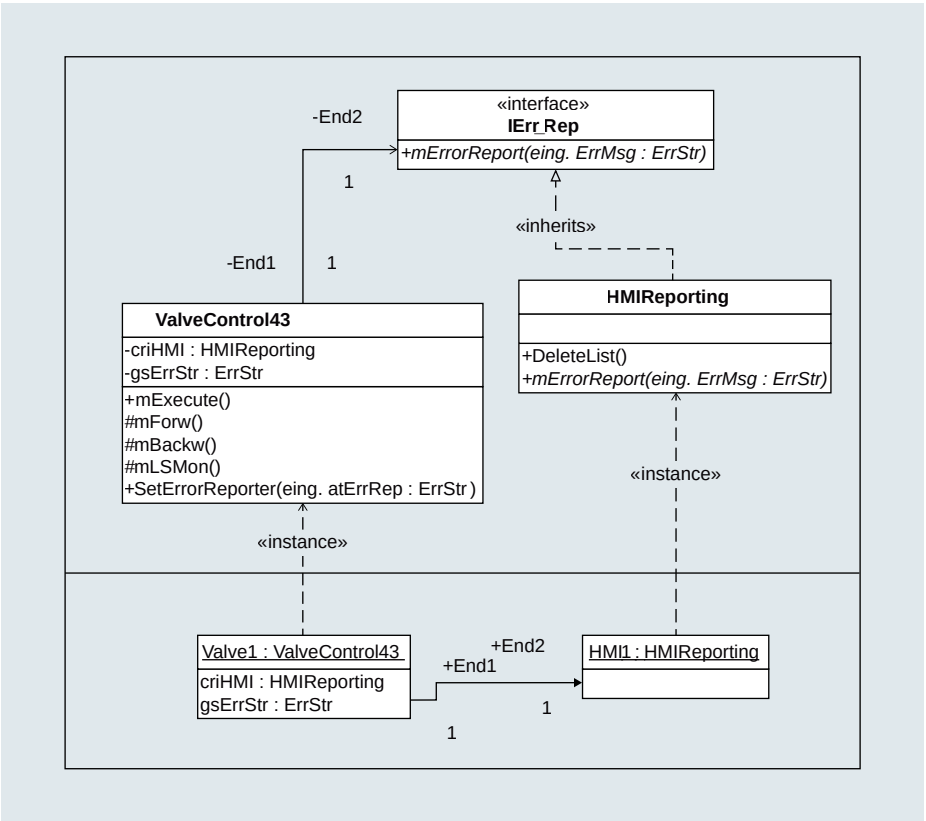


Figure 54 Representation of classes and objects in UML

grammer should only do what is absolutely necessary and no more. The integration of potential future relationships should be avoided at all costs.

Large interfaces must be split up or avoided altogether. The software module uses interfaces that precisely specify the functions actually required, so helping to prevent dependency on functions that are not needed. The definition of an all-singing, all-dancing, all-rounder "mega interface" would directly contradict this principle. By following this approach, we can keep the software lean and manageable.

When we adhere to this principle, we can also develop lightly coupled classes. If the software needs to be refactored, the task is made significantly simpler. If the software needs to be additionally expanded over its lifetime, it is extremely easy to define new interfaces and implement them in the requisite derivation hierarchy. In other words, the new interface is implemented in the class in which the new functions are used. Pre-existing classes are not affected and do not change. These software adaptation mechanisms can be used without any restriction and also optimize the time and effort involved in updating the software. Optimum design is essential in ensuring that software can be flexibly adapted. Precise analysis and definition of interfaces are crucial to the success of a software design.

5.2.6 SOLID principles

Object-oriented design is based on certain principles that were publicized and propagated by, among others, Robert C. Martin, Bertrand Meyer and Barbara Liskov. Robert C. Martin introduced the acronym SOLID to describe a group of principles. Useful information about SOLID can also be found in Wikipedia.[8]

5.3 Reusable and easy-to-maintain software

Software designers should keep three important principles in mind – reusability, ease of maintenance and ease of continued development – and apply these consistently to the task of software development. Object-oriented programming is largely defined by these three principles. In the interests of facilitating their application in SIMOTION, we have summarized various aspects of their implementation in this chapter.

5.3.1 How can software be made reusable?

"Reusability" simply means that the software is designed in such a way that it can be used more than once (we know, we are stating the obvious!). This principle needs to be taken into account in the software design process. Any attempt to render software reusable when the design process is already complete will generally only succeed to a limited extent, or not at all, because by then, too many dependencies on other components will already exist.

For this reason, the software needs to be structured in small, easily manageable modules that can be combined easily for new applications. Components must be so modular in structure that no (concealed) internal dependencies on other modules or specific hardware exist. Any essential connections are neutralized via interfaces for this purpose and each interface is limited to the absolute minimum required functionality.

Restricting the functional scope of interfaces to the absolute minimum can result in two types of interface.

1. *Interfaces for configuring*
   These interfaces are used to link software modules and/or to define initial states. During configuring, for example, references are initialized with appropriate values and then used to establish relationships between objects for operational purposes. Their methods are generally called in StartupTasks or a single time after the control system has powered up.
2. *Operational interfaces*
   This type of interface is used by classes to allow interaction between objects of different types. Their methods are called in normal operation when objects need to exchange information. Depending on the task, methods may be called cyclically or on an event-driven basis, i.e. they are called repeatedly.

[8] "SOLID (object-oriented design)" In: Wikipedia – The Free Encyclopedia. (https://en.wikipedia.org/wiki/SOLID_(object-oriented_design) (viewed on: March 22, 2016, 16:32 UTC)

If the software developer decides to create these two categories of interface, a separate design must be developed for each, but the two designs must be mutually coordinated.

The ability to combine different modules must be planned at the design stage and the interface-based interactions between modules also need to be defined. The designer should always exploit the option of using different interfaces for different tasks and connections. In other words: different functions should never be programmed in a single interface.

5.3.2 Libraries are helpful

At the software design and development stage, it is absolutely essential to give careful thought to the option of storing software elements for general use in libraries. To ensure that this policy can be sensibly applied, programmers must examine the possibility of a system for generating libraries before they start programming. The policy of moving programs to libraries will otherwise not work.

SIMOTION provides the option of moving general-use software components to libraries. In fact, libraries are an integral component of a SIMOTION project. The user can store various different libraries in the library folder of the project and use the content of these libraries in programs. It is also possible to combine and utilize functions between different libraries. For example, it is by all means possible to combine a library of basic functions with libraries containing more complex or specialized functions without the user of the libraries needing to know that they have been combined.

Software components stored in libraries must not contain any hardware-specific links because it is not until the component is used that the system can identify the environment in which the software is running.

SIMOTION libraries can protect know-how. They do this because the creator of a library can protect access to the source code of the programs by a login and password. The program code is encoded by the system and is therefore more secure against unauthorized access. The user has a selection of different protection levels. With the highest protection level, it is even possible to remove the source code of the library. In other words, the library can be supplied without sources. When a library is delivered in this state, it of course cannot be changed by the end user.

5.3.3 What is the best way to develop modules?

When the object-oriented design phase is essentially complete, the results of the design process are put into practice, i.e. the programming phase begins. If we think about the mechanisms already described in this book, we'll remember that a software module has one responsibility (e.g. to "control a valve-cylinder combination"). All the essential functions that the module requires to fulfill this responsibility must be implemented. The functional scope of the module not only includes the basic functions (e.g. controlling the movements of the valve-cylinder combination), but all the additional capabilities as well, such as changing data, detecting errors or communication. We are now going to analyze various different aspects which can be relevant to implementing functions in a class.

Simpler objects can be depicted as a single class. Complex structures are aggregations of further objects with a certain responsibility (e.g. integration of a state machine into a class, see chapter 3.6.2.2). The objective is to ensure that a specific piece of program code exists only once.

Objects of different types interact with one another by different interfaces implemented in the classes. In this case, each interface depicts the functions specified in the class that have been defined for a particular task. By querying the validity of supported interfaces in the classes, it is even possible to adapt objects to different environments during runtime without changing the software (see chapter 3.5.5).

When viewed from outside, each module has its own micro sequence and can be forced to carry out actions in response to commands (command methods). Each module supplies its own methods (Setter/Getter methods) for querying information or changing properties. The connection to I/O devices where necessary is provided in the form of a neutralized interface, but not actually implemented until an object is generated.

The way in which the functions of classes themselves are programmed is determined by the responsibility of the class. This responsibility requires the implementation of methods for individual tasks. The overall responsibility of the class is reflected in the total number of tasks. The task and the way in which the method is implemented determine the context in which the method must be called. There are methods, for example, that are called cyclically, while others are specifically called a single time in order to perform one task. In the case of methods called cyclically, only one method should be programmed for each cyclic execution level. An automation system can have several cyclic execution levels. If it is necessary to implement methods for distribution among various cyclic execution levels, one method must be programmed for each execution level. What the programmer must not do, for example, is implement one method for all cyclic execution levels or two different methods for one cyclic execution level.

With interface methods that are called to allow interaction with other modules, the required call context is not immediately evident from the prototype of the method. This information must therefore be included in the interface definition. This helps to avoid erroneous calls and the users of interface methods will find it easier to work with them. Because it is not always possible to implement the methods defined in the interface in such a way that a single call is enough. For example, if a hand shake has been defined as the interaction mechanism, the method must be called repeatedly until the interaction process is completely finished. In this instance, the interface method call must be integrated into a cyclic context.

Some software modules have the responsibility of providing the functions for a defined machine component. These modules are representatives, i.e. they represent a subcomponent (e.g. control module), a more complex component (e.g. equipment module), or even internal functions. Our valve class, the drive classes or the equipment module all belong to these modules. These modules always contain a method that needs to be called in a cyclic context. The main functions of the object are programmed in this method that is called cyclically. It is the method that organizes the module. We essentially call a method in the cyclic context in order to read inputs, carry out processing in response to signal changes and produce a suitable reaction that is subsequently written to the outputs. We owe this system to the automation engineering processing model. If the I/O components of the control system were

mapped as event-driven devices, the model would have a stronger resemblance to a PC program. In this case, we would only use methods that are called in response to certain events. PC programs do not usually include methods that are called cyclically.

Commands or interactions, and get and set methods, are programmed in such a way that they need not be called cyclically. They have a greater similarity with event-driven methods and are only called when they are needed.

Another type of module has tasks that do not relate to components. These modules or objects tend to be needed for the purpose of organizing the software internally.

These can be objects, for example, that integrate procedural program sections not yet converted to OOP into a class (wrapper). Program sections of this kind behave externally like objects, but are not modeled internally as object-oriented program (potentially as an intermediate solution). The modeling of the wrappers depends on the integrated code. They may require a cyclic context.

Classes that administer or collect pure data, for example, also belong to this category. These need not be designed for a cyclic context. One example of this kind of object is our class "HMIReporting" (Figure 54) which collects the errors of the valve objects. Only one object of this class needs to be generated to perform this function. A method for a cyclic call does not exist in this class. The method for entering an error in the error list is only called when an error event has actually occurred.

Object-oriented design of software modules in an automation system is thus influenced by two basic considerations:

- The cyclic context is a principle typical of PLC engineering. The system reads the signals from input peripherals and stores their states in an internal memory before they are processed in programs. As the programs process the signals from the internal memory, they also store the processing results in an internal memory for the outputs. The system then transfers the data from the internal memory for outputs to the output peripherals. Anyone who programs PLCs knows this context extremely well. PLC programmers think in this context and develop their programs accordingly. Programmers from the PC environment need to become familiar with this principle. Methods developed in this context are designed to transfer essential I/O signals to an object and to pass the results of processing routines to the outputs.
- The event-controlled context is a principle typical of PC engineering. PC programmers understand this behavior and design their programs accordingly. In automation systems, however, there is no means by which the control can selectively register individual events. Events in the control system are identified by the analysis of states in the programs that subsequently necessitate the call of a particular method (e.g. a command to a specific object or transmission of information in the event of an error). These analyses or tests are generally performed in cyclically called methods of other objects.

In other words, both of these considerations need to be taken into account when classes are designed. This will result in a class model that unites both principles (cyclic and event-controlled) and makes optimum use of both of them. Cyclically called methods collect the signals required for an object, receive commands, process them and deduce from them the action that needs to be taken. This action is, on

the one hand, the correct control of actuators and, on the other, the generation of events. Each event calls the appropriate method, including methods of other objects. Other objects can in turn call methods from the class in which they are defined. The class therefore also provides important command methods, status query methods or Setter methods (for changing data). These methods are called on an event-controlled basis and must in some cases trigger appropriate reactions in the cyclic method. All of these considerations form part of the process of optimum class design and, if applied correctly, can result in extremely flexible, easily combinable solutions.

The object-oriented programming principle as it is applied in automation engineering is virtually indistinguishable from its application in Java or other programming languages. There are naturally specific language differences, but while Java programs are used for PC applications, automation engineering programs must execute on dedicated hardware. This hardware is not fully comparable to a PC. Automation systems generally operate in a continually cyclic context and, in some instants, in an equidistant (isochronous) context. This equidistance prohibits the generation or destruction of objects during runtime, for example. In addition, tasks are also executed in parallel. Since the runtime characteristics of automation systems are so different to those of PCs, programmers with a PC background will have to get to grips with this specific runtime behavior because it influences how methods are programmed.

5.4 Organizational and legal aspects

Object-oriented programming opens up a multitude of possibilities for modularizing and structuring software in a more effective way. Software is easier to test and maintain and thus prone to fewer errors. This can only be achieved, however, if the software is carefully planned and if all the people working with the software are also convinced of the benefits of OOP.

5.4.1 Transition to OOP must be planned

Using object-oriented programming mechanisms can be very beneficial in the medium term. If we compare the advantages of OOP with its disadvantages, transition to this new programming style would appear logical and meaningful. Since the programs created to date were necessarily based on the procedural method, however, it is absolutely essential that the transition to new OOP mechanisms is carefully planned.

Training existing programmers in the OOP programming technique in the hope that they will create purely object-oriented programs that are easier to maintain will not be enough.

We shouldn't forget that these same programmers have until now been working perfectly successfully to create and maintain software solutions (procedural or modular) for automation engineering applications. These capable people are now going to be confronted with a completely new way of thinking and working, and will need to internalize and meaningfully deploy this new technique. In addition to support and persuasion, they are also going to need time. If we ignore these issues, they will simply give us object-oriented frames containing functions programmed

by the procedural method. We will not enjoy the success we had expected and will lose any of the promised advantages.

Moreover, if the object-oriented programs are "simply handed over" to commissioning and service personnel, they will probably not understand them. These personnel are also still living in the old procedural world and are not yet familiar with the OOP way of thinking. It is obvious what the consequences and conclusions will be: "Object-oriented programming is no use!"

To prevent all this from happening, the following steps need to be considered and then implemented:

- It is vital that every single member of staff involved becomes familiar with the object-oriented programming technique. *Everyone affected*, i.e. commissioning and service personnel as well as programmers, must be included in this process.
  - It is essential that programmers learn the technique of object-oriented software design and bring appropriate programs to application maturity.
  - Commissioning and service personnel must understand the processes associated with the use of OOP mechanisms. This group of people must also know how to use the relevant program modules.
- It is important to ensure that all those involved are persuaded of the advantages of using OOP. If they are not persuaded, the enterprise will fail.
- Programmers who have been regretting the absence of OOP and are waiting for it to arrive will obviously not need to be persuaded. If programmers with this mindset are available, they can be used to spread OOP knowledge and skills.
- Preparations for transition to OOP must be made by appropriately training all the relevant personnel.
- For a transition to object-oriented programming to be successful, a detailed analysis will need to be performed on plants and/or on existing programs. It will not make any sense to draw up an implementation plan until it is known what proportion of the software is worth converting to OOP.

5.4.2 Software needs to be planned

Careful planning is an essential aspect of all projects and this is also true of software planning. In fact, it is the key to a successful transition to OOP. Object-oriented programming supports the software design process, but does not control it. The software designer should always exploit these supportive aspects so as to facilitate the subsequent coding and programming work. By doing so, moreover, part of the documentation of the software and interrelationships within the software will be generated automatically. This will pave the way for simpler software maintenance and make it easier to continue developing the software when necessary.

5.4.2.1 Analysis of existing programs

Since programs have already been written for the plants and are functioning successfully, they can be analyzed effectively. The purpose of analyzing the software is to identify modules that are potential candidates for conversion to OOP, in other

words, to detect program sections that can be formed into modules and identify extremely lean interfaces between modules. It must be possible to transfer modules from their procedural context into an object model and to encapsulate the module data in such a way that they take on the essential properties of an object.

In other words, the newly created objects must fit into the existing software environment without impairing the functional quality of the overall plant. Analysis of the data is therefore vitally important if proper functioning of the plant is to be maintained. If the data models previously used make encapsulation impossible, it is safe to assume that the modules were incorrectly configured at the early planning stage.

Since object-oriented programming can be regarded as an extension to procedural programming, both programming methods can coexist. It is not absolutely necessary to replace existing, fully functional programs. What is important is to learn how to use the object-oriented programming technique and to gain useful, relevant experience. The transition to OOP will then be a smoothly conducted process and so help to ensure its success.

Once suitable software modules for conversion to OOP have been identified, they are used as a basis for planning classes and their hierarchies. The commonalities for the base classes need to be worked out first at this stage. From the base classes, subclasses are derived (derivation) for further specialization in the extended classes.

When planning classes, it is important to remember that a design error in the base class will be propagated through the entire inheritance chain and, after it has been corrected, may necessitate extensive reworking in the derived classes. For this reason, it is extremely important to plan the base classes with great care.

5.4.2.2 Reuse of software

Better reuse of software is one of the often cited plus points of object-oriented programming. When object-oriented principles are correctly applied, this advantage has a very substantial influence. To exploit this opportunity to the greatest possible extent, it is extremely important to analyze the software and structure it accordingly. We have to emphasize at this point, however, that it is not absolutely necessary to reuse all software. As with all things, the pros and cons must be weighed up to asses whether reuse is worthwhile.

If the software for a plant is completely unique and does not contain any recurring software sections, it can be assumed that it is not a candidate for reuse. The software is used for precisely one particular application and is rendered in a deliverable state for that purpose. New software will then be developed for a new application. The original software will not be reused. This does not mean, however, that it would be pointless to use object-oriented programming in this case. The better structuring options and the encapsulation of software modules could also be extremely beneficial for software that will not be reused (to make it easier to maintain, for example).

If machine elements of the same kind are installed in the plant and operated by appropriate software modules, thought needs to be given as to how these modules can be reused repeatedly in the plant environment. The concept of standardization plays an important role here. Appropriately trained personnel are generally appointed to standardize software modules. These people are responsible for implementing and maintaining the standards. This task also includes the preparation of

documentation describing the correct application and the provision of modules according to defined procedures.

But caution! Do not be tempted to try and standardize everything! Because if you do, you will also need to model every single exception and special feature in the standard. You will end up with cumbersome standard software and generally lose the flexibility of the software design. Standards are merely a collection of recurring commonalities. Object-oriented programming provides enough mechanisms by which specializations (exceptions and differences) can be implemented in the software.

If an organization has taken the decision to standardize certain software components, it is absolutely essential that those responsible for maintaining the standards are involved at an early stage when any changes or upgrades to the plant are made. The standardization might otherwise be ruined unintentionally.

Human nature can cause another problem when it comes to reusing software. We are sometimes reluctant to take a fully functional solution over from somebody else. There are several possible reasons for this reluctance. Either because we don't understand the implementation of the code (perhaps due to the fact that the function is poorly documented or not documented at all), or because we simply think we can do it better, or because we actually don't understand it. Whatever the reason might be, a culture needs to be established within the software development team that makes the creation of reusable software possible.

Every organization has its own culture. The established culture rewards certain practices and punishes others (reward system[9]). This of course encourages employees to behave in a certain way, i.e. to avoid undesirable practices and favor those that are appreciated or rewarded.

For example, organizations still sometimes judge the performance of software developers by the number of code lines they write. This inevitably leads to the creation of superfluous, often unreadable program code that is difficult to reuse.

For one customer, for example, the existing program code had to be analyzed in the course of a debugging assignment. This analysis revealed that general data areas (data blocks) in the control system had been configured to the maximum possible size even though only the first 10 or 20 bytes in the programs were being used. Only during discussions with the employees did the cause of the problem come to light: they were being paid by kilobytes of code.

In such cases, it is extremely difficult to determine the locations in which data are actually used because the people developing the programs will certainly not feel inclined to document the fact that they have included superfluous code. Such programs will also be more difficult to maintain in the field and it will be hard to continue their development. Anyone who has ever had to get to grips with the program code written by other people will know how difficult it is to read the interrelationships contained within the code without overlooking anything important. The conclusion we can draw from the case described above is this: the organization was definitely using the wrong reward system.

It also proves that the culture that is actually lived in any given organization is a crucial issue if we are to obtain reusable programs from development projects. In

[9] In this context, the term "reward system" does not refer exclusively to remuneration for work performed. It also refers in a large part to all forms of behavior of employees/developers for which they are either praised or reprimanded as they carry out their duties. (Source: The C++ Programming Language by Bjarne Stroustrup)

this context, we can ask ourselves "which reward systems prevail in this development environment?", and draw useful conclusions from the answer. These will help us to identify the steps that may be necessary to change the existing culture to one where practices useful to our cause are rewarded. In other words, the prevailing culture must reward or promote the development of reusable program code. This is clearly a task for management. Regulations, rules or processes on their own will not be enough. The developers should themselves strive to develop high-quality and possibly reusable software, but they will only do so if this behavior is rewarded. With the support of the management, therefore, a change in the behavior and practices of the employees must be encouraged and achieved. This certainly won't be easy because management as well as employees will have to abandon what they are used to and learn something new.

If the decision has been taken to use, or at least consider using, the object-oriented programming method to develop software, it is important not to forget that those programmers who have been the experienced, expert developers will suddenly find themselves in the role of beginners again. It is vital to consider this aspect when dealing with employees. A decision to transition to object-oriented programming is not in itself a criticism of the programming methods used in the past, but expresses a desire to exploit the potential of OOP in order to design better software. It could be helpful to call on the support of suitable experts (possibly external specialists) to reinforce this message.

5.4.3 Reuse and ownership of software

In order to derive benefits from creating reusable software, an organization has to take into account many different aspects. These basic requirements need to be established first so that "reuse" becomes economically beneficial.

Social aspects

Conditions need to be created within the software development department that will encourage developers to write reusable software on their own initiative. Any factors that hinder them from doing so (inappropriate reward systems and culture, for example, or poor communication, problems in accepting code written by others) must be eliminated.

Organizational aspects

The development department should be in a position to define its own organizational structure that promotes the development of reusable software. This structure must naturally fit into the company overall. The form that this organization could take depends on the size of the company and the number of developers. This restructuring can in any case be made easier by the drawing up of rules (processes) and the nomination of a person/team responsible for software standardization.

Legal aspects

Legal aspects relating to the software or procedures for software handling must be clarified within the organization. Two basic issues need to be clarified:

- Under what conditions might the software developed within the organization be passed on to other parties, and what rights would be granted to those parties?
- If software is handed over to developers outside the organization, the legal conditions for transferring the software must be checked. The software in this case might have been supplied by other companies (sourced and paid for) or might be software to which rights have been granted (supplied libraries, universities, Internet).

The legal aspect of software is a subject that is often ignored. In certain circumstances, it could pose a real problem to creating reusable software. For this reason, we are going to take a closer look at this subject. We want to make it absolutely clear that this explanation is solely intended to focus attention more sharply on this important aspect, but we are not going to make any legally binding statements. If you have any further queries, please consult an authorized agency.

As an example, here are some informations on the German Copyright Act. The Copyright Act specifies that software (computer programs) is a copyrighted work (i.e. it is treated in the same as a literary work). According to the Copyright Act, software is only deemed worthy of protection if it is the author's own intellectual creation. In other words, a program must be unique or special in some particular way to be deemed worthy of protection. Trivial programs developed scores of times are not covered by the Copyright Act.

As a general rule, the author or right holder grants the user of the software certain usage rights for proper use of the software. Any additional rights are not normally granted. According to the Copyright Act, a criminal offence is committed if sections of the software are extracted for other applications, for example, or if the software is passed on to a third party.

If any programmers now imagine that they can influence what will happen to their software, we have to say immediately that §69b of the Copyright Act is very specific on this point.

(1) If a computer program is created by an employee in the execution of his/her duties or on the instructions of his/her employer, then it is only the employer who is entitled to exercise any economic rights over the computer program unless otherwise agreed.

(2) Paragraph 1 applies accordingly to employment contracts.

5.4.3.1 Distribution of software

When software was initially developed for programmable logic controllers, the programs were still relatively simple. Since the software design was based on circuit diagrams, the programs essentially comprised pure combinational logic. There was very little complexity in the software and end customers were used to having free access to the programs for their own service personnel. The programs were not deemed to have any value at that time.

As advances were made in the field of control engineering, the programs became far more complex, leading to a transition to high-level programming languages such as Structured Text or SCL. Some machine manufacturers have been working for many years to develop programs that might contain functions or algorithms that are really

worthy of protection. Nevertheless, the tendency to attach low economic value to software has persisted for a long time in this environment that is so influenced by mechanical engineering. End customers are still demanding from their suppliers the documented source data of the software so that it can be used by their servicing personnel to safeguard production.

It is exactly this attitude that can sometimes lead to problems. End customers could come to regard the machine manufacturer's software as their own. Modifying the software to supposedly "optimize production" is just a small step, but if it were to happen, the programs would no longer correspond to the delivered state of the software. If errors were to occur, the arguments as to who was responsible for the problems would begin.

In some cases, third parties working for the end customer (programmers, engineering companies) have access to the program source data. Because source data is so easy to copy, the programs could end up being used for other purposes by unauthorized parties (e.g. in the hands of the machine manufacturer's competitors).

To avoid such problems, written specifications regarding the transfer of software are extremely important for all involved.

5.4.3.2 Acquisition of software

The acquisition of software that has not been developed in-house is the second area that can lead to problems. One of the most appealing aspects of reusing software is to cut costs. After all, if it has already been developed, it doesn't need to be programmed again. That all sounds great provided that the software has been legally acquired and the legal situation is clear.

Machine manufacturers sometimes engage third parties to develop software for their machines. These might be engineering companies or external programmers (support staff supplied by the control system manufacturer). These programmers create parts of the machine application on the behalf of the machine manufacturer. According to the Copyright Law, the creators of the software are also the authors (unless an employment contract exists) and are entitled to decide how the programs will be used.

It is absolutely essential for the machine manufacturer that use of the software is regulated by appropriate agreements with third parties before the software is developed. Problems that were avoidable at the outset may arise later if agreements of this kind are not reached.

Another important matter for the machine manufacturer to consider is whether the software integrated into the machinery is actually legal. Machine manufacturers are responsible for verifying the source of the programs. If the software or parts thereof are illegal copies, the machine manufacturer could be accused of copyright infringement.

Software can be obtained from all sorts of sources (e.g. Internet) today. If software from such sources is used by in-house developers, it is essential to clarify the terms of use associated with it. Even if the software is in the public domain, it is not permissible, generally speaking, to remove copyright notices and it may also be necessary to acknowledge the author(s). The conditions specified by the author (which need to

be actively sought in some cases) regulate requirements in detail. Failure to observe these rules will soon end in a copyright infringement.

The creator is always the owner of the software and merely grants usage rights to the user. It is only the creator who may decide what is done with the software.

As a general rule, machine manufacturers or even engineering companies who make their living from software have long-established rules governing software usage. In some cases, however, smaller companies might still be failing to take individual aspects into account or might need to improve their procedures. We have therefore summarized various points again below:

- The distribution of software, including software that is delivered with machinery, should be formulated in writing so that it conforms to the Copyright Act.
- Rules for the acquisition of third-party software for use by in-house developers should be defined and made compulsory for programmers.
- Establishing a standardized procedure for checking the use of third-party software is an effective means of preventing problems later on.
- It is absolutely essential to observe the conditions for transferring software when third-party software is acquired. These include the retention of copyright notices or acknowledgement of authors.
- Employment contracts with external programmers should be examined carefully.

5.4.4 "Good software" and object-oriented design

If the employees are settled in a suitable working environment and the issues raised above have been resolved or do not present any problems, then there shouldn't be any further obstacles to creating high-quality, reusable software, should there?

It would be possible to conclude, for example, that software should at least be "reusable" if it is going to be classed as "good". Since not all software is reused, however, the inverse conclusion, i.e. that software that is not reused is automatically poor software, is also the wrong way of looking at the matter.

So we are faced with the question "what is it that characterizes good software?" or "what are the characteristics that good software must possess?". The only reason we should try to answer this question is to let us judge our own software against certain assessment criteria and find possible ways of improving it. It is not a question of distinguishing between good and bad and therefore passing judgement on our own programmers. These people have already demonstrated their capabilities by writing programs for existing plants and keeping the machinery running with the software they have created.

So, what are the essential characteristics of "good" software? Software quality requirements were defined in standard ISO/IEC 9126 which has since been replaced by ISO/IEC 25000. The software quality requirements defined in the standard apply generally to all software and are thus also applicable to automation system software.

The first and most important characteristic of any software is that it must be functional within a given framework. In other words, it simply has to work. While this may sound trivial, if we examine it more closely, we will find that it is not that

easy to achieve. Implementing fully functioning software is a complex process. To ensure that it can function within a given framework, the framework needs to be precisely defined. What is really important is that all necessary information flows to the right place. In other words, all the functions, requirements, specifications or implementation guidelines that have been agreed with the customer must be made available in suitable form to all those involved in the process. They can be used to define the framework for implementation.

Once the framework has been defined, the software can be structured and then developed on this basis. If boundary conditions change while software development is in progress, the completion date will naturally be delayed. This generally increases the pressure on programmers because they need to deliver by the agreed deadline. As a consequence, the software is handed over to the commissioning engineers in an unfinished state. If software fails to function properly, it can cause a multitude of problems at the commissioning stage and ultimately endanger the entire project. This kind of scenario can be avoided only if the boundary conditions are precisely specified and if any change to the conditions with ensuing delay is accepted by all parties involved. The programmers are otherwise forced to make frequent corrections until the software is rendered fully functional.

When the software has been delivered in a fully functional state, it can always be analyzed and improved once it is operational. Because it will need to be maintained over the entire service life of the plant. Maintenance in this context also includes adaptations that may be required if the customer expresses a wish to increase production capacity, improve error diagnostics to assist troubleshooting or expand the spectrum of products manufactured with the machinery.

By assessing various criteria, therefore, it is possible to formulate the properties software must possess if it is to be deemed “good”.

- The software must be operational and must be reliably fit for the specified operating conditions.

  This point also reveals another important issue that is often not properly heeded in the context of software development. Have the requirements of the software been comprehensively formulated and specified? If programmers are going to implement requirements, they need to know first what these requirements are.

- The software structure must be as modular as possible and each individual module must be capable of functioning autonomously.

  This modularity will make it possible to create suitable environments for testing the software. Software that has been tested during development will cause fewer problems during commissioning. Software with a less modular structure may function perfectly well, but the level of modularity determines the amount of work and time required to maintain or expand the software.

- Control systems often impose restrictions (memory or runtime) with respect to resource availability. The programmer therefore needs to make sparing use of these resources and design resource-efficient software.

  Saving a few milliseconds of runtime can boost the productivity of the plant by 10% or more.

- The software and its scope of functions must be adequately documented.

  This makes life easier when the software needs to be handed over to other people for maintenance or continued development.

- It must be easy for third parties to comprehend the software structure.

  Well structured software is easier to maintain and easier to develop further.

- It must be possible to restructure (refactor) the software over its lifetime.

  Since software is adapted time and again over its lifetime, the programs will need to be revised to make them easier to read and expand. But these revisions must not change the behavior of the software, i.e. they must be compatible with its original design.

Even if programmers are aware of software quality requirements, the software they create will not automatically fulfill these requirements. So we need to answer the question: how do we create the conditions in which developers can write good software? The journey to this goal is certainly not an easy one. Software development is a creative process and individual programmers must be allowed a certain degree of freedom to try out different ideas. Only in this way can they expand and deepen their understanding and knowledge of the underlying system. From this process will emerge new ideas and innovative techniques.

Since individual programmers within the organization are assigned specific jobs and the results of their work must at some point be combined to form a single plant, software development must be regarded as a team task. Whenever teams have to work together, it is vital that individual team members exchange information effectively. Information should be exchanged continuously and not just during meetings or via emails. The exchange of information, especially the informal flow of information (e.g. during coffee breaks, at meeting tables), should not just be tolerated by the management, but actively encouraged.

“Team thinking” should be nurtured. If a team is assigned a shared task and then receives praise for its work as a team rather than as individuals, the team members will feel emboldened to exchange information and cooperate more closely with one another. The quality of the final result will improve. The methods and principles defined by the “agile software development” concept have proven to be extremely successful.

Now we have some sort of idea about what constitutes “good software”, we also need to ask whether use of the object-oriented programming method is absolutely essential to the creation of good software. We can give a clear answer to this one: No! It is possible to write outstanding programs without using object-oriented programming. If that’s the case then, why should we go to all the trouble of learning OOP?

It was exactly this question that PC programmers needed to answer when the object-oriented mechanisms of C++ became available. As demands became more exacting and programs more complex, a paradigm shift took place in programming. In order to manage this increasingly complex software, developers needed to change and learn new techniques and design methods.

The software for automation engineering applications is also becoming more complex as users set ever higher standards and system performance is boosted to meet these demands. Developing software costs a lot of money. Software expenditure

is already determining the price of machinery, and will continue to do so in the future. If new programming techniques that make it easier to optimize and maintain software are now becoming established in the field of automation engineering, it would just be silly not to use them. When it comes to automation applications, OOP is still in its infancy. But it won't even take another 20 years for this programming method to become an indispensable part of control systems. Another factor to be considered is that some automation systems in particular have a significantly longer service life than PC software, for example.

The good news is that the transition to OOP can be planned calmly and implemented at a moderate pace. If we don't use the time available now to get ready, we will be forced to make the transition under greater time pressure in future.

5.5 Software tests are a must!

We are not going to question whether or not it is worthwhile to carry out software tests. Furthermore, a wealth of detailed information about this subject can be accessed on the Internet. We are convinced that software testing is a must. Software tests are not available free of charge and the costs incurred by them must be included in the software development budget. The experts largely agree that failure to perform software tests ultimately costs significantly more money than to carry them out before the software is delivered. Companies who test software as a quality assurance measure often spend as much money on these tests as they do on development.

We therefore want to examine the subject of software testing in a little more detail in this chapter, and give various tips and guidance as to how such tests can be implemented more effectively. We do want to limit our focus to the essential issues, however, because a comprehensive study would require a book in its own right.

Software tests are often the first item to be axed when there are deadline issues or cost problems, but this decision often has far-reaching consequences. In many cases, however, these consequences are not felt until much further down the line and it may be difficult to see the connection between their negative effects and the decision to abandon software testing.

Just like hardware, software undergoes a design process. But in the following respects, it is not possible to make a direct comparison between software and mechanical modules or assemblies:

- Software is significantly more flexible and combinable than mechanical components. The number of possible combinations and thus also the susceptibility to errors is also higher.
- It seems easy to modify software because the changes can simply be typed in, sometimes with fatal consequences.

The task of the programmer is to predict the full potential scope of application of the software and to program functions which reflect this potential. This assessment also includes improper application of the software and the relevant system reactions when the software is improperly used. Predicting future events is not an easy task and it won't take long for a programmer to overlook certain scenarios.

The need to assure the quality of the software by applying specific, measurable criteria is another issue. This quality assurance can naturally only be achieved by testing the software with precisely defined scenarios. The more extensive and complicated the software design, the more difficult it is to define test criteria. This problem can only be overcome by designing a modular software structure so that the tests can be split into different test procedures. As a result, there are normally four different test scenarios:

- *Module test*
  Each module is tested in a test environment.
- *Integration test*
  The interaction between different modules is tested.
- *System test*
  All modules in a specific system are tested.
- *Acceptance test*
  The environment is tested by the customer.

There are also procedures for software tests that can be meaningfully combined with particular test scenarios. For example, the module test will almost certainly be conducted as a white-box-test because the module creator knows exactly how the module functions internally. System testing is usually conducted using the black-box-method. This procedure is designed primarily to test functionality without peering into the internal workings of the software. An integration test would be conducted by a mixture of methods. It is also important to test interruption and error scenarios. It has to be said that these are the most difficult of all tests to perform.

Although we believe that software tests are absolutely essential, we wouldn't wish to deny that no testing regime, however thorough, can ever guarantee that software is absolutely error-free. The multitude of ways in which software can be combined would make this impossible. What is important is to decide on a sensible level of testing with a manageable scope of tests that assure the highest possible degree of reliability. It is ultimately the degree of test coverage achieved that determines the degree of reliability of the software that is finally delivered to a customer. In our opinion, the higher the degree of test coverage, the lower the probability that errors will occur in the field.

On its own, software cannot normally endanger the safety of people. But if it is combined with sensors and actuators of a particular kind, a software error can definitely cause a hazardous state to develop. For this reason, it is important to consider legal aspects and fulfill the obligations that are prescribed by law.

Manufacturers, operators and distributors of machinery are required to abide by the relevant laws[10] and directives[11]. From these must be implied the obligation to ensure that machinery and plant are safe. "Safe" in this context means that such machinery must not endanger the safety of, and certainly not cause injury to, anyone at all. This also includes foreseeable operating errors or machine/plant malfunctions. As a result of this obligation, it is essential to conduct thorough testing. Failure to perform tests is extremely risky and may even be negligent.

[10] E.g. product liability laws

[11] E.g. Machinery Directive, Directive 2006/42/EC of the European Parliament and of the Council of May 17, 2006

5.5.1 Module test

To be able to perform module tests at all, it is essential that the software has a modular structure. Object-oriented programming offers many advantages in this respect. OOP is based on an object-focused view and thus has a modular concept on condition that the design criteria of OOP have been observed. Objects are independent, encapsulated modules capable of functioning independently and generally have just one responsibility. Since the properties and functions of a module (object) are precisely defined, it is easy to work out which tests need to be performed.

This kind of test object is normally embedded in a program written in the same programming language. The sequence of the program in which the test is embedded is divided into individual test steps (e.g. controlled by CASE). Each step is programmed so that the interfaces of the object are supplied with precisely defined data, the object is called and the feedback from the object is analyzed after the program has run. The object feedback is compared with expected reactions. If the result is as expected, the object has successfully completed this step. If the result is negative, the test is aborted and the program stops. In this instance, the error in the object program must be identified and rectified. It might sometimes be necessary to interrupt and adapt the test if the object functionality has been expanded. If all test steps have been conducted with a positive result, the complete test can be deemed a success.

Since data are supplied in steps, troubleshooting in the event of an error is relatively easy. Since the process is easily reproducible, the developer can simply repeat the module test with the input data that caused the error. If the developer gains new knowledge during testing or debugging (test expansions become necessary), new case branches can simply be added to the test steps to provide new input data and the associated expected feedback.

This test should also include a scenario in which incorrect data are input because this is an error that occurs frequently in practice. The module must be capable of dealing with this possibility and reacting accordingly. The expected reactions can be analyzed and evaluated in the test steps.

If interactive links to other objects are also integrated in the test object, the relevant objects must of course be provided in the test environment. “Mock objects” are used for this purpose. These implement interfaces, for example, and simulate the requisite methods with selective, but simplified reaction (see chapter 3.5.7.3). It may well be necessary to selectively manipulate this reaction in the test procedure.

As a general rule, these kinds of module or unit tests are carried out at an early stage of the development process. They are thus frequently devised and applied by the person who developed the software. These tests naturally take a certain amount of time and effort and tie up development resources, but they make it easier to verify that software will function reliably as it is developed. Some developers are not particularly fond of carrying out these tests and don’t do so conscientiously enough at times. A big advantage of the program-embedded test concept described above is, on the one hand, that it automates the test process and, on the other, that the tests integrated in the program can execute after any software modification without the developer having to spend any time on this “dull” task.

5.5.2 Integration test

The next stage of the test program is the integration test. Following successful completion of the module tests, the purpose of the integration test is to verify that mutual dependencies or interactions between different modules have been programmed correctly. This test procedure principally focuses on the interaction between modules and on the data exchange via interfaces that permits them to interact.

To keep this test to within manageable proportions, only those modules that are to interact with one another should be selected for testing. All others should be excluded. As with the module test, this test environment also only includes interacting modules and the program code in which they are embedded. In other words, it is necessary to create program code around the modules that enables testing of the module interactions in individual test steps.

Like the module test, the integration test is performed in steps with appropriate inputs and subsequent evaluation of the results. It takes more time and work to evaluate the results of this test than the result of a module test. This is because large volumes of data may need to be exchanged when modules interact. Checking the correctness of this data naturally involves more work.

The design of the relationships between modules plays an important role for the integration test. If a class has several relationships to other classes, it will implement various different interfaces. Each interface constitutes a relationship that needs to be tested. In order to maintain a clear test structure, the tester should set up the test environment in such a way that only one relationship at a time is tested. In other words, only the class in which the interface is implemented and the other class which is addressed via this interface should be included in each test. On successful completion of the tests, the next test on the second interface can commence. This process continues until all of the implemented interfaces have been successfully tested.

It should now be clear why the design of the classes is such an important factor in determining the scope of tests to be performed. But classes are not designed with testing in mind, but in order to meet specific technical requirements. The amount of work involved in testing classes will highlight their design features, however, and it might be worthwhile considering whether a simpler solution with fewer relationships would be a better option.

Like the module test, the integration test is closely related to the development process. Because any errors detected during testing normally require changes to the program code. The integration test is normally carried out by the developers themselves or by an integration test department attached to the development department. An automated testing system can be very helpful.

5.5.3 System test

The next stage of the test program is the system test. As the name suggests, this involves testing of the entire system. The system test is the continuation of the many integration tests carried out in the previous phase. With mechanical engineering applications, it is generally the test that is performed during initial commissioning of a machine or plant. But it might also be a test plant that comprises part-functions of a larger plant. At least some of the required sensors and actuators are often

installed in the system test environment. For this reason, system testing has a higher hazard potential because mechanical parts of a plant or plant section are actually moved during testing. The risk of injury is high and suitable measures must therefore be taken to minimize this risk. The module and integration tests performed in advance of system testing also help to mitigate this risk.

At the beginning of the commissioning process, the wiring connections of the hardware components are normally inspected to check that they are correct. But the most important job of all is to check that the emergency switching off circuits are functioning reliably including the travel limits of the moving axes. Furthermore, the possible axis motion must be restricted so that there is sufficient travel reserve between the mechanical limit stop and the emergency travel limit. Since we are assuming that these precautions are already well known and widely practiced, we won't go into any further detail. What we are interested in is testing the software.

The first step in the software test process is to check the software components that are responsible for switching on and powering up the control system. These software components also include the code for initializing the connections between different components. By checking that the initialization data are correct, it is possible to ensure that individual software components will interact properly later on.

Even at this early stage, it will be necessary to pay some attention to the user interfaces and the HMI for displaying information. This will ensure that testing will be supported by useful feedback information and displays during the subsequent commissioning process. The HMI is also included in the modularization and independence strategy. Loose coupling is provided by a defined interface to the HMI. The modules store their data in this interface. No data will be transferred to the interface until the modules become active and the tester will not then be overwhelmed by error messages.

In exactly the same way as with the integration test, the system tester focuses on commissioning a single subcomponent. This approach is made easier if the software is modularized. The tester is then in a position to function-test individual software modules step by step. After a component has been successfully tested, the test is extended to include a further module and testing then continues. Thanks to the integration test conducted beforehand, the interaction between mutually dependent modules has already been tested and should not present any problems during system testing.

Special attention must be paid to those modules that are responsible for issuing enable signals and, above all, for shutting down the drives. If these are fully functional, commissioning and optimization of the drives can commence. It may well make sense at this stage to uncouple the motors from the mechanical system for safety reasons. This naturally applies only to motors that cause mechanical components to move. It is not therefore necessary, for example, to uncouple fans. Drives are commissioned and optimized individually. Once they are functioning correctly, the software components responsible for controlling the movements of the drives are tested.

Once all individual component tests have been completed, the operational sequence as a whole can be tested. Increased caution is required at this stage because there is initially a risk of collision between mechanical components due to errors. Once the whole operational sequence has been successfully tested, the plant response to interruption events must be checked. The scenarios to be tested are, on the one hand,

normal interruptions to the process flow as a result of the need to carry out visual inspections in the process area, for example. On the other hand, they also include the plant reaction to emergency trips or power failures and subsequent restart of the process. It is important to remember to test specific operating errors and the plant's response to simulated defects of individual subcomponents. In such cases, the plant must never react in such a way that it poses a hazard to people.

The system test data are documented in appropriate test data records in which the results of individual tests and any other noteworthy events are recorded. These data records provide the necessary proof that the machine is safe. These test results can also help to improve the software if they flow back into the development process.

The system test is ultimately a preparatory step for acceptance testing. In other words, the system test is a means of determining whether the plant meets the customer's requirements.

5.5.4 Acceptance test

The acceptance test is not the most important test, but it is the most significant of all. Because it is the acceptance test that confirms that the customer's requirements have been fulfilled. In many cases, only a preliminary acceptance test is carried out. This is generally followed by a trial production period during which the customer has the opportunity to study the performance of the plant in more detail. Only after the agreed trial production period is successfully completed does final acceptance take place.

The acceptance process must be prepared with great care if it is to be successful. The tests conducted in advance of acceptance are essential to its success. The customer normally demands accurate verification that his or her requirements have been implemented. This can be done only if a detailed record of the individual requirements has been drawn up. Both parties – customer and supplier – must agree with the way that the requirements are formulated. Without documents and records of this kind, it will be possible for either party to "reinterpret" the requirements and the acceptance process will not run smoothly.

Generally speaking, the acceptance test should be a mere formality. By this stage, all modules in the plant and their integration in the system should be completely finished and fully functional. Anyone who delays settling outstanding issues until the completion stage at the end customer's premises is taking a great risk. This also means that certain problems have either not been recognized or have been suppressed at some much earlier time. Systematic testing ensures that difficulties of this kind are promptly revealed so that appropriate solutions can be found. Proper testing is therefore an absolute must.

6 Additional Topics Relating to Software Structuring

In addition to the core topics relating to object-oriented programming that we have already discussed, IEC 61131-3 ED3 has also introduced other new constructs that are useful mechanisms for facilitating the modularization and structuring of software. According to the description in IEC, these are:

- Directly represented variables, partially specified with "*" (referred to below as I/O references)
- Namespaces
- References (referred to below as general references)

These are not yet implemented in SIMOTION Version V4.5. We will discuss these constructs in relation to SIMOTION in the following chapters so that the overall concept becomes clearer. Our readers will thus be able to make preparations for the introduction of the planned functions.

6.1 I/O references

If a developer wishes to write function blocks or classes for a general application and it is necessary to link these to I/O components, then the input and output variables must be transferred in the call interface (VAR_INPUT, VAR_OUTPUT or VAR_IN_OUT).

This is what we have also done in earlier examples in this book. It is obvious that this approach will lead to really extensive call interfaces (see chapter 3.3.1) or interface definitions (see chapter 3.5.9). Direct I/O access at a code point within the modules cannot be implemented as a library-capable solution. It was precisely this type of programming that we condemned in previous chapters about modularization because it would mean abandoning the strategy of making modules completely independent of hardware variants.

The links to I/O components are established via the address list in SIMOTION. The I/Os defined in the hardware configuration are linked to freely definable variable names in this editor. In other words, the user can interconnect each variable with a hardware address.

To make the use of I/Os possible in modules without losing the independence between modules and hardware, we can deploy the construct "partially specified, directly represented variables" defined in the IEC standard. It allows us to declare internal variables neutrally (without an address) in the declaration of function blocks and classes with input and/or output variables, and to use them in the program code. They no longer need to be transferred with VAR_INPUT, VAR_OUTPUT or VAR_IN_OUT. It is only when the instance is defined that it becomes necessary to combine the neutral variables with the actual I/O addresses. Since the instance of the function block or class is initialized with the actual I/Os, the program reads or writes the I/Os directly (without intermediate parameter transfer).

6.1.1 Declaration

SIMOTION uses the asterisk operator "*" to declare I/O references in the variable declaration of the FB or class (as formulated in IEC). In this way it is possible to express an I/O variable generally and link it later on.

The variables are specified in the VAR declaration blocks of FUNCTION_BLOCK or CLASS. Only simple data types or arrays of simple data types may be entered in these blocks (the same applies to the SIMOTION address list). It is not permissible to specify them in VAR_INPUT, VAR_IN_OUT and VAR_OUTPUT for FUNCTION_BLOCK. To ensure that the I/O references can be assigned from an external source, the relevant variable declaration block must either be declared PUBLIC, or at least released for initialization (OVERRIDE see also 8.3.1).

```
// UNIT fb_def;
INTERFACE
    FUNCTION_BLOCK fb_iocopy;
END_INTERFACE
IMPLEMENTATION
    FUNCTION_BLOCK fb_iocopy
        VAR PUBLIC
            invar  AT %I* : INT; // declaration input variable
        END_VAR
        VAR PRIVATE OVERRIDE
            outvar AT %Q* : INT; // declaration output variable
        END_VAR
        outvar := invar;         // example how to use them
    END_FUNCTION_BLOCK
END_IMPLEMENTATION
```

Variables declared in this way can be used in methods or in the function block body. The typical restrictions for I/O variables (INPUT cannot be written) apply. Using this syntax, it is now possible to program function blocks or classes independently of I/O addresses.

6.1.2 Linking references to I/O variables

IEC 61131-3 ED3 makes provision for VAR_CONFIG blocks for connecting incompletely specified I/O variables to the actual I/O variables. The actual I/Os are linked to the instance-specific inputs in these blocks. This IEC requirement has been resolved in a different way in SIMOTION. The same result has been achieved without needing to implement VAR_CONFIG in SIMOTION.

The link to existing I/Os is programmed in the location in which the programmer would normally declare a global instance. This is generally done in the VAR_GLOBAL block in the interface section, or in the implementation section of an ST unit, or in the VAR declaration block of a PROGRAM. It is here at the latest that the link to the actual I/O entries from the address list must be established. It is of course possible to link local instances declared in classes or function blocks to I/O variables. In this case, however, the same I/O link then initially applies for all instances of the relevant class types. The programmer needs to take this behavior into account when aggregating modules in higher-level modules. We therefore recommend that the initialization of I/O references should always be programmed at the highest level so that neutrality can be maintained.

The I/O variables declared in the address list (cf. chapter 8.9.5) are linked to the I/O references of the classes and function blocks using the syntax of the instance-specific initialization mechanism defined in IEC. It is therefore possible to initialize any nested instances. This is also demonstrated by the initialization of gFbNested in the following example in which FB fb_iocopy (programmed beforehand) is aggregated in function block fb-nested.

```
INTERFACE
    USES fb_def;
    VAR_GLOBAL
        gFbCopy1 : fb_iocopy := (invar  := InputVarW0,
                                 outvar := OutputVarW0);
    END_VAR
END_INTERFACE
IMPLEMENTATION
    FUNCTION_BLOCK fb_nested
        VAR OVERRIDE
            locInst1 : fb_iocopy;
            locInst2 : fb_iocopy;
            i        : INT;
        END_VAR
        locInst1();  // call of the first instance
        locInst2();  // call of the second instance
        i := locInst1.invar;  // the access of public I/O-references
                              // outside of the fb is possible
    END_FUNCTION_BLOCK
    VAR_GLOBAL
        gFbCopy2  : fb_iocopy := ( invar  := InputVarW2,
                                   outvar := OutputVarW2);
        gFbNested : fb_nested := (locInst1:= (invar  := InputVarW4,
                                              outvar := OutputVarW4),
                                  locInst2:= (invar  := InputVarW6,
                                              outvar := OutputVarW6) );
        gFbCopy3  : fb_iocopy := ( invar  := InputVarW8,
                                   outvar := OutputVarW8);
    END_VAR
    PROGRAM P1
        VAR
            progFbInst : fb_iocopy := (invar  := InputVarW10,
                                       outvar := OutputVarW10);
        END_VAR
        // program body
        ;
    END_PROGRAM
END_IMPLEMENTATION
```

The actual I/O variables – InputVarW0 to InputVarW10 and OutputVarW0 to OutputVarW10 – were declared in this case in the address list of the SIMOTION device.

In order to visualize access to the I/O variables, the FB instance calls would also need to be programmed, but these were omitted from the example to keep it simple.

Where I/O references are declared in the form described above, it is essential to link them to I/O variables. This rule can become a particular burden when it comes to implementing modular machine concepts in which certain sensors or actuators are installed only as optional equipment. It is precisely for this kind of application that SIMOTION has made provision for declaring I/O references that have no compulsory link to an I/O variable.

```
FUNCTION_BLOCK fb_ioref
    VAR PUBLIC
       invar_opt AT %I* : INT := NULL; // optional connectable input var
       outvar    AT %Q* : INT;         // declaration output var
    END_VAR
    IF invar_opt <> NULL THEN // check that I/O-var is valid
       outvar := invar_opt;   // example for access
    END_IF;
END_FUNCTION_BLOCK
```

By initializing the I/O reference with NULL, it is identified as a reference “without compulsory link” when it is declared. By programming a comparison with NULL in the program code, it is possible to determine during runtime whether an I/O reference is linked to an I/O variable. This extension has made it possible to program a class or a function block with “optional link” I/O variables.

6.2 Namespaces

By offering a means of linking data type declarations and constants to classes and function blocks, SIMOTION already has a mechanism for declaring these types of definition outside the global namespace. If it is necessary, however, to go further and group various classes and function blocks (for the provision of libraries, for example), additional constructs are needed.

The user-defined namespace as defined by IEC 61131-3 is an example of this type of construct. The following elements can be defined for SIMOTION in a user-defined namespace:

- User-defined data types (via keyword TYPE)
- Functions, function blocks and classes
- Interfaces
- Global constants and variables
- Other user-defined namespaces

Namespaces can be nested. A nested namespace can be defined in various different ways, as illustrated by the example below.

```
// UNIT ns_defs;
INTERFACE
    NAMESPACE ns1       // a simple namespace
        FUNCTION f_inc;
        FUNCTION INTERNAL f_add;
    END_NAMESPACE
    NAMESPACE ns2.ns21  // a namespace declared by full qualified name
        FUNCTION f_inc;
    END_NAMESPACE
    NAMESPACE ns3
        NAMESPACE ns31 // a nested namespace declaration
            VAR_GLOBAL
                g_i : INT;
            END_VAR
        END_NAMESPACE
    END_NAMESPACE
END_INTERFACE
```

```
IMPLEMENTATION
    NAMESPACE ns1
        FUNCTION INTERNAL f_add : INT
            VAR_INPUT
                in1, in2 : INT;
            END_VAR
            f_add := in1+in2;
        END_FUNCTION
        FUNCTION f_inc : INT
            VAR_INPUT
                in : INT;
            END_VAR
            f_inc := f_add(in, 1);  // call without namespace name
        END_FUNCTION
    END_NAMESPACE
    NAMESPACE ns2.ns21
        FUNCTION f_inc : REAL
            VAR_INPUT
                in : REAL;
            END_VAR
            f_inc := in +1;
        END_FUNCTION
    END_NAMESPACE
    PROGRAM prog
        VAR
            i : INT;
            r : REAL;
        END_VAR
        // example of a function call with namespace prefix
        i := ns1.f_inc(i);
        r := ns2.ns21.f_inc(r);
        // access to a global variable with namespace prefix
        ns3.ns31.g_i := i;
        // compile error , f_add is INTERNAL in ns1
        // i := ns1.f_add(i,1);
    END_PROGRAM
END_IMPLEMENTATION
```

The keyword INTERNAL can be used to identify elements of a namespace that may be used only within the namespace itself. The function f_add in namespace ns1 is an example of this kind of element. It can be called in namespace ns1, but it cannot be called from outside, for example, from the implementation of the program prog. The following language elements can be identified as INTERNAL within a NAMESPACE:

- TYPE (acts on all user-defined data types of the TYPE block)
- VAR_GLOBAL, VAR_GLOBAL CONSTANT (acts on all variables within the declaration block)
- INTERFACE
- FUNCTION, FUNCTION_BLOCK and CLASS
- METHOD
- VAR, VAR CONSTANT within CLASS and FUNCTION_BLOCK (acts on all variables within the declaration block)
- NAMESPACE (the keyword INTERNAL can only be used if the NAMESPACE is subordinate to a global NAMESPACE)

The same namespaces can be specified multiple times within a single source and as cross-source namespaces, allowing further functions to be added. Let's take a look

at the following example in which a function block is added in namespace ns1. In this case as well, the function f_add from source ns_defs can be used because the newly created function block belongs to the same namespace.

```
INTERFACE
    USES ns_defs;
    NAMESPACE ns1  // extend the content of ns1 with fb_test
        FUNCTION_BLOCK fb_test;
    END_NAMESPACE
END_INTERFACE
IMPLEMENTATION
    NAMESPACE ns1
        FUNCTION_BLOCK fb_test
           VAR_INPUT
               in1, in2 : INT;
           END_VAR
           VAR_OUTPUT
               out : INT;
           END_VAR
           out := f_add(in1, in2); // call of INTERNAL function possible
        END_FUNCTION_BLOCK
    END_NAMESPACE
END_IMPLEMENTATION
```

Namespaces can be used to combine functionally interrelated elements. They thus offer a range of different structuring options, particularly when it comes to creating libraries. Namespaces can also be used to implement machine modules in the application.

While namespaces are very useful for structuring the software, one of their main advantages is that they also help to prevent name collisions in the global scope. This is a particularly helpful feature when software needs to be extended because it prevents collisions between identifiers which can lead to errors.

6.3 General references

Virtually every programming language has mechanisms that allow the programmer to implement references to any data in the computer's memory without creating a copy of it. These mechanisms are known as a "pointer" or "reference". In the control programming concept defined according to IEC 61131, use of references was restricted to the transfer of parameters to POUs (program organization units) using VAR_IN_OUT until the 3rd Edition was approved. No special language constructs were required. The current edition also includes references of a general kind. These can be applied type-safely to any data element. By contrast with pointers which typically allow arithmetic operations ("pointer arithmetic") and conversion operations ("pointer casts"), type-safe references ensure that program behaviors are well defined. This is a very important feature, particularly with respect to control systems in which runtime errors must always be prevented. The restriction laid down by the IEC that references must not point to temporary data elements further reduces the risk that they will be used incorrectly. It is therefore possible to completely eliminate those runtime errors caused by problems that are difficult to pinpoint.

All of these aspects have been taken into account in the implementation of the references for SIMOTION. It has been ensured that references can only ever point to a valid storage space of the correct data type, or that they are assigned the value NULL. This characteristic is guaranteed by the creation system.

6.3.1 Declaration and initialization

A reference is a variable that does not have a value itself, but is implemented as the physical address of where a variable is stored in memory. When defining the reference, the programmer specifies the data type of the referenced variable. The keyword REF_TO is used to define a reference.

```
INTERFACE
    TYPE
        S1 : STRUCT
            x : INT;
            y : REAL;
        END_STRUCT;
        refIntType : REF_TO INT;  // type-declaration reference to INT
    END_TYPE
    VAR_GLOBAL
        gRefInt1 : REF_TO INT; // variable, reference to INT
        gRefInt2 : refIntType; // like above but by type-declaration
        gRefS1   : REF_TO S1;  // variable, reference to S1
    END_VAR
END_INTERFACE
```

As the above example illustrates, references can be declared both as user-defined data types and as variables. References can also be used within structures or as an ARRAY. It is not permissible to declare reference variables in RETAIN declarations.

References can point type-safely to the following elements in SIMOTION:

- Variables of standard data types (BOOL, BYTE, INT, ...)
- Variables of user-defined data types (enums, structures, arrays)
- Instances of classes or function blocks

If a reference variable is created, the system initializes it with the value “NULL”. This means that while the reference itself exists, it does not yet point to any valid storage location (it is thus assigned the invalid address NULL). It is of course possible to program a reference variable to point to a valid variable of the correct data type at the same time the reference variable is defined. The standard function REF() is used to create a reference.

```
VAR_GLOBAL
    gVarInt : INT;
    gRefInt : REF_TO INT := REF(gVarInt);
END_VAR
```

Variable gRefInt in the above example points (according to its declaration) to variable gVarInt. During program runtime, the current reference may be changed by reassignment to another reference of type INT or by assignment to another variable altogether.

It is possible to initialize reference variables with a value other than “NULL” for global variables and all the static variables of POUs. Temporary variables and elements in TYPE declarations may only be initialized with the value “NULL”.

6.3.2 Working with references

Working with references in the user program involves various tasks which can be roughly broken down into the following categories:

- Creation of the reference to an existing variable using the standard function REF()
- Access to the content of the reference using the caret-symbol “^” (also referred to as “dereferencing”)
- Standard operations on references such as assignment and comparison
- Special operations with references of classes such as the dynamic cast “?=”

As we mentioned briefly in the previous section, a reference is created with the standard function REF(). REF() may be programmed anywhere in the code for the purpose of initialization or implementation. It is permissible to specify all the static variables of POUs or global variables as arguments for the function REF().

Once we have created a reference with which a pointer can be transported to a data element (so obviating the need to copy the data), we just need to figure out how to access the data. The IEC has defined the caret operator “^” for this purpose.

```
PROGRAM MyReference
    VAR
        A, B   : INT;
        result : INT;
        refI   : REF_TO INT;
    END_VAR
    A:=5;  // A contains the value 5
    B:=10; // B contains the value 10
    result := A + B;     // result has value 15

    refI   := REF(A);    // refI points to A
    refI^  := 3;         // A contains the value 3
    result := A + B;     // result has value 13
    result := refI^ + B; // result has value 13 too

    refI   := REF(B);    // refI points now to B
    result := refI^ + B; // result has value 20
END_PROGRAM
```

To ensure that a reference is used safely, the validity of its assigned address needs to be checked by comparison with the value “NULL”. It is always advisable to do this prior to dereferencing in cases where references are deployed as transfer parameters, or when no measures have been implemented in the program to guarantee that a reference will have a valid initialization or assignment value. Comparing addresses is a simple mechanism for determining whether two different references point to the same variable.

```
PROGRAM RefCheck
    VAR
        A, B   : INT := 5;
```

```
        ref1    : REF_TO INT := NULL;
        ref2    : REF_TO INT := REF(B);
        bTest   : BOOL;
    END_VAR
    // initialization of ref1
    IF ref1 = NULL THEN
        ref1 := REF(A);
    END_IF;

    bTest := ref1 = ref2;   // bTest FALSE; ref1 points to A, ref2 to B
    bTest := ref1^ = ref2^; // bTest TRUE; values of A and B are the same

    ref1  := REF(B);
    bTest := ref1 = ref2;   // bTest TRUE; ref1 and ref2 points to B

    ref1  := NULL;
    bTest := ref1 = ref2;   // bTest FALSE; ref1 is NULL, ref2 points to B

    IF ref1 <> NULL THEN
        ref1^:= 10;         // never executed; test with NULL
                            // prevents the execution fault
    END_IF;
END_PROGRAM
```

Like other variables, references may also be assigned to one another. In this case, the data reference is copied. In other words, the stored address is copied rather than the data stored at the address. References can normally only be assigned to one another if they point to the same data type. No provision is made for implicit type conversions of the kind used for numerical standard data types, with one exception. Class references are subject to additional rules relating to implicit type conversion that have arisen as a result of inheritance between classes (these rules are also generally referred to as polymorphism). It is thus possible to assign a reference that points to a derived class to a reference that points to a base class. This rule applies in the same way for transfer to VAR_IN_OUT variables of type CLASS and can also be used in version 4.5 even before the introduction of general references. This again illustrates that references are formed implicitly when variables are transferred with VAR_IN_OUT.

```
CLASS clBase (* ... *)  END_CLASS
CLASS clDerived EXTENDS clBase (* ... *)  END_CLASS
FUNCTION_BLOCK fbRef
    VAR_IN_OUT
        cl_io : clBase;
    END_VAR
    (*  ... *) ;
END_FUNCTION_BLOCK

PROGRAM classRef
    VAR
        cl_1 : clBase;
        cl_2 : clDerived;
        fb_1 : fbRef;
        ref_base    : REF_TO clBase;
        ref_derived : REF_TO clDerived;
    END_VAR
    // polymorphy with VAR_IN_OUT
    fb_1(cl_io := cl_1);
    fb_1(cl_io := cl_2);
```

```
    // correct type of the CLASS-reference
    ref_base    := REF(cl_1);
    ref_derived := REF(cl_2);
    // polymorphy with CLASS-references
    ref_base    := REF(cl_2);
    ref_base    := ref_derived;
END_PROGRAM
```

References can be used as transfer parameters and as VAR_OUTPUT at functions, function blocks and methods. It is thus possible, for example, to supply references to instance data externally and then to change them. The following example illustrates how a reference to a private class instance can be accessed by an OUTPUT variable and can therefore be changed outside the FB implementation.

```
CLASS clBase
    VAR PUBLIC   v_pub   : INT;            END_VAR
END_CLASS
CLASS clDerived EXTENDS clBase    (* ... *)
END_CLASS

FUNCTION_BLOCK fbRef
    VAR_INPUT    cl_in   : REF_TO clBase;  END_VAR
    VAR_OUTPUT   cl_out  : REF_TO clBase;  END_VAR
    VAR          cl_inst : clDerived;      END_VAR
    IF cl_in <> NULL THEN
        cl_in^.v_pub := 3;     // access by Input-reference
    END_IF;
    cl_out := REF(cl_inst);
END_FUNCTION_BLOCK

PROGRAM classRef
    VAR          cl_1    : clBase;
                 cl_2    : clDerived;
                 fb_1    : fbRef;          END_VAR
    VAR_TEMP     ref_base:  REF_TO clBase; END_VAR
    // polymorphy with REF_TO in VAR_INPUT
    fb_1(cl_in := REF(cl_1));
    fb_1(cl_in := REF(cl_2));
    // access to a private fb-member by Output-Reference
    ref_base := fb_1.cl_out;
    IF (NULL <> fb_1.cl_out) THEN
        fb_1.cl_out^.v_pub := 2;
    END_IF;
END_PROGRAM
```

Apart from the assignment of class references using the classic assignment operator (:=), there is another interesting operation that can be performed with class references. This is the dynamic type conversion, or the “dynamic cast” using the operator “?=” according to the IEC. The programmer can use this operation to attempt to obtain a reference to a derived class from a valid reference of the base class. If the class instance behind the reference is actually of the correct type, the operation will create a valid reference in the target variable, otherwise the cast will return NULL. The behavior described here is similar to the dynamic type conversion behavior between interfaces (chapter 7.2.7). In this case as well, the type information stored in the SIMOTION Runtime system about a class instance is used to determine whether a specific interface is implemented in the class. In exactly the same way, it is possible to determine the type of a class instance during runtime and of course use this

information to work out which base classes the class instance was derived from. It is even possible to identify the implementation of the class associated with an interface during runtime.

```
CLASS clBase (* ... *)  END_CLASS
CLASS clDerived EXTENDS clBase (* ... *)  END_CLASS

PROGRAM tryRef
    VAR
        cl_1 : clBase;
        cl_2 : clDerived;
        ref_base    : REF_TO clBase;
        ref_derived : REF_TO clDerived;
    END_VAR
    ref_base    := REF(cl_2);
    ref_derived := REF(cl_2);
    // assignment attempt
    ref_derived ?= ref_base;
    IF (ref_derived <> NULL) THEN
        ; // access to cl_derived possible wihout execution fault
    END_IF;
END_PROGRAM
```

References constitute a powerful tool for efficient programming and can also assist with the structuring of the software. When working with class references in particular, the programmer will discover a huge scope of options (similar to those provided by interfaces) for separating software components. To provide a comprehensive overview of all the applications of references, we would need to show a very large number of examples but have decided not to do this here. In this chapter, we have discussed the fundamental aspects of references to give our readers an approximate picture of their potential applications.

7 Description of the Extended Functionality in SIMOTION

In this chapter we are going to explain how the object-oriented programming mechanisms implemented in SIMOTION have been extended beyond the scope defined in IEC 61131-3 ED3. This chapter documents to some extent the ways in which the implementation of OOP in SIMOTION has been modified and extended by comparison with the IEC standard.

The SIMOTION extensions are designed to make programming significantly easier. Nonetheless, OOP can be used in SIMOTION exactly as prescribed by the IEC standard. In this case, the SIMOTION extensions cannot be used.

The source text examples in the following sections merely serve to describe the syntactic options and we have therefore kept them brief. The associated functional explanations and application examples can be found in the first chapters of this book.

7.1 General extensions to the programming model

According to IEC 61131-3 ED3, there are two methods of implementing OOP. One possible option is to add OOP mechanisms to function blocks. The other option is to introduce classes as an addition or alternative to function blocks. SIMOTION decided to use object-oriented features (derivations, implementation of interfaces, method override mechanisms, etc.) only in conjunction with classes. A line has thus been deliberately drawn between object-oriented features and general extensions of the procedural programming technique.

But even in relation to the classic programming method, the scope for software structuring has been expanded. For this purpose, SIMOTION-specific features such as the declaration of local constants and data types in function blocks have been developed further.

The following new constructs for function blocks have been added:

- The programmer can use the access identifier PUBLIC for constants, user-defined data types and static variables that are declared in the FB context.
- The static data of function block instances can be initialized on an instance-specific basis.
- Private static data can be released for initialization via OVERRIDE.
- Methods can be structured more finely using methods.

We have given a more detailed description of these constructs in the following section because they are identically implemented for function blocks and classes.

It is not admissible to implement derivations or override mechanisms for function blocks, nor is it permissible for function blocks to implement interfaces. This will ensure that an FB can be used and treated in the same way it has been in the past. SIMOTION treats all calls as static calls, regardless of whether these are implemented by specification of the instance or are programmed in other methods of the function block.

By utilizing these features, it will be possible to structure software more effectively by binding data types and constants to function blocks on the one hand, and by distributing the code among methods on the other. There is no need to rethink anything with respect to definition and use. A positive side effect of this approach is that these features are available for all SIMOTION-RT versions with SCOUT V4.5.

7.2 Classes in SIMOTION

It was necessary to expand the runtime system of the SIMOTION controller in certain respects to allow implementation of object-oriented programming with classes. For this reason, it will only be possible to use classes on controllers with kernel version V4.5.

SIMOTION supports the following constructs for use with classes:

- Declaration of constants, user-defined data types and static variables with access identifier (PRIVATE, PROTECTED, PUBLIC)
- Definition and use of methods.
- Classes may be derived from other classes.
- Implementation of interfaces in classes.
- Definition of methods or classes, including ABSTRACT methods and classes.
- Provision of runtime type information to support type conversions during runtime (operator “?=”).
- Instance-specific initialization of the static data of class instances.
- Enabling of non-PUBLIC static data for initialization by OVERRIDE.

It is only when classes are implemented and used that the programmer needs to rethink the approach according to object-oriented principles. The programmer must anticipate (or take deliberate measures to prevent) that method overrides will take effect, or that overrides might expand or modify the program code that he or she has implemented.

7.2.1 Constants and user-defined data types in classes

One of the additional features of SIMOTION that is not defined in the IEC standard is that it allows the definition of constants and user-defined data types with a specified access identifier within classes. It is thus possible to define the required data types in the class or class methods directly in the scope of the class. They are linked to the namespace of the class. It is thus that only locally required data types and constant values do not need to be declared globally or in special namespaces. If they are

assigned the access identifier PUBLIC, then they can be accessed via the class name. In this instance, the class name acts like a namespace identifier. This feature is also available for function blocks.

```
CLASS cl_typedef
    VAR CONSTANT PUBLIC
        end_array : DINT := 3;
    END_VAR
    TYPE PUBLIC
        t_array : ARRAY [0..end_array] OF INT;
    END_TYPE
    VAR // usage of local type defined in the own class
        v_array : t_array;
    END_VAR
END_CLASS
CLASS cl_other
    VAR // usage of type t_array defined in class cl_typedef
        v_test : cl_typedef.t_array;
    END_VAR
    METHOD m : VOID
        VAR
            idx : DINT;
        END_VAR
        FOR idx := 0 TO cl_typedef.end_array DO
            ; // iterate through the elements
        END_FOR;
    END_METHOD
END_CLASS
```

7.2.2 Naming of variables in classes and methods

With regard to variable names within methods (VAR_INPUT, VAR_OUTPUT etc.), the IEC standard stipulates that these must not coincide with the names of the static variables of the class.

This rule is very inconvenient, particularly with respect to the extension of base classes or the implementation of interfaces to classes that are also derived from other classes. When the programmer chooses names, therefore, he or she not only has to guarantee that the non-PRIVATE instance data and methods have unique names, but must also take into account all method arguments as well. This kind of rule will quickly lead to error scenarios during compilation that are unmanageable and difficult to understand.

SIMOTION has decided to implement a different strategy. Each method in SIMOTION has its own scope that is subordinate to the scope of the CLASS or the INTERFACE. This rule is also commonly applied in other programming languages. The names of variables within methods (each variable in the method must naturally have a unique name) are selected independently of the names assigned to the static instance data of the class and the derived subclasses.

In order to gain access to the instance data of the class in cases where the same name has been assigned more than once, the instance data can be clearly identified for reading or writing by the fact that the data name is preceded by the class name. This rule applies in the same manner to the methods of function blocks. It is not possible to access the variables of the class with THIS (as it is in C++ for example). According to the IEC definition, THIS is reserved exclusively for method calls.

```
CLASS cl_names
    VAR
        v_test : INT;
    END_VAR
    METHOD PUBLIC m
        VAR_OUTPUT
            v_test : LREAL;
        END_VAR
        v_test          := LREAL#1.0; // access to OUTPUT-variable
        cl_names.v_test := INT#2;     // access to CLASS-variable
    END_METHOD
END_CLASS
```

7.2.3 Method calls

The standard provides three options for method calls:

- External method call using the instance name,
- Internal method call using THIS,
- Internal method call using SUPER.

The IEC standard stipulates that the internal method call using THIS must be implemented with dynamic binding. Unlike existing object-oriented programming languages, there is no option in the IEC standard for programming static calls of methods defined in the class independently of existing overrides.

SIMOTION offers two options for doing the latter (see also chapter 3.3.6). One variant allows methods implemented in the class to be called only via their method name (without a preceding THIS or SUPER). The other variant allows any implementation of a base class method to be called by specifying the class name first. In this instance, the compiler selects (in a similar way to a SUPER call) the effective method to call in the specified class implementation during compilation.

```
CLASS cl_base
    METHOD m_b ; END_METHOD
    METHOD m_x ; END_METHOD
END_CLASS

CLASS cl_derived EXTENDS cl_base
    METHOD OVERRIDE m_b ; END_METHOD
    METHOD PUBLIC m_testDerived
        THIS.m_b();       // call of implementation of cl_derived
                          // or of cl_upper (depends on the instance)
        SUPER.m_b();      // call of implementation of cl_base
        m_b();            // call of implementation of cl_derived
        SUPER.m_x();      // call of implementation of cl_base
    END_METHOD
END_CLASS

CLASS cl_upper EXTENDS cl_derived
    METHOD OVERRIDE m_b ; END_METHOD
    METHOD OVERRIDE m_x ; END_METHOD
    METHOD PUBLIC m_testUpper
        SUPER.m_b();      // call of implementation of cl_derived
        m_b();            // call of implementation of von cl_upper
        cl_derived.m_b(); // call of implementation of von cl_derived
        cl_base.m_b();    // call of implementation of von cl_base
        SUPER.m_x();      // call of implementation of von cl_base
        m_x();            // call of implementation of cl_upper
```

```
        cl_derived.m_x(); // call of implementation of cl_base
        cl_base.m_x();    // call of implementation of cl_base
    END_METHOD
END_CLASS

PROGRAM prog
    VAR
        v_clderived : cl_derived;
        v_clupper   : cl_upper;
    END_VAR
    v_clderived.m_testDerived(); // method call via instance
    v_clupper.m_testDerived();
END_PROGRAM
```

The SIMOTION compiler generates dynamic calls for methods only when necessary. It is therefore only calls using interfaces, calls using THIS and calls to class instances transferred via VAR_IN_OUT that are generated dynamically today. Calls to class references are also generated dynamically.

7.2.4 FINAL for methods and classes

Methods or even entire classes can be declared as FINAL in order to protect the methods against being overridden or prevent the derivation of subclasses from the classes. Preventing the overriding of methods or the derivation of subclasses should be done for functional reasons.

This also has a positive side effect: The SIMOTION compiler can use this information to determine which method calls actually need to be generated dynamically. For this reason, a FINAL declaration can also help to improve runtime performance at those points where dynamic calls would normally need to be executed.

7.2.5 Declaration of abstract classes and methods

If a method is identified as ABSTRACT in a class, SIMOTION automatically transfers this characteristic to the class. It is not necessary to identify the class as ABSTRACT as well. This also applies to all derived subclasses provided that not all the abstract methods have a valid implementation. Furthermore, a class can also be declared directly as ABSTRACT even if all the methods it contains have an implementation. A class identification of this kind is not automatically transferred to the subclasses derived from it.

```
// abstract CLASS because method is abstract
CLASS cl_Not_Instantiable_1
    METHOD ABSTRACT m_abstract
    END_METHOD
END_CLASS

// abstract derived CLASS, method of base class is still abstract
CLASS cl_Not_Instantiable_Ext EXTENDS cl_Not_Instantiable_1
END_CLASS

// abstract CLASS by declaration
CLASS ABSTRACT cl_Not_Instantiable_2
    METHOD m_impl
        ; // code of the method
    END_METHOD
END_CLASS
```

```
// CLASS is instantiable
CLASS cl_Instantiable EXTENDS cl_Not_Instantiable_2
END_CLASS
```

7.2.6 Interface implementation and class derivations

If a class implements an interface using the keyword IMPLEMENTS, then SIMOTION requires that all methods of the interface must either be implemented or at least identified as ABSTRACT in the class. A combination of derivation and INTERFACE implementation is supported. A method that is already implemented in a base class can also be bound to an INTERFACE only in a subclass derived from the base class. This can only be done, of course, if the method signatures (i.e. all method arguments with name and data type) are the same.

```
INTERFACE IArith  // interface-definition with 3 methods
    METHOD m_add  : INT
        VAR_INPUT  in1, in2 : INT;  END_VAR
    END_METHOD
    METHOD m_sub : INT
        VAR_INPUT  in1, in2 : INT;  END_VAR
    END_METHOD
    METHOD m_mul : INT
        VAR_INPUT  in1, in2 : INT;  END_VAR
    END_METHOD
END_INTERFACE

CLASS cl_add  // CLASS implements add
    METHOD PUBLIC m_add : INT
        VAR_INPUT  in1, in2 : INT;  END_VAR
        m_add := in1 + in2;
    END_METHOD
END_CLASS

CLASS cl_addsub EXTENDS cl_add IMPLEMENTS IArith
    // use the base-class implementation to execute add-operation
    METHOD PUBLIC OVERRIDE m_add  : INT
        VAR_INPUT  in1, in2 : INT;  END_VAR
        m_add := cl_add.m_add(in1,in2);
    END_METHOD
    // sub is implemented here
    METHOD PUBLIC m_sub : INT
        VAR_INPUT  in1, in2 : INT;  END_VAR
        m_sub := in1 - in2;
    END_METHOD
    // mul is not implemented
    METHOD PUBLIC ABSTRACT m_mul : INT
        VAR_INPUT  in1, in2 : INT;  END_VAR
    END_METHOD
END_CLASS

CLASS cl_arith EXTENDS cl_addsub
    // first implementation of mul in derived class
    METHOD PUBLIC OVERRIDE m_mul : INT
        VAR_INPUT  in1, in2 : INT;  END_VAR
        m_mul := in1 * in2;
    END_METHOD
END_CLASS
```

A class cannot implement two interfaces both containing a method having the same name but different signatures.

```
INTERFACE IArithReal
    METHOD m_div : REAL
        VAR_INPUT  in1, in2 : REAL;  END_VAR
    END_METHOD
    METHOD m_mul : REAL  // method with different signature to IArith
        VAR_INPUT  in1, in2 : REAL;  END_VAR
    END_METHOD
END_INTERFACE
// declaration is not possible; m_mul has different signature
// in IArith and IArithReal
CLASS cl_arith_all EXTENDS cl_arith IMPLEMENTS IArithReal
END_CLASS
```

7.2.7 Type conversions for classes and interfaces

Implicit type conversions are defined for classes and interfaces as they are for the standard data types and the Technology Objects. The implicit type conversion options are based on the derivation hierarchy and the IMPLEMENTS specifications.

Implicit type conversions are possible for class instances when they are transferred to VAR_IN_OUT variables. An implicit type conversion is possible for interface variables if the interfaces are derived from one another. Another means of implementing implicit type conversion is to assign class instances to interface variables.

```
INTERFACE iface_base    (* ... *)
END_INTERFACE

INTERFACE iface_derived EXTENDS iface_base   (* ... *)
END_INTERFACE

CLASS cl_base IMPLEMENTS iface_base          (* ... *)
END_CLASS

CLASS cl_derived EXTENDS cl_base IMPLEMENTS iface_derived  (* ... *)
END_CLASS

FUNCTION f_ifbase : VOID
    VAR_INPUT  in : iface_base;    END_VAR ; (* ... *)
END_FUNCTION

FUNCTION f_clbase  : VOID
    VAR_IN_OUT io : cl_base;       END_VAR ; (* ... *)
END_FUNCTION

CLASS cl_implicit_cast_instances
    VAR
        v_clbase    : cl_base;
        v_clderived : cl_derived;
        v_ifbase    : iface_base;
        v_ifderived : iface_derived;
    END_VAR
    METHOD m_test
        // Implicit casts between INTERFACES
        v_ifbase     := v_ifderived;
        f_ifbase( in := v_ifderived );
        // Implicit casts between classes and INTERFACES
        v_ifbase     := v_clbase;
        v_ifbase     := v_clderived;
```

```
        f_ifbase( in := v_clbase    );
        f_ifbase( in := v_clderived );
        // Implicit cast classes using VAR_IN_OUT
        f_clbase( io := v_clderived );
    END_METHOD
END_CLASS
```

THIS can be used in method implementations as a reference to the class instance in which the method is defined. A reference to the class in which the method is defined can thus be passed to other functions and methods.

```
CLASS cl_implicit_cast_this EXTENDS cl_derived
    VAR
        v_ifbase    : iface_base;
    END_VAR
    METHOD m_test
        // Implicit casts using THIS
        v_ifbase     := THIS;
        f_ifbase( in := THIS );
        f_clbase( io := THIS );
    END_METHOD
END_CLASS
```

In addition to implicit type conversions, SIMOTION also supports type conversions between interfaces based on type information stored in the runtime system of the controller and supplied by the operator “?=”. In this case, the compiler does not check whether this kind of conversion can be meaningfully executed. Instead, a check is performed during runtime to determine whether this is possible. More detailed information about its application can be found in chapter 3.5.5.

```
INTERFACE iface1    (* ... *)
END_INTERFACE

INTERFACE iface2    (* ... *)
END_INTERFACE

CLASS cl_base       (* ... *)
END_CLASS

CLASS cl_try_assign
    VAR
        v_test : iface2;
    END_VAR
    METHOD m_test
        VAR_INPUT
            vi_iface1 : iface1;
        END_VAR
        VAR_IN_OUT
            vio_cl    : cl_base;
        END_VAR
        // usage of the Assignment attempt operator
        v_test ?= vi_iface1;
        v_test ?= vio_cl;
        v_test ?= THIS;
    END_METHOD
END_CLASS
```

The scope of associated potential will increase significantly again when general references become available. Conversion options between class references and from interfaces to class references will then be added.

7.3 Instantiation of classes and function blocks

7.3.1 User-defined initialization of instances

If class instances are set up, the static variables of the class can be initialized specifically. In this case, the IEC standard currently stipulates that only PUBLIC variables may be initialized. In order to offer more flexibility in this respect, SIMOTION has decided that a variable declaration block that is PROTECTED or even PRIVATE may be released for initialization by an OVERRIDE. This allows setting values to be input irrespective of the level of protection of the instance data. This feature can also be used with function blocks.

```
CLASS cl_data
    VAR PRIVATE OVERRIDE
        v_priv : INT := 5;
    END_VAR
END_CLASS

// redefinition of initial values of the base class
CLASS cl_derived EXTENDS cl_data := (v_priv:= 10)
    VAR PUBLIC
        v_pub : INT := 1;
    END_VAR
END_CLASS

PROGRAM prog
    VAR // definition of initial-values in instance declaration
        v_cldata1    : cl_data;
        // v_cldata1.v_priv = 5
        v_cldata2    : cl_data    := (v_priv := 15);
        // v_cldata2.v_priv = 5
        v_clderived1 : cl_derived;
        // v_clderived1.cl_derived.v_priv = 10;  v_clderived1.v_pub = 1
        v_clderived2 : cl_derived := (v_pub:= 2);
        // v_clderived1.cl_derived.v_priv = 10;  v_clderived1.v_pub = 2
        v_clderived3 : cl_derived := (v_pub:= 3, cl_data:=
                                      (v_priv:=20));
        // v_clderived1.cl_derived.v_priv = 20;  v_clderived1.v_pub = 3
    END_VAR
    ;
END_PROGRAM
```

The instance-specific initialization mechanism defined in the IEC standard should be regarded as a substitute for the constructors that are normally provided in other programming languages. This mechanism can be used to initialize variables with a standard data type as well as interface variables contained in the instance data and variables containing references to Technology Objects. For variables of this type, measures can be taken in SIMOTION to ensure that initialization is performed compulsorily when the instance is created. The “*” operator is used to identify the specified value. All instance data identified in this way for global variables and variables within programs are monitored during compilation to ensure that they are initialized. Initialization of the static instance data of function blocks can also be forced in this way.

```
INTERFACE iface
        (* ... *)
END_INTERFACE
```

```
CLASS cl_ifimpl IMPLEMENTS iface
        (* ... *)
END_CLASS

CLASS cl_init
    VAR PRIVATE OVERRIDE
        v_iface : iface     := *;
        v_toref : ANYOBJECT := *;
    END_VAR
END_CLASS

CLASS cl_embed
    VAR PRIVATE OVERRIDE
        v_clpriv : cl_init; // we do not need an initial value here
    END_VAR
END_CLASS

PROGRAM prog
    VAR
        v_ifImpl  : cl_ifimpl;
        // we need an initial value in a PROGRAM
        v_clembed : cl_embed := ( v_clpriv:= ( v_iface := v_ifImpl,
                                               v_toref := Achse_1  ));
    END_VAR
    ;
END_PROGRAM

VAR_GLOBAL
    vg_ifImpl  : cl_ifimpl;
    // we need an initial value on global instances
    vg_clembed : cl_embed := ( v_clpriv := ( v_iface := vg_ifImpl,
                                             v_toref := Achse_1   ));
END_VAR
```

If we look at the specification of the initialization values in the previous example, we will notice that the specified values are much more difficult to read because they are typed on separate “parenthesis levels”. To remedy this problem, SIMOTION allows the programmer to write the value specifications for structured elements in a similarly compact form to the code section. The initialization value can thus be specified in an alternative form as shown below.

```
VAR_GLOBAL
    // alternative syntax to initialize structured elements
    vg_clembed : cl_embed := ( v_clpriv.v_iface := vg_ifImpl,
                               v_clpriv.v_toref := Achse_1);
END_VAR
```

This method of specifying initialization values is not restricted solely to class and function block instances. Even structures can be initialized in this way. Since the extension is limited to the multi-stage selection of the structure component, it is fully compatible with the IEC method of specifying initialization values, and mixtures of both methods can be used.

7.3.2 Initialization of interface variables

In SIMOTION, interface variables can always be initialized first with the default value NULL. It is moreover possible to initialize static variables in classes and function blocks, and INPUT variables with class instances. In this case, the instances must

either be globally accessible or belong to the instance data of the class in which they are defined or of the function block.

```
INTERFACE iface       (* ... *)
END_INTERFACE

CLASS cl_impl IMPLEMENTS iface  (* ... *)
END_CLASS

VAR_GLOBAL
    vg_cl : cl_impl;
END_VAR

CLASS cl_iface_init
    VAR
        v_loc : cl_impl;
        v_if1 : iface := vg_cl; // Init with global instance
        v_if2 : iface := v_loc; // Init with local instance of own class
    END_VAR
    METHOD m_test
        VAR_INPUT
            // global instance as a default argument of a method
            vin_iface1 : iface := vg_cl;
            // local instance as a default argument of a method
            vin_iface2 : iface := v_loc;
        END_VAR
        ;
    END_METHOD
END_CLASS
```

7.3.3 Creating class and function block instances

The IEC standard makes provision for including function blocks and classes in structures. But it prohibits, for example, the creation of function blocks and classes in VAR_TEMP declaration subsections. In some instances, it is not possible to copy structures with the instances they contain. If a system function block reserves resources in the runtime environment, for example, some means of creating the copy needs to be found. This ultimately makes it necessary to introduce constructors, copy constructors and assignment operators that would need to be programmed by the user.

So as to define practicable rules for the user, SIMOTION does not make provision for the use of function block and class instances in user-defined data types (structures and array definitions). This rule has been devised to ensure that user-defined data types can be freely used as normal. It also prevents the copying of class and function block instances. The rule that classes may not be used in structures does not impose any limitations on the user. A good solution in this case is to use a class without methods and containing variables that are all declared PUBLIC as a quasi substitute.

Class and function block instances can be set up in other classes, function blocks and programs only as static or array variables. They can also be instantiated as global variables.

It should be noted that these rules will not apply to references to classes and function blocks. Like interface variables, these can also be used as elements of user-defined data types.

7.3.4 RETAIN data in classes and function blocks

The RETAIN property can also be assigned to static instance data in classes, function blocks and programs. The values of these variables are automatically stored in non-volatile memory and read back during restart. Declaration of such variables as PUBLIC is not supported. If they need to be accessed from an external source, they can always be reached by implementation of appropriate methods. It is important to note that the storage space for this kind of data is limited on a control system. Large volumes of this kind of data should not therefore be stored.

```
CLASS cl_retain_var
    VAR
        v_loc    : INT;
    END_VAR
    VAR RETAIN
        v_retain : INT;
    END_VAR
END_CLASS
```

If a class or a function block has a RETAIN component, instances of it cannot be created in the RETAIN sections of other classes or globally as RETAIN. It is also impossible to use elements that do not contain any restorable component in RETAIN sections. To give an example, this would apply to structures that contain nothing but interface variables.

7.3.5 Arrays of variable length

SIMOTION users have for a number of years been able to use arrays with an index range that is unknown when the program is compiled. The associated language construct has now been adopted in the 3rd edition of the IEC standard and is now also available as a standards-compatible feature. SIMOTION supports variable-length arrays within VAR_IN_OUT for functions, function blocks and methods and at VAR_INPUT of functions and methods. It has been decided not to support variable-length arrays at VAR_OUTPUT because it cannot be ensured that OUTPUT variables will be supplied correctly when they are called. Since there are no differences in behavior vis-à-vis VAR_IN_OUT, this does not restrict the user in any way.

```
FUNCTION f_copyarray : INT
    VAR_INPUT
        v_in : ARRAY [*] OF INT;
    END_VAR
    VAR_IN_OUT
        v_out : ARRAY [*] OF INT;
    END_VAR
    VAR_TEMP
        idx1, idx2 : DINT;
    END_VAR
    idx1 := LOWER_BOUND(arr := v_in);
    idx2 := LOWER_BOUND(arr := v_out);
    WHILE ( idx1 <= UPPER_BOUND(arr := v_in) AND
            idx2 <= UPPER_BOUND(arr := v_out)    ) DO
        v_out[idx2] := v_in[idx1];
        idx1 := idx1 + 1;
        idx2 := idx2 + 1;
        f_copyarray := f_copyarray + 1;
    END_WHILE;
END_FUNCTION
```

7.4 Tips for creating compatible and efficient software

7.4.1 Methods and function calls

Methods and functions can be called by various different methods. It is possible, on the one hand, to supply only values to all the INPUT and IN_OUT variables in the order in which they are declared in the call. On the other hand, the relevant parameter name assigned to the value can also be specified in the call. Using this form of call means that OUTPUT parameters can also be read and that parameters to which defaults are assigned in the declaration can be omitted from the call.

After the explanation we have given above, we recommend the second form of call. Apart from the additional options for assigning OUTPUT parameters and using default values, there are, however, other reasons for recommending this form of call.

- Erroneous assignments in calls are prevented because there is a clear assignment between the argument and value.
- Additional arguments with default value can be added to functions and methods without requiring the adjustment of all call points.
- If the meaning of a parameter is changed, all call points can be easily found simply by changing the parameter name at the same time. The compiler helps during the search by issuing an error message at the relevant points of use.

7.4.2 Use of enum values and constants

The elements of an enumeration type are specified in its declaration. Elements of an enumeration type do not need to be unique as compared to other types. In other words, the same element identifier may be defined in various different enumeration types. An element of an enumeration can be accessed simply by using the identifier of the enumeration value. Alternatively, an enumeration element can also be accessed using the associated data type name followed by "#" and the name of the element (<enumtype_name>#<enumvalue_name>). While this notation is slightly longer, it does ensure that unintentional compilation errors caused by identifier ambiguity can be avoided if the enumeration type is later expanded. This can occur, for example, if a newly added variable or data type hides the identifier of the enumeration value. We can demonstrate how this works if we remove the marked comments in the following example.

```
VAR_GLOBAL
    // on : BOOL;  // removal of the comment will cause an
                   // error in METHOD m
END_VAR

CLASS cl_enum
    TYPE
        state : ( on, off );
    END_TYPE
    VAR
        vl_state : state;
        // on : BOOL;  // removal of the comment will also cause an
                       // error in METHOD m
    END_VAR
```

```
    METHOD m
        vl_state := on;        // not recommended notation
        vl_state := state#on; // recommended notation to access enums
    END_METHOD
END_CLASS
```

The data type with which an expression is calculated is specified on the basis of the data types of the variables and constants involved. In this case, it is normally not possible to uniquely assign value specifications to a data type. If the compiler determines the data type for performing the calculation on the basis of the specified value, the calculation may produce unexpected results. This situation can be avoided by specifying the data type and the value together. This is done by typing the data type name first followed by "#".

```
FUNCTION f : LREAL
    f := 1/4;              // SINT operation;  result is 0
    f := LREAL#1/LREAL#4; // LREAL operation; result is 0.25
END_FUNCTION
```

If the data type for performing the calculation is defined by a constant which cannot be uniquely assigned to a data type, the SIMOTION compiler issues an alarm to warn that the calculation might produce the wrong result.

7.4.3 Use of predefined namespaces

The SIMOTION controller has a structured data model (see chapter 8.7.2). This data model allows identifiers with the same name to coexist on different structuring levels (scopes). These scopes are mapped by namespaces in the programming model. Most of them can be addressed directly using predefined identifiers. The variables in the relevant namespace can thus be addressed and used selectively in the program. When scope identifiers are used to access variables, compilation errors or any kind of unexpected behavior do not occur when identifiers with the same name are added to other levels. Because an identifier that is not preceded by a namespace can be resolved differently after program changes and the following compilation, an undesirable variable access could occur.

Table 5 shows an overview of existing scopes. Starting from the current scope, the compiler searches through these for the specified identifier. If one of the predefined scope identifiers is used, only the symbols available in the relevant scope (variable identifiers, user-defined data types, POUs, etc.) will be found. The predefined scope identifiers are defined in the unit-global scope and cannot be overwritten there.

Table 5 Predefined namespaces (scopes)

Structuring level (ascending)	Scope identifier (namespace)
Method scope	–
POU scope (class, function block, ...)	Public elements can be accessed from external sources using the POU name.
Unit-global scope (unit variables, global data types and POUs in the source and imported via USES, USELIB or USEPACKAGE)	–

Structuring level (ascending)	Scope identifier (namespace)	
Device-variable scope (system, I/O, global device variables)	_device _device._image _device._quality	– all device variables – direct I/O access – status I/O access
Device scope (Technology Objects of the device)	_to	
Project scope (Technology Objects with unique names across device and project)	_project	

This table indicates the sequence in which identifiers defined in different scopes are found during compilation. If the programmer wants to ensure that quite specific variables are used, this can be done quite simply by inserting the relevant scope identifier in front of the variable.

All device variables can be specifically addressed, for example, by programming the namespace "_device". "_to" can be programmed to specifically address the Technology Objects of the device. If namespaces are used to program device variables and TOs, the user program in which they are used does not need to be reworked if the unit-global scope is extended in a way that results in ambiguous identifiers. The right variable will always be addressed. This is an especially useful feature if different editions of supplied libraries are integrated into the system.

Furthermore, the execution levels of existing tasks can be exclusively accessed using the scope identifier "_task". The scope "_alarm" can be used in the same way to access messages defined in the project. These two scope identifiers are also defined in the unit-global scope and cannot be overwritten there.

7.4.4 Declaration of data types, variables and methods

There are various ways of working with arrays. It is possible to input variables directly as an array. Alternatively, an array can also be declared as a user-defined data type and this definition then used to declare the relevant variables. To ensure that the dimension of the array can be changed centrally, it is advisable to at least use a symbolically declared constant to specify the dimension. An even better solution, however, is to declare a type definition and use this. The only case in which a type definition cannot be used is when arrays need to be handled by function blocks or classes.

If arrays contain a very large number of elements, care should be taken when defining functions and methods that these do not need to be copied unnecessarily during runtime. A good solution is to transfer these to function and method interfaces using VAR_IN_OUT. When values are transferred to functions and methods in SIMOTION, the following generally applies:

- VAR_INPUT
  Copy the values during the call
- VAR_OUTPUT
  Copy the values after the call when the output value is assigned
- VAR_IN_OUT
  Transfer the value implicitly as a reference

With large volumes of data, it is thus convenient to transfer them using VAR_IN_OUT. Variables of a simple data type, or structures that do not contain extensive array elements, should only be transferred as a reference if this is functionally essential. The programmer can significantly influence the program runtime if he or she adheres to these principles and designs the interfaces appropriately.

Methods can be defined in an interface, implemented in classes and overridden. It can then be ensured that methods with the same name also have the same method signature. This means that the return value and all elements in VAR_INPUT, VAR_OUTPUT and VAR_IN_OUT for these methods are identical and need to be declared in the same sequence. It can thus be ensured that all these methods have the same call interface.

It is also possible to specify a default value in the method declaration for variables in VAR_INPUT. These variables can be omitted from a call on the basis of this default. Since these values do not belong to the method signature, they can always be defined differently in a method override mechanism. But caution! The user of the methods needs to be absolutely clear about which default assignment is applicable for each call. It is advisable to select identical default assignments across all method declarations or to dispense altogether with default values in methods that can be overridden.

```
INTERFACE iface
    METHOD m_int
        VAR_INPUT in : INT := 1;    END_VAR
    END_METHOD
END_INTERFACE

CLASS cl_base IMPLEMENTS iface
    METHOD PUBLIC m_int
        VAR_INPUT in : INT := 3;    END_VAR
        ;
    END_METHOD
END_CLASS

CLASS cl_derived EXTENDS cl_base
    METHOD PUBLIC FINAL OVERRIDE m_int
        VAR_INPUT in : INT := 5;    END_VAR
        ;
    END_METHOD
    METHOD m_check
        VAR
            v_if : iface;
        END_VAR
        v_if := THIS;
        v_if.m_int();   // call of cl_derived.m_int( in := 1 )
        SUPER.m_int();  // call of    cl_base.m_int( in := 3 )
        THIS.m_int();   // call of cl_derived.m_int( in := 5 )
    END_METHOD
END_CLASS
```

7.4.5 Preparing structured data for transmission

It is often necessary to transfer data from the control system to another controller or other systems (chapter 3.5.8). SIMOTION also has a range of useful features for doing this in addition to its actual communication functionality. We are going to explain the potential offered by these features in more detail using a slightly more

complicated example. Our task is to send either a data block or a command after a header with a predefined format.

```
CLASS cl_sendData
    TYPE PUBLIC
        s_data : STRUCT // public struct to define the data-content
            v_s1 AT %B0 : DINT;  // use explicite offsets
            v_s2 AT %B4 : INT;
            v_s3 AT %B8 : LREAL;
        END_STRUCT
        s_cmd  : STRUCT // public struct to define a command
            v_cmd    AT %B0 : INT;
            v_subcmd AT %B2 : INT;
        END_STRUCT
    END_TYPE
    TYPE PRIVATE
        e_version : (ver_1:= 1, ver_2:= 2);          // version type
        e_content : (empty:= 0, data := 1, cmd:= 3); // content type
        s_header : STRUCT    // definition of communication-header
            v_ver AT %B0: e_version;
            v_cnt AT %B4: e_content;
        END_STRUCT
        s_sendBuf : STRUCT OVERLAP  // use a union to cast to byte-array
            head AT %B0 : s_header; // header at the begin
            data AT %B8 : s_data;   // data or command after the head
            cmd  AT %B8 : s_cmd;
            buf  AT %B0 : ARRAY [0.. _sizeof(s_header) +
                   MAX(_sizeof(s_data), _sizeof(s_cmd))-1] OF BYTE;
        END_STRUCT
    END_TYPE
    VAR CONSTANT PRIVATE  // constants for the supported header types
        c_headdata_v1 : s_header := (v_ver := e_version#ver_1,
                                     v_cnt := e_content#data);
        c_headcmd_v1  : s_header := (v_ver := e_version#ver_1,
                                     v_cnt := e_content#cmd);
    END_VAR
    VAR
        conID : DINT;     // the id of the connection;
    END_VAR
    METHOD PUBLIC FINAL sendData : DINT   // data-transfer
        VAR_INPUT   data   : s_data;    END_VAR
        VAR         sndbuf : s_sendBuf; END_VAR
        sndbuf.head := TO_BIG_ENDIAN(c_headdata_v1);
        sndbuf.data := TO_BIG_ENDIAN(data);
        sendData := _tcpSend( connectionId := conID
         ,nextCommand := EnumTcpNextCommandMode#IMMEDIATELY
         ,dataLength  := DINT_TO_UDINT(_sizeof(s_header)+
                         _sizeOf(s_data))
         ,data        := sndbuf.buf);
    END_METHOD
    METHOD PUBLIC FINAL sendCmd : DINT    // command-transfer
        VAR_INPUT   cmd    : s_cmd;     END_VAR
        VAR         sndbuf : s_sendBuf; END_VAR
        sndbuf.head := TO_BIG_ENDIAN(c_headcmd_v1);
        sndbuf.cmd  := TO_BIG_ENDIAN(cmd);
        sendCmd := _tcpSend( connectionId := conID
         ,nextCommand := EnumTcpNextCommandMode#IMMEDIATELY
         ,dataLength  := DINT_TO_UDINT(_sizeof(s_header)+_sizeOf(s_cmd))
         ,data        := sndbuf.buf);
    END_METHOD
END_CLASS
```

When defining the structure of the data to be transferred, the user can also specify the memory layout. The offsets of the elements in question are explicitly defined in the relevant definitions of s_data, s_cmd and s_header. The definition of s_sendBuf makes it possible to pass the data as a byte-array to the communication functions. The keyword OVERLAP is included in the declaration of this structure to permit an overlap in the memory between the elements contained in the structure. This solution permits a data block or command block to be programmed after the header, while the element buf allows access to the entire contents in the form of a byte-array. The size of the array is calculated according to the sizes of the structure elements involved. Here we can see that SIMOTION also allows an expression that can be calculated during compilation to be used at all those points where constants are required.

The constant values for version and content are explicitly specified in the ENUM definitions e_version and e_content. Constants can naturally be used instead of an enum.

These definitions help to determine the data structure as well as certain values for the communication services. They are now used in the two functions sendData and sendCmd in order to assemble the transfer buffer efficiently. For this purpose, the header required in each case and the data or command are copied to a temporary element of type s_sendBuf in the endian required by the communication service. The actual communication function is transferred to this data area as a byte-array by means of the member buf. Using this system, the transfer buffer can be assembled with a minimum number of copy operations.

8 Introduction to SIMOTION

In the chapter above we discussed the object-oriented programming mechanisms recently added to SIMOTION. Users who already know the SIMOTION system will not have any difficulties in using the example programs. They will be familiar with the SCOUT engineering environment that is supplied with SIMOTION and will have enough knowledge of SIMOTION to be able to test the programs.

Users who are not yet acquainted with SIMOTION will get to know this system as well as SCOUT in this section. To help new users get to grips with object-oriented programming later on, we would advise them to study this introduction first.

The demand for ever more flexible and sophisticated production machines means that the control systems themselves need to satisfy ever more challenging requirements.

SIMOTION is a motion control system developed by SIEMENS. It has been on the market now since 2002 and used in many different kinds of machine. It is mainly used for applications in which motion control plays a central role. The SIMOTION system has been designed primarily to meet current and future motion control requirements.

But what is so special about SIMOTION? Before we can answer this question, we need to take a brief look at the development history of control systems. We discussed this subject in some detail in chapter 1.

8.1 Classic development of control systems

The possibility of using control systems in mechanical engineering applications first became a reality in the mid-1970s with the advent of programmable logic controllers (PLC). Thanks to this technology, it became possible to program the required functionality using programming devices that had in many cases been specially developed for the relevant PLC. As a general rule, the controllers were equipped with connections for digital I/O components via which they were connected to the machine sensors or actuators. A program running cyclically in the controller combined the signals and controlled the actuators (motor contactors or valves, for example).

Further advances in the technology such as bus system integration or analog signal processing ensured that PLCs became more widely accepted. The integration of bus systems later allowed visualization components (Human Machine Interface HMI) to be directly connected as well, thus providing a means of operating and diagnosing the machinery. Special function modules (e.g. motion control or technology modules) with their own program code were later developed to perform specific tasks. While these needed their own drivers, they could be integrated into the control systems via appropriate interfaces or buses. Figure 55 shows the interaction between components.

As it became possible to integrate additional special modules, the programmable logic controllers became even more flexible in their application. As a result of this

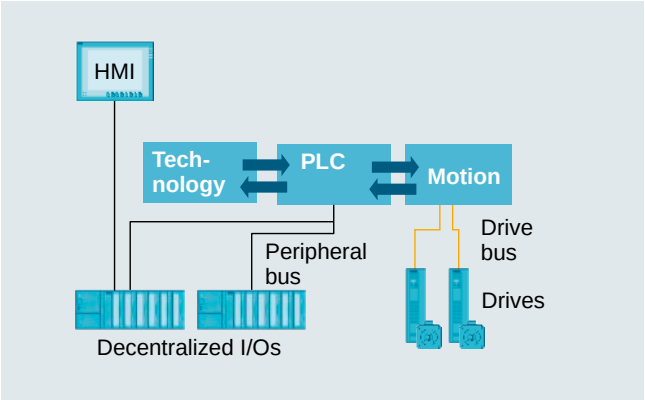


Figure 55 Interaction between PLC, technology modules and motion control

development, the tasks of the PLC programs have significantly changed over the intervening years. In addition to the actual control task, an ever growing number of data management functions were added to the program.

The increasing numbers of additional functions integrated in the programmable logic controller had an impact on the program and the programmer. Since it is now possible to connect specialized modules and visualization equipment to the controller, it is inevitably true that a significant proportion of the control software to be written by the programmer is dedicated to module control and management tasks. The more extensive the functional scope of the special modules, the more complicated the control software.

Visualization systems can be used to operate machinery, but they also provide valuable troubleshooting support to the service engineer when the process is interrupted by an error. Error diagnostics and the associated data management functions are an integral component of the modern PLC program.

The connection of actuators (frequency converters or servo controllers, for example) to the programmable logic controller via bus systems is now state of the art. New mechanisms for accessing data stored in the drive components are being added continually in order to allow dynamic adaptation of the actuators to changes in the process environments in the machine (e.g. new products). This functionality must also be reflected in the PLC program that manages, reads out or changes the drive data or diagnoses the drives themselves.

8.2 New control concepts required

The trend in favor of isolating machine movements so that they are performed by controlled drive axes that are coupled to controllers via buses provided the impetus for developing a new control system. The design concept was to allow functions such as PLC programming, axes with motion control and technological functions (temperature control, for example) to be implemented in a homogeneous system with standardized programming procedures. Since the system would need to be

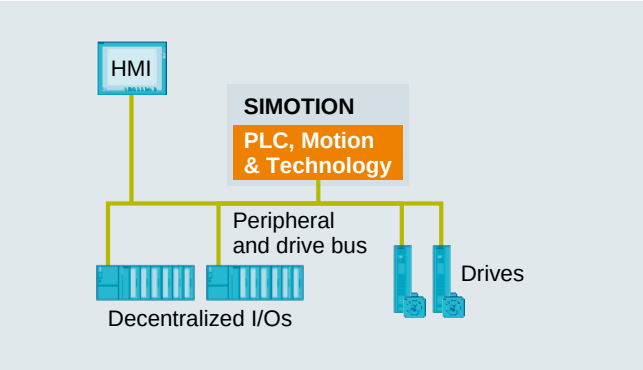


Figure 56 Integration of PLC, motion and technology

deployable in a wide variety of machines, it would need to be a high-performance, scalable system.

SIMOTION was developed to satisfy these demands. To save the user the trouble of programming the synchronization between different processes that might have different time responses, all of the disciplines (PLC, motion and technology) have been combined in a single system (Figure 56). The components work in a shared execution system and can be operated in different tasks depending on their requirements. As a result, all components and processes are fully synchronized at all times. The possibility of connecting drives via a bus system makes the system highly scalable – from just a few individual axes to large quantity structures with more than one hundred axes.

I/O components and visualization equipment are also connected to SIMOTION via bus systems. In this case, the communication link may be isochronous (for the drives) or asynchronous (for the I/O devices).

8.3 Technology Objects in SIMOTION

In order to increase the flexibility of the SIMOTION system, technological components such as axes, encoders, measuring inputs, output cams, but also other functions such as synchronous operation connections, are represented by Technology Objects (TO). These objects and their data form a configurable unit. These data include all the information required for the object including the coupling mechanisms. An axis thus knows, for example, how it must communicate with the connected drive component. The requisite communication process is thus a property of the axis and the user need not program anything extra with respect to communication (Figure 57). This is a standard feature of the SIMOTION system that helps to reduce programming requirements.

Each object is a software representative of some feature of the machine's hardware. Objects can be addressed via a function call interface in the program code. The programmer can therefore issue commands to the object such as "travel from A to B at velocity F". Couplings between axes can be activated or deactivated via

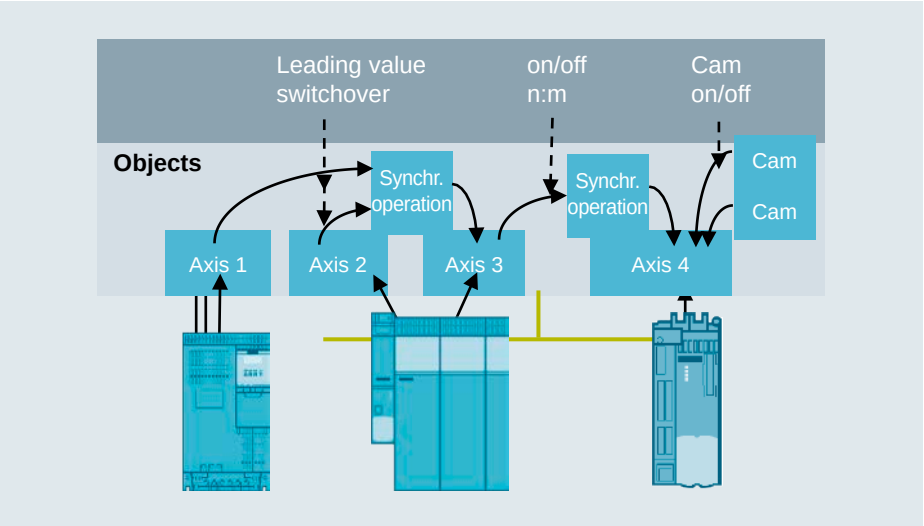


Figure 57 Technology Objects in SIMOTION

commands. Once parameterized, output cams and measuring inputs perform their function independently when the axis reaches the defined positions. The number of Technology Objects that can be used in SIMOTION is not limited by the capacity of the system. But each object occupies (depending on the type and version selected) corresponding memory and computing time in a CPU. As a result, the CPU used does impose certain limits according to the amount of available memory and the performance of the hardware used.

SIMOTION is available in various different hardware platforms with varying degrees of performance and scalability. The degree of integration of the runtime kernel and the possibility of coupling TOs across SIMOTION platforms makes it extremely easy to distribute functionality. If a hardware constraint is encountered, the user can simply choose a higher-performance CPU or distribute the TOs among different CPUs. Existing Technology Objects can easily be relocated to another CPU in the engineering system.

The engineering system features wizards that guide the user through the process of creating Technology Objects. TOs are not programmed, but simply configured.

With this Technology-Object-based model, it is extremely easy to flexibly adapt and scale the functions required within a machine.

8.4 Three hardware platforms

SIMOTION features a standardized scope of functions in three different hardware platforms (Figure 58). All platforms have the same basic functionality and the same basic interfaces. The only differences between the platforms relate to their performance and the associated expansion options.

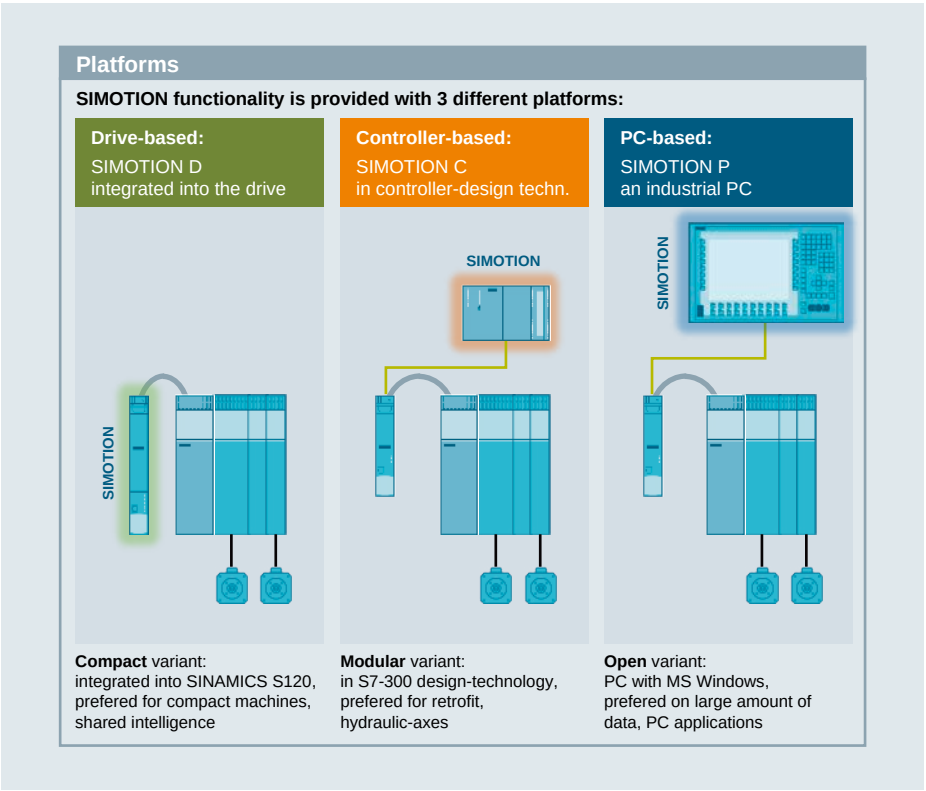


Figure 58 The 3 hardware platforms of SIMOTION

Drive-based variant SIMOTION D

SIMOTION D is a hardware platform that combines SIMOTION and drive functions in a single module. Since the open-loop control and closed-loop drive control functions are integrated in a single hardware unit, the drive power units can be directly connected to the control system so that the overall design is extremely compact. The system can be extremely finely scaled starting with the D410-2 single-axis controller up to the high-performance multi-axis system D455-2 with up to 128 axes. The user can use Profibus and Profinet to couple I/O components or other equipment to the SIMOTION system.

Controller-based variant SIMOTION C

The user can directly connect analog axes (hydraulic axes, for example) to the SIMOTION C without the need for any additional hardware. The encoder interfaces required are an integral part of the system. I/O modules from the SIMATIC product range can be directly connected. The system can be expanded or coupled to further equipment by means of Profibus or ProfiNet (depending on variant).

PC-based variant SIMOTION P

A user who needs an industrial PC that is an open system will prefer SIMOTION P. The SIMOTION kernel runs on a Windows operating system and can therefore be expanded by additional software components. A Profibus or Profinet board is available for coupling the system to other components or I/O devices.

8.5 Connecting drives and I/O devices to SIMOTION

A control system requires a machine to have actuators (such as drives or cylinders) in order to convert motion commands into movements, and sensors or control keys that are connected to the control system.

This connection is provided by appropriate I/O devices such as input or output modules. These are linked in turn to the SIMOTION controller via bus systems (Figure 59). All SIMOTION hardware platforms offer the PROFIBUS or PROFINET bus systems for coupling I/O components and/or drives.

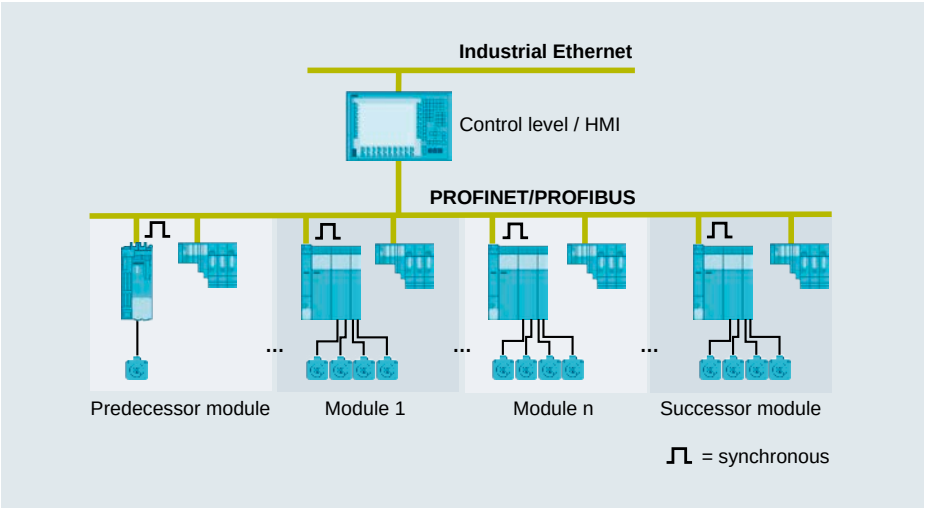


Figure 59 SIMOTION with drives and I/O devices

8.6 Handling kinematics in SIMOTION

A large number of functions are integrated in the SIMOTION system. The upper end of the functional range is rounded off by the standard kinematics for handling equipment. Figure 60 shows the kinematics supported by SIMOTION.

These kinematics are directly integrated in the SIMOTION system. Like axes or technological functions, kinematics are implemented in the form of Technology Objects. The user inserts a TO path object and sets its parameters in dialog boxes specifically provided for the object. The path object utilizes the axis objects already

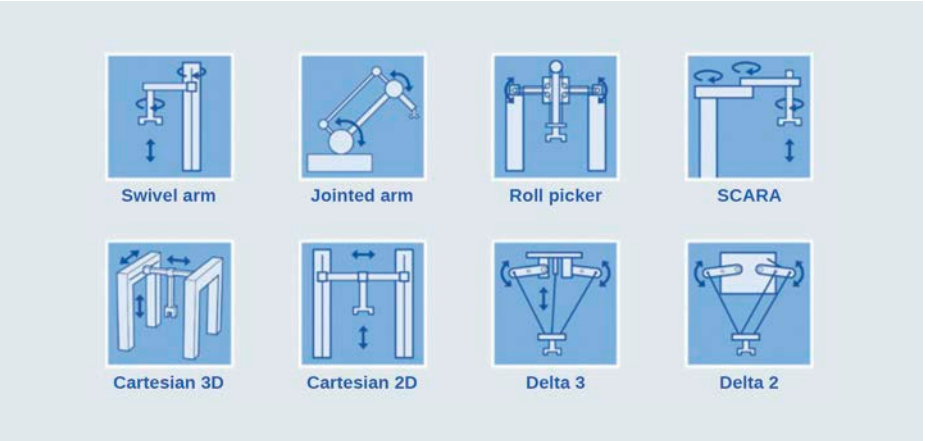


Figure 60 Kinematics supported by SIMOTION

programmed in the system to control the motion. The path object can calculate the kinematic transformations itself. This makes it easy to program the motions in a Cartesian coordinate system.

8.7 SIMOTION's programming model

The three SIMOTION hardware variants are easily exchangeable. This exchangeability defines a fundamental characteristic of the runtime system. Each SIMOTION variant must basically offer the same scope of functions. In other words, none of the hardware variants limits the number of Technology Objects or sets any defined programming limits. All SIMOTION systems have the same command set and the same data model. Only if all SIMOTION devices share this property is the user able to transfer programs between hardware variants.

The hardware variants naturally differ in terms of their performance and memory capacity. A given hardware variant may only be able to compute a specific number of axes per clock cycle or hold a certain size of program in its memory.

If the number of programs exceeds the storage capacity of one hardware variant, another variant with more memory and higher performance can be used instead. If the highest-performance hardware is already installed, the quantity structure will need to be distributed among several SIMOTION systems. The fact that functions are implemented on the basis of Technology Objects in SIMOTION is also helpful in such cases. TOs can be assigned to the relevant hardware and thus reused without any problems. The simplicity with which user programs can be distributed among several SIMOTION CPUs depends very heavily on the programming method used. A modular software concept that has been thought through and carefully programmed in advance significantly reduces the work involved in distributing user programs.

The modular design of the user software therefore plays an important role when software changes need to be made. It is therefore important that the programmer is familiar with the programming model implemented in SIMOTION.

8.7.1 The units of SIMOTION

The user generally programs functions in blocks. SIMOTION provides the following blocks for the user:

- Function blocks
- Functions
- Programs
- Classes (as of V4.5)

IEC 61131-3 refers to these blocks as "Program Organization Units" (POU). This type of block is an enclosed unit with the properties (variables) and programmed behavior (function) specified by the user.

These blocks are not handled as individual elements in SIMOTION, however, but organized in a "unit". A unit therefore functions as a container in which the user programs blocks (comparable to the usual PC programming practice of storing source texts). Several blocks (programs, functions or function blocks) can be stored in one unit. The user can use a unit to group individual blocks according to logical criteria and to form collections of interrelated functions.

Nevertheless, we still need to explain the specific advantages of this kind of modeling. The units in SIMOTION are two-part containers comprising an interface section and an implementation section. A C or C++ programmer would recognize a comparable sectionalization between header and source text files.

The programmer stores the blocks (POUs) that he or she has programmed in the implementation section. In the interface section, the user can specify the functions (blocks) and properties that must be available outside the unit. The user can therefore adopt this units approach to implement modular software concepts without needing to apply object-oriented programming mechanisms.

If the data (blocks or variables) of a specific unit need to be used in other units, it is possible to establish connections between units by means of the keyword "Uses" (Figure 61). If a unit has no connections, its variables and functions cannot be accessed.

It would be possible, for example, to program various machine modules in units. In this case the connected units can "see" only the data that the programmer has made available in the interface section of the unit. In Figure 61, the public data of unit A are visible in unit B. Unit C can also see the data of unit B, but also the data of unit A via the connection in unit B. If the programmer does not want the data of unit A to be visible in unit C, however, he or she must store the Uses connection in unit B in the implementation section. If this is done, the data of unit A are still visible in unit B, but not in unit C. In this case, the calculation in unit C "Diff:=Speed_A-Speed_B" would no longer function because Speed_A is no longer visible. But the programmer can easily solve the program by making public a new variable (Speed_of_A) in B and using this in the formula.

It is thus possible to achieve software modularization with data encapsulation (information hiding). The Autocomplete mechanism of the editor knows the connections and displays only the relevant variables (data). Variables with the same name may be stored in different units provided that the units are not connected. If the programmer accidentally programs elements with conflicting names, the compiler will output a warning or error message during program compilation.

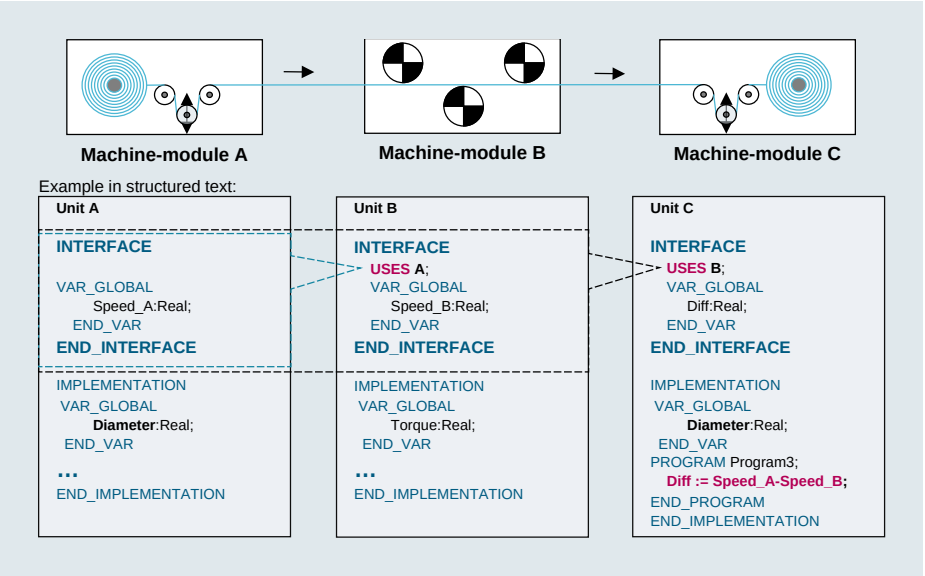


Figure 61 Programs and data are organized in units

With this means of modularizing the software at unit level, the user has a certain degree of control over the access to variables. This mechanism does not work for global elements (such as function blocks) because the instance data for globally accessible function blocks are also set up as global data. More effective encapsulation that is also applicable to global data can only be achieved by object-oriented programming or FBs with methods.

Note about interface

It is essential for readers not to confuse the interface section of a unit with the interface used in object-oriented programming. A unit in SIMOTION is basically divided into the interface section and the implementation section.

SIMOTION uses the interface section of a unit that begins (as it does with OOP interfaces) with INTERFACE and ends with END_INTERFACE to identify the data types, variables, constants and POUs that a unit makes visible for use outside the unit. The programmer can now define one or more interfaces for OOP within this section that will also be available for use in other units. But an OOP interface can also be defined in the implementation section of a unit. This OOP interface can be implemented in classes and used as a variable only within this unit.

8.7.2 The variable model in SIMOTION

A SIMOTION controller provides the user with a variety of ranges of variables for use in his or her application (Figure 62). All variables belong to one device and can be declared in appropriate editors by the user.

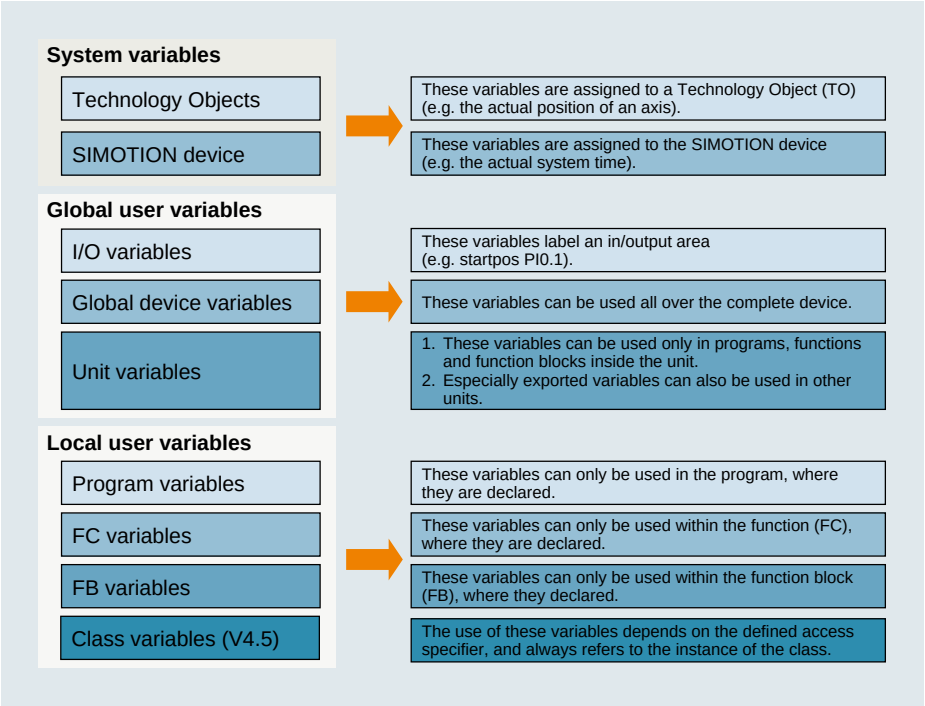


Figure 62 Variable model of SIMOTION

System variables

The first range of variables are the system variables. As the name suggests, these are variables that the system makes available to the user. The programmer can therefore use them, but not generate them.

System variables are in turn divided into two ranges. The system variables of the device are set up automatically when a device is configured. The second range of variables are those that belong to a Technology Object (TO).

System variables of the SIMOTION device

Each SIMOTION controller has system variables that can have different features depending on the device. In this range, for example, the system makes variables such as the system time, the current time values of individual tasks or information about the system fans available to the user for analysis.

The system variables of the device are visible in the detail view of SCOUT when the device is selected in the project navigator (see chapter 8.9.1).

System variables of TOs

The system variables of a TO are set up when the user configures a Technology Object (TO) in SCOUT. Different types of TOs with varying functionality are available in SIMOTION. As a consequence, each TO has its own specific data. These data naturally vary between the different types of TO.

The following Technology Objects can be set up in a SIMOTION device:

- Drive axis
- Positioning axis
- Following axis or synchronous axis
- Path axis
- Following object
- Path object
- External encoder
- Measuring input
- Output cam, cam track
- Cam
- Temperature controller
- Addition object
- Formula object
- Fixed gear
- Sensor
- Controller object

The procedure for creating Technology Objects is described, for example, in chapter 8.9.6 “Creating axes” or the relevant SIMOTION documentation.

Global user variables

Global user variables are valid in all of the units within a SIMOTION device. In other words, they can be used anywhere within programs. Since they are user variables, they must of course be defined by the user. Global user variables are divided into two ranges. I/O variables and global device variables belong to the first range that is assigned to the SIMOTION device. Each variable must have a unique name within this range. Global variables declared in the units belong to the second range. Since these ranges are managed hierarchically, variable names may sometimes be duplicated. If this happens, the compiler issues an alarm to indicate that the identifier is hiding another variable. Unless an additional name is inserted in front, therefore, only the locally defined variable with the same name can be accessed in the program.

I/O variables

These are all variables that refer to input or output devices. The SCOUT engineering system supplies the user with an address list for defining input and output variables, i.e. a table editor in which the variables can be defined. Variable names in the address list are normally assigned to an input or output. Before this can be done, the relevant I/O devices must first be connected to the SIMOTION system. I/O component connections are engineered in the hardware configuration. Chapter 8.9.4 explains how the hardware configuring system is operated.

Global device variables

Like I/O variables, global device variables belong to the device. The user can define these variables in a table editor (similar to the address list). The table editor is opened using the menu option "Global device variables" in the SCOUT project navigator. The variables defined in "Global device variables" can be used anywhere within the device.

Unit variables

These variables belong to a unit. A unit is a program container in which the user can set up programs for programming. The unit consists of two sections – the interface section and the implementation section. If the user defines variables in the interface section, they will also be visible outside the unit and may therefore also be used in other units (see chapter 8.7.1).

Local user variables

Local user variables refer to the programs, function blocks, classes with their methods, and functions within a unit. Local user variables are thus valid within the declared range. In the same way as global user variables, local variables are also defined by the user. If the user accidentally or deliberately assigns a name to a local variable that has already been assigned to an existing global variable, the compiler issues an alarm to notify the user that the local variable is hiding a global identifier. This means that the global identifier with the same name cannot be accessed in this program section.

Program variables

These variables can be declared for a specific program, but they are only valid within this program.

FC variables

The variables of a function are valid only within the function. If values need to be transferred to a function, the user needs to do this by means of VAR_INPUT variables. Results can be returned with variables in the VAR_OUT range or via a variable implemented in the function itself. It must be noted that all the variables of a function are temporarily stored in the stack and reinitialized every time they are called. In other words, these variables are temporary.

FB variables

FB variables belong to and are used within a function block. Since an instance needs to be created in order to call a function block, the FB variables are contained in this instance. The variables are thus initialized only at certain times.

Class variables

Classes will be available as of SIMOTION software version V4.5. The variables of a class belong to the class and can be assigned various access identifiers so that they can be used for different purposes. As with function blocks, an instance (object) of

a class must be created before it can be used. Use and initialization of classes has been described in detail in previous chapters.

8.7.3 Libraries in SIMOTION

Using libraries in a project presents a wonderful means of providing reusable software. Various libraries can be stored under the “Libraries” folder in a SIMOTION project (Figure 63).

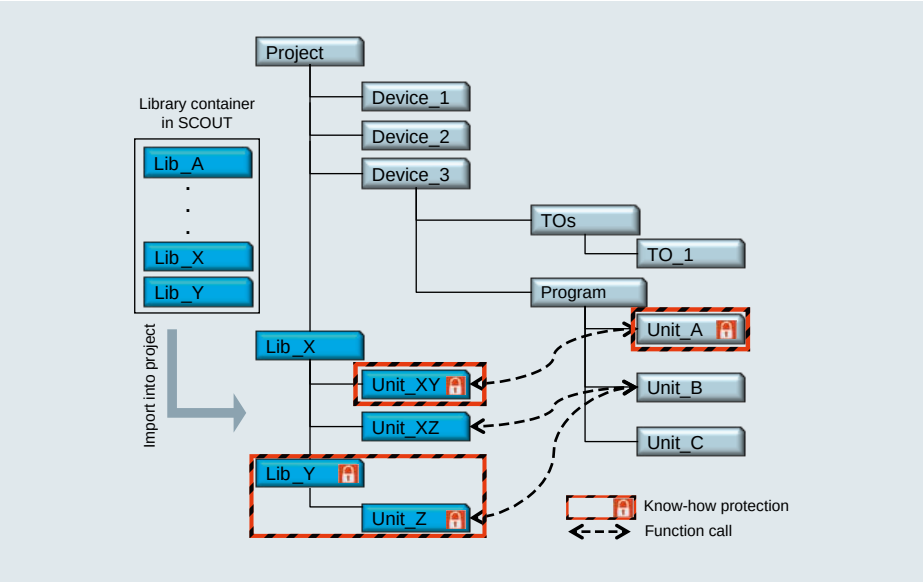


Figure 63 Libraries in the SIMOTION project

Library functions can be used multiple times in each device in the project. Function blocks that can be used over and over again can thus be stored in a library. The program code need only exist once in the project. To use library contents in the device programs, the user simply has to set up a link to the library (USELIB *<Library_Name>*) in order to gain access to the blocks and variables in the library. When the device is compiled, the associated library elements are automatically added to the device data.

Full or partial know-how protection can be applied to libraries, allowing the functions to be used, but not opened. Creators of libraries can use this mechanism to safeguard their know-how. The programmer can of course apply know-how protection to the normal units of a device as well.

8.8 The SIMOTION SCOUT engineering system

The SIMOTION system is planned, configured and programmed by means of the SCOUT engineering system. All editors and tools required to configure machinery are integrated into this engineering environment.

SCOUT is an option package for SIMATIC STEP 7 and requires STEP 7 as its basic operating software. SCOUT TIA is the engineering environment combined with the TIA Portal. SCOUT TIA is supplied as standard with the SCOUT package.

The SIMOTION SCOUT stand-alone software package is available for users who only want to configure the SIMOTION system. SCOUT stand-alone comes with all the necessary STEP 7 components to permit configuring of SIMOTION with the required I/O components. The STEP 7 basic package is not required.

The user can also choose WinCC Flexible as additional software for configuring machine control operator panels. The configuring process for HMI systems is then directly integrated into the SCOUT engineering software.

SCOUT provides you with a complete engineering environment including all the editors and tools that you might need to configure a plant (Figure 64).

This environment is designed to help you configure SIMOTION systems with all necessary components such as I/O devices and drives including HMI within a shared data management environment. You program the systems in the editors integrated in SCOUT. But commissioning and troubleshooting are also directly supported by appropriate diagnostic tools.

The descriptions below focus on the essential points so that readers will have all the information they need in order to try out the example programs.

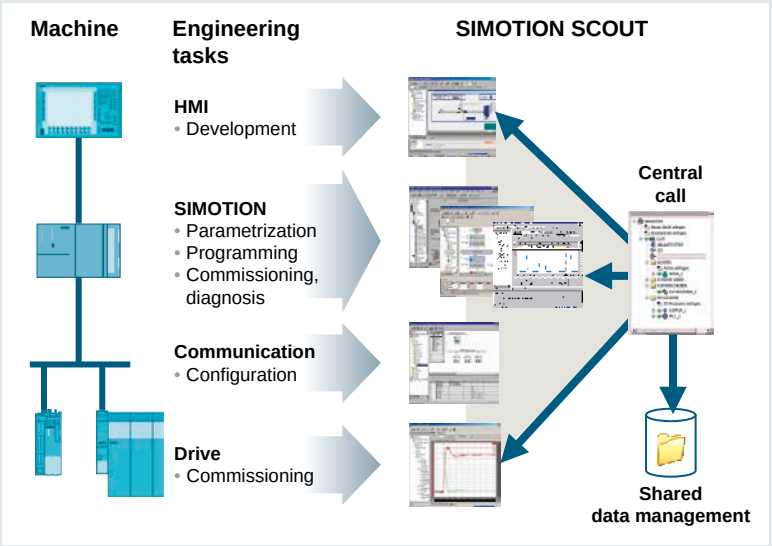


Figure 64 SCOUT engineering system

8.9 Components of SCOUT

SCOUT is a complete configuring environment for SIMOTION which is structured according to a workbench principle (Figure 65). In other words, its structure is based on Windows Explorer.

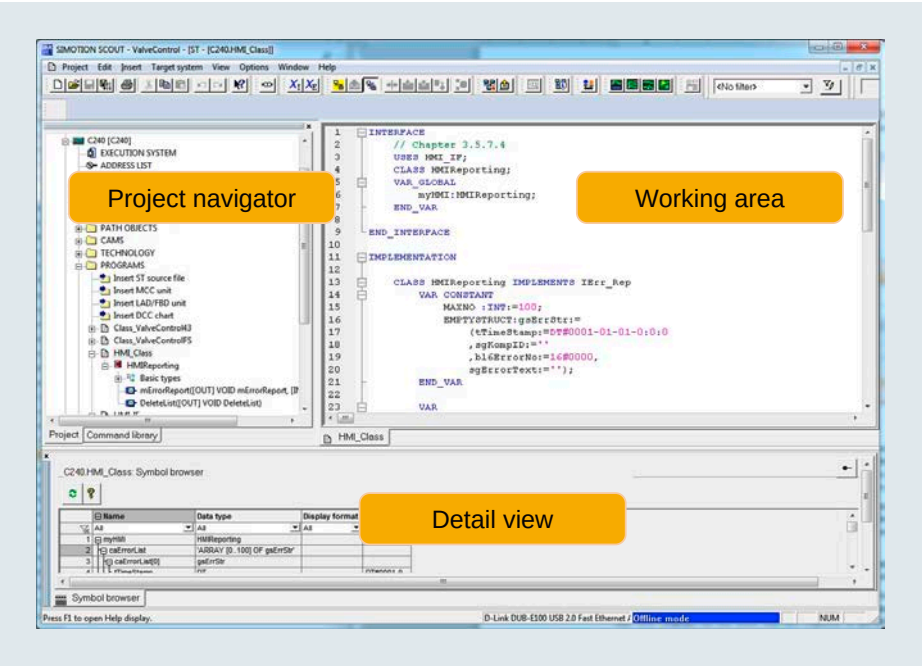


Figure 65 SCOUT workbench

You can see the project navigator (PNV) in the left-hand window. When you open an existing project, the SIMOTION devices with their subordinate components are listed in a tree structure in the PNV. When you double-click on an entry in the tree, the editor for the relevant component opens on the right-hand side, i.e. in the working area.

The detail view is displayed at the bottom of the workbench. This view provides you with additional information. When you select an entry in the PNV, SCOUT displays, for example, the variables belonging to the component you have selected (if the object has public variables).

Further tabs might also appear in the detail view. The tab “Compile/check output” appears automatically with the compilation results when compilation is in progress.

Various table editors such as the address list, global device variables, watch tables, alarms or the module online status are also shown in the detail view. As a general rule, this view displays all the tools that are required in parallel to the working area for programming purposes.

8.9.1 The SCOUT project navigator

The project navigator (PNV) (Figure 66) is central to the operation of the SCOUT engineering system. You can insert SIMOTION devices in the PNV. When you insert a SIMOTION device, SCOUT automatically enters other main folders underneath the device.

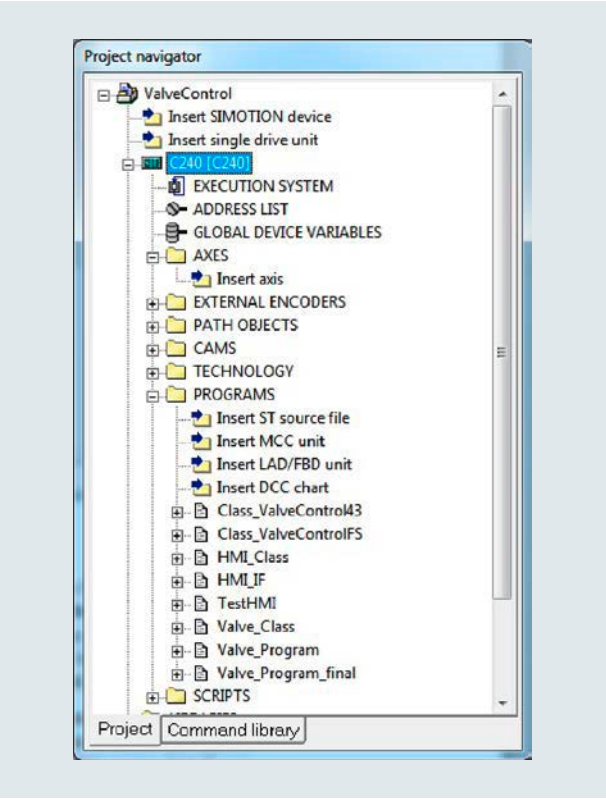


Figure 66 Project navigator

You can insert and configure or program elements belonging to a SIMOTION device in the main folders listed underneath that device. These main folders may or may not contain information (elements), but you cannot delete them. You can expand a folder by clicking on the plus symbol in the tree. When the main folder is open, you will find that it contains entries under which other elements can be inserted.

When you double-click on the Insert command, the appropriate insert dialog will open. The insert dialogs will request you to enter data for the element to be inserted. SCOUT proposes a standardized name for the element. You can change this name, but if you don't change it, you must at least accept the proposed name. The element is inserted in the main folder when you double-click on OK. The insert function is explained in detail in the following chapters.

8.9.2 Creating a new project

This is how you create a new project

You start a new configuration in SIMOTION SCOUT by creating a new project.

1. Select menu items Project > New.
   The dialog New Project appears.
2. Enter the project name, e.g. Sample_1, under Name.
3. Under Storage Location (path), enter the path in which you want the project to be stored. The default path is already set.
4. Confirm the new project with OK.
   The dialog box closes.

Default path

The default path for projects is:

- C:\Program Files (x86)\Siemens\Step7\s7proj

Project name and project directory name

The name of a SIMOTION SCOUT project can contain a maximum of 24 characters. The project appears under the full name in the dialogs.

When initially saving the project, SIMOTION SCOUT creates the directory name from the first 8 characters of the project name. SIMOTION SCOUT uses a numerical counter "_1", "_2", ... to resolve conflicting names resulting from the abbreviation of the 8 characters.

The counter replaces the last characters of the directory name.

Note: It is useful to select project names in such a way that they can be uniquely distinguished by their first 8 characters. The project name and the directory name derived from it thus uniquely identify the same project.

Example (Figure 67)

The project Sample_1 has been set up in the project tree (PNV) in SIMOTION SCOUT. The project folder "Sample_1" is visible in the project navigator.

Further folders (LIBRARIES, SINAMICS LIBRARIES and MONITORING) have been automatically set up beneath. The entries "Insert SIMOTION device" and "Insert single drive unit" are also displayed automatically.

Insert a SIMOTION device

You can insert a new SIMOTION device in the project by double-clicking on this entry. Various devices from the SIMOTION C, D and P product ranges are available for selection. The procedure for inserting a SIMOTION device is demonstrated in the next section.

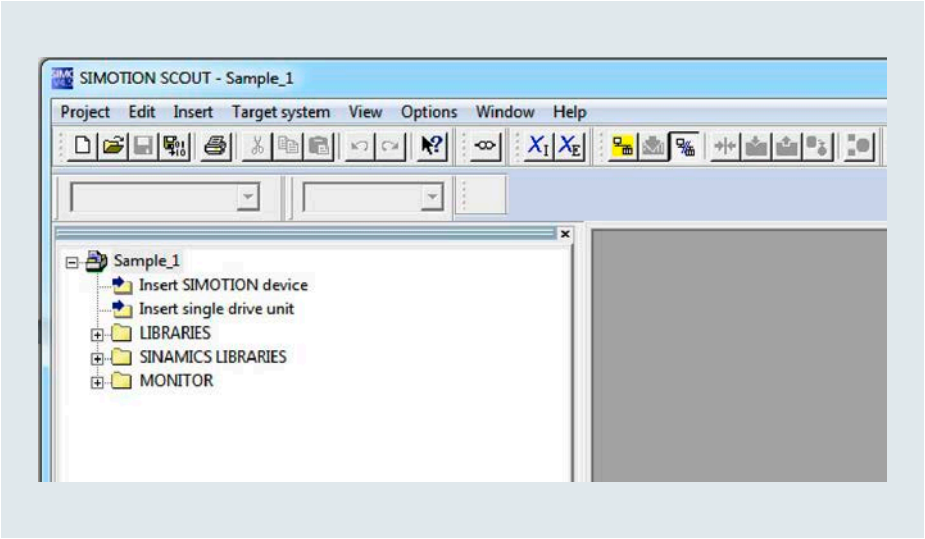


Figure 67 Result of creating a new project

Insert a single drive unit

With this entry you can insert stand-alone drives (e.g. SINAMICS S120) in the project navigator. Single drive units are those drives that will be installed in the plant (e.g. fans or roller conveyors), but are not connected to a SIMOTION axis. You commission them, however, using the wizards in the working area of the workbench in exactly the same way as drives with axes.

8.9.3 Creating a new device

This is how you create a SIMOTION device (e.g. D435-2) in the project

Double-click on Insert SIMOTION device in the project navigator. The Insert SIMOTION device dialog will appear (Figure 68).

1. The SIMOTION D platform is the default setting in the Device list box. If you want to choose a device of another platform, you can change the setting in the “Device” list box.
2. Select the SIMOTION device D435-2 DP/PN from the “Device version” list and choose the firmware version V4.5 of the device used under “SIMOTION version”.
3. If you check the box “Open HW Config”, the hardware configuring window “HW Config” will open when you have finished setting up the new device. In this case, uncheck the box.
4. Confirm the new project with OK. The “Insert SIMOTION device” dialog is closed. SIMOTION SCOUT takes you to the next step. Configure the PROFINET interface.

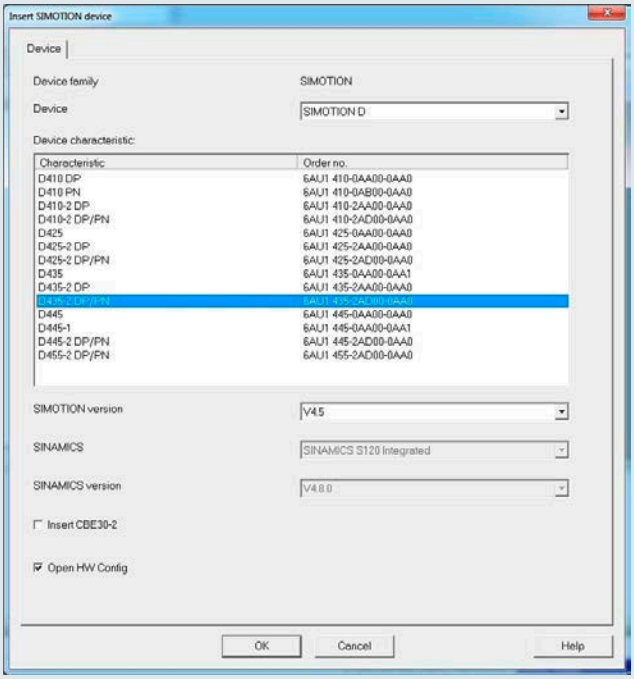


Figure 68 Inserting a device

Note: The configured device version must match the firmware version on the memory card of the SIMOTION device. You will otherwise receive an error message when you go online with the device.

Configure the PROFINET interface

If the SIMOTION device has a PROFINET IO interface, the dialog “Properties – Ethernet interface PNxIO” is displayed. This dialog can be used to integrate the SIMOTION device into an existing PROFINET IO subnet. If no subnet is known, you can create it here.

PROFINET is not used in the sample project (Figure 69). Click on “Cancel”. The dialog box closes. SIMOTION SCOUT takes you to the next step “Set up PG/PC communication”.

Interface selection dialog

SIMOTION SCOUT opens the dialog “Interface selection” (Figure 70). You configure the network communication between the PG/PC and the SIMOTION device in this dialog.

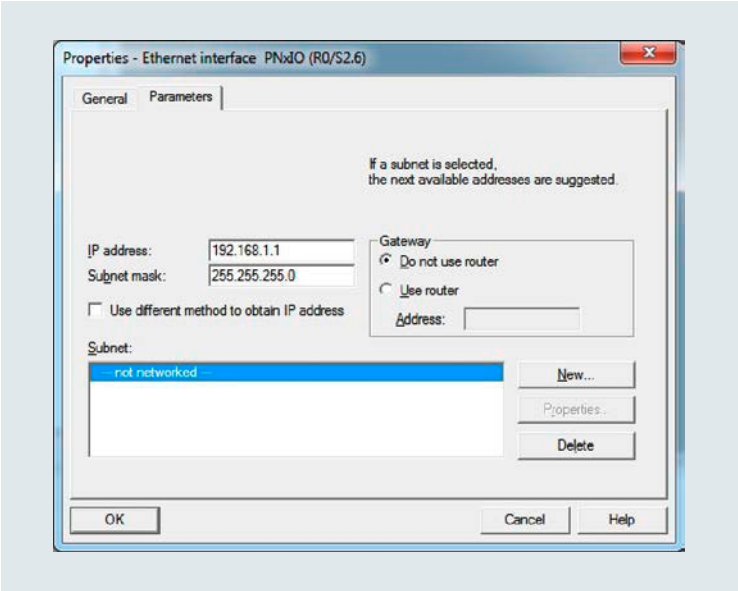


Figure 69 Properties – Ethernet interface PNxIO

The list box “Interface selection for PG/PC connection” in the upper half of the dialog provides a selection list of SIMOTION device interfaces. Listed here are all the interfaces that the SIMOTION device possesses. We will select the interface with X127.

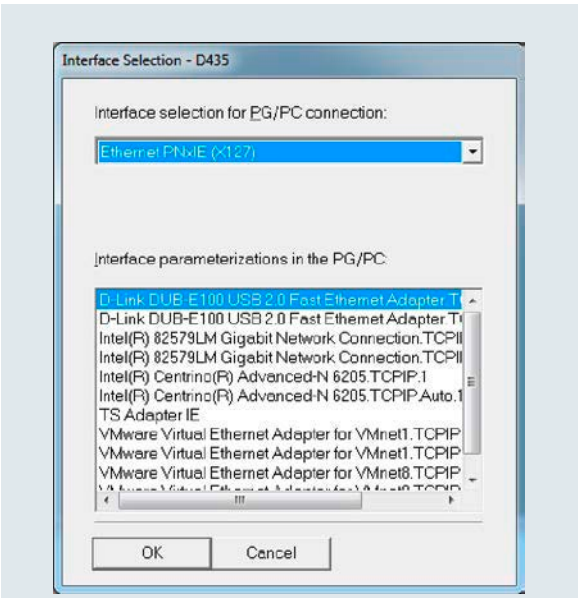


Figure 70 Setting up PG/PC communication

The list box underneath “Interface parameterizations in the PG/PC” lists the interfaces of the PG/PC. The upper field functions as a filter so that only the interfaces with suitable transmission protocol are available for selection. The contents of the selection list depend on the computer hardware and installation. It will therefore vary depending on the computer. The interface of the PG or PC via which the SIMOTION device will be connected must be selected here.

A SIMOTION device is generally connected via an Ethernet interface. The PC on which the SCOUT engineering software is installed often has an Ethernet interface to connect it to the company network. This interface can of course also be connected to the SIMOTION device, but it cannot be used to access the company network at the same time. More than one Ethernet interface is required to connect the PC to SIMOTION and the company network. The simplest solution is an Ethernet adapter with USB. The interface with the company network can then be retained and the SIMOTION device connected at the same time via the USB adapter.

1. Select the Ethernet interface “Ethernet PNIE (X127)” for the SIMOTION D435-2.
2. Then select the prepared Ethernet interface of the PG/PC.
   Confirm this configuration with OK.
3. The dialog “Interface selection” then closes.

8.9.4 Hardware configuration

If you checked the box “HW Config” at the “Insert SIMOTION device” stage (Figure 71), the hardware configuration for the device you have inserted will be displayed when you click on the OK button.

You can configure and parameterize the required bus systems (PROFIBUS and PROFINET) in the HW Config screen. You can select any additional I/O component requirements (such as input and output modules) from a catalog for connection to the bus systems.

HW Config provides a large number of different configuration options. Instructions for operating the HW Config can be found in the documentation “SCOUT.pdf” for SIMOTION devices or in the online help of HW Config itself.

Inputs for the examples

It is not necessary to change the data in HW Config in order to test the example programs. If you checked the box “Open HW Config” at the “Insert SIMOTION device” stage, the interface described below will open.

The HW Config consists of a three-pane window (Figure 72):

- The hardware catalog is visible on the right.
- The working window has two panes:
  - In the upper half, you can see the rack or station frame with the CPU already inserted automatically in slot 2 (example for D435-2). Insert the objects from the hardware catalog here and process them.
  - In the lower half, you can see detailed information about the selected objects.

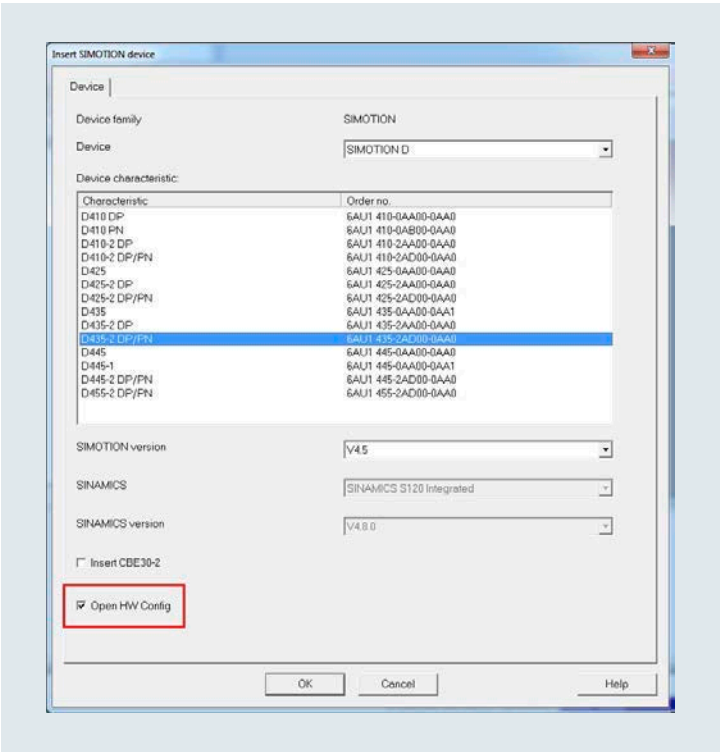


Figure 71 Insert SIMOTION device with “Open HW Config”

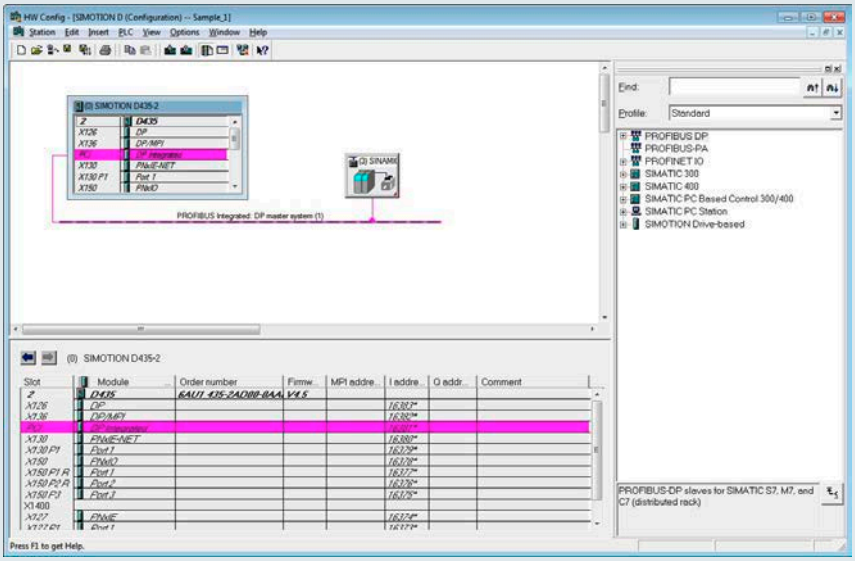


Figure 72 HW Config with the SIMOTION device

Inputs required for the examples

If the window shown in the screenshot above is open, you can close it with the “Close” button or select menu items Station->Exit.

If a real plant were being configured here, it would of course be necessary to add the necessary I/O modules and possibly other drive components.

Note: The variant of SCOUT described here is linked to STEP 7 V5.5 and displays the hardware configuration of this STEP 7 version. You can also use the SCOUT TIA variant. Version 4.5 of the SCOUT TIA is linked to the TIA Portal V14 and the hardware (including SIMOTION hardware) is therefore configured with the TIA Portal.

8.9.5 The SIMOTION address list

You specify your I/O component requirements in the hardware configuration. After defining the I/Os in this way, you can assign an identifier to them for use in SIMOTION. You configure the assignment between hardware and symbolic identifier in the address list in the project.

When you double-click on the entry “Address list” in the project navigator, the address list opens in the lower part of the SCOUT engineering interface (Figure 73).

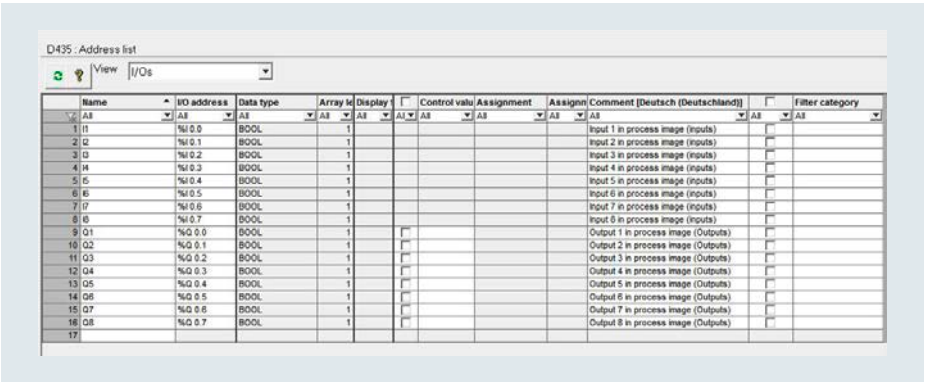


Figure 73 Address list

You assign a symbolic name in the column headed “Name”. You enter the reference to the actual hardware address in the column headed “I/O address”. The engineering system performs a plausibility check on your inputs and displays feedback information immediately.

If you have not yet connected any I/Os, you can use the process image to perform tests. The process image is an internal CPU memory in which specific I/O areas are stored (e.g. byte address 0-64). This internal memory can be used to perform tests even if no I/Os are connected. Further information about the address list can be found in the online help and in the documentation.

8.9.6 Creating axes

Once the SIMOTION device has been inserted and the HW Config interface closed (if it was open), you are now in SCOUT. Figure 74 is a screenshot of the interface.

Now select the entry “Insert axis” under “Axes” in the tree in order to create a Technology Object for an axis. The SCOUT axis wizard is launched when you double-click this entry (Figure 75).

Inputs for the examples

- Enter the axis name for the object in the open window.
- Select the axis name “SpeedAxis_1” so that you will be able to use the axis later in the example programs.
- Check only the box marked “Speed control” as the technology.
- Confirm your inputs with OK and the next input window will appear.

Brief description of the technology

You can specify the type of axis by checking the boxes under “Which technology do you want to use?”. Your selections will determine the characteristics of the axis. After you have entered this data, the axis will be set up with the data required for the Technology Object.

Speed control

Once a drive axis has been created, it is controlled by means of a speed setpoint which it accepts via the integral speed controller.

Motion control is performed using a speed specification without position control.

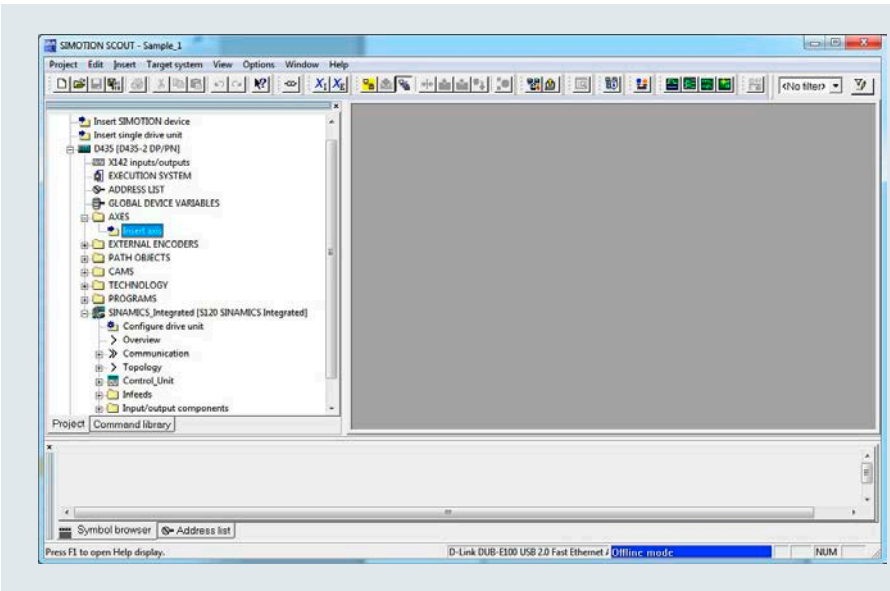


Figure 74 SCOUT with inserted D435-2 device

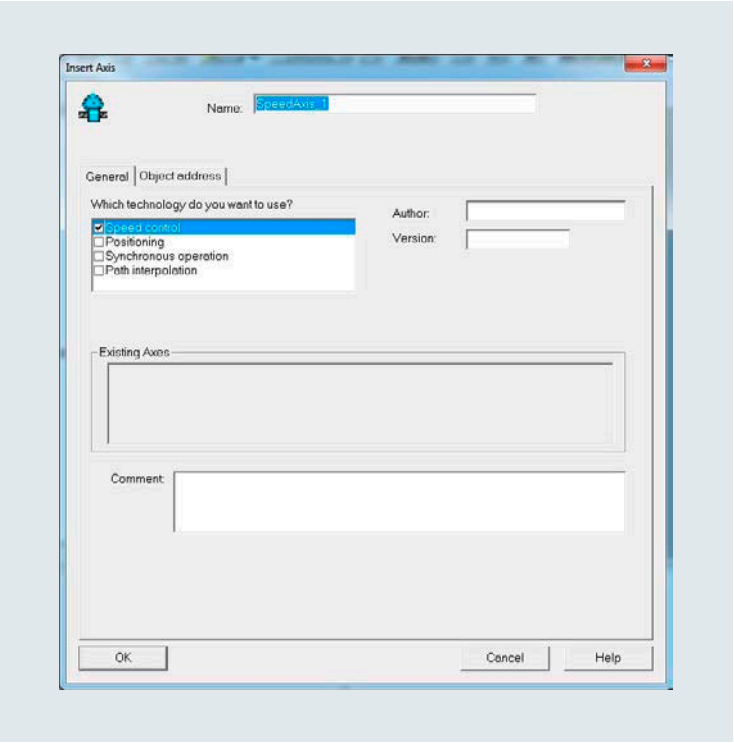


Figure 75 Creating a drive axis

This axis technology corresponds to the minimum axis functionality available in SIMOTION.

The drive axis is referred to by the data type “driveAxis” in reference lists and the program code.

Positioning

SCOUT sets up a position axis when you select “Positioning”. A complete position controller is implemented in an axis of this kind and it can thus be commanded to approach a position at a specific velocity. The position axis is capable of independent positioning. This selection represents the next extended level of an axis and it therefore inherits all the properties of the drive axis. A position axis is thus also capable of processing speed setpoints.

Motions are position-controlled.

The position axis is referred to by the data type “posAxis” in reference lists and the program code.

Synchronous operation

The synchronous axis encompasses an even larger scope of axis functions. It can be interconnected with other axes or master values in order to create axis groupings

with coupling methods of different types. A synchronous axis is also capable of positioning or operating according to a speed setpoint.

The synchronous axis creates a grouping of the following axis and synchronous object. Functions such as master value coupling, synchronization and desynchronization of the synchronous operation, gearing and camming are provided via the synchronous object. The synchronous object can be interconnected with different master values.

Information about synchronous axis applications can be found in the Technology Objects Synchronous Operation, Cam manual.

The following axis is referred to by the data type “followingAxis” and the synchronous object by “followingObjectType” in reference lists and program code.

Path interpolation

A path axis is the highest selectable level of an axis and inherits all the properties of the synchronous axis.

The path axis type can be interconnected with a path object.

The path object can be used to calculate and traverse a linear, circular or polynomial path in the 2D/3D coordinate system for at least two path axes and up to three path axes. A synchronous axis can be traversed in parallel.

A description of how to combine a path axis with a path object can be found in the Technology Object Path Interpolation manual.

The path axis is referred to by the data type “_pathAxis” and the path object by the data type “_pathObjectType” in reference lists and the program code.

After you have confirmed the axis name input and the technology selection with OK, the wizard displays the next input window “Axis configuration” (Figure 76).

The axis type is specified in the axis configuration window. 3 different types are available for selection:

- Electrical
- Hydraulic
- Virtual

Inputs for the examples

The example programs do not require a real axis. Please therefore select the button “Virtual” and then “Next”.

Electrical

An electrical axis is connected to an electric drive. Various methods can be used to connect it to the drive. As a general rule, drives are linked to the SIMOTION controller via a bus system. Before this coupling can be parameterized, a suitable drive component must be configured at a SIMOTION bus system in HW Config. If you have selected “Electrical”, the next window displays a list of the drives that can be connected. The next window appears when you click on “Next”. SIMOTION also supports coupling on the basis of an analog setpoint, but this option is rarely used today. A suitable I/O device must be configured in HW Config beforehand so that SIMOTION can transfer analog setpoints.

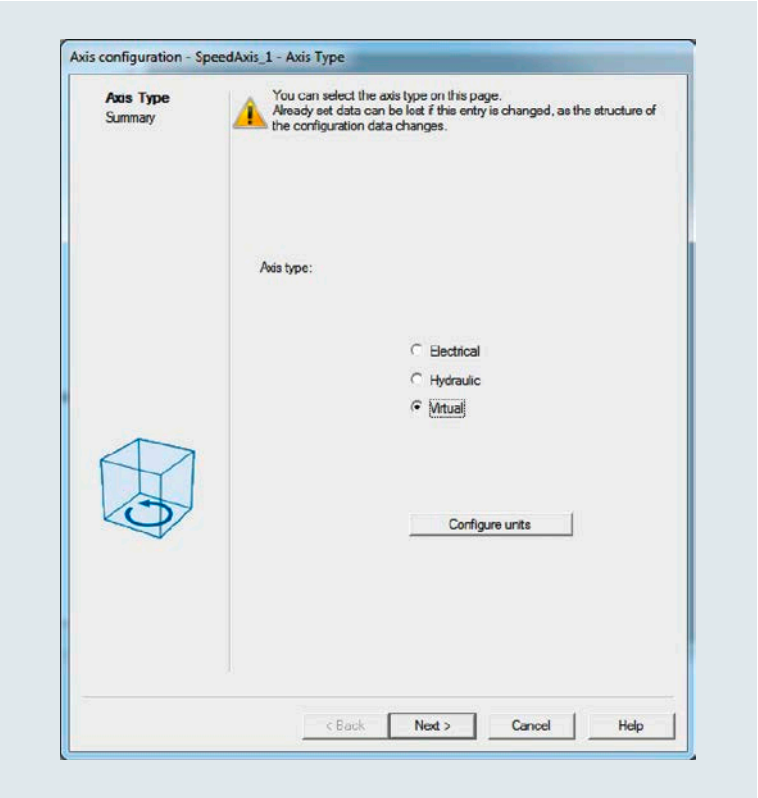


Figure 76 Axis configuration – axis type

Hydraulic

Hydraulic axes operate with control valves and analog setpoints. When you select this option, a different input window with additional selections is displayed. The following valve types are possible:

- Q valve (pressure valve)
- P+Q valve

The following closed-loop controls are possible for hydraulic axes depending on the selected valve type:

- Standard:
  Hydraulic axis without force control or pressure control (Q valve only)
- Standard with pressure control:
  Hydraulic axis with pressure control
- Standard with force control:
  Hydraulic axis with force control

As with analog drives, a suitable analog module must be set up and parameterized in HW Config before the hydraulic axis option can be used.

Virtual

Virtual axes do not have a real actuator (drive or valve) but are axes calculated by the system. The behavior of virtual axes is “ideal” and they are therefore often used as a master value for coupled axes. Virtual axes are useful for testing programs and program sections in development and test environments. This does not of course apply to program sections that are required to process a reaction to axis or drive errors.

Inputs for the examples

After you have entered all the data for a virtual axis, SCOUT displays a summary of your settings. If these are OK, click on the button “Finish” (Figure 77). You have now finished creating the axis and the wizard will close.

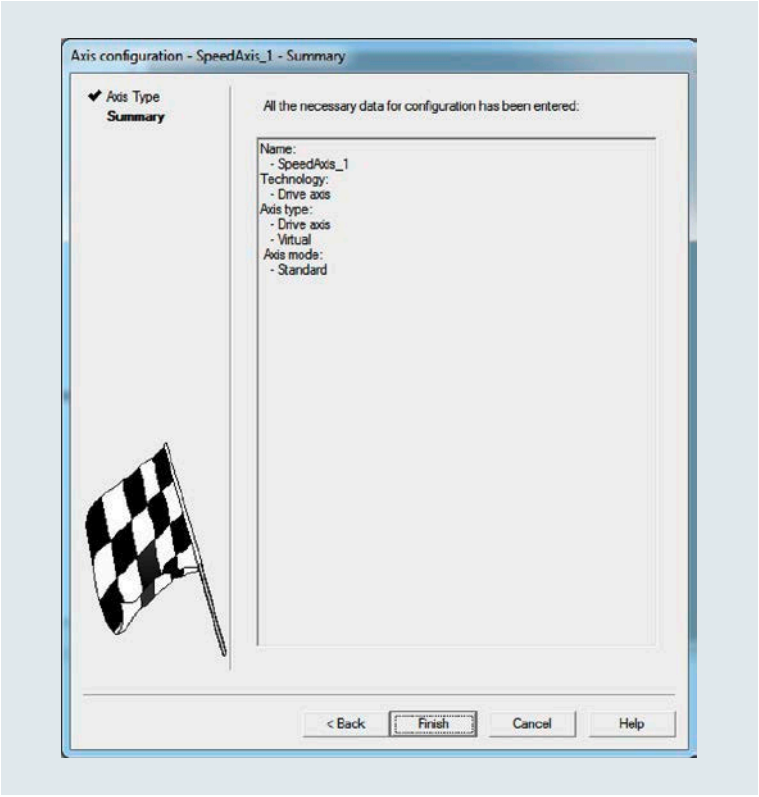


Figure 77 Axis configuration – summary

Subsequent changes to settings

The axis type can be changed subsequently. The configuration window opens in the working area when you double-click on the axis name in the project navigator. You can call the axis configuration window by clicking on the “Change” button in this window.

If you have selected an electrical axis with drive, the drive will also need to be parameterized. The procedure for setting up a drive in SIMOTION is described in brief in the following chapter.

8.9.7 Creating drives

If you have selected the option “Electrical” as the axis type, the wizard displays the drive assignment window when you select “Next” (Figure 78).

If a SIMOTION D435-2 device has been inserted and you have entered no other inputs or parameter settings in HW Config, the axis wizard does not display a drive for selection.

Define assignment later

When this option is selected, you can exit from the wizard and assign the drive later. A programmer involved with the mechanical engineering design will often do this if no information is yet available about the drive to be installed.

Create a drive

A SIMOTION D (drive-based) has the hardware properties of an integrated drive. With this SIMOTION module, therefore, it is possible to create a drive without adding any parameter settings to HW Config.

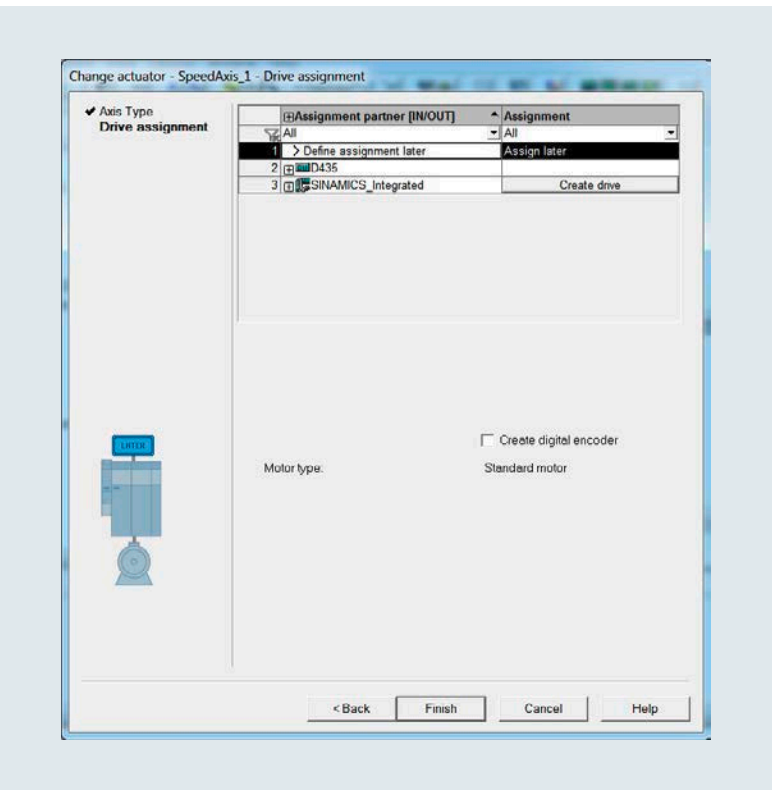


Figure 78 Assigning a drive to the axis

To do this, click on the button “Create drive” next to the entry “SINAMICS_Integrated”. SCOUT opens a second wizard in which the drive can be configured.

You can configure the entire drive train in this drive wizard. The wizard will guide you through various configuration screens that consist of the following steps:

- Configure the infeed
- Insert a drive
- Define the controller structure
- Select a power unit
- Select an enable signal
- Select the motor
- Motor with or without holding brake
- Select an encoder
- Configure the data exchange with the drive
- Summary

After you have worked through the drive wizard, the axis wizard for the axis parameter settings will appear again (Figure 79).

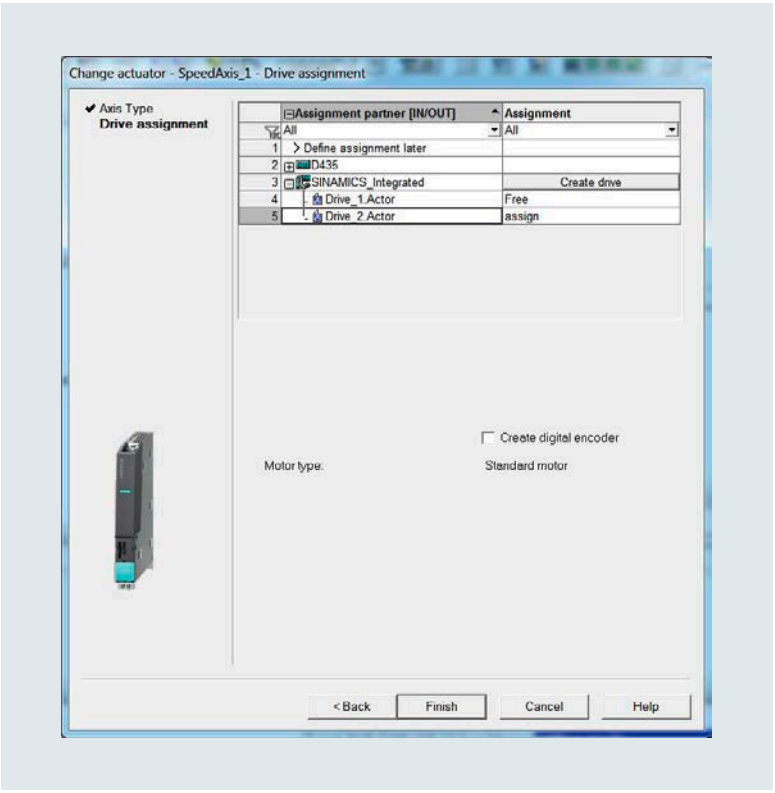


Figure 79 Axis wizard for assigning a drive

Now connect the actuator to the axis. If you have selected a positioning, synchronous or path interpolation axis, you will also have to connect the encoder after you have connected the actuator. This configuration step does not need to be performed for the drive axis and the completion screen is displayed when you click on “Next”. After you have completed these tasks, SCOUT will automatically configure all the relevant data during the next compilation operation. The drive and motor must of course already be physically present in the hardware before the program is compiled. You can find further information in the SIMOTION documentation.

8.9.8 Creating path objects

Like all other Technology Objects, the path object is created in SCOUT. You will find an entry “Insert path object” in the folder “PATH OBJECTS” underneath the SIMOTION device (Figure 80).

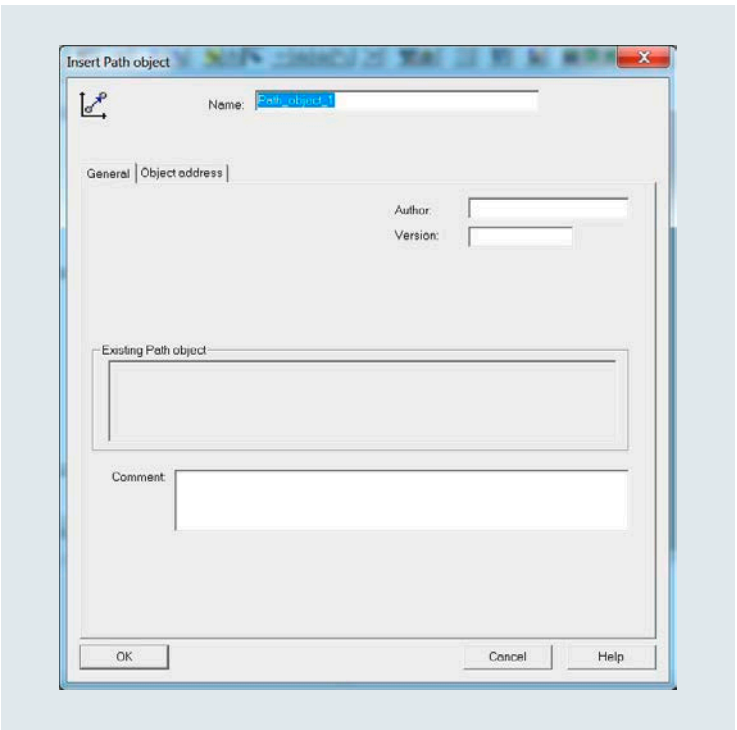


Figure 80 Inserting a path object

Enter the name of the path object by overwriting the default name. The path object is created when you click on the button “OK”. SCOUT inserts the path object in the project navigator (PNV) with a variety of subelements (Figure 81).

Figure 82 shows the configuration screen in which you specify the required kinematics. The drop-down list shows the kinematics that are supported by SIMOTION.

Figure 81 Path object in the PNV

A variety of further configuration screens enable you to assign additional parameters to the path object. These include

- Geometry specifications
- Offsets and rotations
- Initial position definitions
- Specification of units for calculating the kinematics

You can select additional support (such as limits or defaults) for handling kinematics, or functions such as calibration or path control panel for aiding commissioning in the PNV. Certain commissioning functions are available to the user only if an online connection has been set up to the SIMOTION system (e.g. path control panel).

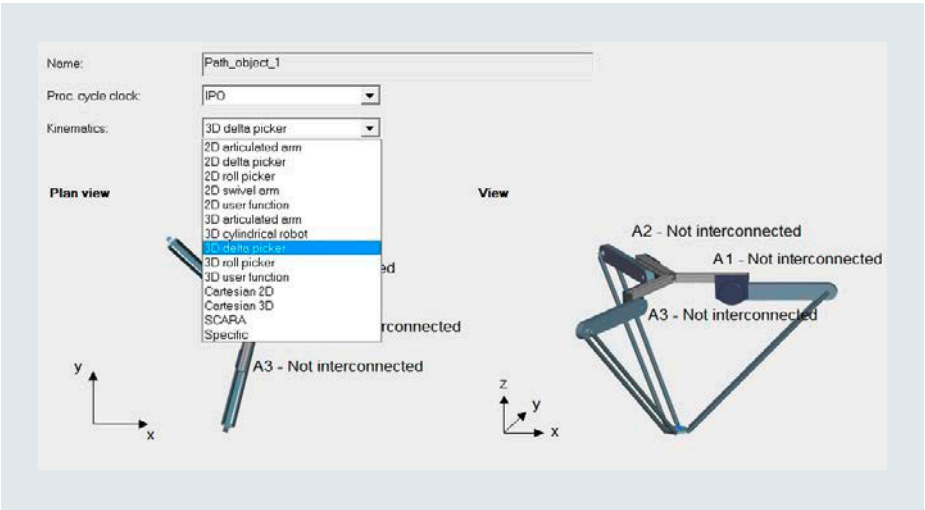


Figure 82 3D delta picker

8.9.9 Language editors in SCOUT

SCOUT provides you with editors for various programming languages (Figure 83). Every programming language has its strengths and should be used wherever it can be of most benefit.

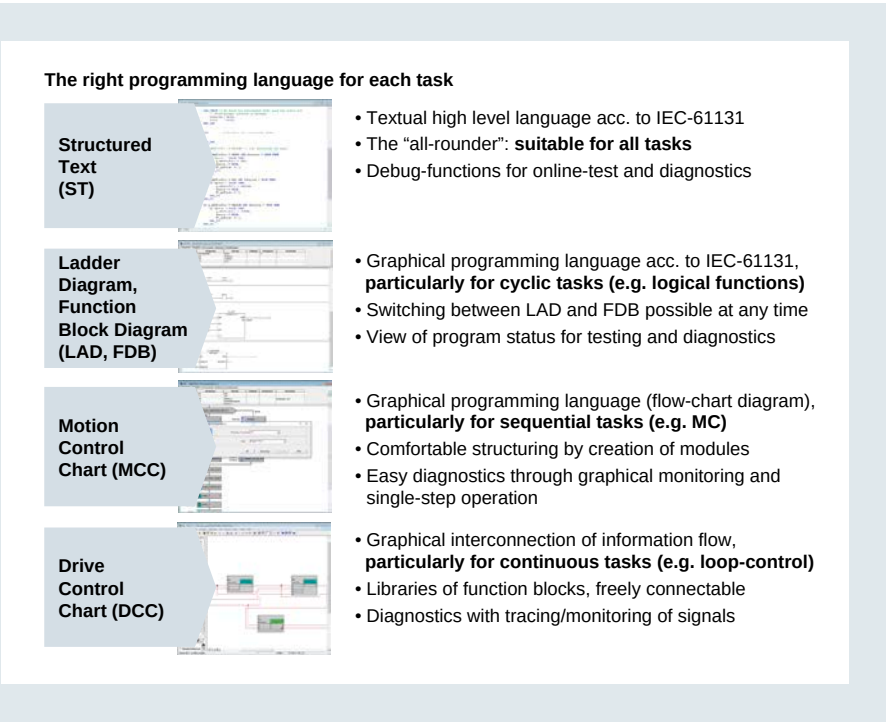


Figure 83 Programming languages in SCOUT

Structured Text (ST)

The high-level language Structured Text is very similar to PASCAL. ST is one of the languages that is supported and standardized by IEC 61131-3. The programmer can use this language to program all the tasks that need to be implemented in SIMOTION. The editor provides all the necessary programming functions such as indent, outdent, folding, automatic completion and formatting. Diagnostic options such as "program status" and "breakpoints" are available for testing programs.

Ladder diagram, function block diagram (LAD, FBD)

LAD and FBD are graphical programming languages that are ideal for programming combinational logic. They are thus the perfect languages for expressing logic combinations. It is extremely easy to diagnose programs using the "Program status"

function. But breakpoints can also be used in these languages for troubleshooting purposes.

Motion Control Chart (MCC)

The graphical programming language MCC is represented in a flowchart format and has been designed to help first-time users get to grips with programming motion functions (motion control – MC). Programs are executed according to the defined program flow. Predefined command boxes are inserted in the program at the points chosen by the programmer. A double-click on the command box opens a window in which further parameters can be assigned to the command. Users find it extremely easy to program motion commands, even if they are complex, with MCC. The „Monitor“ function is activated in order to track program execution. If monitoring is selected, the editor highlights the currently active command box in yellow. Other functions such as program status, single-step tracking or breakpoints allow the user to perform in-depth analysis on a program. MCC is optimized for creating motion programs for MotionTasks.

Drive Control Chart (DCC)

DCC has been designed for special applications and is *not* a standard feature of the engineering system. It is available as an option package for SCOUT and is ideal for programming control tasks. An “interconnection editor” can be additionally installed to set up graphical interconnections between signal flows. It is possible to analyze programs assisted by the status display and signal status tracking over time. The DCC option package comes with an extensive standard library that contains predefined functions such as logic operations, mathematical functions as well as more complex elements such as diameter calculator, PID controller or winder.

8.9.10 Support for programming languages

All the programming languages included in the basic system contain various test aids for analyzing programs. These include, for example, the status display in programs, the setting of breakpoints or the monitoring function in MCC. Automatic completion of commands or variables also help the programmer to work faster.

Cross-project software checks can be carried out using the convenient comparison function implemented in SCOUT (Figure 84). The programmer can use this tool to compare projects or to transfer program sections from existing projects to new projects.

The comparison shows a detailed summary of differences using a color coding system. Sections of program can be copied from the comparison project over to the reference project (currently open project) with a single click. This user-friendly comparison function is also available online from the SIMOTION CPU provided that the source data were also stored on the CPU.

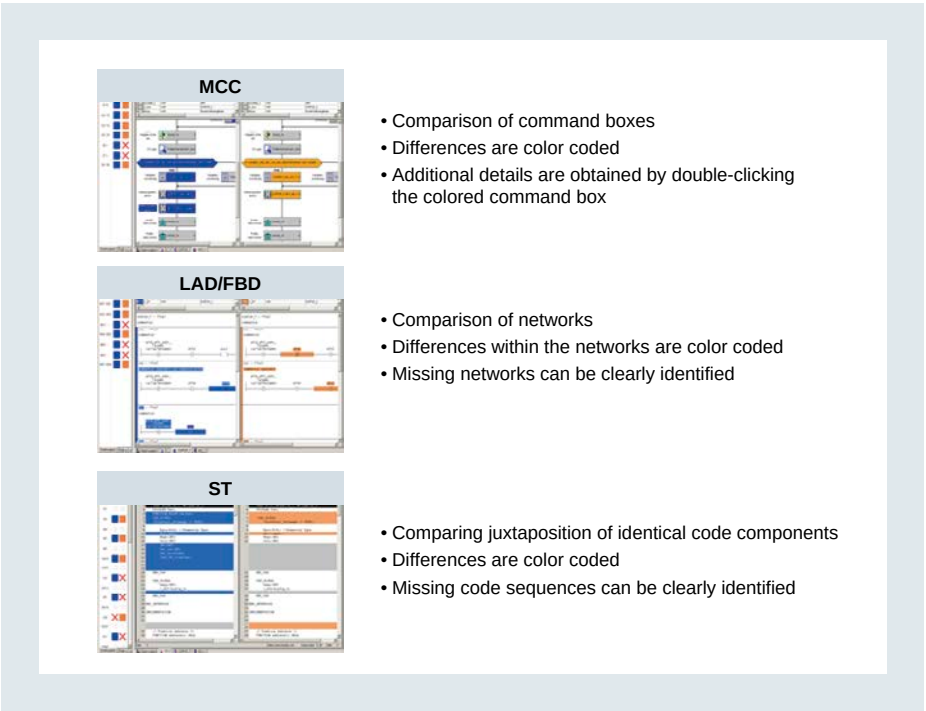


Figure 84 Comparison function in SCOUT

8.9.11 Inserting program sources (units)

Before you can input a program in SCOUT, you first need to insert a program source (unit) in the project navigator.

To do this, open the folder “Programs” in the PNV (Figure 85). You will find four insert dialogs underneath “Programs”. Each of these dialogs refers to a particular programming language and allows you to insert a source (unit) in the relevant language. In other words, each of the source containers in SIMOTION contains only one language. Programs written in a different language can, of course, be called from any program.

Insert ST source file

Select the entry “Insert ST source file” in order to insert a new source for the language ST (Structured Text).

When you double-click on the entry, a dialog box opens in which you can parameterize the ST source (Figure 86).

The dialog contains various tabs for setting the properties of the source. The overarching property is the name of the source. Enter a name of your choice in this box. We have used the name “ST_Examples” in our example.

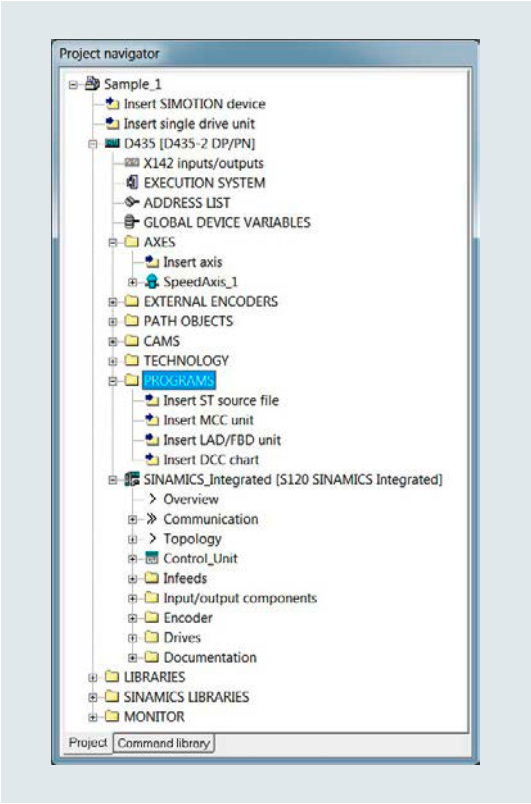


Figure 85 Inserting program units

General tab

You can enter information relating to the source such as author, version or comments in this tab. The engineering system will provide you with property details such as code size, "last modified on" time and date, storage location and the SCOUT version with which the source was created.

Compiler tab

The Compiler tab defines the options that are admissible in this source. The settings can be applied globally to all sources, or set separately to a value different from the global setting for each individual source. You must check the "Permit object-oriented programming" box in this screen. You must also check "Permit program status" so that you can monitor programs with the "Program status" function. Other details about the settings can be found in the online help or the documentation.

Additional settings tab

The active compiler options are stored as a string in the "Additional settings" tab. You can also implement preprocessor statements at the source. Preprocessor statements are a mechanism for defining conditional translations and text substitutions for specific variants of the environment. Further details can be found in the documentation.

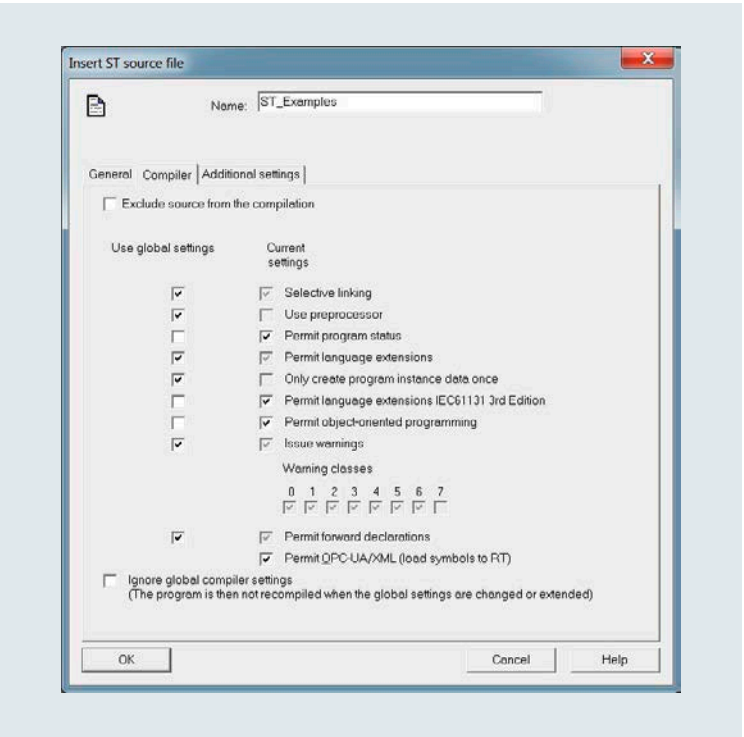


Figure 86 Inserting an ST source file (unit)

8.9.12 Entering programs

Once you have specified the properties of the program source and closed the dialog with "OK", the program editor will normally open automatically (Figure 87). You can now write the program in this editor.

As we have already mentioned several times, a source (unit) is a two-part container for programs (IMPLEMENTATION) and for providing public access to data (INTERFACE). An OOP interface ITF1 has been defined in the INTERFACE section in this source. This is available to all programs that are connected to this unit "ST_Examples". In the implementation section of the unit, we have programmed the class CL01 that in turn implements the interface ITF1.

If we want to use the class, we need to create an object (instance of the class). To do this, we create a program in which we define this instance (Figure 88).

We enter the PROGRAM App01 for class CL01. We define the instance of class CL01 as a variable in this program and also create Inp01 and Outp01 as BOOL variables. Then we call method mM1 in the program. We need to make this public so that the program can be assigned in the execution system. We declare the program as public in the interface section of the unit simply by entering "PROGRAM App01;" in the interface. The next chapter explains how the program is assigned to the execution system.

If the class is to be used again outside the unit, it can also be declared as public with the entry "CLASS CL01;" in the interface section.

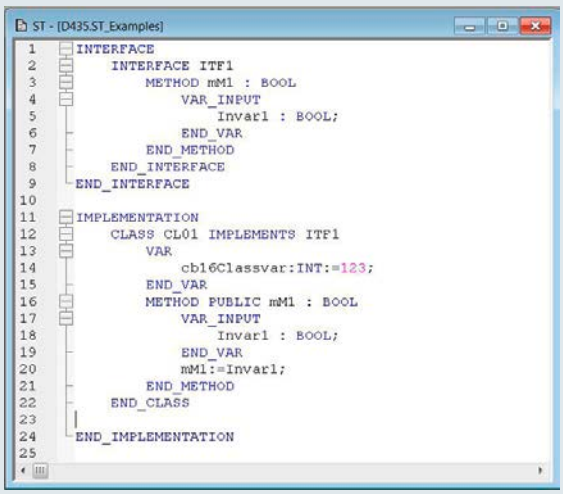


Figure 87 ST programming editor

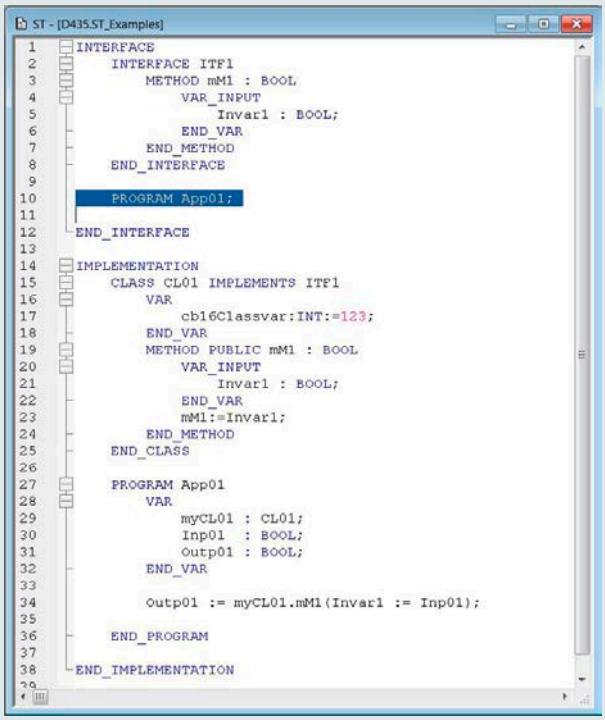


Figure 88 Program for execution system

8.9.13 Assigning programs to the execution system

Programs can be executed in the SIMOTION system only if they have been assigned to the execution system. The SIMOTION execution system has several execution levels. Some of these are defined by the system, while others can be influenced by user settings. A detailed description of the execution system can be found in the online help and the documentation.

You can go to the execution system screen by double-clicking on the entry “Execution system” in the project navigator (PNV). The window shown in Figure 89 appears. The different tasks are represented by symbols in this screen. The startup and shutdown tasks are in the center of the screen. The cyclic and parallel tasks are arranged to the right. All of the tasks can be selected in the tree view. Programs that are assigned to a task are also visible in the tree view.

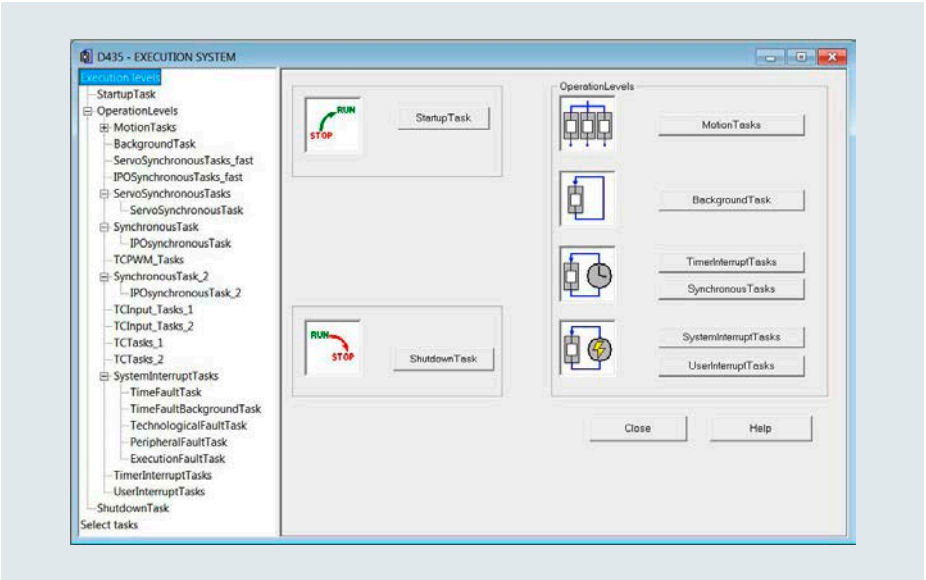


Figure 89 The execution system of SIMOTION

The Help button displayed in the dialog will take you directly to the relevant page in the online help. You will also find a detailed description of the SIMOTION task system there.

The “BackgroundTask” is a frequently used execution level. The BackgroundTask is executed cyclically by the system and is used to assign programs that always need to be processed, but are not time-critical. This task is ideally suited for testing the sample programs.

When you click on the button “BackgroundTask”, the setting screen for this task will open (Figure 90).

It is extremely easy to assign a program to a task. Programs that can be assigned are listed on the left-hand side with <source name>.<program name>, followed by

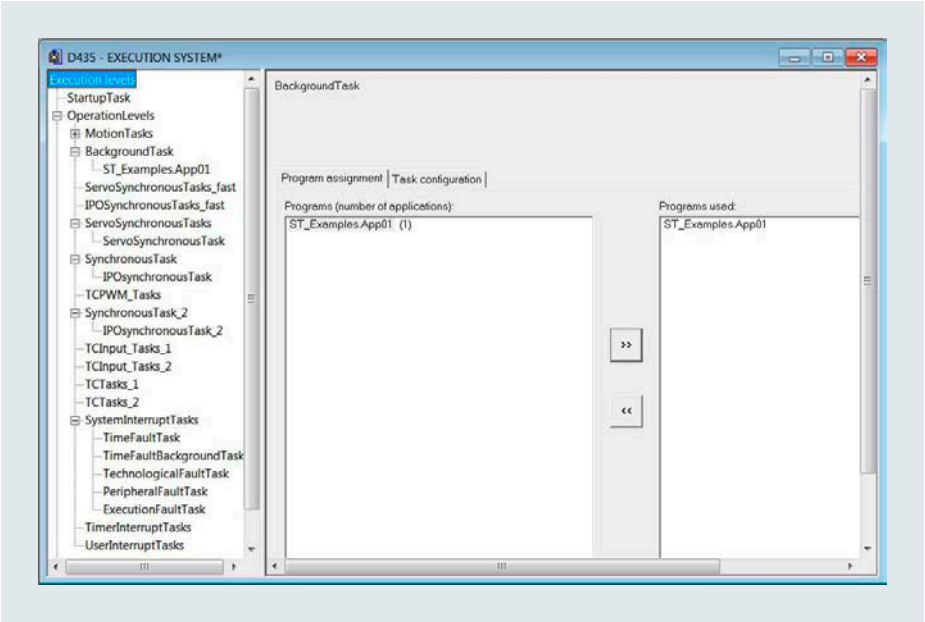


Figure 90 BackgroundTask: assigning programs

a number in parentheses. The number indicates the number of times the program has already been assigned to tasks. It is permissible to assign a program to more than one task. The programmer must however take task assignments into account.

The programmer can program function blocks, functions, programs and, as of V4.5, classes or interfaces. Only programs may be assigned to a task in the SIMOTION system. Programs are displayed in the left-hand list only if they have already been declared public in the unit. By selecting the program App01 and clicking on the button “arrows to the right”, you can assign App01 to the BackgroundTask.

8.9.14 Integrated test functions

At least one SIMOTION CPU is required to test programs. The test programs can be downloaded to this CPU. They are processed on this CPU.

A wide range of mechanisms for supporting tests have been integrated. These include, for example:

- Program status
  (requires activation via compiler switch)
- Variable status
  (in the symbol browser and in the source text)
- Breakpoints
  Extensive range of trace functions
- User-friendly watch tables

These functions are available only if an online connection is established between the engineering computer and the SIMOTION CPU. A detailed description of how to set up online connections can be found in the documentation or the online help. We would therefore advise our readers to study the documentation relating to communication or the online help.

8.9.15 Testing with “program status”

The most common means of testing programs is to use the “Program status” function. An online connection must be set up between the engineering computer and the SIMOTION CPU before this function can be used. The “Program status” function is essentially a mechanism that records the values of variables in real time as the program sequence selected by the user is executed. In other words, the values recorded indicate the exact processing sequence of the program in the CPU. These values are transferred to the engineering computer, processed there and displayed in SCOUT in coordination with the program sequence. “Program status” is suited primarily to testing programs that are called cyclically.

Figure 91 shows the status displays of the SCOUT engineering system. When “Program status” is active, the program editor window is split and the current status values of the variables are displayed in the right-hand window pane. The values

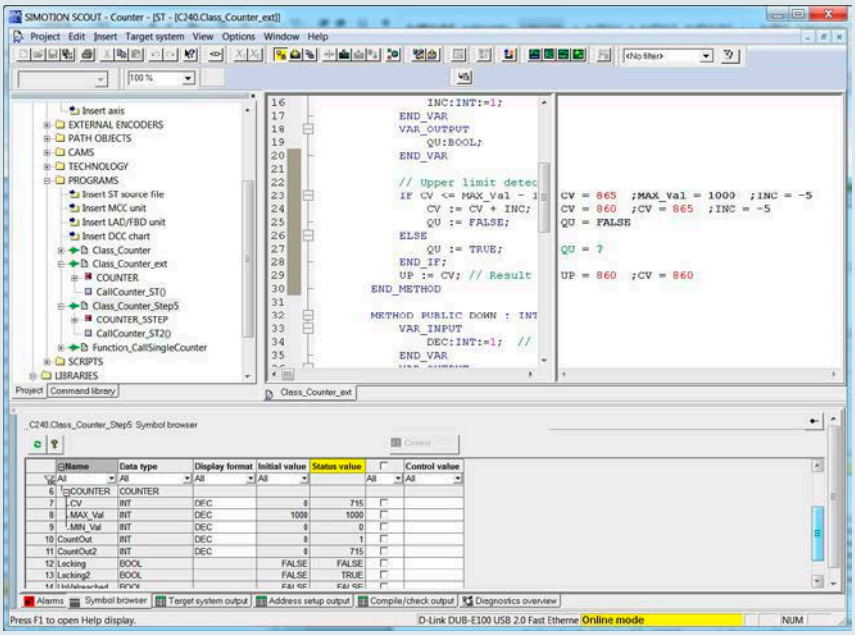


Figure 91 Status displays in SCOUT

are assigned to the relevant program line. The function also displays correct values stored temporarily in the CPU stack even if these no longer exist after a function or method has been executed. The values are recorded in real time as the selected program sequence is executed. The gray bar in the left-hand window shows the progress of the active recording.

Various information (compilation results, watch tables or variables, for example) is displayed at the bottom of the SCOUT screen. In this part of the screen, you can display the variables of the element selected in the PNV. To do this, you select the program source in the PNV and click on the tab “Symbol browser”. If the tab “Symbol browser” does not display any content after you have selected the program source, it is an indication that the selected source does not contain any global variables. You can select the element “Execution system” in the PNV as an alternative. In this case, the symbol browser displays the variables for all the programs that are integrated in the execution system. These data in the symbol browser are transferred acyclically from the CPU and displayed.

Setting up the “Program status” function

You must set up the “Program status” function before you can activate it. We want to explain this process using a specific test example. We will use the COUNTER classes from chapters 3.3.6 and 3.3.7 for this purpose.

A reminder: We developed there a class named COUNTER with methods UP() and DOWN(). We then extended these by deriving class COUNTER_5STEP from class COUNTER. These two classes were used in the program CallCounter_ST2 and we implemented count up and count down functions for each class. The core feature of these classes is that the only working program is the UP method of the base class.

We now want to monitor this method as it works and will do this using the “Program status” test mechanism. To utilize the “Program status” function, we first need to activate the compiler options “Enable Program status” at the program source (Figure 92). You can find this setting by selecting the source in the PNV, clicking on the right mouse button and selecting “Properties” in the context menu. The switches are displayed for selection/deselection in the “Compiler” tab.

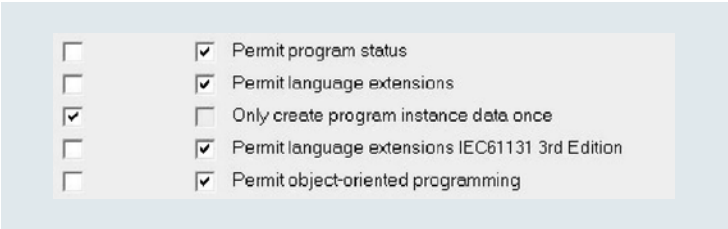


Figure 92 Enable Program status

This same procedure must naturally be carried out for every program source to be monitored. If the base classes and the derived class are stored in different program sources, the option must be set for both sources.

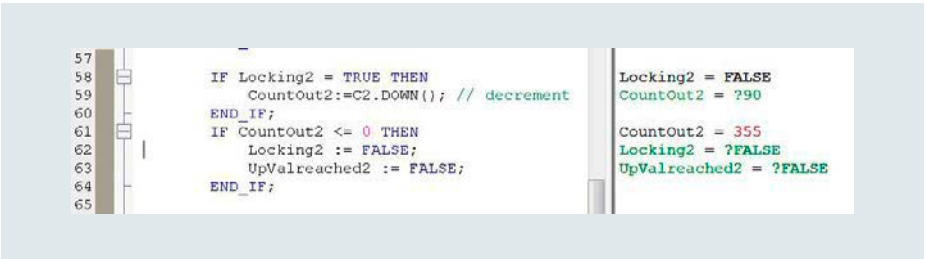


Figure 93 Program status display

We then open the program source which contains the program CallCounter_ST2. In the source we navigate to the point at which counting down in increments of 5 is programmed. In our example, the relevant program code is stored in line 59 (Figure 93). We position the cursor exactly on this line in the editor and then click.

The rest of the procedure is relatively simple. You can activate the "Program status" function using keyboard shortcut "CTRL+F7" or the button "Program status" in the toolbar  .

After you have activated the function, the window is split into panes and values are displayed automatically. We can now observe the effect of call C2.DOWN() in line 59. It should be noted, however, that we can only observe the effect of the methods here, but not the method itself.

Setting the call path/task selection

The same procedure as described above is also used to monitor methods. To do this, you navigate to the source in which the class Counter with the method UP() is programmed. After placing the cursor in the editor, you can activate "Program status" again with "CTRL+F7". When you do this, however, you will notice that the function is not activated, but the dialog "Call path/task selection" is displayed instead.

SCOUT always displays this dialog if the program sequence to be monitored can be called from several positions. You use this dialog to tell SCOUT which data to use for monitoring. The simplest option to select is "all call positions as of this call level".

If you do this, the system will record the data with all possible call positions. When you select this option, the system will supply a value display with various different data. Our method UP() of the class COUNTER is called by a total of four different positions, i.e. by the functions that count up and down in increments of one and those that count up and down in increments of five.

But we only want to observe how the method UP() counts down in increments of five. Before we do this, let's take another look at the entire functional interaction between the two classes. It is clearly illustrated in Figure 21.

The diagram shows that when the method DOWN() of the class COUNTER_5STEP is called, the inherited method DOWN() of the base class is called. This in turn uses THIS to call the method UP() of the class COUNTER_5STEP. UP() of the base class COUNTER is called with SUPER in this method.

We have presented this call chain again in Figure 94 which also shows the program line numbers.

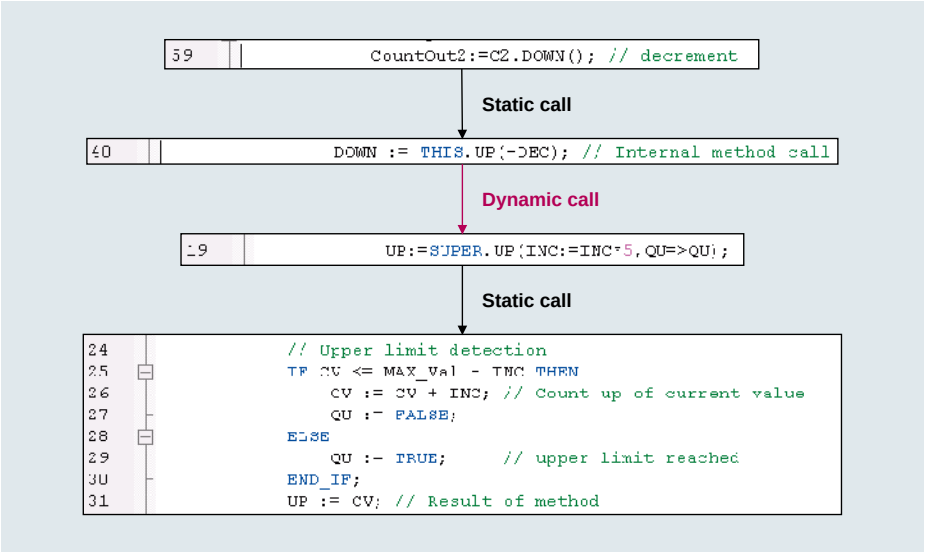


Figure 94 Method call chain

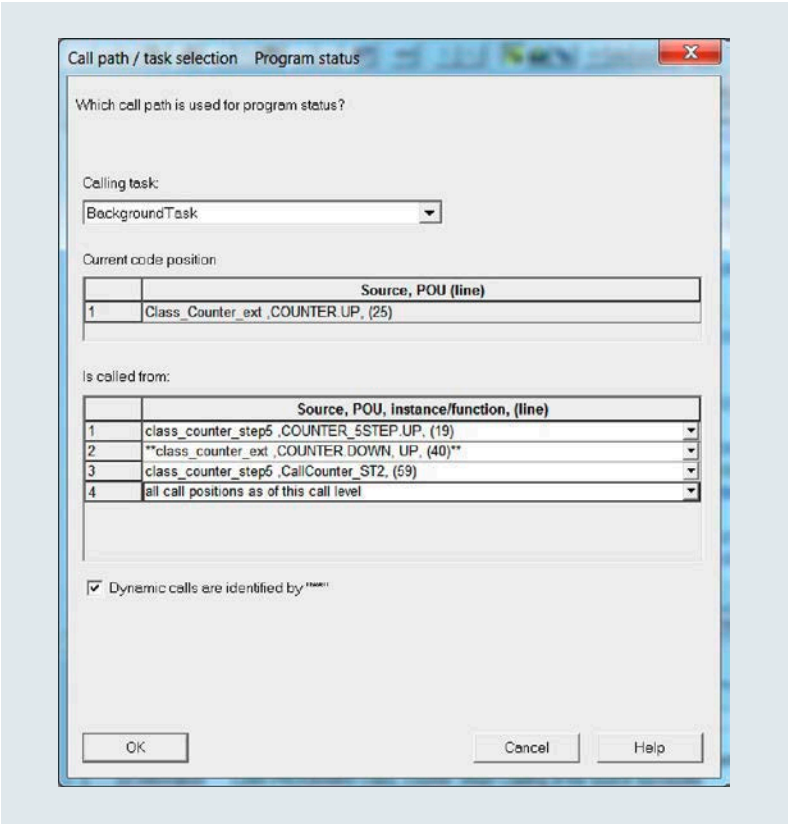


Figure 95 Setting the call path/task selection

You now need to use the dialog “Call path/task selection” to pass this information to the engineering system, but in the reverse sequence (Figure 95).

The dialog “Call path/task selection” offers you a variety of different setting options.

Calling task

In this box you must select the calling execution level for the program to be monitored. It is of course possible to call a program in different tasks.

Current code position

The point in the program to be monitored is displayed in this box. You have already selected this point by positioning the cursor in the editor.

Is called from

In this box you select the relevant call points as illustrated in Figure 95, starting at the bottom. All the points you can possibly select are displayed when you expand the list box. In this example, we will choose program line 19.

When you have finished this selection, the next list box is activated and again displays the possible call points. Since the next call is dynamic, you must also check the box “Dynamic calls ...”. The list box will otherwise only show the static calls that you can select. We will choose program line 40.

The last setting in our example is program line 59 that shows us the call of COUNTER_5STEP with the method DOWN().

When you click on the button “OK”, the dialog closes and you return to the status display. The status is then displayed as it is appears in the screenshot in chapter 8.9.15.

With a little practice, you will be able to enter these settings very quickly. When you have learned how to precisely specify the data you want to display, you will find that the “Program status” function is an extremely useful tool. Since “Program status” is not a suitable test mechanism in all situations, we will demonstrate its limits below.

The limits of “Program status”

As we have already mentioned, “Program status” is an ideal tool for testing programs that are executed cyclically. It is also designed to permit several status tasks to be processed simultaneously (depending on settings). “Program status” is thus a valuable tool for programmers.

When this test tool is used in a variety of scenarios, however, its underlying principle imposes a few limits on its capabilities. We want to demonstrate these here so that our readers will know when “Program status” can be used expediently as shown in Figure 96.

In order to use “Program status”, you need to set up the task. We have described how this is done in the previous chapter. Once the task has been set up, SCOUT requests the CPU to record the selected code range. The engineering system then waits until the code in the selected range has been executed once in the CPU. Once the data are available, SCOUT uploads these from the CPU and then processes and displays them. The cycle is then repeated. The “Program status” function is terminated if the online connection between the engineering computer and CPU is interrupted or deactivated.

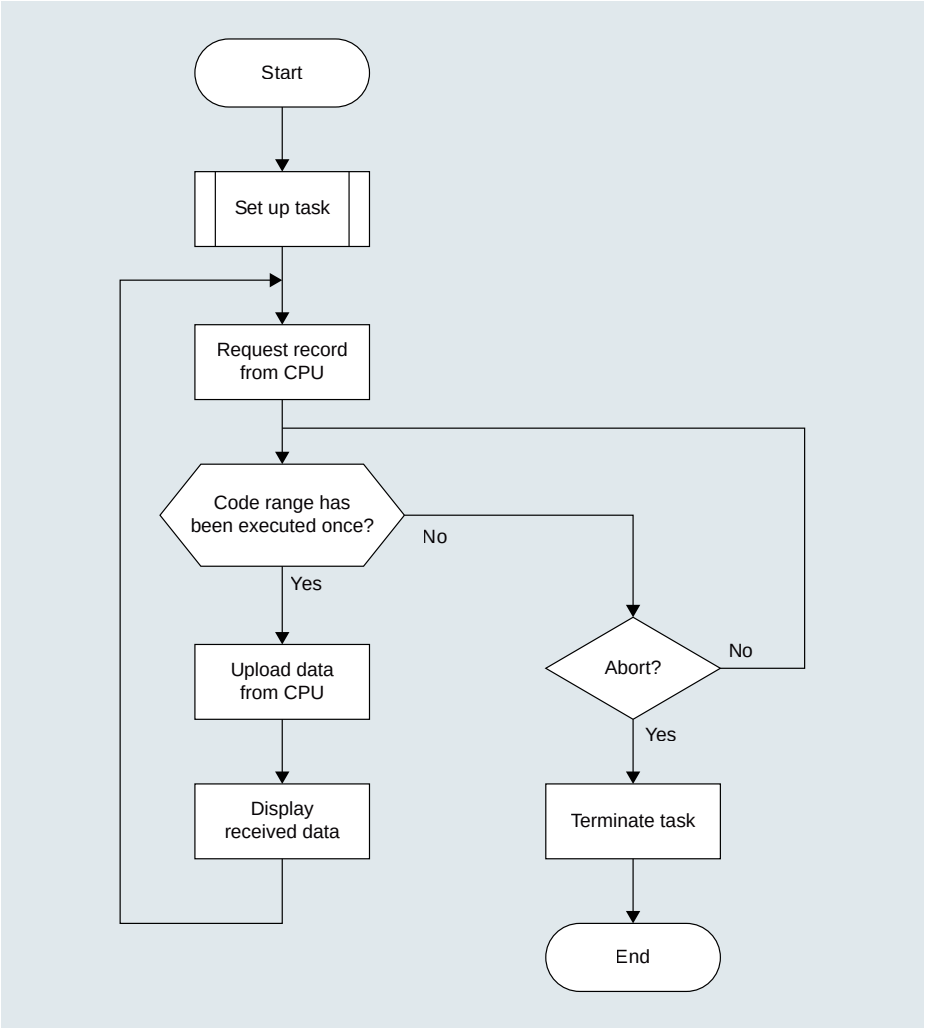


Figure 96 Operating principle of "Program status"

From the workflow illustrated above, it is clear that "Program status" requires the engineering system and the SIMOTION CPU to interact. The systems are naturally designed to do just this. But the way in which programs are programmed and used has an impact on the capabilities of this test tool. For this reason, we want to present specific scenarios and their consequences here. In the following cases, it is impossible to display status information even if "Program status" has been set up correctly.

- If you set up a task for "Program status", the engineering system waits until the selected code range is executed in the CPU. But if the selected program sequence never executes in the CPU, no data will be displayed in the right-hand window pane.

- This could happen if the CPU could either not execute, or exit from, the program sequence because it was preceded by a statement (such as IF).
- The selected program sequence is contained in a program, function block, function or method that is not currently called.
- The program sequence has been executed once or even multiple times, but the task was only set up after it was executed for the last time.
- The program is assigned to a task, but to a task that is not currently active (e.g. InterruptTask or MotionTask).
- The CPU has branched to STOP owing to an error.
- Using “Program status” in combination with breakpoints does not generally work. If one or more breakpoints are set in the range that also corresponds to the selected program sequence, the CPU will stop at the breakpoint. The status task thus waits for the complete program sequence to run, but this cannot complete execution because the CPU has stopped at the breakpoint.
- If the status task cannot be completed, the engineering system does not receive any data nor display any values.
- The data for the status might also have been delivered and displayed just prior to activation of the breakpoint. In this case, the displayed values do not correspond to the values that are actual when the CPU stops at the breakpoint.
- If breakpoints are located at a different point in the program that is not within the selected status display range, the approach to a breakpoint might prevent execution of the program sequence. No values are displayed in this case either.

It is clear from the description above that users of the “Program status” tool must have a certain level of knowledge about the system behavior and the program structure and sequences. Only programmers with this knowledge can fully exploit the potential of these valuable test functions. Further information about the behavior of the “Program status” tool can be found in the SCOUT online help. We would in particular like to draw your attention to the various operating modes of the SIMOTION system for testing. Selection of the appropriate operating mode influences the behavior and the information displayed by “Program status”.

Note about using the example programs

The example programs contained in this book have been developed to explain certain issues and they should be understood as such. The minimum engineering environment required for use of the examples is SIMOTION SCOUT V4.5.

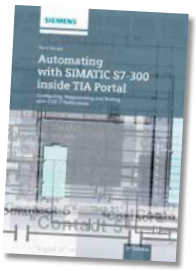
The reader is granted the non-exclusive, non-transferable right to use the software free of charge; this includes the right to modify the software, to reproduce it in modified or non-modified form and to link it to the reader's own software.

The software has not been tested with the normal system test by Siemens AG. We do not assume any liability – irrespective of the legal grounds – especially for faults in the software or associated documentation or damage on account of consulting unless the same is to be attributed to us, for example, for willful damage, gross negligence, damage to life, body or health, taking over of a procurement guarantee, willful concealment of defects or violation of important contractual liability. This does not entail a change in the burden of proof to the disadvantage of the customer.

German law applies. The courts Erlangen shall have exclusive jurisdiction.

Index

The page numbers specified in the index refer to places in the text in which the relevant term is defined or described for the first time, or to places in which it has special relevance in examples. The index would have become overcomplicated if we had listed all the pages on which the relevant terms appear.

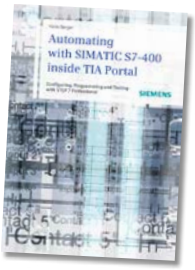


Hans Berger

Automating with SIMATIC S7-300 inside TIA Portal

Configuring, Programming and Testing with STEP 7 Professional

Second edition, 2014, 725 pages, € 79.90
ISBN 978-3-89578-443-9



Hans Berger

Automating with SIMATIC S7-400 inside TIA Portal

Configuring, Programming and Testing with STEP 7 Professional

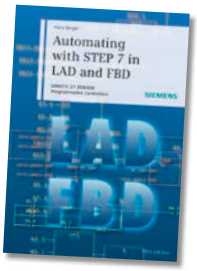
2013, 746 pages, € 69.90
ISBN 978-3-89578-383-8

Hans Berger

Automating with STEP 7 in STL and STL

SIMATIC S7-300/400 Programmable Controllers

Sixth edition, 2012, 553 pages, € 69.90
ISBN 978-3-89578-412-5



Hans Berger

Automating with STEP 7 in LAD and FBD

SIMATIC S7-300/400 Programmable Controllers

Fifth edition, 2012, 451 pages, € 69.90
ISBN 978-3-89578-410-1

Hans Berger

Automating with SIMATIC

Hardware and Software, Configuration and Programming, Data Communication, Operator Control and Monitoring

Sixth edition, 2016, 306 pages, € 44.90
ISBN 978-3-89578-459-0

Hans Berger

Automating with SIMATIC S7-1200

Configuring, Programming and Testing with STEP 7 Basic
Visualization with WinCC Basic

Second edition, 2013, 575 pages, € 49.90
ISBN 978-3-89578-385-2

Hans Berger

Automating with SIMATIC S7-1500

Configuring, Programming and Testing with STEP 7 Professional

Second Edition, Autumn 2017,
ca. 880 pages, ca. € 89.90
ISBN 978-3-89578-460-6

Nicolai Andler

Tools for Project Management, Workshops and Consulting

A Must-Have Compendium of Essential Tools and Techniques

Third edition, 2016, 488 pages, € 49.90
ISBN 978-3-89578-447-7