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.

Reproduction or transmission of this document or extracts thereof is not permitted unless expressly authorized.

The authors and publisher have taken great care with all texts and illustrations in this book. Nevertheless, errors can never be completely avoided. The publisher and authors accept no liability, regardless of legal basis. Designations used in this book may be trademarks whose use by third parties for their own purposes could violate the rights of the owners.

www.publicis-books.de

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

Publisher: Publicis Publishing, Erlangen, Germany
© 2017 by Publicis Pixelpark Erlangen – eine Zweigniederlassung der Publicis Pixelpark GmbH

This publication and all parts thereof are protected by copyright. Any use of it outside the strict provisions of the copyright law without the consent of the publisher is forbidden and will incur penalties. This applies particularly to reproduction, translation, microfilming or other processing, and to storage or processing in electronic systems. It also applies to the use of individual illustrations or extracts from the text.

Printed in Germany

Table of contents

Information for readers 13

1 Developments in the Field of Control Engineering 18
1.1 The early days of programmable logic controllers (PLCs) 19
1.2 The PLC learns to communicate 22
1.3 Development of fieldbus systems 24
1.4 Integration of display systems in PLCs 25
1.5 Integration of motion control in PLCs 27
1.6 Drives become fully-fledged bus system nodes 30
1.7 PLC and PAC – what is the difference? 31
1.8 General conclusions about past developments 31

2 Basic Principles of Object-Oriented Programming 33
2.1 The basis of object-oriented programming 33
2.1.1 History 33
2.1.2 What's different? 34
2.1.3 What does object orientation mean? 35
2.1.4 Objects and their interactions 36
2.2 General principles of OOP 37
2.2.1 Objects 37
2.2.2 Classes 39
2.2.3 Inheritance 39
2.2.4 Overriding 41
2.2.5 Interfaces for object interaction 42
2.2.6 Summary 44
2.2.7 Advantages of using OOP 45
2.2.8 Disadvantages of OOP 45
2.3 Tips about defining classes 46

3 Object-Oriented Programming 49
3.1 Implementation of OOP with SIMOTION 49
3.2 Function blocks with methods 50
3.2.1 Modularization without OOP extensions 51
3.2.2 Program and data are separate 53
3.2.3 Advances in the life cycle of software 55
3.2.4 Disadvantages of programming without OOP extensions 56
3.2.5 Extensions to FBs and their access specification 57
3.2.6 Use of methods to improve program structuring 59
3.2.6.1 Example of FB with methods 60

3.2.6.2 Example of a function block call ... 61
3.2.7 Function block with methods for placing commands ... 62
3.2.7.1 Example of the FB with command methods ... 63
3.2.7.2 Example of an FB call with command methods ... 65

3.3 Classes (CLASS) ... 66
3.3.1 Keywords supported for a class ... 67
3.3.1.1 Example of a CLASS declaration ... 69
3.3.2 Methods (METHOD) ... 69
3.3.3 Methods and their access specification ... 70
3.3.4 Declaration of instances of a class ... 71
3.3.5 Rules for identifiers in a class ... 72
3.3.6 Use of class methods ... 72
3.3.6.1 Example of a CLASS COUNTER ... 73
3.3.6.2 Use of the method of CLASS COUNTER ... 74
3.3.6.3 Extension of the CLASS COUNTER and use of THIS ... 75
3.3.6.4 Use of the methods UP and DOWN ... 76
3.3.7 Classes and inheritance ... 76
3.3.7.1 Example of derivation of a class ... 78
3.3.7.2 Example of how to use base and derived classes ... 79
3.3.7.3 Other aspects of the method call ... 80
3.3.7.4 Example of base and derived classes in a function ... 81
3.3.8 Abstract classes ... 82

3.4 Examples of valve applications with OOP ... 84
3.4.1 Example with 4/3-way valve ... 84
3.4.1.1 Example of a class for 4/3-way valves ... 85
3.4.1.2 Example of a valve call ... 87
3.4.1.3 Example with 4/3-way valve with fast/slow speed ... 88
3.4.1.4 Example of a derived class ValveControl43FS ... 89
3.4.1.5 Example of calls of base class and extended class ... 90
3.4.1.6 Example of call of extended class with basic function ... 91

3.5 Interfaces ... 92
3.5.1 Supported features ... 93
3.5.2 Principles of interfaces ... 94
3.5.2.1 Example of an interface declaration ... 95
3.5.3 Representation of interfaces in the PNV of SCOUT ... 97
3.5.4 Benefits of interfaces ... 99
3.5.5 Interfaces as a reference to classes ... 100
3.5.6 Valve classes with interfaces ... 103
3.5.7 Declaration of the valve interface ... 105
3.5.7.1 Example of ValveControl43 with limit switch monitoring ... 105
3.5.7.2 Example of ValveControl43 with error reporting ... 108
3.5.7.3 Example of ValveControl43 with test error reporting ... 112
3.5.7.4 Example of class HMIReporting ... 113
3.5.7.5 Example of ValveControl43 with error reporting ... 115
3.5.8 Interface for neutralizing I/O components ... 116
3.5.8.1 Connection of cameras to the control system ... 116
3.5.8.2 Interface definition for a camera connection ... 122

3.5.9 Interface for neutral I/O connection (condensed example) 123
3.5.9.1 Interface definition for neutral I/O connection 125
3.5.9.2 Implementation in classes 125
3.5.9.3 Interface definition and mapping table program 126
3.5.9.4 Program for implementation and use of classes 127
3.5.9.5 Interface for fast/slow speed switchover 129
3.5.9.6 Implementation of classes for fast/slow speed 130

3.6 Further optimization of the valve class 131
3.6.1 Existing implementation of ValveControl 131
3.6.2 Design of a state machine 132
3.6.2.1 Example of ValveControl43ST – state machine using CASE 134
3.6.2.2 Example of ValveControl43ST – state machine with classes 140

3.7 Abstract class for different drives 143
3.7.1 Functional differences between various drive solutions 144
3.7.2 Class model for connecting different drives 146
3.7.2.1 Example of abstract class “CDrive” 147
3.7.2.2 Example of class for direct-on-line starting drives 148
3.7.2.3 Example of class for drives with star-delta starters 149
3.7.2.4 Example of class for speed-controlled drives 151
3.7.2.5 Example program for controlling drives of different types 155

3.8 Abstract class versus interface 157

3.9 OOP opens up the world of design patterns 159

4 OOP Supports Modular Software Concepts 161

4.1 Assembling projects for real machines 162
4.1.1 Module design 163
4.1.2 The role of the software developer 163
4.1.3 Modularizing software 164
4.1.3.1 Creating equipment modules 166
4.1.3.2 Software design of the equipment module 167
4.1.3.3 Example of the class “CEMPusher” 169
4.1.3.4 Example of an equipment module call 174
4.1.4 Preparations for multiple reuse 175
4.1.4.1 Example of the neutralized equipment module 176

4.2 SIMOTION easyProject project generator 177
4.2.1 Adding your own modules to the project generator 181
4.2.2 Creating a user interface for the project generator 182
4.2.3 XML description of the equipment module 184

5 Guide to Designing and Developing Software 188

5.1 Establishing requirements 188
5.1.1 Starting point – user interfaces 189
5.1.2 Starting point – process operations 189
5.1.3 Starting point – mechanical engineering elements 190
5.1.4 Existing solutions 191

5.2 Object-oriented design ... 192
5.2.1 Encapsulation ... 192
5.2.2 Responsibility of a class ... 193
5.2.3 Commonalities and differences between objects ... 194
5.2.4 Principle of replaceability with derived classes ... 194
5.2.5 Determining relationships ... 195
5.2.6 SOLID principles ... 197

5.3 Reusable and easy-to-maintain software ... 197
5.3.1 How can software be made reusable? ... 197
5.3.2 Libraries are helpful ... 198
5.3.3 What is the best way to develop modules? ... 198

5.4 Organizational and legal aspects ... 201
5.4.1 Transition to OOP must be planned ... 201
5.4.2 Software needs to be planned ... 202
5.4.2.1 Analysis of existing programs ... 202
5.4.2.2 Reuse of software ... 203
5.4.3 Reuse and ownership of software ... 205
5.4.3.1 Distribution of software ... 206
5.4.3.2 Acquisition of software ... 207
5.4.4 “Good software” and object-oriented design ... 208

5.5 Software tests are a must! ... 211
5.5.1 Module test ... 213
5.5.2 Integration test ... 214
5.5.3 System test ... 214
5.5.4 Acceptance test ... 216

6 Additional Topics Relating to Software Structuring ... 217

6.1 I/O references ... 217
6.1.1 Declaration ... 218
6.1.2 Linking references to I/O variables ... 218

6.2 Namespaces ... 220

6.3 General references ... 222
6.3.1 Declaration and initialization ... 223
6.3.2 Working with references ... 224

7 Description of the Extended Functionality in SIMOTION ... 228

7.1 General extensions to the programming model ... 228

7.2 Classes in SIMOTION ... 229
7.2.1 Constants and user-defined data types in classes ... 229
7.2.2 Naming of variables in classes and methods ... 230
7.2.3 Method calls ... 231
7.2.4 FINAL for methods and classes ... 232
7.2.5 Declaration of abstract classes and methods ... 232
7.2.6 Interface implementation and class derivations ... 233
7.2.7 Type conversions for classes and interfaces ... 234

7.3 Instantiation of classes and function blocks ... 236
7.3.1 User-defined initialization of instances ... 236
7.3.2 Initialization of interface variables ... 237
7.3.3 Creating class and function block instances ... 238
7.3.4 RETAIN data in classes and function blocks ... 239
7.3.5 Arrays of variable length ... 239

7.4 Tips for creating compatible and efficient software ... 240
7.4.1 Methods and function calls ... 240
7.4.2 Use of enum values and constants ... 240
7.4.3 Use of predefined namespaces ... 241
7.4.4 Declaration of data types, variables and methods ... 242
7.4.5 Preparing structured data for transmission ... 243

8 Introduction to SIMOTION ... 246
8.1 Classic development of control systems ... 246
8.2 New control concepts required ... 247
8.3 Technology Objects in SIMOTION ... 248
8.4 Three hardware platforms ... 249
8.5 Connecting drives and I/O devices to SIMOTION ... 251
8.6 Handling kinematics in SIMOTION ... 251
8.7 SIMOTION's programming model ... 252
8.7.1 The units of SIMOTION ... 253
8.7.2 The variable model in SIMOTION ... 254
8.7.3 Libraries in SIMOTION ... 258

8.8 The SIMOTION SCOUT engineering system ... 259
8.9 Components of SCOUT ... 260
8.9.1 The SCOUT project navigator ... 261
8.9.2 Creating a new project ... 262
8.9.3 Creating a new device ... 263
8.9.4 Hardware configuration ... 266
8.9.5 The SIMOTION address list ... 268
8.9.6 Creating axes ... 269
8.9.7 Creating drives ... 274
8.9.8 Creating path objects ... 276
8.9.9 Language editors in SCOUT ... 278
8.9.10 Support for programming languages ... 279
8.9.11 Inserting program sources (units) ... 280
8.9.12 Entering programs ... 282
8.9.13 Assigning programs to the execution system ... 284
8.9.14 Integrated test functions ... 285
8.9.15 Testing with "program status" ... 286

Note about using the example programs ... 293
Index ... 294

List of Figures

Figure 1 Example of a ladder logic program ... 20
Figure 2 Example of SIEMENS STL ... 20
Figure 3 SIMATIC S5-150K ... 22
Figure 4 WF470 with compact operator panel ... 26
Figure 5 S5-150K with WF625 and WS600G ... 28
Figure 6 Communication between objects – object-oriented ... 34
Figure 7 Communication between objects – procedural ... 35
Figure 8 Hydraulic aggregate ... 38
Figure 9 Class and object ... 39
Figure 10 Inheritance principle with classes ... 40
Figure 11 Hydraulic aggregate with HMI display ... 43
Figure 12 Valve-cylinder combination ... 51
Figure 13 FB_Valve ... 52
Figure 14 Program and data are separate in function blocks ... 55
Figure 15 Function blocks need to be copied and adapted ... 56
Figure 16 Programming FB Valve43 with methods ... 59
Figure 17 Further development FB Valve43 extended ... 63
Figure 18 CLASS in the PNV ... 66
Figure 19 Access definition for methods (source: IEC 61131-3 ED3) ... 71
Figure 20 Classes and their derivations ... 77
Figure 21 Derivation and counter call principle ... 78
Figure 22 Plant with a 4/3-way valve ... 85
Figure 23 4/3-way valve with fast/slow speed ... 89
Figure 24 Interfaces (source: IEC 61131-3 ED3) ... 95
Figure 25 Interface representation in PNV ... 98
Figure 26 Interfaces in classes ... 99
Figure 27 Overview of valve and HMI development ... 104
Figure 28 Interface for error reporting ... 105
Figure 29 Delta picker with two belts ... 118
Figure 30 Conveyor belt with parts ... 119
Figure 31 Product register of SIMOTION handling ... 120
Figure 32 Proposal for a standard telegram for cameras ... 121
Figure 33 Interface for camera ... 122
Figure 34 Principle of signal transfer in layers ... 124
Figure 35 Neutral interface ... 125
Figure 36 Valve with neutral I/O connection ... 126
Figure 37 Valve with signal interconnection ... 132
Figure 38 Valve state machine ... 133
Figure 39 Different drive types in one plant ... 144

Figure 40 Class model CDrive ... 146
Figure 41 SIMOTION Technology Objects ... 152
Figure 42 Hierarchy as defined by ISA-88-01 ... 165
Figure 43 Equipment module for conveyor belt with ejector ... 167
Figure 44 Software design of the equipment module ... 168
Figure 45 States of the equipment module ... 169
Figure 46 Functions of the easyProject project generator ... 178
Figure 47 “easyProject” project generator ... 178
Figure 48 User interface of the project generator ... 179
Figure 49 Equipment modules of the project generator ... 180
Figure 50 Generating a project ... 180
Figure 51 Structure of the project generator data ... 181
Figure 52 Equipment module PusherX ... 182
Figure 53 User interface of the equipment module ... 183
Figure 54 Representation of classes and objects in UML ... 196
Figure 55 Interaction between PLC, technology modules and motion control ... 247
Figure 56 Integration of PLC, motion and technology ... 248
Figure 57 Technology Objects in SIMOTION ... 249
Figure 58 The 3 hardware platforms of SIMOTION ... 250
Figure 59 SIMOTION with drives and I/O devices ... 251
Figure 60 Kinematics supported by SIMOTION ... 252
Figure 61 Programs and data are organized in units ... 254
Figure 62 Variable model of SIMOTION ... 255
Figure 63 Libraries in the SIMOTION project ... 258
Figure 64 SCOUT engineering system ... 259
Figure 65 SCOUT workbench ... 260
Figure 66 Project navigator ... 261
Figure 67 Result of creating a new project ... 263
Figure 68 Inserting a device ... 264
Figure 69 Properties – Ethernet interface PNxIO ... 265
Figure 70 Setting up PG/PC communication ... 265
Figure 71 Insert SIMOTION device with “Open HW Config” ... 267
Figure 72 HW Config with the SIMOTION device ... 267
Figure 73 Address list ... 268
Figure 74 SCOUT with inserted D435-2 device ... 269
Figure 75 Creating a drive axis ... 270
Figure 76 Axis configuration – axis type ... 272
Figure 77 Axis configuration – summary ... 273
Figure 78 Assigning a drive to the axis ... 274
Figure 79 Axis wizard for assigning a drive ... 275
Figure 80 Inserting a path object ... 276
Figure 81 Path object in the PNV ... 277
Figure 82 3D delta picker ... 277
Figure 83 Programming languages in SCOUT ... 278

Figure 84 Comparison function in SCOUT 280
Figure 85 Inserting program units 281
Figure 86 Inserting an ST source file (unit) 282
Figure 87 ST programming editor 283
Figure 88 Program for execution system 283
Figure 89 The execution system of SIMOTION 284
Figure 90 BackgroundTask: assigning programs 285
Figure 91 Status displays in SCOUT 286
Figure 92 Enable Program status 287
Figure 93 Program status display 288
Figure 94 Method call chain 289
Figure 95 Setting the call path/task selection 289
Figure 96 Operating principle of “Program status” 291

List of Tables

Table 1 Keywords for classes 67
Table 2 Declaration of instances of a class 71
Table 3 Keywords for interfaces 93
Table 4 Comparison between abstract class and interface 158
Table 5 Predefined namespaces (scopes) 241

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.

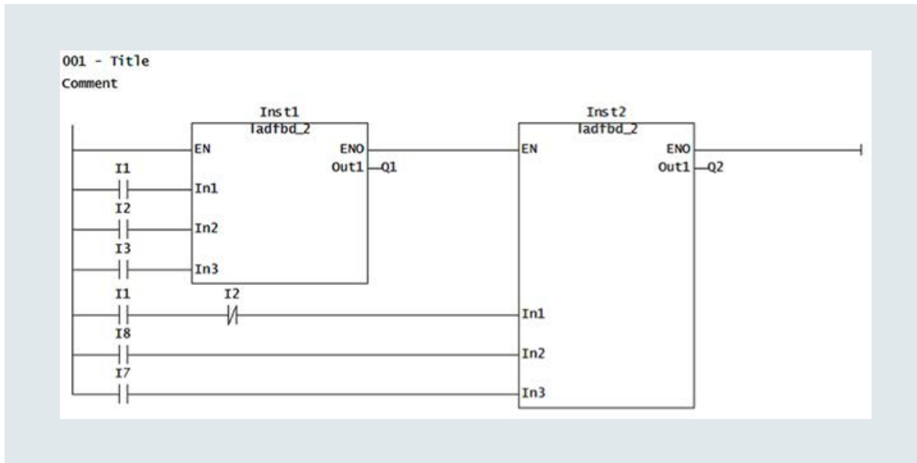

Figure 1 Example of a ladder logic program

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.

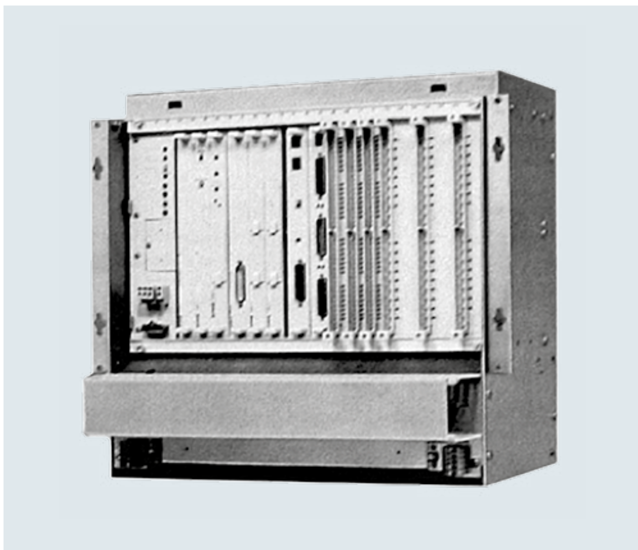

Figure 3 SIMATIC S5-150K

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.

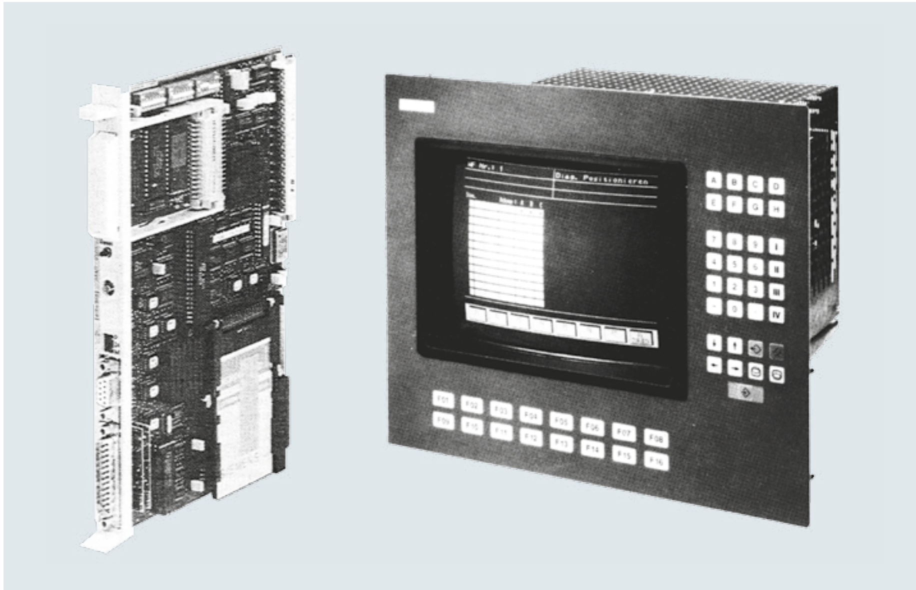

Figure 4 WF470 with compact operator panel

- 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.

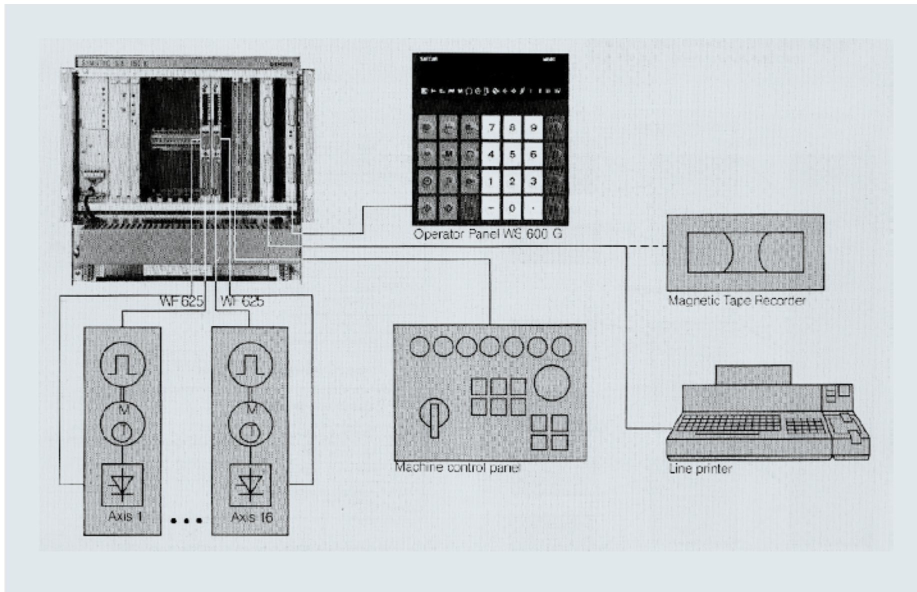

Figure 5 S5-150K with WF625 and WS600G

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.