

# **CODESYS Control V3 Manual**

**Document Version 19.0**

**CONTENT**

<b>1</b>	<b>INTRODUCTION</b>	<b>11</b>
1.1	Runtime System in the Overall System	11
1.2	Mission of the Runtime System	12
<b>2</b>	<b>ARCHITECTURE</b>	<b>13</b>
2.1	Overview	13
2.2	Technology: ANSI-C and C++	14
2.3	Single Source	15
2.3.1	Compact runtime	15
2.3.1.1	Compact download format	16
2.4	Error Codes	18
2.5	Object Handles	18
2.6	Linkage	18
2.7	Component Interface Architecture	18
2.7.1	Calling convention	19
2.7.2	Export convention	19
2.7.3	Import convention	19
2.7.4	Check functions	20
2.7.5	Summary	20
2.8	M4 Mechanism	20
2.8.1	Interface description file	21
2.8.2	Dependency description file	25
2.9	Source Tree of the Runtime System	26
2.10	Files on the Target System	27
2.10.1	Boot projects	27
2.10.2	I/O manager	27
2.10.3	User management	27
2.10.4	Visualization	28
<b>3</b>	<b>OVERVIEW OF THE KERNEL COMPONENTS AND MAIN FUNCTIONS</b>	<b>29</b>
3.1	Start up and Shutdown	29
3.1.1	Startup	29
3.1.2	Operating mode	30
3.1.3	Shutdown	30
3.2	Component Manager	31
3.3	Application Handling	31
3.3.1	Overview	32
3.3.2	Application management	33
3.3.3	State behaviour (Start/Stop/Error)	33

3.3.4	Boot project	33
3.3.4.1	Create boot project at download implicitly	33
3.3.4.2	Create boot project after online change implicitly	33
3.3.4.3	Create boot project on demand	33
3.3.4.4	Create boot project offline	33
3.3.5	Retain variables	34
3.3.6	Debugging	34
3.3.7	Download and online change	35
3.3.7.1	Download	35
3.3.7.2	Online Change	35
3.3.8	Events related to application handling	35
3.3.9	System variables for controlling critical runtime services	36
3.3.10	Accessing project and application information	36
3.3.10.1	Project information	37
3.3.10.2	Application information	37
<b>3.4</b>	<b>Watchdog Handling</b>	<b>37</b>
3.4.1	Monitoring	38
<b>3.5</b>	<b>IEC Task Management</b>	<b>38</b>
3.5.1	Data Format of the task description	39
3.5.2	Creating IEC tasks	39
3.5.3	Creating an external event task	39
<b>3.6</b>	<b>Scheduling</b>	<b>40</b>
3.6.1	Single tasking	40
3.6.2	Timer scheduler	40
3.6.3	Multitasking	41
<b>3.7</b>	<b>Task management</b>	<b>42</b>
<b>3.8</b>	<b>Configuration (Settings)</b>	<b>42</b>
3.8.1	INI file backend	42
3.8.2	Embedded Backend	43
<b>3.9</b>	<b>Logging</b>	<b>43</b>
<b>3.10</b>	<b>Hardware and Operating System Abstraction Layer (Sys-Components)</b>	<b>44</b>
3.10.1	Time access (SysTime)	44
3.10.2	Serial interface (SysCom)	44
3.10.3	Exception handling (SysExcept)	44
3.10.4	File access (SysFile)	45
3.10.5	File access using flash (SysFileFlash)	45
3.10.6	Flash access (SysFlash)	45
3.10.7	Directory handling (SysDir)	45
3.10.8	Memory access	46
3.10.8.1	Heap and static memory (SysMem)	46

3.10.8.2	Physical memory access and shared memories (SysShm)	46
3.10.9	Dynamic loading module (SysModule)	46
3.10.10	Ethernet sockets (SysSocket)	46
3.10.11	Debug console outputs (SysOut)	46
3.10.12	Message queues (SysMsgQ)	46
3.10.13	Interrupt handling	46
3.10.14	PCI bus access (SysPCI)	46
3.10.15	Device port access (SysPort)	46
3.10.16	Timer handling (SysTimer)	47
3.10.17	Target information (SysTarget)	47
3.10.18	Task handling	47
3.10.18.1	Synchronization and semaphores (SysSem)	47
3.10.18.2	Operating system events (SysEvent)	47
3.10.18.3	Task handling (SysTask)	47
3.10.19	Optional system components for target visualization	47
3.10.19.1	Window handling (SysWindow)	47
3.10.19.2	Basic graphic routines (SysGraphic)	47
3.10.20	Process handling	47
3.10.20.1	Processes (SysProcess)	48
3.10.20.2	Process synchronization (SysSemProcess)	48
3.10.21	Direct Ethernet controller access (SysEthernet)	48
<b>3.11</b>	<b>Memory Management</b>	<b>48</b>
<b>3.12</b>	<b>Events</b>	<b>48</b>
<b>3.13</b>	<b>Exception Handling</b>	<b>49</b>
3.13.1	Structured exception handling (rts_try / rts_catch)	50
<b>3.14</b>	<b>License Check</b>	<b>50</b>
<b>3.15</b>	<b>Online User Management</b>	<b>53</b>
<b>4</b>	<b>PORTINGS</b>	<b>54</b>
<b>4.1</b>	<b>Windows Specific Information</b>	<b>54</b>
4.1.1	Windows runtime services	55
4.1.1.1	CODESYS Control Win V3 (soft real time)	55
4.1.1.2	CODESYS Gateway Service V3	55
4.1.1.3	CODESYS Service Control V3	55
4.1.1.4	Brand labeling	55
4.1.2	CODESYS Control RTE V3 (hard realtime)	56
4.1.3	CODESYS integrated runtime systems	56
4.1.3.1	CODESYS simulation	56
4.1.3.2	CODESYS HMI	56

<b>4.2</b>	<b>Windows CE Specific Information</b>	<b>56</b>
<b>4.3</b>	<b>VxWorks Specific Information</b>	<b>57</b>
4.3.1	Distributed clocks	58
4.3.1.1	Timer sources	58
4.3.1.2	Performance & Accuracy	58
4.3.1.3	Jitter	59
4.3.1.4	Static Memory Areas	59
4.3.2	Global object pools	59
<b>4.4</b>	<b>Linux specific information</b>	<b>60</b>
<b>5</b>	<b>COMMUNICATION</b>	<b>61</b>
<b>5.1</b>	<b>Overview</b>	<b>61</b>
5.1.1	Usage scenarios	62
<b>5.2</b>	<b>General</b>	<b>63</b>
<b>5.3</b>	<b>Communication Layers</b>	<b>63</b>
5.3.1	Block driver (Layer 2)	63
5.3.2	Router (Layer 3)	64
5.3.3	Channel management (Layer 4)	64
5.3.4	Application services (Layer 7)	65
<b>5.4</b>	<b>Network Topology and Addressing</b>	<b>66</b>
5.4.1	Topology	66
5.4.2	Addressing and routing	66
5.4.2.1	Parallel routing	67
5.4.3	Address determination	67
5.4.4	Address structure	67
5.4.4.1	Network addresses	67
5.4.4.2	Node addresses	68
5.4.4.3	Absolute and relative addresses	69
5.4.4.4	Broadcast addresses	70
<b>5.5</b>	<b>Router Communication</b>	<b>70</b>
5.5.1	Hop count	71
5.5.2	Router signaling	71
5.5.3	Variable maximum block length for a transmission route	71
5.5.4	Multiple router instances	72
<b>5.6</b>	<b>Layer 3 Services</b>	<b>72</b>
<b>5.7</b>	<b>Gateway and client</b>	<b>72</b>
<b>5.8</b>	<b>Implementation Aids</b>	<b>73</b>
5.8.1	Implementation of own block driver	73
5.8.2	Interface	73
5.8.3	Addressing	74
5.8.4	General implementation procedure	75

5.8.5	Synchronisation	75
<b>5.9</b>	<b>Implementation of Own Communication Driver</b>	<b>75</b>
5.9.1	Communication driver for the gateway	76
5.9.2	Communication driver for the client	77
5.9.2.1	Connection parameters	79
5.9.2.2	Implementation of BeginConnect	81
<b>5.10</b>	<b>Standard block drivers and their network addresses</b>	<b>82</b>
5.10.1	Overview	82
5.10.2	UDP block driver	82
5.10.3	Serial block driver	83
<b>5.11</b>	<b>Modules</b>	<b>84</b>
<b>5.12</b>	<b>Client API Interfaces</b>	<b>84</b>
5.12.1	Channel client (CmpChannelClient)	84
5.12.2	Gateway client (GwClient)	84
5.12.3	PLCHandler	85
<b>6</b>	<b>DEVICE- / I/O CONFIGURATION</b>	<b>86</b>
<b>6.1</b>	<b>Graphical Configuration</b>	<b>86</b>
<b>6.2</b>	<b>Devices</b>	<b>87</b>
<b>6.3</b>	<b>Device Descriptions</b>	<b>87</b>
6.3.1	Connectors	87
6.3.2	Parameters	92
6.3.3	I/O mapping	93
<b>6.4</b>	<b>Device Description Files</b>	<b>94</b>
6.4.1	Defining types	95
6.4.1.1	Bitfields	95
6.4.1.2	Range types	95
6.4.1.3	Array types	95
6.4.1.4	Simple structures	96
6.4.2	Defining strings for localization	96
6.4.3	Defining files and adding icons and images	97
6.4.4	Defining the device itself (identification, connectors, driver, parameters)	97
6.4.4.1	Device	98
6.4.4.2	Device identification	98
6.4.4.3	Device info	99
6.4.4.4	Driver info	100
6.4.4.4.1	Adding libraries and function blocks	101
6.4.4.5	Defining connectors	102
6.4.4.5.1	Slave with 1 connector	102
6.4.4.5.2	Master with 2 connectors	102
6.4.4.5.3	Multiple parent connectors	102

6.4.4.6	Defining parameters and parameter sections	103
6.4.4.7	Functional, defining child objects	107
6.4.4.8	Compatible Versions	107
6.4.5	Target description	108
6.4.5.1	Target settings	108
6.4.5.1.1	Runtime features	108
6.4.5.1.2	Memory layout	111
6.4.5.1.2.1	Some Use cases of memory layout settings	117
6.4.5.1.2.2	Child applications	120
6.4.5.1.3	Online	120
6.4.5.1.4	Task configuration	121
6.4.5.1.4.1	Application tasks	122
6.4.5.1.5	Network variables	126
6.4.5.1.6	Code generator	127
6.4.5.1.7	Device configuration	132
6.4.5.1.8	Library management	134
6.4.5.1.8.1	Placeholder Libraries	134
6.4.5.1.8.2	Placeholderlib, for replacing 3S-libraries by customer-specific libraries	135
6.4.5.1.8.3	Exclude library category	135
6.4.5.1.9	Visualization	135
6.4.5.1.10	Online Manager	141
6.4.5.1.11	Recipe manager	142
6.4.5.1.12	Symbolconfiguration	143
6.4.5.1.13	Trace	143
6.4.5.1.14	Object Type Restrictions	144
6.4.6	Custom tags	144
6.4.7	Strings	145
6.4.8	Types	145
<b>6.5</b>	<b>Device administration</b>	<b>146</b>
<b>6.6</b>	<b>Save and Restore Changed IO Configuration Parameters</b>	<b>147</b>
<b>7</b>	<b>I/O DRIVERS</b>	<b>148</b>
<b>7.1</b>	<b>Concept</b>	<b>148</b>
<b>7.2</b>	<b>Main I/O Driver Interfaces</b>	<b>150</b>
7.2.1	IBase	150
7.2.2	ICmploDrv	150
7.2.3	ICmploDrvParameter	158

<b>7.3</b>	<b>Optional Interfaces</b>	<b>158</b>
<b>7.4</b>	<b>I/O Manager</b>	<b>158</b>
<b>7.5</b>	<b>Access to the I/O Configuration</b>	<b>158</b>
<b>7.6</b>	<b>I/O Drivers in C/C++</b>	<b>159</b>
<b>7.7</b>	<b>I/O Drivers in IEC</b>	<b>160</b>
<b>7.8</b>	<b>Diagnostic Information</b>	<b>160</b>
7.8.1	General diagnostic information bit-field	160
7.8.2	Extended diagnostic parameter	161
7.8.3	Extended diagnostic acknowledge parameter	162
7.8.4	Implementation notes	162
<b>7.9</b>	<b>IO Consistency</b>	<b>163</b>
7.9.1	Consistency in the IO Driver	164
<b>7.10</b>	<b>External CAN Sync</b>	<b>165</b>
7.10.1	CAN L2 API	165
7.10.2	Timer ISR	165
7.10.3	Motion Cycle Time	166
<b>7.11</b>	<b>Byte order specific data handling in IO driver</b>	<b>166</b>
7.11.1	Bits handling in BYTE/WORD/DWORD	166
7.11.2	Helper functions for I/O update	168
7.11.3	Representation of bit-fields in IO configuration	168
<b>8</b>	<b>SYMBOLIC IEC VARIABLE ACCESS</b>	<b>170</b>
<b>8.1</b>	<b>Architecture</b>	<b>170</b>
<b>8.2</b>	<b>Database of Symbolic Information</b>	<b>171</b>
<b>8.3</b>	<b>Variable Access Interfaces</b>	<b>172</b>
8.3.1	Functional interface	172
8.3.2	Online interface	172
<b>8.4</b>	<b>Data Consistency</b>	<b>172</b>
<b>8.5</b>	<b>Behaviour at Download/Online Change</b>	<b>172</b>
<b>8.6</b>	<b>Usage on Small Embedded Systems</b>	<b>172</b>
<b>9</b>	<b>CUSTOMER ADAPTATIONS AND EXPANSIONS</b>	<b>174</b>
<b>9.1</b>	<b>Configuration</b>	<b>174</b>
9.1.1	Link type of the runtime system	174
9.1.2	Choice of components	175
9.1.3	Static configuration of components	176
9.1.4	Dynamic configuration (CmpSettings)	176
9.1.5	Typical configurations of the CODESYS Control WinV3 runtime system	179
9.1.5.1	Embedded Runtime System	179
9.1.5.2	Timer runtime system	179
9.1.5.3	Full runtime system	180



9.1.5.4	Gateway runtime system	180
9.1.5.5	Visualization runtime systems (target visualization CODESYS HMI)	181
9.1.6	Create your own configuration with the RtsConfigurator	181
<b>9.2</b>	<b>Implementing own components</b>	<b>182</b>
9.2.1	Global include files	182
9.2.2	Include files of the components	182
9.2.2.1	Interface file	182
9.2.2.2	Dependency file	182
9.2.3	Generation of include files	183
9.2.4	Source code file	183
9.2.4.1	General interface	183
9.2.4.2	Component specific implementation	185
<b>9.3</b>	<b>Implementation Notes</b>	<b>186</b>
9.3.1	Error returning	186
9.3.2	Memory	186
9.3.3	Allocation of IDs	186
9.3.3.1	Vendor ID	186
9.3.3.2	Component ID	186
9.3.3.3	Interface ID	187
9.3.3.4	Class ID	187
9.3.4	Importing of functions	187
9.3.5	Calling of imported functions	187
9.3.6	Exporting of functions	187
9.3.7	Linkage with the runtime system	188
9.3.8	Order of the INIT Hooks	188
<b>9.4</b>	<b>Libraries</b>	<b>188</b>
9.4.1	Creating a library in CODESYS	189
9.4.2	Implementation of external POU's in the runtime system	189
9.4.2.1	Declaration in the runtime system	189
9.4.2.2	Declaration of functions	189
9.4.2.3	Declaration of function blocks	190
9.4.2.3.1	Declaration of methods	192
9.4.3	Implementation	193
<b>9.5</b>	<b>Adaptations to Specific Operating Systems or Processors</b>	<b>194</b>
<b>9.6</b>	<b>Licensing</b>	<b>194</b>
9.6.1	Derivate based licensing	194
<b>10</b>	<b>CODING GUIDELINES</b>	<b>196</b>

<b>10.1</b>	<b>General</b>	<b>196</b>
<b>10.2</b>	<b>Naming conventions and identifier</b>	<b>196</b>
<b>10.3</b>	<b>Data types</b>	<b>198</b>
<b>10.4</b>	<b>Component interfaces and dependencies</b>	<b>200</b>
<b>10.5</b>	<b>Startup sequence</b>	<b>201</b>
<b>10.6</b>	<b>Alignment</b>	<b>201</b>
<b>10.7</b>	<b>Use of special macros</b>	<b>201</b>
<b>APPENDIX A: MIGRATING FROM CODESYS CONTROL VERSION 2 TO VERSION 3</b>		<b>202</b>
<b>CHANGE HISTORY</b>		<b>204</b>

## 1 Introduction

A runtime system is referred to as the software, that is running on a system that controls a device or a machine. *CODESYS Control* is here the product name of the runtime system of 3S-Smart Software Solutions GmbH.

**CODESYS** is the product name of the complete software family of IEC 61131 programming tools.

**Control** stands for the runtime system to control a machine.

The runtime system CODESYS Control V3 is a completely new development of 3S. It is based on more than 10 years experience in the automation technology and is provided for a wide range of operating systems and processors.

In this document, the runtime system CODESYS Control Version V3 is declared in detail. With this document we give you all basic information to start working with it and to understand the internal mechanisms.

### 1.1 Runtime System in the Overall System

The runtime system is running typically in a PLC (programmable logic controller). The runtime system is connected via a communication media to communicate with clients (CODESYS, HMO, OPC-Server, etc.).

**WARNING:** For security reasons, controllers, specifically, their TCP/IP programming ports (usually UDP-Ports 1740..1743 and TCP-Ports 1217 + 11740 or the controller specific ports) **must not be accessible from the Internet or untrusted Networks** under any circumstances! In case Internet access is needed, a safe mechanism has to be used, like VPN and password protection of the controller. When you are going to start CODESYS Control Win V3 from the systray menu, you will get an appropriate warning in a dialog box, where you still can cancel the startup.

On the other side, the runtime system is typically connected to the IO-system (field busses, local IOs) and/or drives of the machine that is controlled.

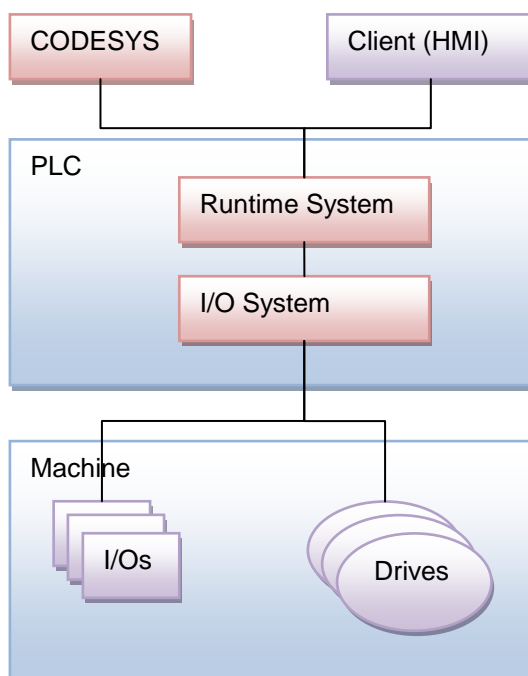


Figure 1. Runtime system in the overall system

## 1.2 Mission of the Runtime System

The runtime system CODESYS Control provides the following main functions:

- Execution of the IEC application(s), that are created with CODESYS V3
- Debugging of the IEC application
- Connection to the IO-system and/or drives
- Communication with the programming tool CODESYS V3 or another client (e.g. HMI)
- Routing for communication to subordinate runtime systems
- Runtime system to runtime system communication

## 2 Architecture

The architecture of the runtime system CODESYS Control V3 based on a component oriented model. Each logical, functional part of the runtime system is separated in one (or more) components.

Each component has a well defined interface. Via this interface, other components can have access to this component. Because a component has a specified interface, it can be replaced by another component that implements this interface.

Additionally a component can have a dependency to other components, if this component uses their interfaces to operate.

Because of this component model, components can be arranged together to a complete runtime system. This runtime system can be a very small one with less functionality or a full ranged runtime system with all features! This depends only on the number and kind of components that are arranged together.

Most of the components are system and processor independent and they can run on small 16-Bit up to powerful 32-Bit systems. This is realized by a "single-source" of the runtime system.

### 2.1 Overview

In the following figure you can see an overview of the architecture of the runtime system.

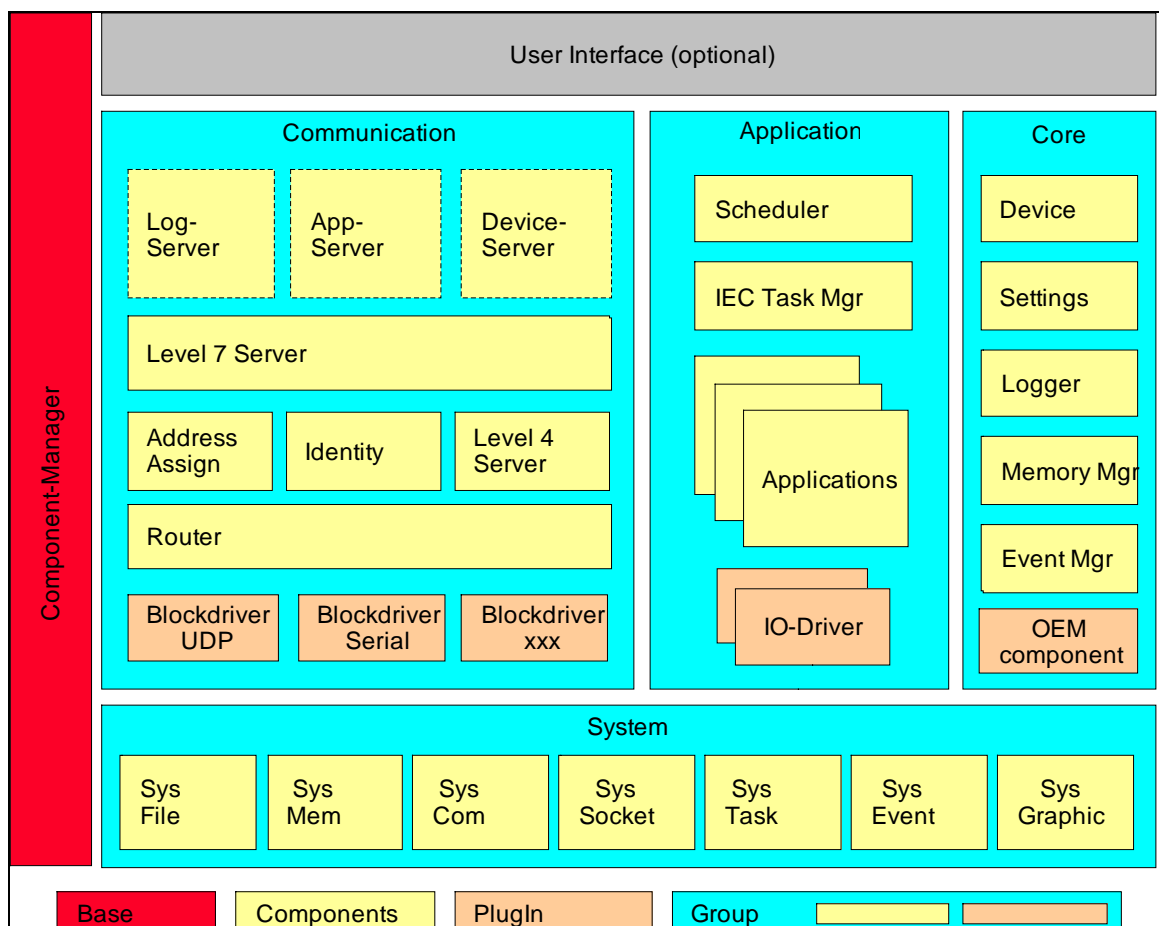


Figure 2. Architecture overview

As you can see, the base component is the **component manager**. This is the central component that must exist in every runtime system. The component manager loads and initializes all components and enables that each component can call each other.

The second important part consists of the so called **system components**. These components represent the hardware and operating system abstraction layer and hide processor and operating

system specific stuff from all other components. Via this system interface, all other components can be written absolutely portable and independent of processor and operating system internals. Every system component has its specific functions, e.g. file access, access to heap memory, access to a serial RS232 interface, operating system tasks, etc. All system components are described in chapter 3.10.

The third part is the **communication stack**. It consists of several components to enable the communication to clients (CODESYS, HMI, etc.) or to other runtime systems. For the last issue, a routing mechanism is additionally provided by the communication stack.

At the bottom of the stack there are the so called block drivers. Each of them provides one single communication medium (e.g. UDP, a TCP/IP protocol via Ethernet). The block drivers use the system components to get a neutral access to the physical layer.

The next layer in the communication stack is the router. It provides routing to each runtime system in a hierarchical network that consists of CODESYS Control V3 runtime systems.

Above the router, there are special service components e.g. for address assignment in the network, identification of a runtime system or the level 4 server for the block communication.

Each service request from a client first is received by the level 7 server. Each service is assigned to a specific group of services and then is forwarded to the special server, that was registered to the special service group (see chapter 5 for details of the communication stack).

In the architecture overview figure shown above you can see, that the communication stack is an integrated part of the runtime system. The gateway from version 2.x of CODESYS on version V3 is a special derivate of the runtime system that consists only of the component-manager, some system components and the communication stack (see chapter 5 for detailed information)!

The fourth part consists of the **application management** components. Here the execution, debugging and monitoring of the IEC application(s) is done. A completely new feature in CODESYS Control V3 is the management of several applications on ONE runtime system! This offers a lot of new application possibilities. The PLC task handling is separated up in two components: the IEC task manager and the scheduler. So a new scheduler algorithm can be provided only by replacing the scheduler component.

The last but not least part consists of the **core** components. These components provide the remaining functionality of a runtime system. The configuration of the runtime system, the logger to log all states and the device component for authentication reside here. The memory management and the event handling (for runtime events) are two additional components of the core.

For OEM customers there are several ways to extend the runtime system. They can:

- Replace existing components
- Write own Plugins (block driver, IO-driver)
- Write OEM components with external libraries, own service handling, etc.

## 2.2 Technology: ANSI-C and C++

All components are implemented in ANSI-C, because a C-Compiler is available for nearly all controllers, processors and operating-systems. So ANSI-C typically is the language to write own components. But you can write your own component in C++ with a C Interface.

But the architecture of CODESYS Control V3 has one additional big benefit: All components can alternatively be used with C++-Interfaces in object oriented environments! This is realized as a C++ wrapper around all components.

So it is on your choice to use the runtime system with C or with C++ interfaces. If you would like to write your own component in C++, the only thing you have to do is to compile the runtime system with the specific define CPLUSPLUS. After this you see around your component only C++ interfaces!

How this was realized in the component interface, this is explained in detail in chapter 2.7.

## 2.3 Single Source

To support the full range of runtime systems for small 16-bit controllers up to guidance systems with a single source of the runtime system, the key is the component orientation. For all systems, a set of base kernel components are always used. These base components are for example:

- Component manager
- CmpSettings
- CmpMemPool
- CmpLog
- And some system components

For special characteristics of the runtime system, the complete set of components is different, e.g. for the Gateway, the CODESYS Control RTE or an Embedded Runtime.

We defined the smallest configuration of the V3 runtime system (we call it “Compact Runtime”) for very small systems with a minimum set of functionality and some restrictions. This is declared in detail in chapter 2.3.1.

### 2.3.1 Compact runtime

For the smallest configuration of the V3 runtime system (we call it “Compact Runtime”), we defined a minimum set of functionality to run on small systems. As such a small system we defined *80 kB RAM* and *100 kB Code* (Flash).

#### Important:

The compact runtime is no special derivate or a fork of the V3 runtime system! It is only a defined set of components with a minimal configuration!

The restrictions of the compact runtime are:

1. Only one plc application
2. No Online Change
3. Only one online communication channel
4. No routing (CompactRuntime is endpoint of the routing line)
5. No breakpoints
6. No forcing of variables (i.e. setting a variable on the plc to a fix value)
7. Restrictions for the online monitoring:
  - No pointer dereferencing
  - No component access
  - No stack relative monitoring (only active, if breakpoints are enabled)
  - No property monitoring
  - No operator monitoring
8. No 64-bit support (no LREAL, LINT data types in IEC)
9. No external system libraries
10. No file transfer feature
11. No PlcShell
12. No Trace

In the future, the plc program can be executed directly out of the flash memory to save RAM. Additionally it is planned that an Online Change will be possible, if the plc program is executed out of RAM (not in flash).

#### Important:

All restrictions are not fix! Each restriction can be broken by using different components or by changing the configuration of the runtime system.

In the starter-package of the runtime system you can find a template for a Compact Runtime with the corresponding "RTS Configurator" project. This can be used as a template to create own compact runtime systems.

### 2.3.1.1 Compact download format

With the version V3.4.1.0 of the runtime system, a new download format is designed especially for embedded runtime systems to fulfill the following requirements:

- Easy to implement the download service in the runtime system
- Easy to implement storing and loading the boot project
- Ability to run the boot project directly out of flash memory

For this, a new download format is created. It consists of a contiguous binary stream with the following different content segments:

Segment	Description
CodeHeader:	Header of the stream:
RTS_UI32 ulHeaderTag;	HeaderTag=0x1234ABCD
RTS_UI32 ulHeaderVersion;	HeaderVersion=0x00000001
RTS_UI32 ulHeaderSize;	HeaderSize=104
RTS_UI32 ulTotalSize;	Total size of header including all segments
RTS_UI32 ulDeviceType;	Device type of the selected device
RTS_UI32 ulDeviceId;	DeviceID of the selected device
RTS_UI32 ulDeviceVersion;	Device version of the selected device
RTS_UI32 ulFlags;	Download Flags
RTS_UI32 ulCompilerVersion;	Compiler version of used CODESYS version
RTS_UI32 ulCodeAreaSize;	Code area size
RTS_UI16 usCodeAreaIndex;	Code area index
RTS_UI16 usCodeAreaFlags;	Code area flags
RTS_UI32 ulOffsetCode;	Offset in bytes, where the code segment begins
RTS_UI32 ulSizeCode;	Size in bytes of the code segment
RTS_UI32 ulOffsetApplicationInfo;	Offset in bytes, where the application info segment begins
RTS_UI32 ulSizeApplicationInfo;	Size in bytes of the application info segment
RTS_UI32 ulOffsetAreaTable;	Offset in bytes, where the area table segment begins
RTS_UI32 ulSizeAreaTable;	Size in bytes of the area table segment
RTS_UI32 ulOffsetFunctionTable;	Offset in bytes, where the function table segment begins, Here the link function are specified:
	- CodeInit
	- GlobalInit
	- GlobalExit
	- Reloc
	- DownloadCode
	- TargetInfo
RTS_UI32 ulSizeFunctionTable;	Size in bytes of the function table segment
RTS_UI32 ulOffsetExternalFunctionTable;	Offset in bytes, where the external function table segment begins to link c functions against iec code



RTS_UI32 ulSizeExternalFunctionTable;	Size in bytes of the external function table segment
RTS_UI32 ulOffsetRegisterIecFunctionTable;	Offset in bytes, where the iec function table segment begins to link iec functions against the runtime system
RTS_UI32 ulSizeRegisterIecFunctionTable;	Size in bytes of the external function table segment
RTS_UI32 ulOffsetSourceCode;	Offset in bytes, where the optional sourcecode segment begins
RTS_UI32 ulSizeSourceCode;	Size in bytes of the optional sourcecode segment
RTS_UI32 ulCrc;	RC32 of the complete download stream including the header with ulCRC written to 0!
Code segment	Code POU's
ApplicationInfo segment: All names are transmitted 4 byte aligned: Project name (ASCII string) Author (ASCII string) Version (ASCII string) Description (ASCII string) Profile (ASCII string) Date of last change (DATE_AND_TIME)	Application info
Area segment: List of AREA_INFO	Table of all used areas
Function table: 1. pfCodeInit 2. pfGlobalInit 3. pfGlobalExit 4. pfReloc 5. pfDownloadCode 6. pfTargetInfo	Function table to link to runtime
External Function table: List of EXT_REF_INFO (4 Byte aligned for the next entry!)	Table of external references
IEC Function table: List of FUNCTION_INFO with following name of the IEC function at the end (4 Byte aligned for the next entry!)	Table of IEC functions that can be called from C
Source code segment: Actually not used	Optional source code segment for the source code of the used project
CRC	CRC over the complete download stream

The code POU's can be relocated in CODESYS, if the address of the Data Area is fix and is known. This is the main requirement to execute the plc program in flash.

## 2.4 Error Codes

All functions in the runtime system return an error code. This error code has the type `RTS_RESULT`. All error codes can be found in `CmpErrors.h`.

Standard error codes that can be used in each component are defined at the beginning of `CmpErrors.h`. Each error code has a prefix with `ERR_`.

Each kernel component can specify its own error codes in `CmpErrors.h`.

Each OEM components can specify its own error codes in the range `f 0x8000..0xFFFF`.

## 2.5 Object Handles

All handles to data structures or objects have the type `RTS_HANDLE`. An invalid object can be checked with `RTS_INVALID_HANDLE`. All functions that return a handle must additionally return an error code to the caller to specify the reason of an `RTS_INVALID_HANDLE`.

## 2.6 Linkage

All components of the runtime system can be linked together in 3 different ways:

- **Static Link:** All components are linked together to one executable file (e.g. `*.exe`). This executable file is closed, so no additional component can be added afterwards to this package.
- **Dynamic Link:** Each component is linked here separately to a dynamically loadable module (e.g. `*.dll` or `*.o`). The components are loaded dynamically during startup of the runtime system. So at every start it can be decided, which components are to be loaded and linked to the runtime system.
- **Mixed Link:** This is a mixture of both issues above. The basic components are linked statically together to one executable file, but can get extended by dynamically loaded components.

How the linkage is done can be specified with compiler flags:

- `STATIC_LINK`: For static linking
- `MIXED_LINK`: For mixed linking
- `DYNAMIC_LINK` (or nothing): This is the default, if none of the two defines above is set

The source code of the components is identical in all linkage models. How this is realized is described in the next chapter.

## 2.7 Component Interface Architecture

The component interface of the runtime system must match the following requirements:

1. For embedded targets, all runtime system components should be linked static with direct C-functions calls in order to have a maximum of performance.
2. For more powerful systems with an operating system, the components should be loaded dynamically in order to have the flexibility to select components that should be used during startup time. Here, all functions of other components must be called via function pointers that are investigated during start up of the system.
3. For C++ runtime systems, the C++ calling conventions (name mangling, virtual function calls, methods) should be used.

To perform all these issues, 3S has developed an own interface technology based on C-macros.

Every function call must be done not directly with a call of the C-function. You always have to use a special `CAL_` prefix for that.

Additionally the export and import of functions must be done by special `EXP_` and `GET_` macros.

The usage of these macros is explained in the following:

### 2.7.1 Calling convention

If a component Cmp1 wants to call a function Fct1 of component Cmp2 and Cmp1 does not call the function directly, it will have to call the function via the specific CAL\_ macro:

```
CAL_Fct1();
```

The CAL\_ macros can now be resolved in different ways:

```
Static Linking:      #define CAL_Fct1      Fct1
Dynamic Linking:    #define CAL_Fct1      pfFct1 // Function pointer to Fct1
C++:                #define CAL_Fct1      ICmp2::Fct1
```

As you can see, the CAL\_ macro reaches the goal to adapt the calling convention to the linkage mechanism. On the other hand you can see that the caller does not see any difference, in which linking environment the component is embedded.

### 2.7.2 Export convention

To use a function from another component, the function first must be exported by this component. This is done with a second macro, the export macro. Therefore we created the so called EXP\_ macro. Referring to the example shown above, Cmp2 has to export its Fct1 with the following macro at a specific moment during start up of the runtime system:

```
EXP_Fct1;
```

This macro is expanded in different environments like:

```
Static Linking:      #define EXP_Fct1      ERR_OK
Dynamic Linking:    #define EXP_Fct1      CMRegisterAPI( "Fct1", (void*)Fct1, 0, 0)
C++:                #define EXP_Fct1      ERR_OK
```

Only in case of dynamic linking, component Cmp2 has to register its function pointer of the exported function at the component manager to use it later from other components.

### 2.7.3 Import convention

If a component needs to call functions from other components, it has a dependency to other components. This issue is covered by the third macro called GET\_ macro.

In the example shown above, Cmp1 has to import Fct1 with the following macro at a specific moment during start up of the runtime system:

```
GET_Fct1;
```

This macro is expanded in different environments like:

```
Static Linking:      #define GET_Fct1      ERR_OK
Dynamic Linking:    #define GET_Fct1      CMGetAPI( "Fct1", (void **) &pfFct1, 0)
C++:                #define GET_Fct1      ERR_OK
```

Only in case of dynamic linking, component Cmp1 has to resolve the function pointer of the imported function from the component manager to use it later.

In the case of dynamic linking you can see that there is a function pointer needed to hold the pointer to the function. For this a second macro is needed, that declares these function pointers. It is called USE\_ macro.

In the example shown above, Cmp1 has to declare a function pointer for Fct1 with the following macro at the beginning of its C-file:

```
USE_Fct1;
```

This macro is expanded in different environments like:

```
Static Linking:      #define USE_Fct1
```

```
Dynamic Linking:    #define USE_Fct1    PFFCT1 pfFct1;
C++:                #define USE_Fct1
```

### 2.7.4 Check functions

The last category of macros is the so called check macros. To check, if an interface function is available, there is always the possibility to use the **CHK\_** macro. The macro will return 1, if the interface functions is available and 0 if not. In your code, this macro can be used as follows:

```
if (CHK_Fct1)
    CAL_Fct1();
```

### 2.7.5 Summary

As you can see, the component interface is basing on special C-macros. To pursue the example described above, the macros must be used in the two components as follows:

Cmp1.c:

```
USE_Fct1;
int ImportFunctions(void)
{
    GET_Fct1;
}

int Code(void)
{
    ...
    CAL_Fct1();
    ...
}
```

Cmp2.c:

```
int ExportFunctions(void)
{
    EXP_Fct1;
}
```

As you can see, there are a lot of macros necessary to realize the component interface mechanism. To simplify this, we automate this process with the GNU m4-Compiler. This is declared in the next chapter.

## 2.8 M4 Mechanism

To reduce the effort for the generation of the component interface macros, we use the GNU M4 pre-processor. Therefore, two different types of description files must be used:

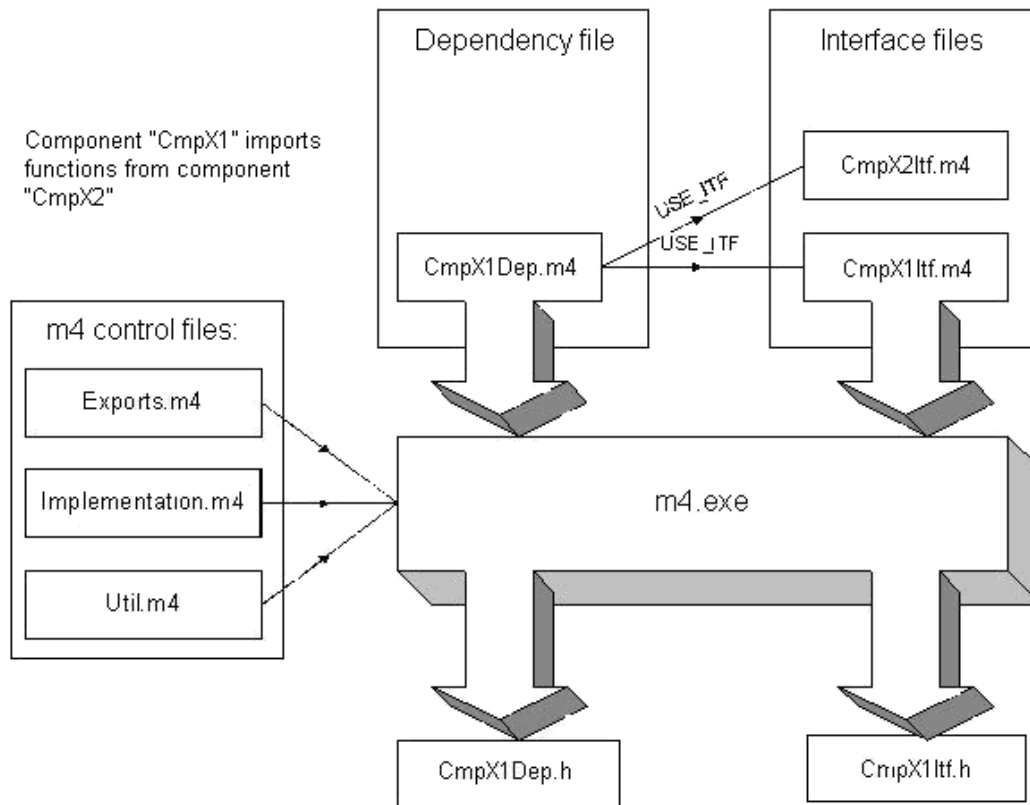
- **\*Itf.m4:** The interface description file contains all information about an interface, that a component provides

- \*Dep.m4: The dependency file contains all dependencies on other components

From these description files (interface and dependency) the GNU M4 macro pre-processor generates the associated header files, using the three control files Export.m4, Implementation.m4 and Util.m4.

For generating the dependency header file the macro pre-processor accesses all interfaces used. No further file is required for generating the interface header file.

The diagram below illustrates the generation of the header files for component CmpX1, where this component wants to import functions from component CmpX2.



### 2.8.1 Interface description file

The interface description file must contain an entry for each function to be exported from the component. Each function must be defined with one of a set of specific keywords. The keywords always have the same structure:

```
<KEYWORD>(`<Return type>',`{CDECL|STDCALL}' , `<Function name>',
`(<Signature>)' [ `<LinkToIECApp>{1|0}' ] )
```

Example for the function Fct1 with prototype **int Fct1(int a)**:

```
DEF_API(`int',`CDECL',`Fct1',`(int a)',1,0)
```

An interface function can be specified with one of the following keywords:

Keyword	Description
DEF_API	C-functions (can be used for external libraries too)  In the runtime system we use the convention that a lower case function name indicates that this function can be called from the IEC application. With the lower case we separate C-functions that can be used by other components from the functions that can be used by the IEC program.

Keyword	Description
DEF_STATIC_API	Static C-functions These interface functions cannot be used in the static linkage case. These functions are static and can only be used with dynamic linking and functions pointers.
DEF_ITF_API	Singleton classes Most of the classes in the runtime system are singleton classes. They implement one interface and always one instance is available.
DEF_STATICITF_API	Static class methods of Singleton classes This is only relevant for C++ implementations. This interface is implemented static, that no instance pointer is required to call this interface function.
DEF_CREATEITF_API	Instantiable class constructors This is used for interfaces that are defined once and are implemented only by one class in one component (e.g. all System classes like SysFile)! The return value of a function, that is specified by this keyword, is the handle to the object (class instance)
DEF_HANDLEITF_API	Instantiable class methods This is used for interfaces that are defined once and are implemented only by one class in one component! The first parameter of a function, that is specified by this keyword, is the handle to the object (class instance)
DEF_CREATECLASSITF_API	Instantiable class constructors This is used for interfaces that are defined once and can be implemented by more then one class in several components! One example is the CmpIoDrv-Interface that can be implemented by more then one I/O-driver. The return value of a function, that is specified by this keyword, is the handle to the object (class instance)
DEF_CLASSITF_API	Instantiable class methods This is used for interfaces that are defined once and can be implemented by more then one class in several components! One example is the CmpIoDrv-Interface that can be implemented by more then one I/O-driver. The first parameter of a function, that is specified by this keyword, is the handle to the object (class instance)

All keywords are expanded by the m4-compile process in the corresponding Itf-header files.

The optional parameter *LinkToIECApp* indicates whether this function can be linked from an IEC application. If the parameter is not defined, '0' (i.e. not linkable to IEC applications) will be assumed. The above examples are also included in file CmpTemplateltf.m4.

From each entry a function prototype, the definition of the function pointer, and the macros USE\_XXX, EXT\_XXX, GET\_XXX, CAL\_XXX, EXP\_XXX and CHK\_XXX are generated so that the functions can be used in a second module. In addition the signature checksum required by the component manager for registration and import of the function is calculated.

- for declaration of the function pointer (USE\_...)
- for declaration of external function pointers (EXT\_...)
- for importing (GET\_...)
- for calling the function (CAL\_...)
- for checking the function pointer (CHK\_...)

- for exporting (EXP\_...).

The exported component functions should be defined in a uniform manner such that the return value always is an error code. If the function is to return variable values, parameters with pointer to the return value must be used.

The macro pairs are distinguished by the following compiler flags:

1. **STATIC\_LINK**: If the components are linked statically, **STATIC\_LINK** must be specified in the preprocessor options.
2. **MIXED\_LINK**: Must be specified if the components are linked statically with the option of expansion through dynamic reloading of further components.
3. **CPLUSPLUS**: Must be specified if the runtime system is to be compiled in C++.
4. **DEFAULT**: If no compiler flag is specified, the components later will be reloaded as modules.
5. **<component name>\_NOTIMPLEMENTED**: Can be used to completely deactivate the interface of a component.
6. **<function>\_NOTIMPLEMENTED**: Can be used to deactivate individual functions of a component.

As an example the function `CmpTemplateOpen` of component `CmpTemplate` is examined:

The function declaration can be taken over from the header file into the source code file, where it is also implemented:

```
RTS_HANDLE CDECL CmpTemplateOpen(char *pszTemplate, RTS_RESULT *pResult);
```

Function pointer to this function:

```
typedef RTS_HANDLE (CDECL * PFCMPTEMPLATEOPEN) (char *pszTemplate,
RTS_RESULT *pResult);
```

Macro definition if the whole component or this function is to be deactivated:

```
#if defined(CMPTEMPLATE_NOTIMPLEMENTED) ||
defined(CMPTEMPLATEOPEN_NOTIMPLEMENTED)

    #define USE_CmpTemplateOpen
    #define EXT_CmpTemplateOpen
    #define GET_CmpTemplateOpen ERR_NOTIMPLEMENTED
    #define CAL_CmpTemplateOpen(p0,p1) (RTS_HANDLE)ERR_FAILED
    #define CHK_CmpTemplateOpen FALSE
    #define EXP_CmpTemplateOpen ERR_OK
#endif
```

Macro definition for the statically linked runtime system:

```
#elif defined(STATIC_LINK)
```

In the statically linked runtime system no function pointers are required.

```
#define USE_CmpTemplateOpen
```

The same applies to the external function pointer.

```
#define EXT_CmpTemplateOpen
```

In the statically linked runtime system the function pointer import is minimal.

```
#define GET_CmpTemplateOpen ERR_OK
```

The call is made directly as a function call.

```
#define CAL_CmpTemplateOpen CmpTemplateOpen
```

In the statically linked case the function pointer is always present.

```
#define CHK_CmpTemplateOpen TRUE
```

The function pointer does not have to be exported.

```
#define EXP_CmpTemplateOpen ERR_OK
```

Macro definition for the runtime system with mixed links:

```
#elif defined(MIXED_LINK)
```

The runtime system with mixed links contains elements of the statically and the dynamically linked runtime system. The core components are statically linked, while optional modules can be reloaded.

The core is statically linked, which means that no pointers are required.

```
#define USE_CmpTemplateOpen
```

```
#define EXT_CmpTemplateOpen
```

The core is statically linked, which means that no function pointers have to be imported.

```
#define GET_CmpTemplateOpen ERR_OK
```

The call to the function can also be replaced with a direct function call.

```
#define CAL_CmpTemplateOpen CmpTemplateOpen
```

Not required either

```
#define CHK_CmpTemplateOpen TRUE
```

In order to enable access to the functions of other components for dynamically linked optional components, the component manager must be notified of the function pointer.

```
#define EXP_CmpTemplateOpen s_pfRegisterAPI( "CmpTemplateOpen",
(void*)CmpTemplateOpen, 0, 0)
```

Definition for the C++ runtime system (statically linked):

```
#elif defined(CPLUSPLUS)
```

No function pointers are required since the C++ runtime system is also statically linked:

```
#define USE_CmpTemplateOpen
```

```
#define EXT_CmpTemplateOpen
```

The import of function pointers is also minimal

```
#define GET_CmpTemplateOpen ERR_OK
```

In the C++ runtime system function calls correspond to method calls.

```
#define CAL_CmpTemplateOpen
((ICmpTemplate*)s_pfCreateInstance(CLASSID_CCmpTemplate, NULL))-
>ICmpTemplateOpen
```

Before the call the existence of an object for the class must be verified:

```
#define CHK_CmpTemplateOpen (s_pfCreateInstance != NULL)
```

The export of functions is also minimal

```
#define EXP_CmpTemplateOpen ERR_OK
```

Definition for the dynamically linked runtime system:

```
#else /* DYNAMIC_LINK */
```

If a function is to be used, the function pointer must be declared with the USE macro:

```
#define USE_CmpTemplateOpen PFCMPTEMPLATEOPEN pfCmpTemplateOpen;
```

Declaration of an external function pointer

```
#define EXT_CmpTemplateOpen extern PFCMPTEMPLATEOPEN
pfCmpTemplateOpen;
```

The function pointer fetches the GET macro via the GetApi function of the component manager:

```
#define GET_CmpTemplateOpen s_pfGetAPI( "CmpTemplateOpen",
(void **)&pfCmpTemplateOpen, 0)
```



A function pointer is used for the call:

```
#define CAL_CmpTemplateOpen  pfCmpTemplateOpen
```

Before each call the validity of the function pointer should be verified.

```
#define CHK_CmpTemplateOpen  (pfCmpTemplateOpen != NULL)
```

During export a function pointer is registered in the component manager.

```
#define EXP_CmpTemplateOpen  s_pfRegisterAPI( "CmpTemplateOpen",
(void*)CmpTemplateOpen, 0, 0)
```

```
#endif
```

## 2.8.2 Dependency description file

The file starts with a comment containing a textual description of the component. The tags defined above should be used.

- 1 SET\_COMPONENT\_NAME(`<name of component>').  
Sets the name for this component
- 2 COMPONENT\_SOURCES(`<source file>')  
Specifies the source file for this component.
  - IMPLEMENT\_ITF(`<relative path to interface file>.m4').  
This entry specifies which interface this component implements. This entry is used to generate an #include statement for the associated header file and the EXPORT\_STMT macro, which should be called in the ExportFunctions function and which registers the interface functions in the component manager.
  - USE\_ITF(`<relative path to interface file>.m4').  
These entries specify interfaces from which functions are imported. They are mainly used to generate #include statements for the associated header files. The component manager also needs this information for checking the signatures of the imported functions.
  - REQUIRED\_IMPORTS(`<function1>', `<function2>', ...)  
This is used to generate the IMPORT\_STMT macro. The component must call this macro in the ImportFunctions function. The specified functions are then imported by the component manager. If an error occurs an appropriate message is written into the log, and ExportFunctions is terminated with an error.
  - OPTIONAL\_IMPORTS(`<function3>', ...)  
Specifies non-critical functions for import, i.e. the component is executable without these functions. Special assert macros (see below) can be used to restrict this function. This macro also extends IMPORT\_STMT with associated instructions.
  - ASSERT\_ONE\_OF(<A>,<B>,<C>, ...)  
At least one of the specified functions must exist.

CmpTemplateDep.m4 is illustrated as an example below:

Component name

```
SET_COMPONENT_NAME(`CmpTemplate')
```

Source file for the component.

```
COMPONENT_SOURCES(`CmpTemplate.c')
```

Interfaces implemented by this component.

```
IMPLEMENT_ITF(`CmpTemplateItf.m4', `CmpEventCallbackItf.m4')
```

Interfaces from which this component imports functions.

```
USE_ITF(`CmpLogItf.m4')
```

```
USE_ITF(`CmpSrvItf.m4')
```

```
USE_ITF(`CmpBinTagUtilItf.m4')
```

```

USE_ITF(`SysTaskItf.m4`)
USE_ITF(`SysMemItf.m4`)
USE_ITF(`CmpSettingsItf.m4`)
USE_ITF(`CmpAppItf.m4`)
USE_ITF(`CmpEventMgrItf.m4`)

```

Functions that must be imported.

```

REQUIRED_IMPORTS(
  LogAdd,
  ServerRegisterServiceHandler,
  . . .
  EventUnregisterCallback,
  EventPost,
  EventPostByEvent)

```

Functions that can be imported optionally.

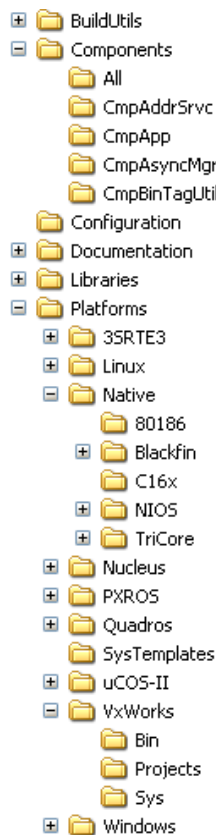
```

OPTIONAL_IMPORTS(
  SysMemIsValidPointer)

```

## 2.9 Source Tree of the Runtime System

The source tree of the runtime system is built as followed:



**Figure 3: Source tree of the runtime system**

All m4-Definitions and the m4 pre-processor you can find in the **\$\BuidlUtils** directory.

The base directory is **\$\Components**. Here you can find all Itf.h files in the root.

Under `$(Components)`, you can find a directory for each kernel component, like **`$(Components\CmpApp)`**. In each directory you can find all `c-`, `cpp-`, `Dep.h` and `Dep.m4` files for every kernel component.

One special directory is the directory **`$(Components\All)`**. This directory contains all kernel components and header files in flat order in single directory. This can be used on systems that have problems to compile and link the source files split in several directories. This simplifies the generation of make files too.

Under the directory **`$(Platforms)`** you can find all operating system and processor adaptation components. The components are separated by the operating system. All native adaptations without an operating system, you can find under **`$(Platforms\Native)`**.

Each adaptation part is separated into the directories **`$(Platforms\<OS>\Sys)`** with the system implementations, **`$(Platforms\<OS>\Projects)`** with the make files and project files and **`$(Platforms\<OS>\bin)`** with the optional binaries that are shipped with the runtime system.

## 2.10 Files on the Target System

Which files are necessary or created on your target file system, hardly depends on the type of runtime and the extended which you are using. At the absolute minimum, you will use an embedded runtime system, with a static configuration, no extended features and a flash file system for your boot projects. So you won't need any file system at all. But contrary to this, the following chapter will describe a full runtime system with all features.

### 2.10.1 Boot projects

By default, boot projects are saved as files, named `<application>.app` on the target file system. On embedded runtime systems they can be also saved in flash with the runtime component `SysFileFlash`.

If we are using a target file system, the runtime system will generate the following two files when a boot project is created:

- `<application>.app`

This file contains the application code and data.

- `<application>.crc`

This file contains a CRC value of `<application>.app` which is verified at boot time.

With the setting `[CmpApp]->RetainType.Applications=OnPowerfail`, the target will save the "retain" and "retain persistent" data to the file system. For this to work, the target needs to be shut down correctly. On power fail, this can be done with a big enough condensator, which can buffer the target long enough to shut it down. In this case the following file is generated for every application on shutdown:

- `<application>.ret`

This file contains the complete retain data segment of the corresponding application, and is therefore as big as the retain-segment is configured in the corresponding device description.

### 2.10.2 I/O manager

The I/O manager forwards all configuration-, read- and write-requests to the attached device drivers. Please refer to chapter 7 for a more detailed description.

Every Parameter which is set in the I/O Configuration is saved to the file "IoConfig.par". Those values are then restored at the next power up from this file.

### 2.10.3 User management

The User management of the CODESYS Runtime encapsulates the whole rights- and user management into the component `CmpUserDB`. This component can be implemented by the OEM

customer himself to match his special requirements. But there is also a fully complete user database included in the runtime.

This uUser database saves it's user and rights tables in simple CSV files on the target file system:

- UserMgmtDB.csv

This file contains all users and groups, which are downloaded by the CODESYS programming system.

- UserMgmtRightsDB.csv

The content of this file is downloaded from the CODESYS programming system, too. It contains all configured user rights for the different files and devices.

#### **2.10.4 Visualization**

If the target visualization is running in a different process as the runtime system, it needs a way to exchange the bitmap pool between the two processes. The bitmap pool consists of IDs, associated with image file names on the file system. This means, that when you are using a target visualization with images on your target, you will get also the following file, beside all the necessary image files:

- IPCBitmapPool.cfg

This file is used to share an image pool between the two processes of the target visualization and the soft PLC.

### 3 Overview of the Kernel Components and Main Functions

As described in the previous chapter, the runtime system V3 consists of single components that can be arranged and configured for different targets to reach every set of features.

All components can be assigned to one of the following categories:

1. **Component-Manager:** This is the central component that must always be part of a runtime system.
2. **System-Components:** Adaptation to the operating system and the processor
3. **Communication-Stack:** The component realizes the complete communication stack.
4. **Application-Management:** The component realizes the handling and management of the PLC applications and the PLC tasks.
5. **Core:** These components are further components of the core of the runtime system.
6. **Optional components:** Optional components can be integrated optionally in the runtime system. The usage depends on the target conditions or the feature set that should be reached. For example the target visualization components refer to this category.

In this chapter in the following the main components and their functionality is described in detail.

#### 3.1 Start up and Shutdown

The startup of the runtime system is executed in a strong layer based way in order to avoid that a component calls another component that is not initialized.

A defined shutdown is typically not possible on embedded targets. The power is switched off there and the runtime system has no time to make a defined shutdown. But on systems with e.g. an UPS a defined shutdown is possible. For these cases, the runtime systems make a defined and orderly shutdown.

##### 3.1.1 Startup

The startup is done in the following main steps:

1. Basic settings of the runtime system are verified.
2. Components get loaded and initialized.
3. Boot project gets loaded, IEC tasks get created, application(s) get(s) started.
4. Communication server gets started and corresponding tasks get created.

See in the following a more detailed description of the particular steps:

1. The basic runtime system settings are checked (data types, byte order setting, etc.).
2. The component manager loads all so-called *system components*. The system components are components that are always loaded and initialized before all other components are loaded. For example the logger component for logging all start up events is such a system component.
3. The `ComponentEntry()` routine is called by every system component in order to link all components with the component manager.
4. The system components will be called to export their interface functions and to import the needed interface functions.
5. The component manager loads all other components (in the static linkage case, nothing must be done; in the dynamic linkage case, the components are loaded dynamically).
6. The `ComponentEntry()` routine is called by every component in order to link all components with the component manager.
7. The other components will be called to export their interface functions and to import the needed interface functions.

## 8. Initialization:

- 8.1. The initialization of the system components is done with the hook CH\_INIT\_SYSTEM. In this hook, all systems components do their init code.
- 8.2. The initialization of all other components is done with the hook CH\_INIT. Here all local variables should be initialized.
- 8.3. The third level of initialization is done with the hook CH\_INIT\_DONE. In this hook, you can use all interfaces to other components! You can expect here, that all other components are initialized and can be used.
- 8.4. The fourth level of initialization is done with the hook CH\_INIT\_TASKS. Here all active parts can be started. Typically here the boot project is loaded and the PLC tasks are created.
- 8.5. The last level of initialization is done with the hook CH\_INIT\_COMM. Here all level 7 servers are started to open the communication to the other systems.

After the last hook (CH\_INIT\_COMM), the runtime system is completely initialized and started.

### 3.1.2 Operating mode

After the startup the runtime system goes into the operating mode. This mode is done with a cyclic call of the hook CH\_COMM\_CYCLE. This hook is called in the idle and main loop of the runtime system. In this hook every component can do some background jobs.

In a single tasking system the hook is the only position where all jobs are done, like communication, PLC task execution, etc.

The cycle time of the CH\_COMM\_CYCLE hook cannot be predicted. It belongs on the jobs that are done in all components. So try to minimize the jobs that are done in this hook to avoid an overloaded system!

### 3.1.3 Shutdown

An orderly shutdown of the runtime system is only possible on targets with an UPS or with a defined shutdown. The shutdown is done by the following steps, which basically are the same as performed during startup but in reverse order:

1. Deinitialization:
  - 1.1. The first level of deinitialization is done in the hook CH\_EXIT\_COMM. Here all level 7 servers should close their communication channels and the server tasks.
  - 1.2. The second level of deinitialization is done in the hook CH\_EXIT\_TASKS. Here all active tasks should be deleted.
  - 1.3. The third level of deinitialization is done in the hook CH\_PRE\_EXIT. In the prepare exit hook, all components should unregister from other components to dismiss the dependency to other components.
  - 1.4. The fourth level of deinitialization is done in the hook CH\_EXIT. Here all local resources can be released and the deinitialization of all components can be done.
  - 1.5. At least, the hook CH\_EXIT\_SYSTEM is called to deinitialize all system components.
2. After the deinitialization, the components (except the system components) are unloaded.
3. At least the system components are unloaded.

After the unloading of the system components the runtime system is completely down and released.

## 3.2 Component Manager

The component manager is the central component of the runtime system. It has the following job:

1. Loading components statically or dynamically during startup sequence
2. Initializing all components by calling the ComponentEntry() routines
3. Calling the ImportFunctions() routine for each component and hold a list of all API interface functions that are registered by the ImportFunctions().
4. Calling the ExportFunctions() routine for each component and Provides all interface API functions
5. Calling the CH\_ hooks of all components in the right manner of start up and shutdown and in the operating mode.

The component manager can be started by the main module with the CMInit() function. If this function returns with no error, the runtime system is completely started.

The component manager can be shutdown by the main module with the CMExit() function. If this function returns with no error, the runtime system is completely released and shutdown.

In the operating mode, the main module can call the CMCallHook() routine like:

```
CMCallHook( CH_COMM_CYCLE, 0, 0, FALSE);
```

In this hook, the component manager calls the HookFunction() routines of all loaded components.

## 3.3 Application Handling

The CODESYS VV3 runtime system is to be able to handle several applications. Each of these applications can be loaded and run independently of each other.

Each application defines one or more tasks. The tasks are processed in accordance with their properties (event-driven, time-driven, free running or driven by external events).

The tasks should behave pre-emptively if a multitasking operating system is available. The tasks for all applications are managed in a common task pool.

The time control of these tasks can either be carried out by the operating system or by the runtime system's scheduler. This depends, for instance, on whether the operating system is capable of calling tasks cyclically with high precision.

Event-driven tasks should be run promptly and in direct response to the occurrence of the trigger; it must not be necessary to poll them cyclically. The event may be set as edge-driven or level-driven.

Two different basic scheduling procedures must be possible:

- Cyclic processing is managed by the operating system; time-slicing for freewheeling tasks is handled by the scheduler of the runtime system, as is the runtime monitoring task (watchdog), and possibly also the time-slicing.
- Cyclic processing and time-slicing are carried out by the runtime system scheduler. A version is also to be possible here in which only one task is active at a time (one task scheduling).

It must be possible optionally to select a variety of time-slicing procedures:

- External time-slicing for all tasks. Time-slicing is managed by another task outside the runtime system.
- Internal time-slicing:  
A slice for all the IEC tasks alternates with a slice for the rest of the system. The available execution time can in turn be assigned in time-slices to the IEC tasks.

Tasks with the same priority are handled in a round robin procedure. This means that each task gets assigned a time-slice and is activated during that time-slice. Once the time-slice has elapsed, the next task in the list will be activated. In this case, the length of a time-slice is always the schedule interval.

Watchdog handling for cyclic tasks is carried out by the runtime system scheduler. In this case, the watchdog time and the CPU usage of all the IEC tasks must be taken into account. The hardware watchdog is also managed by the scheduler.

The exception handling system must provide a defined response on all exceptions that might occur (DivByZero, Access Violation, InvalidOpcode, etc.). When an exception occurs, the PLC is halted and the call stack is determined (including in the case of an endless loop). An entry is also created in the logger at this point.

As an option, a callback can be appended in order to provide more accurate control of the behaviour (abort or continue).

### 3.3.1 Overview

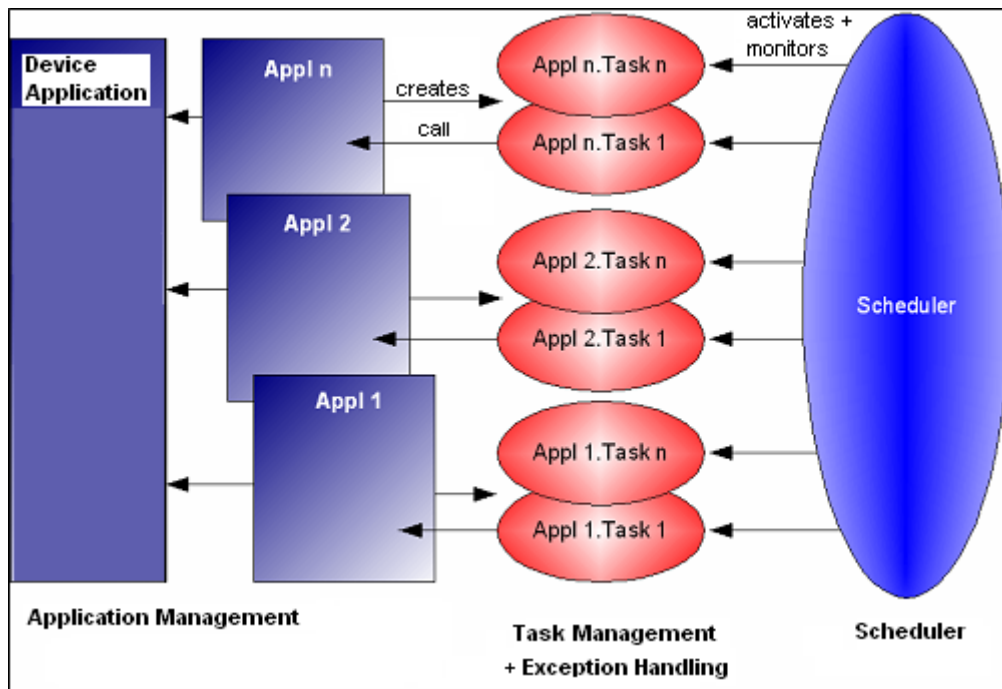


Figure 4: Overview

Figure 1 shows the general structure of the individual units. The four primary units of application management, task management, exception handling and the scheduler can be seen.

Application management contains all the application code and the data for all the applications, as well as the current status of each application. Each application is uniquely identified here with a name and an ID.

When an application is loaded, the application management generates the relevant IEC tasks in the task management. All the tasks from all the applications run there together.

The exception handling system is used by the task management. This has all the information needed (task, application, task context) to determine the call hierarchy through the exception component. The exception component is specific to the operating system and processor.

The scheduler carries out time control, the optional time-slicing and the watchdog management.

The advantage of this structure is its modularity. It is possible, for instance, to introduce a new scheduling procedure by exchanging the scheduler, without having to change the other components. This allows different schedulers having different features to be implemented and optionally included. This makes maintenance a good deal easier and solves one of the biggest problems of the former monolithic scheduler.



### 3.3.2 Application management

The application management system handles the application services to and from the programming system and the management data for all the applications, together with the current state of all the applications (Start, Stop, Reset, Exception).

The application component implements the application management precisely. It also receives services that concern an application (Login/Logout, Create/Delete, Download, Start, Stop, Reset, etc.). This involves support for a number of applications running at the same time. The component therefore registers a callback function with the server for online services of the SG\_APPLICATION group.

Each application can define one or more memory areas for user code and user data. The content is then managed by the programming system.

### 3.3.3 State behaviour (Start/Stop/Error)

The Start, Stop and Reset commands each operate on one application. A program can be started if it has been loaded and an associated application is present.

If an application has generated an exception, the application is going into Stop state. The application can then be started directly! It must be done a reset first!

### 3.3.4 Boot project

Once the runtime system has been initialized, an attempt is made to load the boot project. Each application has its own boot project.

The format of the boot project corresponds exactly to the download format. The same functions are therefore called to load the boot project as to download it. The content of the download message will be written into a file. This file is then available as boot project.

There are three times at which the boot project may be created:

- optionally immediately after download
- while running, when requested by the user (after online change, too!)
- Offline

These three different mechanisms to create a boot project are described below. The event indicating a successful creation is EVT\_CreateBootprojectDone.

#### 3.3.4.1 Create boot project at download implicitly

A boot project is automatically generated with the download of an application.

The CodeGUID and DataGUID are transmitted at the beginning of the download service. This is needed to avoid the repositioning on flash file systems to the beginning of the file to store these GUIDs.

#### 3.3.4.2 Create boot project after online change implicitly

A boot project is automatically generated after the online change of an application. Here only a description of the code needed for the boot project is transferred to the target. So no complete transfer of the code is needed.

#### 3.3.4.3 Create boot project on demand

A boot project can be created in online mode on demand via command "Create boot project for <application>", which per default is available in the Online menu in CODESYS.

Not the complete boot project is transmitted in this case, but only a description of the used POUs of the application that is actually running on the target!

#### 3.3.4.4 Create boot project offline

In offline mode the same command as described in chapter 3.3.4.2 ("Create boot project for <application>") can be used to create a boot project. The boot project can be stored as a file \*.app and be loaded via file transfer to the runtime system. The user will be asked whether a possibly already

available compile info should be overwritten. This should be done, if the intention is to load the boot project via an external tool to the PLC (because then no further download is necessary), because: In this case the compile info of a previous download would not match to the new boot project.

Don't forget to add this boot project to the configuration file of the runtime system (e.g. CODESYSControl.cfg). The following entry is needed:

```
[CmpApp]
Application.x=<Application name>.app
X stand here for an index, starting with 1.
```

### 3.3.5 Retain variables

Retained runtime system data (e.g. PLC variables) retain their value even when the controller is restarted. Two different storage procedures are distinguished here:

1. storage in static memory (e.g. SRAM)
2. storing the retain data in a non-volatile storage medium (such as Flash memory or hard disk) when the controller is shutdown. The data is held in RAM at runtime.

The latter procedure can only be used if it can be guaranteed that, when the controller is switched off, the power supply will remain on for a certain time in order to store the retain data (e.g. by means of a UPS<sup>1</sup> or battery).

The management of these storage areas for retained data is located in the CmpRetain component.

This also contains the management of the static memory that assigns individually requested areas to the retain memory (e.g. one area for a PLC device application and an area for a PLC application).

Note the possibility of using SRAMemory, already mapped by another component, for retains: By setting "Retain.SRAM.AddressMapped" in section [CmpRetain] a retain area is set to the specified address without mapping this area before:

```
Example: [CmpRetain]
Retain.SRAM.AddressMapped=0xABCD1234
```

Regard in this context that the following setting must always be specified to allow retains in SRAM:

```
[CmpApp]
RetainType.Applications=InSRAM
```

**ATTENTION:** If you use the mechanism to store retain variables in SRAM, you have to configure 24 bytes less memory for the application in the target description than is physically available! These bytes are reserved at the beginning of each retain memory segment to store the data GUID of the corresponding application into the SRAM, in order to check integrity after the application has been loaded! This is necessary, because an SRAM can be moved from one controller to another and so it must be checked if the data content matches with the assigned application.

**Note:** Regard further settings concerning retains in the device description: chap. 6.4.5.1.2, **Fehler! Verweisquelle konnte nicht gefunden werden.**

### 3.3.6 Debugging

If a breakpoint is set in CODESYS, the original code at the breakpoint position will be stored and a jump will be patched at this position on over this code. This has some advantages:

- no context switch necessary like on interrupt breakpoints
- no interrupt handling necessary, so it is more easy
- can be implemented on every controller

<sup>1</sup> UPS: Uninterruptible Power Supply

To step over a breakpoint the original code is restored before the step and then the original code is executed until the next breakpoint was reached.

In CODESYS V3 conditional breakpoints can be set, e.g. to specify the number of hit counts until the breakpoint is reached.

### 3.3.7 Download and online change

Downloading an application overwrites the application with the same name on the runtime system.

At Online Change, only the changed program units will be downloaded to the runtime system. These changed POU's will be downloaded in the same code as the remaining POU's, so no special reorganization is necessary with copying remaining POU's from the old to the new buffer.

After that, the copy code will be executed and the new code will be integrated in the still running application without stopping.

**ATTENTION:** If you make many Online Changes with increasing POU's, a fragmentation of the code area can occur, so that an Online Change is no more possible! After that only a reboot solves this problem.

#### 3.3.7.1 Download

The following steps are processed during an application download:

1. Check whether an application with the same name is already available on the controller:
  - If an application with the same name exists already, this will be terminated (IEC tasks are deleted, breakpoints and force lists are released, areas are deallocated (except PERSISTENT area!!!)). Then it will be replaced by the new one.
  - If an application with the same name does not yet exist: The new application will be created.
2. Area(s) are created. Application gets initialized and IEC tasks get created.

Events:

EVT\_PrepareDownload  
EVT\_DownloadDone

#### 3.3.7.2 Online Change

1. Modified POU's will be transferred to the controller.
2. Blanking interval is searched (Time at which no IEC tasks are active any longer).
3. Within the blanking interval the CopyCode gets executed and the modified code gets activated.
4. POU's which are no longer used or which are obsolete or removed will be marked in the area as "deleted", but will leave gaps. These gaps will be used by the next online change, but a segment fragmentation might result.

Events:

EVT\_PrepareOnlineChange  
EVT\_OnlineChangeDone

### 3.3.8 Events related to application handling

The events listed are members of the event class EVTCLASS\_INFO that is represented by the high word of the event Id.

Event	Description	Event Id
EVT_PrepareOnlineChange	Sent before the online change of the specified application will be executed	7
EVT_OnlineChangeDone	Sent after an online change of the specified application has been executed	8

Event	Description	Event Id
EVT_PrepareDownload	Sent before the specified application will be downloaded	9
EVT_DownloadDone	Sent after the specified application has been downloaded	10
EVT_CreateBootprojectDone	Sent after a boot project of an application has been created successfully	16
EVT_LoadBootprojectDone	Sent after a boot project of an application has been loaded successfully	17
EVT_DenyLoadBootproject	Sent to deny the loading of a boot project of an application.	18
EVT_PrepareLoadBootproject	Sent before a boot project of an application will be loaded	19
EVT_DenyStartBootproject	Sent to deny the starting of a boot project of an application	20
EVT_PrepareStartBootproject	Sent before a boot project of an application will be started	21
EVT_StartBootprojectDone	Sent after a boot project of an application has been started	22
EVT_DenyStart	Sent to deny the starting of an application	23
EVT_DenyStop	Sent to deny the stopping of an application	24
EVT_AllBootprojectsLoaded	Sent after all boot projects have been loaded	25

### 3.3.9 System variables for controlling critical runtime services

Certain commands from the programming system concerning application services on the RTS might be dangerous in a critical condition of the machine (e.g. stop application, set a breakpoint, online change or reset). They can be disabled/enabled by using the following system variables in the application program; they are properties of the "PlcOperationControl" module (component manager.library):

- xDisableApplicationOnlineChange
- xDisableApplicationDownload
- xDisableApplicationStop
- xDisableApplicationBP
- xDisableApplicationWrite (disabled via PLCHandler/IecVarAccess too!)
- xDisableApplicationForce
- xDisableApplicationReset
- xDisableAll

By default these variables are FALSE. By setting them to TRUE the corresponding runtime service gets disabled. This means it will be rejected and a message box will inform the user correspondingly.

Example:

```
PlcOperationControllInstance.xDisableApplicationStop := TRUE; //disables stopping the application
```

<b>Note:</b>	The use of these variables might be completely suppressed by the following entry in the RTS cfg-file: [ComponentManager] DisablingOperations=1
--------------	--

### 3.3.10 Accessing project and application information

In CODESYS it is possible to specify information about a project and its applications. These entries can be accessed in the runtime system.

### 3.3.10.1 Project information

In the project information dialog, you can specify the following entries:

- Company
- Title
- Version
- Author
- Description

These information can be extended by own entries of the type, text, number, bool and version. To make these information available in the runtime system, in the dialog there is a check box "Automatically generate POU's for property access". If you enable this check box, a set of access functions is generated in the POU pool, so you can access this information out of the plc program.

Additionally, these access functions are available in the runtime system at the next download. In the runtime system you can use the API function `AppGetProjectInformation()` to get default information. To use the extended information, you can use the property access functions `AppGetBooleanProperty`, `AppGetTextProperty`, `AppGetNumberProperty` or `AppGetVersionProperty` (see [1] for details).

### 3.3.10.2 Application information

The information about an application can be specified in the property page of an application. Here the following entries are possible:

- Project name
- Author
- Version
- Description
- Used CODESYS Profile
- Date of last change

The entries are not extendable by the user. This information will always be downloaded to the runtime system. In the runtime system you can access this information be the API function `AppGetApplicationInfo()`.

## 3.4 Watchdog Handling

All IEC-tasks of an application can be supervised by the runtime system to avoid cycle time overruns. The watchdog function can be enabled and configured in the task configuration of the PLC application. The task will be terminated with error status ("Exception"), when the currently set watchdog "Time" gets exceeded, whereby the currently set "Sensitivity" is included. Two cases are possible:

1. Contiguous time overruns; the following is true:

Sensitivity	Exception in cycle ...
0,1	1
2	2
...	...
n	n

2. Single time overrun: Exception if the cycle time for the current cycle is greater than (Time \* Sensitivity). Example: Time=t#10ms, Sensitivity=5 -> Exception as soon as the task (once) runs longer than 50ms. This serves to detect endless loops in the first cycle.

If the watchdog expires, the outputs are reset to its default values, if in the device dialog of the PLC application the entry "IO update while in stop" and "set outputs to default values" is specified.

By use of the library function in `CmpIecTask.library` a watchdog may be suspended for particular PLC cycles; this may be helpful for cycles requiring an extra amount of time due to initialization processes.

Having declared an appropriate variable of type `RTS_IEC_HANDLE` to handle the task

```
hIecTask : RTS_IEC_HANDLE;
```

deactivating and subsequent reactivating of the watchdog can be realized with help of the interface functions:

```
hIecTask := IecTaskGetCurrent(0);
IecTaskDisableWatchdog(hIecTask);
... // Code that is protected against watchdog
IecTaskEnableWatchdog(hIecTask);
```

### 3.4.1 Monitoring

The monitoring of the IEC variables is always done task-inconsistently. This is done because here no influence of the IEC tasks on the runtime can be guaranteed.

But in the future there will be an option to read IEC variables task-consistently.

## 3.5 IEC Task Management

The IEC task management is done in the component `CmpIecTask`.

Four different types of tasks are possible:

1. Cyclic tasks:  
These tasks are executed in a specified interval.
2. Event tasks:  
These tasks are executed once every time, if a specified IEC BOOL variable changed its value from FALSE to TRUE.
3. Freewheeling tasks:  
These tasks are executed without a specified cycle in a loop. To enable running tasks with a lower priority, the freewheeling task frees the processor in a specified way:
  - Default: 20% of its execution time, the task will sleep at the end of a cycle
  - If a processor load maximum is specified in the scheduler, this is the load value for the freewheeling tasks too, e.g.  
processor load maximum = 60%  
execution time of the freewheeling task = 10ms  
-> Sleep time = (100% - processor load maximum) \* 10ms = 40% \* 10ms = **4ms**
  - Fix sleep time:  
A fix sleep time can be specified for the freewheeling task.  
This value can be specified additionally in the scheduler with the following setting:  
[CmpSchedule]  
Task.Freewheeling.Cycletime=10  
In this example, every freewheeling task sleeps at the end of its cycle 10ms.
4. External Event tasks:  
An external event task is executed every time, an external (runtime event, hardware event) occurred. The event is specified by name in the device description of the device. This event can be specified in the task configuration in CODESYS, if external event task is selected.

The list of all IEC-tasks in an application is generated by CODESYS in the form of initialized IEC data structures. At download, this list is created during initialization and is transmitted to the `CmpIecTask` component after initialization. The tasks are reported then to the scheduler and are created specifically, in accordance with their type, as operating system tasks.

The format of the task description is explained in more detail below.

### 3.5.1 Data Format of the task description

The download message contains information about an application's tasks in the form of initialized IEC data structures.

The IEC data structures are initialized with the other variables after the download.

The data structures are defined as follows:

```
#define Cyclic          0x0000
#define Event          0x0001
#define External      0x0002
#define Freewheeling  0x0003

typedef struct
{
    char stName[51];
    short nPriority;
    short KindOfTask;
    char bMicroseconds;
    unsigned int dwInterval;
    unsigned int dwEventFunctionPointer;
    char stExternalEvent[51];
    unsigned int dwTaskEntryFunctionPointer;
    int tLastCycleTime;
    int tAverageCycleTime;
    int tMaxCycleTime;
    int tMinCycleTime;
    int tJitterMax;
    int tJitterLast;
}Task_Info;

typedef struct
{
    short nTasks;
    char stApplicationName[51];
    Task_Info* ptaskinfo;
}sys_setup_tasks_struct;
```

The sys\_setup\_tasks\_struct structure contains the entire task configuration of an application.

### 3.5.2 Creating IEC tasks

```
void CDECL __sys__setup__tasks(sys_setup_tasks_struct* p)
```

This function is called after a download or after the boot project has been loaded. A pointer to a variable in the sys\_setup\_tasks\_struct structure is passed. This function saves a pointer to each task and adds this task to the scheduler component.

### 3.5.3 Creating an external event task

As an external event we describe an event in the runtime system, on which a task can be activated. This event is typically generated out of a runtime system component, but could also be generated out of the IEC-Code. An external event is always identified by its name, so the name of the event must be known.

To realize an external event task, three things are necessary:

1. You need a description of this external event in your device description (see additionally chapter 6.4.5.1.4.1):

To specify an external event, you have to add the following section in your device description. This is an example of an external event named "ExternalEvent1".

```
<ts:section name="taskconfiguration">
    <ts:setting name="externalevents" type="cdata" access="hide">
        <ts:value><![CDATA[
```

```

        <externalevents>
            <externalevent>
                <name>ExternalEvent1</name>
            </externalevent>
        </externalevents>
    ]]></ts:value>
</ts:setting>
</ts:section>

```

2. You need a component in the runtime system, that recognizes and store the event, that was assigned to the specified task:

In the runtime system, the CmpSchedule component offers two events called EVT\_ExternalEventTaskCreateDone and EVT\_PrepareExternalEventTaskDelete, if an external event task is created or deleted. So the first thing that you have to do is to register to these events (how to handle an external event is implemented as an example in the CmpTemplate.c module of the starter package).

In your callback routine of the EVT\_ExternalEventTaskCreateDone event, you have to store the hEvent parameter. And in the callback routine of the EVT\_PrepareExternalEventTaskDelete event, you have to remove the hEvent.

3. Last but not least you have to sent this event always, if the corresponding external event occurred in the runtime system (like an interrupt, etc.). To activate the task once, that is assigned to this external event task, you have to sent this event by CAL\_SysEventSet(hEvent).

After this, you can assign the external event specified in the device description to your task in your IEC program. After downloading your application this task should be executed every time, the event is sent in the runtime system.

Equivalent to the handling in the runtime system, you can handle an external event in IEC-Code. All needed runtime system interfaces for that are available as external System-Libraries.

## 3.6 Scheduling

Scheduling in the runtime systems means to call the PLC tasks at the exact specified cycle times or if special events occurred. The scheduler here has the function to do the time control of each task and to check the occurrence of specified events.

The scheduler is actually available in the following three different implementations for different target.

### 3.6.1 Single tasking

All the tasks are called here in an endless loop in the sequence that corresponds to their priorities. This procedure is also called cooperative multitasking, as every task is carried out to the end without interruption. Watchdog monitoring cannot be carried out here.

This implementation is typically used in very small embedded systems with no operating system. But the implementation has one big disadvantage: The communication is done here in the main loop too! So higher communication load here will lead to larger jitters of the IEC tasks. This disadvantage can be dismissed by the next two implementations of the scheduler.

### 3.6.2 Timer scheduler

The timer scheduler implementation bases on the principle that each IEC task will be executed by one timer device on the target. Some embedded targets have several timers on board, that can be used for that. Typically this implementation is used on systems with no operating system.

The runtime scheduler is used in this context only to supervise the IEC tasks by a software watchdog.



### 3.6.3 Multitasking

In multitasking systems, every IEC task is mapped to an operating system task. The SysTask component of each operating system provides priorities between 0..255 (see chapter 3.7 for details). The priorities of the IEC-tasks (0=Highest..31=Lowest) are mapped into the system priority range of 32=Highest..63=Lowest (TASKPRIO\_REALTIME\_BASE.. TASKPRIO\_REALTIME\_END).

In multitasking systems it is typical for the real-time operating system's scheduler to carry out the time-driven activation of tasks. In this case, however, it is not generally possible to execute cyclic tasks with high precision, only to react very quickly to specific events or interrupts. Because, however, a PLC often must handle cyclic tasks, dedicated time control with high resolution is needed here. In the system under consideration, this is implemented through a dedicated scheduler.

Not every operating system, what is more, supports a software watchdog to monitor task runtimes. This is required in order, for instance, to detect endless loops reliably and to react to them. A PLC scheduler therefore usually also provides a software watchdog functionality.

To provide a third level of safety, the scheduler should also, optionally, be able to trigger a hardware watchdog, so that failure of the basic software in the runtime system can also be detected.

It must be possible for three activation procedures and two time-slicing procedures to be optionally activated for the pure scheduling:

#### Activation:

1. Scheduling involving activation of all tasks at each schedule tick, where the activation time has elapsed. Activation is typically carried out here by means of OS events. After that, the lower-level OS scheduler activates and executes the tasks, in accordance with their priorities. Precise task execution depends on the PLC scheduler being called at accurately measured, evenly spaced times.
2. As 1, except that only the highest priority task is activated. This can simplify the time-slicing described later, since only one active task has to be suspended and resumed.
3. Certain real-time operating systems (such as RTLinux) are already able to call tasks cyclically with high precision. In this case, the PLC scheduler only needs to monitor the tasks (watchdog), not to execute time control.

#### Time-slicing:

- External time-slicing: An external task extracts time slices here from the processing of the PLC task. No PLC tasks are therefore activated during the suspend phases, and all the PLC tasks that are currently running at the start of this phase are suspended. All the tasks are then regularly activated again and processed in the resume phase. This mechanism based on message queues (see SysMsgQ) for synchronisation.
- Internal time-slicing: The scheduler itself reserves time-slots here, e.g. for the rest of the system (communication). This makes it possible to specify a fixed period for PLC processing along with the time-slot for the rest of the system (e.g. 800us PLC, 200us rest).

#### Note for devices with imprecise microsecond timing:

The multitasking scheduler tolerates microsecond timing errors of 25 %. That means a cyclic task with 1 ms interval is scheduled as expected if the microsecond timing (performance counter) and the scheduler tick are synchronized with a deviation < 25%. However, there are modern CPUs (Cortex A8-ARMs, Via X86 1.2 GHz, Atom) with e.g. CE 6 where the tick and the performance counters very soon show deviations of e.g. 751 microseconds. So a 1 ms task will miss some ticks (e.g. 3 % of a 1 ms task are missed). For such cases, the following setting in the runtime system configuration file is available to schedule the IEC tasks based on millisecond times even if microsecond timing is implemented (if 0 or setting not available, microsecond timing will be maintained)

```
[CmpSchedule]
DontUseMicrosecondTiming=1
```

### 3.7 Task management

Each task in the runtime system has a specified logical priority. This priority is separated into the following 8 task segments, each with 31 priorities to categorize tasks:

Usage	Defines	Priority
Task segment 1 For system tasks like scheduler	TASKPRIO_SYSTEM_BASE	0
	TASKPRIO_SYSTEM_END	31
Task segment 2 For real time tasks like IEC-tasks	TASKPRIO_REALTIME_BASE	32
	TASKPRIO_REALTIME_END	63
Task segment 3 High priority tasks like high prior communication tasks	TASKPRIO_HIGH_BASE	64
	TASKPRIO_HIGH_END	95
Task segment 4 Above normal tasks	TASKPRIO_ABOVENORMAL_BASE	96
	TASKPRIO_ABOVENORMAL_END	127
Task segment 5 Normal priority tasks like standard communication task	TASKPRIO_NORMAL_BASE	128
	TASKPRIO_NORMAL_END	159
Task segment 6 Below normal tasks	TASKPRIO_BELOWNORMAL_BASE	160
	TASKPRIO_BELOWNORMAL_END	191
Task segment 7 Low task priorities.	TASKPRIO_LOW_BASE	192
	TASKPRIO_LOW_END	223
Task segment 8 Lowest task priorities for background tasks.	TASKPRIO_LOWEST_BASE	224
	TASKPRIO_LOWEST_END	255
	TASKPRIO_IDLE	
	TASKPRIO_MIN	

So each task priority should be assigned to one of these 8 task segments.

A task can be specified with a special m4-Macro in its component Dep.m4 file. These macros are described in 10. If you specify your task with this macro, the list of used tasks in your system can be generated by the RTS-Configurator. Additionally this entry will be exported in the Reference-Documentation by the RTS-Configurator too!

So if you would like to know, which tasks with which priority are used in the runtime system, please look in the corresponding Reference-Documentation, in the Dep.m4 files for the macros or in the Dep.h files for the categories "Task", "Task prefix" or "Task placeholder".

### 3.8 Configuration (Settings)

The configuration of the runtime system is done in the CmpSettings component. The settings component has the possibility to use different backends. Actually available are the embedded backend and ini-file backend.

#### 3.8.1 INI file backend

The INI file backend operates on one or several INI files (examples: CODESYSControl.cfg, Gateway.cfg). Each component has one section in the INI file to specify configuration settings. See also: Chapter 9.1

The configuration file(s) to be regarded must be specified when starting the runtime system.

Example:

```
[CmpApp]
Bootproject.StoreOnlyOnDownload=0
```

During runtime, the names of the boot projects are written to the runtime configuration file. Due to this writing process the configuration file eventually may get damaged. Unfortunately, a corrupt configuration file will preclude the runtime system from being started, even not for communication purposes. This undesirable situation can be circumvented by employing one of the following procedures:

- a) By CMPSETTINGS\_MASTER\_CONFIG the name of a master configuration file can be defined, which will be copied and used in case of failure of the standard configuration file.
- b) The standard configuration file is set to read-only. In return, it has to contain a link to an additional cfg-file, the names of the boot projects can be written to.

The following steps have to be executed for specifying references to additional cfg-files:

1. Add the following entry to the file CODESYSControl.cfg of the SP runtime:
 

```
[CmpSettings]
FileReference.0=ExtConfig0.cfg
FileReference.1=ExtConfig1.cfg
...
```
2. Delete the complete section entitled [CmpApp] from CODESYSControl.cfg.
3. Create the files referenced by the master configuration file (ExtConfigx.cfg) within the same directory.
4. Set CODESYSControl.cfg to read-only.
5. Start the runtime system.

In consequence, the master configuration file will be write-protected and new entries will be written to the first file referenced that has no write protection.

### 3.8.2 Embedded Backend

The Embedded backend has no file access. Here all settings are returned "hard coded" in the corresponding c-file CmpSettingsBackendEmbedded.c. This is typically used on small systems with no file system.

## 3.9 Logging

The logger component (CmpLog) has the possibility to log all events of the runtime system like the start up and shutdown and all application downloads.

The logger can be instantiated, so each component can create its own logger.

One logger instance is always available and this is called the *standard logger*. In the standard logger, all components log by default.

One log entry consists of:

Entry	Description
Timestamp	Can be a RTC, microsecond or millisecond value (depends on the log options)
ComponentId of the component, that specifies the log entry	Each component in the runtime system has a unique Id. This Id must be specified here to recognize the source of the log entry.
Log class	The following log classes are available: #define LOG_INFO                   0x00000001

Entry	Description
	<p>For general information</p> <pre>#define LOG_WARNING      0x00000002   For warnings  #define LOG_ERROR        0x00000004   For errors  #define LOG_EXCEPTION    0x00000008   For exceptions  #define LOG_DEBUG        0x00000010   Only for debug log entries  #define LOG_COM          0x00000040   For communication entries  #define LOG_INFO_TIMESTAMP_RELATIVE   0x00000080   For entries with a timestamp, that is calculated as a difference from   the log entry before</pre>
Error Id	The error Id, if an operation failed
Info Id	An unique Id per component, that can be used to specify a longer test for the log entry in the target description of CODESYS. This is used to save resources in the runtime system to store the log entries
Info string	Info string with optional and variable information, e.g. application names or task names

The log entries can be added to the standard logger with the interface function LogAdd(). It can be used for example:

```
CAL_LogAdd(STD_LOGGER, COMPONENT_ID, LOG_ERROR, ERR_FAILED,
  LOGID_CmpApp_OpenBootprojectFailed, "<app>%s</app>", pszAppName);
```

### 3.10 Hardware and Operating System Abstraction Layer (Sys-Components)

The hardware and operating system abstraction layer is covered by the so called system interface components. Each system component is designed for a special assignment and a logical operation system object.

The description of all available system interfaces declared in detail in the system reference guide of the runtime system ("CODESYS Control Runtime System Reference.pdf").

#### 3.10.1 Time access (SysTime)

In this component, you can have access to ticks with different resolutions (millisecond, microsecond and nanosecond) for e.g. time measurements and access to some functions of the real time clock (RTC).

#### 3.10.2 Serial interface (SysCom)

This component provides access to a serial device (RS232). You can open and close a device, specify settings to the device and read and write data to this device.

#### 3.10.3 Exception handling (SysExcept)

The first level exception handling is done in this component. This component realizes on different platforms and operating system a defined exception handling.

If the exception is caught, the register set of the processor will be investigated (and the task, if available) and will be forwarded to the task or timer component to investigate the source component of the exception.

With the processor context a call-stack can be investigated by higher level exception handling components.

### 3.10.4 File access (SysFile)

This component provides routines for file access. You can open or close a file, read from or write into a file, delete or rename a file, set and get the actual file pointer in a file and so on.

One thing is quite important:

At the routines where a file name is specified as a parameter, there are two different implementations. E.g. to open a file can be done in with two different interface functions:

```
SysFileOpen(char *pszFile, ACCESS_MODE am, RTS_RESULT *pResult)
```

```
SysFileOpen_(char *pszFile, ACCESS_MODE am, RTS_RESULT *pResult)
```

The only difference between the SysFileOpen and the SysFileOpen\_ is the handling of the file names. If a path (absolute or relative) is specified in the file name, the path is used unchanged.

If no path is specified in the file name, the two functions work different:

In SysFileOpen, a default path is added to the file name by the runtime system. For the SysFile component, paths can be specified for each ending of the file name.

In SysFileOpen\_, no default path is added and the actual working directory is used.

This difference in path handling is identical for all SysFileX and SysFileX\_ routines.

### 3.10.5 File access using flash (SysFileFlash)

As an alternative to SysFile (see 3.10.4), this component provides routines for file access. You can open or close a file, read from or write into a file, delete or rename a file, set and get the actual file pointer in a file and so on.

As a difference, this component does not use the file system of an operating system, but uses the SysFlash component.

A basic file system is implemented. It is based on a static table of files, with fixed file names and maximum file sizes.

The file table has to be defined in sysdefines.h in variable FILE\_MAP. An example is provided in the header file of the component.

### 3.10.6 Flash access (SysFlash)

This component provides routines for flash access. The usage of this component is optional. It provides functions to write a data buffer to flash, read from flash, and get the address and size of the flash. It contains also a function that is called when a flash based file is closed.

The flash can be used as

- a basis for the SysFileFlash component, and
- for optional execution of the user code in flash. This can be used to reduce the RAM memory requirements.

### 3.10.7 Directory handling (SysDir)

The directory component provides access to a file system. Directories can be scanned, created, deleted or renamed.

### 3.10.8 Memory access

The handling of memory is provided by the SysMem component.

#### 3.10.8.1 Heap and static memory (SysMem)

To access the heap memory (dynamic memory allocation), there are different routines to allocate and free code and data memory. Code memory can be used to execute code in this memory.

To access static memory for the PLC application, there is a function called SysMemAllocArea(). Here a static memory for the code and data area for the PLC application can be specified.

#### 3.10.8.2 Physical memory access and shared memories (SysShm)

To get access to physical memory (e.g. dual port RAM of a card), the SysShm provides routines to map the physical memory into the memory of the runtime system.

### 3.10.9 Dynamic loading module (SysModule)

The dynamical load of components can be used on targets with an operating system. This functionality is provided by the operating system and can be used via the SysModule component.

The components must be available in a dynamical loadable format (e.g. under Windows a Dll, under VxWorks an o-file).

#### 3.10.10 Ethernet sockets (SysSocket)

The access to the TCP-stack, the SysSocket can be used. The socket interface provides TCP, UDP and RAW sockets.

The SysSocket component provides additionally more higher level interfaces, to set up a TCP or UDP client and server.

#### 3.10.11 Debug console outputs (SysOut)

To print messages at the console (if available), you can use the SysOut component. This component is used for example by the logger, to print all log entries at the standard console output.

#### 3.10.12 Message queues (SysMsgQ)

MessageQueues are a special higher level object to use for inter thread and task communication. The message queues are thread-safe and are used in the runtime system for example for the time slicing implementation of the Scheduler.

#### 3.10.13 Interrupt handling

Interrupt handling is a very low level and platform dependent component. This component provides routines to open an interrupt and to register an interrupt handler to this interrupt. All interrupt can be enabled or disabled with this component too.

#### 3.10.14 PCI bus access (SysPCI)

This component provides access to the PCI bus. The PCI configuration can be read and written for example to auto detect cards that are plugged in the PCI bus.

#### 3.10.15 Device port access (SysPort)

Accessing devices is typically done via so called ports. The SysPort component provides access to devices via these ports. Port values can be read and written.

### **3.10.16 Timer handling (SysTimer)**

Timer devices can be used for processes with a strong time constraint. The SysTimer component provides access to timer devices on the target. Timers are typically used in the runtime system for the cyclic call of the runtime scheduler or to execute IEC tasks with the specified cycle times.

### **3.10.17 Target information (SysTarget)**

The SysTarget component provides access to the target information like vendor and device name, target Id and version or the node name for the communication. The node name is detected in some implementations from the node name of the operating system that was assigned to the target.

### **3.10.18 Task handling**

These components are only available on operating systems with tasks or threads

#### **3.10.18.1 Synchronization and semaphores (SysSem)**

To synchronize access to common data between several tasks or threads, the SysSem component can be used to handle synchronization objects, called semaphores.

#### **3.10.18.2 Operating system events (SysEvent)**

To activate tasks or threads, the operating systems provides events to activate a task from another task.

#### **3.10.18.3 Task handling (SysTask)**

The SysTask component provides routines to create and delete tasks, to suspend (set into sleep) and resume (wake up) tasks and so on.

### **3.10.19 Optional system components for target visualization**

There are optional system components that are needed by the CODESYS target visualization. These components can be added to the runtime system, if a graphical interface on the target is available.

#### **3.10.19.1 Window handling (SysWindow)**

The SysWindow component provides access to the window handling of the graphical environment. The target visualization typically runs in a window.

#### **3.10.19.2 Basic graphic routines (SysGraphic)**

The target visualization uses some basic graphic operations to display all visualization objects. These basic operations are provided by the SysGraphic component.

With earlier versions, the size of JPEG images in the Windows and Windows CE target visualization was limited to 1024\*768\*3 bytes (approx. 2.3 MB). As from V3.5.1.0 the limit can be configured by an entry in the PLC configuration file CODESYScontrol.cfg:

```
[SysGraphic]
Win32.MaxJpegByteArraySize=5000000
```

Note however: The limit should not be chosen too big in order not to waste memory and because the CE CreateDIBSection function may fail with 10 MB pictures.

### **3.10.20 Process handling**

On operating systems with the possibility to use processes, there are some components to use these processes. This components are typically used in the runtime system to spawn an own process of the target visualization to separate the runtime system from the part that displays the graphic information of the target visualization.

### 3.10.20.1 Processes (SysProcess)

The SysProcess component provides access to create and delete processes.

### 3.10.20.2 Process synchronization (SysSemProcess)

To synchronize access to common data objects from several processes, the SysSemProcess component provides access to process semaphores.

### 3.10.21 Direct Ethernet controller access (SysEthernet)

This component provides direct access to an Ethernet controller. This is typically used by the EtherCAT drivers to get direct access to an Ethernet controller.

## 3.11 Memory Management

In embedded system, the usage of the memory must reach the following requirements:

- Usage of dynamic memory only at start up
- Avoid memory fragmentation with usage of alloc and free
- Static memory should be preferred

These requirements are covered by the CmpMemPool component. This component allows to use a static array that can be split up into a chained list of memory objects with the same size and this is called a memory pool. One memory block from the pool can then be used and free-ed with the CmpMemPool component. The unused blocks are incorporated in the memory pool to use it for the next object.

If all block of the pool are used, there is the possibility to allocate new block from the heap memory. If you release such memory block from the heap, the memory is not physically release! The block is incorporated in the memory pool to use it for the next object.

## 3.12 Events

All type of events are handled in the runtime system by the event manager component CmpEventManager. We separate here two different kind of event users:

- Provider: Component, that provides an event
- Consumer: Component, that uses an event

The CmpEventManager has routines for providers to create and delete and event.

For the consumer, there are routines to open and close events and to register callbacks to an event. Callbacks can be functions or methods of objects (this works with C, C++ and IEC consumers).

One event is always structured as followed:

```
typedef struct
{
    EVENTID EventId;
    CMPID CmpIdProvider;
    unsigned short usParamId;
    unsigned short usVersion;
    void *pParameter;
    void *pUserParameter;
} EventParam;
```

The event Id consists of the high word with the event class and the low word with the event Id. The event class can be one of the following:

```
#define EVTCLASS_INFO           0x00010000
#define EVTCLASS_WARNING       0x00020000
#define EVTCLASS_ERROR         0x00040000
```



```
#define EVTCLASS_EXCEPTION          0x00080000
#define EVTCLASS_VENDOR_SPEC       0x10000000
```

The event Id is unique for each component, but not unique in the runtime system! Only the combination of the event Id and the component Id makes an event unique!

The second parameter of an event is the component Id of the provider.

The parameter Id and the version specify the parameter that is provided by the pointer pParam. Every provider has to specify a structure with a specific parameter Id that is sent in pParameter.

The last parameter can be a parameter that is specified by the consumer at register a callback. This parameter is sent back with the event.

### 3.13 Exception Handling

Exception handling refers to the way processor exceptions or serious errors in program execution are handled. When such an exception occurs, it is not usually possible for normal program execution to continue.

The exception handling system is managed by the SysExcept component. This performs all the exception handling in a manner specific to the operating system. The task that triggered the exception, the exception itself and the context (IP register, base pointer and stack pointer) are determined here:

- Determining the triggering task
- Exception number (specific to the operating system)
- Determining the exact error location (current instruction pointer)
- Call stack (base and stack pointers)

The name of the exception in the operating system-dependent name is then converted to the name or number that conforms to the runtime system.

Every component can register itself for exception handling to the SysExcept component.

Typically, the SysTask component will register itself, in order to obtain all the exceptions. The operating system handle for the task is then converted into the SysTask handle in the SysTask component. An exception handler can be given to every task when it is created (SysTaskCreate()). In the event of an exception, this is then called from the SysTask component. The reason for this is that it is only the component that set up a task that is able to handle the exception properly.

If, for instance, a PLC task is set up, the exception handler will be appended by the multitasking or application component. When this handler is then called, the PLC would stop, etc. From the context it is then possible to determine the call hierarchy in the PLC program.

Every task is monitored for at least the following exceptions:

- division by 0
- access violation (access to invalid addresses)
- invalid opcode
- error in dynamic memory allocation (malloc)

When an exception occurs, the following steps will be carried out with the aid of the SysExcept component:

1. the task or application will, according to the specification or setting, be halted, or the IEC program will first be asked about how to continue
2. the call hierarchy is determined (call stack of all POUs)

### 3.13.1 Structured exception handling (rts\_try / rts\_catch)

The runtime system has a build in structured exception handling. It can be activated by the define `RTS_STRUCTURED_EXCEPTION_HANDLING`.

The structured exception handling based on the standard C-Lib API functions `setjmp` and `longjmp`. With `setjmp`, an actual context can be saved. With `longjmp` you can jump back to this previous saved context. Because these functions are available on nearly every platform, this is a highly portable mechanism.

To use this mechanism, you have to import the following interface and the functions in your `Dep.m4` file:

```
USE_ITF(`SysExceptItf.m4')

SysExceptRegisterJmpBuf
SysExceptCatch
SysExceptUnregisterJmpBuf
```

After this you can use it like the following code snippet:

```
rts_try          /* Mandatory */

{
    ... /* Code where the exception can occur */
}
rts_catch       /* Mandatory */
{
    RTS_UI32 exceptionCode = EXCPT_GET_CODE();
    RegContext *pExceptionContext = EXCPT_GET_CONTEXT();
    ... /* Exception handling code */
}
rts_finally     /* Optional */
{
    ...
}
rts_try_end     /* Mandatory */
```

With this mechanism, you can handle every exception exactly at the place, where it occurred without losing the context! This can be helpful for example to handle exceptions at layer 7 services and to send back an online error code without disturbing the communication stack and without losing the communication.

## 3.14 License Check

Extra license fees have to be paid for various components of the runtime system and the programming system.

This chapter describes the protection of features that have been licensed for a certain device type.

The check is performed by components which an OEM has already purchased and is to prevent the customer of an OEM from using non-free components on controllers of other manufacturers who have not purchased such components. Furthermore, the license check is to make sure an end user can only work with components which have actually been tested and activated for the device in question.

A working license check needs a license file (`3s.dat`) which must be part of the controller firmware. It defines which features are activated and also contains the exact ID of the runtime system. So it can only be used on the device in question. 3S-Smart Software Solutions will provide the OEMs with such a license file for each controller type. This file must be made available in the directory of the runtime system.

The correct license file is generated by 3S-Smart Software Solutions. So following informations are needed:

Manufacturer:	Customer			
<b>Name of controller:</b>		...	...	...
<b>Device type*:</b>		...	...	...
<b>Vendor-ID:</b>				
<b>Device-ID*:</b>		...	...	...
<b>CPU family:</b>	x86			
	ARM			
	PowerPC			
	MIPS			
	SH			
	Blackfin			
	Tricore			
	Nios			
	C16x			
<b>Operating system:</b>	Windows 2000/XP			
	Windows CE			
	Linux			
	VxWorks			
	QNX			
	µCOS			
	others:			
<b>Supported features**:</b>				
	<b>Visualization:</b>			
	Target Visualization			
	Web Visualization			
	<b>Field bus (Master / Slave)**:</b>			
	CANopen			
	EtherCAT			
	SercosIII			
	Profibus			
	Profinet			
	Modbus (TCP)			
	Modbus (Serial)			
	DeviceNET			
	EtherNet/IP			
	FDT			
	<b>SoftMotion:</b>			
	SoftMotion			
<p><b>*) This information can be found in the RTS component SysTarget or in the device description file (devdesc.xml) of your controller:</b></p> <pre> &lt;Device&gt;   &lt;DeviceIdentification&gt;     &lt;Type&gt;4102&lt;/Type&gt;          &lt;-Device type     &lt;Id&gt;0000 0002&lt;/Id&gt;        &lt;-Vendor-ID and Device-ID </pre>				

```
<Version>3.4.0.0</Version>  
</DeviceIdentification>
```

...

**\*\*) Please fill out only 3S Field busses (stack or configurator) – no own developments!**

Every single controller type has its own ID consisting of vendor ID and device ID. An OEM can freely define the device IDs for their controllers but needs to register them at 3S to enable 3S to generate the license files.

If the license file is missing or is incorrect, the non-free components will either not run at all or run in a demo mode and the logger will issue an error message. The visualization in demo mode displays a box on the upper right of the screen that identifies the demo mode. The demo mode of the field bus stacks are indicated with an orange icon (instead of the green icon) in the device tree.

This works on components only from CODESYS V3.4 or higher. Devices with an older version of CODESYS which work with non-free components of an older version are not affected by this licensing mechanism

The actual license check takes place at runtime. If OEMs want to prevent that customers using a non-licensed field bus when programming their application, they can do so by making an entry in the device description (see "allowonly" property of a Connector in device description file)..

### 3.15 Online User Management

In the runtime system, you can use a User Management similar to the management in a CODESYS Project. With this User Management, all important Online Operations are protected.

You can create Users and Groups and you can assign Users to these groups. For details see the Online Help of CODESYS in the chapter:

Editors > Device Editors > Generic Device Editor > Users and Groups

You can assign access rights on all available runtime system objects to these Groups. For details see the Online Help of CODESYS in the chapter:

Editors > Device Editors > Generic Device Editor > Access Rights

To enable this feature, you have to enable the corresponding Device Editor page in your Device Description:

```
<DeviceDescription xmlns="http://www.3s-software.com/schemas/DeviceDescription-1.0.xsd">
<Device>
    <Connector moduleType="256" interface="Common.PCI" role="parent"
explicit="false">
        <Appearance>
            <ShowEditor>UserManagementPage</ShowEditor>
        </Appearance>
    </Connector>
</Device>
</DeviceDescription>
```

Additionally you have to integrate the following components in your runtime system:

1. CmpUserMgr
2. CmpUserDB (worked on a simple file)
  - This stores the complete user management in two files:  
UserMgmtDB.csv and UserMgmtRightsDB.csv
  - or
  - CmpUserEmbedded (for hardcoded Users, Groups and Rights)  
See this component as a template. You have to adapt this to your specific implementation.
3. CmpCryptMD5

After this work is done, you can use the online user management.

## 4 Portings

The runtime system is written completely portable. With the system interfaces, it can be ported very easy to an operating system or a special processor. The following portings are still available. Operating systems or processors/controllers that are not on the list can be provided on request.

**bold** and **green**: Standard product platforms of 3S

*italic*: Available platforms, release must be done custom specific

Operating System	Version	Platforms	Optional
<b>CODESYS Control Win V3</b>	<b>NT .. XP</b>	<b>X86</b>	<b>Target-Visu</b>
<b>CODESYS Control RTE V3</b>	<b>NT .. XP</b>	<b>X86</b>	<b>Target-Visu</b>
<b>Windows CE</b>	<b>4.0 .. 5.x</b>	<b>X86, ARM</b>	<b>Target-Visu</b>
<b>VxWorks</b>	<b>5.4 .. 6.x</b>	<b>X86, PPC</b>	<b>Target-Visu (planned)</b>
<b>Linux</b>	<b>Kernel 2.6 Standard Linux OSADL Linux</b>	<b>X86, ARM, PPC</b>	<b>Target-Visu (planned)</b>
<b>--</b>	<b>--</b>	<b>Infineon Tricore</b>	
<b>--</b>	<b>--</b>	<b>ARM</b>	
<b>PXROS</b>		<b>Tricore</b>	
<b>--</b>	<b>--</b>	<b>Blackfin</b>	
<b>--</b>	<b>--</b>	<b>Altera NIOS II</b>	

### 4.1 Windows Specific Information

The windows runtime runs on NT..XP systems. There are the following derivatives:

- CODESYS Control Win V3: soft real time behaviour
- CODESYS Control RTE V3: hard real time behaviour
- CODESYS Simulation: runtime system for the simulation feature of CODESYS (soft real time behaviour)
- CODESYS HMI: runtime system without I/O handling, for the purpose of running visualization applications on external devices

Also the Gateway consists of runtime system components:

- CODESYS Gateway V3: runtime system with gateway server functionality

### 4.1.1 Windows runtime services

The following windows runtime systems run as a windows service. They have all a command interface with the following calling options:

-i:	Service gets installed
-l:	Specification of the Windows user account (add the username), under which the service should be started (only together with option -i)
-p:	Password of the Windows user (only together with -i)
-u:	Service gets de-installed
-s: [<CODESYSControl.cfg>]:	Service gets started. A cfg-file can be specified optional.
-e:	Service gets terminated
-d:	Service gets started as application in the console mode (running in a DOS box)
-r:	Debugging of the service without console window
--startmode=auto:	Service gets started automatically by Windows
--startmode=demand:	Service can be started manually under Windows

#### 4.1.1.1 CODESYS Control Win V3 (soft real time)

Windows soft real time runtime system (product name: CODESYS Control Win V3) based on a Windows service that is running with REALTIME process priority in user mode. All hardware access routines are handled by a kernel driver called SysDrv3S.sys. This driver must be installed for each (PCI) card that is plugged into the PC and wants to be used by the runtime system.

#### 4.1.1.2 CODESYS Gateway Service V3

After a CODESYS standard installation, at system start the Gateway Server will be started automatically as a service. In addition and also automatically a separate application (GatewaySysTray) will be started, providing the gateway symbol in the system tray and – to be opened via this symbol – the gateway menu.

The gateway symbol indicates whether the gateway service is stopped (🛑) or running (🟢).

The gateway menu contains commands for explicit starting and stopping the gateway service as well as command „Exit Gateway Control“ for terminating the GatewaySysTray application (not however the gateway service !). The GatewaySysTray application also might be started via the Programs menu.

The start mode for the gateway can be set in the Windows service manager or via a call option (see above: call options).

**Note:** The Windows firewall should be deactivated for the gateway systray application.

#### 4.1.1.3 CODESYS Service Control V3

As authorization for starting/stopping services is strictly limited by Windows Vista, from CODESYS Version V3.2 SP1 Patch 2 onwards the services of CODESYS Control Win V3 and CODESYS Gateway Service V3 will be controlled by CODESYS Service Control V3.

#### 4.1.1.4 Brand labeling

The default names of the services controlled by CODESYS Service Control V3 may be modified by editing the following resources:

ServiceControl.exe:

ID 3: vendor name (used for shared memory name!)

ID 4: name of the CODESYS Control service

ID 5: name of the Gateway service

GatewaySysTray.exe:

ID 8: vendor name (used for shared memory name!)

ID 9: name of the Gateway service

CODESYSControlSysTray.exe:

ID 8: vendor name (used for shared memory name!)

ID 9: name of the CODESYSControl service

#### 4.1.2 CODESYS Control RTE V3 (hard realtime)

Windows hard real time runtime system (product name: CODESYS Control RTE V3) operating with an own scheduler that runs windows as an idle task. All other real time tasks are executed before Windows. A special time-slicing mechanism distributes the process consumption in a fix quota between the plc tasks and the windows operating system. This quota can be for example 70% plc and 30% windows.

#### 4.1.3 CODESYS integrated runtime systems

##### 4.1.3.1 CODESYS simulation

Windows soft real time runtime system integrated in CODESYS, used for the simulation feature.

##### 4.1.3.2 CODESYS HMI

Windows soft real time runtime system (product name: CODESYS HMI) which corresponds to the CODESYS Control Win V3 runtime system, however with deactivated I/O-handling functions. Serves to run a visualization application on a corresponding device (standard device: CODESYS HMI for Win32).

### 4.2 Windows CE Specific Information

Under Windows CE, we use the timer interrupt for the tick source of the runtime scheduler. All PLC tasks are created as operating system threads and are triggered by events from the runtime scheduler. The scheduling of the tasks is done by the Windows CE scheduler.

Win CE specific settings in the runtime system configuration file (\*.cfg):

- The **communication cycle interval** can be configured in order to control CPU load:

```
[ComponentManager]
WinCE.CommCycleInterval=5
```

The sleep duration after each comm cycle will be 5ms in this case, default: 1.

- Under Windows CE it is not possible to handle relative paths. So the default behaviour of file transfer and access under Windows CE has changed as from V3.4 SP4. From this version on, by default in the device dialog there is no access to paths beyond the "working directory". For activating the old behaviour (access to all files, e.g. also in the root directory), the following setting can be added to the runtime system configuration file:

```
[SysFile]
WinCE.AllowRootAccess=1
```

- An explicit folder can be defined, to which the **Target-Visualization files** (image files, textlists etc.) get redirected. By default, that is when the setting is missing, they get stored in a subfolder "visu" of the runtime system folder!

```
[SysFile]
WinCE.VisuFilePath=\Hard Disk\SpecialFolder
```

Because CE cannot handle relative paths, the complete path of the desired visu files folder must be specified in the setting!

**Example: Setting „WinCE.VisuFilePath=\Hard Disk\SpecialFolder\SpecialVisuFolder“ effects that folder „SpecialVisuFolder“ will be created as a subfolder of „\Hard Disk\SpecialFolder“ (in case it does not yet**



exist) and the visualization files get stored there. The path "Hard Disk\SpecialFolder" must already exist!

- If SRAM is used for retain handling with a specified address it might be necessary to have the following additional setting.

```
[SysShm]
WinCE.DisableMapPhysicalInVirtualAllocCopyEx=1
```

As the Sysdriver uses the VirtualCopy function for mapping hardware addresses to user program the behavior here depends on the CE image.

This setting has to be set on some devices where it is necessary to have the "PAGE\_PHYSICAL" flag in the call of "VirtualCopy" but not in the call of "VirtualAllocCopyEx". Otherwise it is possible that the address that the Sysdriver returns to the user program is not correct and retains do not work.

- Per default, the CE runtime system tries to change the system memory division to get more program memory and less storage memory. This behaviour can be suppressed by the setting

```
[SysGraphic]
WinCE.DontChangeMemorySettings=1
```

So other applications will get more program memory.

- Up to version 5, Windows CE processes have a 32 MB address space which may be too small for big applications. To overcome this restriction (only if the device has more than 32 MB physical RAM), the OEM can change the memory layout of the device description (example for a 32 MB area)

```
<ts:setting name="minimal-area-size" type="integer" access="visible">
<ts:value>0x2000000</ts:value> </ts:setting>
```

```
<ts:setting name="maximal-area-size" type="integer" access="visible">
<ts:value>0x2000000</ts:value> </ts:setting>
```

and the runtime configuration file

```
[SysMem]
WinCE.CE5BigAreaSize=0x2000000
```

As from V3.5 there is a new source file **SysDrv3SCECustom.cpp** for **customer specific adaptations**. A new project file SysDrv3SCECustom.vcproj for CE6 and SysDrv3SCE7Custom.vcproj for CE7 is available to build the dll by including the SysDrv3SCE.obj and the customer specific source file.

### 4.3 VxWorks Specific Information

Under VxWorks, we can use different timers as the tick source of the runtime scheduler:

- Auxiliary Clock (if available)
- System Clock
- High priority task

All PLC tasks are created as operating system threads and are triggered by events from the runtime scheduler. The scheduling of the tasks is done by the VxWorks scheduler.

Multicore CPUs under VxWorks 6.6 and newer are generally supported. However, currently the runtime has to be bound exclusively to one core. Load balancing of PLC tasks is not supported.

### 4.3.1 Distributed clocks

Distributed Clocks are used by some field busses (like EtherCAT) to synchronize the cycle start times for all participants on the bus. For this feature, we need some special support from the operating system, because we need to modify the cycle times of the tasks with a fine granularity of a few microseconds.

VxWorks doesn't support this, because such a feature would need some kind of one-shot timer and a non-cyclic scheduler. Because VxWorks doesn't support both, there is an own timeout scheduler implemented in the CODESYS Runtime. To simulate a one-shot timer, this timeout scheduler uses the cyclic timers, which are provided by VxWorks.

These are:

- **The System Timer:**  
This timer is used by VxWorks for its own cyclic scheduler. Other tasks (from VxWorks or the User) may run into trouble when using this timer for distributed clocks. Because they may assume that they have a timer with a fixed period.
- **The Auxiliary Timer:**  
This timer is an additional timer, which is provided by VxWorks on some boards. It's based on a completely independent timer source and therefore doesn't interfere with other subsystems, when using it in one-shot mode.

The Distributed Clocks feature can be enabled with the following setting in the runtime configuration file:

```
[SysTimer]
VxWorks.TimerMode=OneShot
```

**Note** that if using this option, the clock which is assigned to the CODESYS runtime by setting "VxWorks.TimerSource" will no longer tick periodically. This might have influence on other applications running on the same platform.

#### 4.3.1.1 Timer sources

When using distributed clocks, the selection of the timer source for the runtime is even more critical than usually.

Using the system timer as a timing-source is in most cases very accurate because the system timer can be programmed very accurate on most systems. But if other programs are running on the same machine, this may lead to an undefined behaviour, because the system tick will occur randomly for the second application.

Auxiliary timers need to be able to be programmed linearly with any cycle time. For example on an x86, this is true for an Auxiliary APIC timer but not for an Auxiliary RTC timer, because the RTC can only be programmed with frequencies exponentially to the base of two.

If such an Auxiliary Timer is available, it might be the better choice, because we don't need to modify the system tick and we don't disturb other simultaneously running applications.

#### 4.3.1.2 Performance & Accuracy

Because we need to reprogram the timer on every tick, the overall system performance will obviously decrease slightly. Also the number of interrupts in our system will be higher, because the number of timer interrupts increases. Beside our standard 1ms interrupts of the Runtime Scheduler, we get additional interrupts for the new high-precision IEC tasks.

To overcome this problem a little bit, it is recommendable to increase the tick period of the Runtime Scheduler as much as possible. Because all periodic IEC tasks are now scheduled over the high-precision timer, this will only have effects on tasks which are triggered by event variables. So increasing this to 10ms or even 100ms might be possible on most systems. This can be done with the following setting in the Runtime Configuration:

```
[CmpSchedule]
```

SchedulerInterval=100000

When using one-shot timers, the tasks are still scheduled by the VxWorks Scheduler, but are triggered from different „high-level schedulers“:

- **Cyclic Tasks:** Triggered by the high-precision timeout scheduler.
- **Event Tasks:** Triggered by the Runtime Scheduler. The Variable is checked in the Interval given in the configuration file.
- **Freewheeling:** Those tasks are running all the time, and are interrupting themselves after every cycle for a small amount of time to give control to the communication subsystem. The tasks are woken up by the high-precision timeout scheduler.
- **External Event Tasks:** Triggered by an external Event. This can be generated in every context and therefore depends on the context where the event is generated.

#### 4.3.1.3 Jitter

Accuracy for all periodic tasks is obviously much higher now, but we might notice some jitter when the timeouts of two tasks are overlapping. In any case the task with the highest priority will be served first, but this can lead to situations where a low-priority task is started and slightly after this a high-priority task get's ready.

Without the Distributed Clocks, the execution of the tasks was synchronized by the periodic system tick. Therefore all tasks were executed serialized, one after another. This means, that the task cycle time was optimal and shorter before. Now low-priority tasks can be interrupted by higher priority tasks, and therefore we will notice a higher jitter in those low-priority tasks than before.

#### 4.3.1.4 Static Memory Areas

In VxWorks, all memory areas from the application are by default allocated dynamically on the heap. If this is not intended for some reasons, it is possible to specify one or more static memory areas in the runtime configuration file.

The user is able to move any area to a fix address (Code, data, input, output, retain,...), but it is especially usefull to define his own areas.

For this purpose, we have the reserved upper 4 bits of the area flags which can be used by the user to move some data to his own special areas. He can do this by specifying the following attribute:

```
{attribute 'location' := '16#8000'}
```

The 16#8000 is the area flag and should be the same as it is specified in the configuration file. e.g.:

```
[SysMem]
VxWorks.Area.0.Flags=0x8000
VxWorks.Area.0.Address=0x60540010
VxWorks.Area.0.Size=0x10000
```

Note, that the areas are always parsed from top down, and that the first area that matches the requested area flags is used. Therefore, you should not define the first area with the flags 0xFEFF or similar, but 0x0EFF instead.

### 4.3.2 Global object pools

If you have two or more instances of the CODESYS Runtime, running on the same PLC, you might want to share Events and SharedMemories. Especially this is necessary if you want to use our CmpBlkDrvShm to do real communication and routing on a CODESYS level between the Instances. But even if you are running only one instance of CODESYS, but want to communicate with some of your own external firmware components, which are not running inside of our CODESYS Runtime, you might have the need to share some memory or send events between each other.

For these scenarios, CODESYS on VxWorks provides a global object pool for:

- events
- semaphores
- Shared memories

To share the global symbol table across application image boundaries, the runtime emits a new symbol at first startup, called "g\_PlObjTab". This symbol is created only on the first instantiation of the runtime. All memory within this table does not belong to the process image. It is allocated dynamically on the heap.

Between different instances of the CODESYS runtime, those objects are shared automatically, based on their name. To use the objects independently from the CODESYS runtime, you can read the data from the table manually in your part of the firmware. The shared object table has the following structure:

```
#define OBJTAB_VERSION 0x00000001

typedef enum objTabType_e {
    e_shm,
    e_sem,
    e_evt
} objTabType_t;

typedef struct shmEntry_s {
    unsigned long ulAddress;
    unsigned long ulSize;
} shmEntry_t;

typedef struct semEntry_s {
    SEM_ID hSem;
} semEntry_t;

typedef struct evtEntry_s {
    SEM_ID hEvt;
} evtEntry_t;

/* base type, which "contains" the above */
typedef struct objEntry_s {
    objTabType_t tType;
    char *pszName;
    union {
        shmEntry_t tShmEntry;
        semEntry_t tSemEntry;
        evtEntry_t tEvtEntry;
    } u;
} objEntry_t;

typedef struct objTab_s {
    unsigned long ulVersion;
    SEM_ID hSem;
    objEntry_t *ptEntries;
    unsigned long ulSize;
    unsigned long ulRefCnt;
} objTab_t;
```

#### 4.4 Linux specific information

Under Linux we base on the kernel 2.6 with the pre-emption patches from Ingo Molnar and Thomas Gleixner to get realtime behaviour. But the integration is transparent, so a naked vanilla standard kernel could be used with the runtime system too.

The runtime system runs under Linux in the user mode. Physical and hardware access is done directly from the user mode or is handled via a kernel driver.

## 5 Communication

### 5.1 Overview

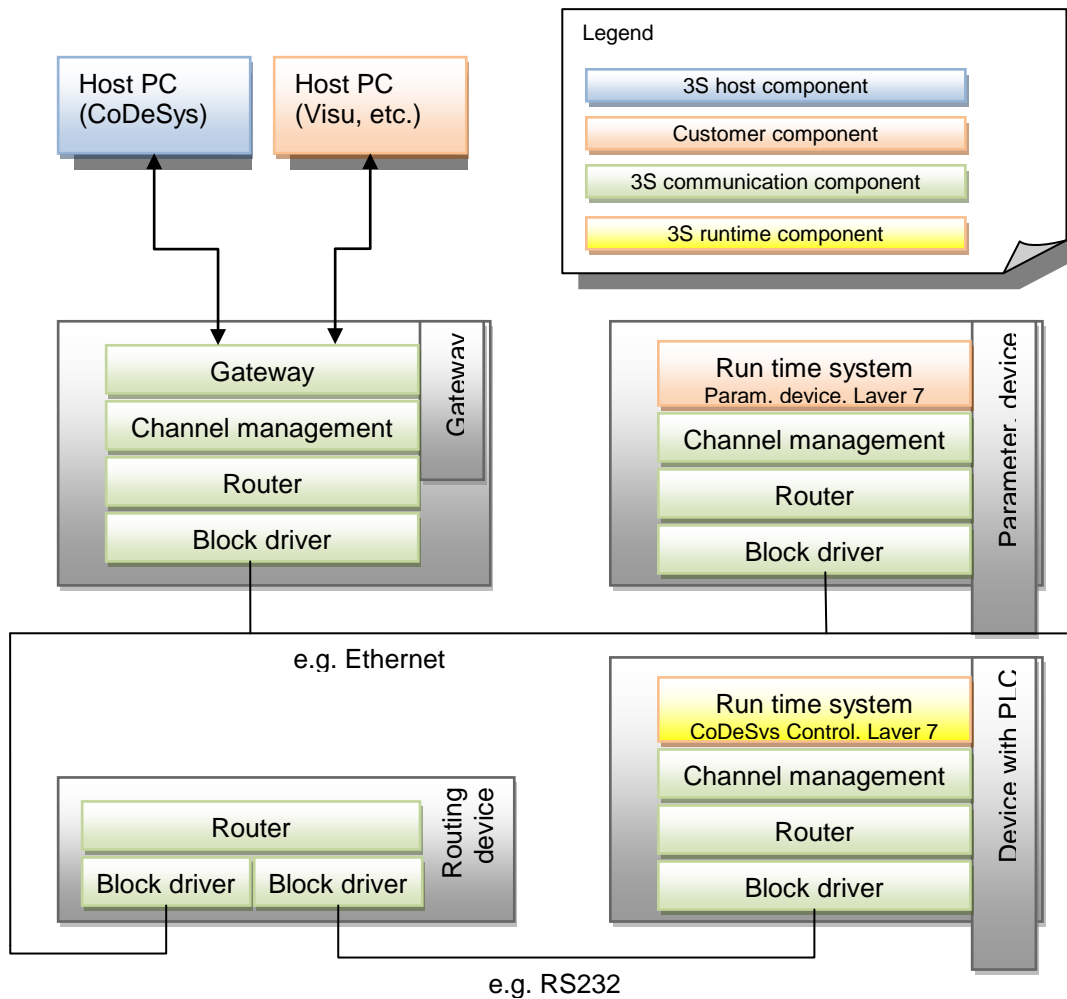
In the communication system for CODESYS V3 a distinction is made between four main communication device types. Simple **nodes** form the core of the communication system. They form a logic tree referred to as the **control network** and implement the CODESYS communication stack. A **gateway** is itself a node within this tree and in addition offers an access point to this network from outside. **Clients**, on the other hand, are not part of the network. Instead they use a gateway to communicate with nodes in the network. **Proprietary devices** generally have no CODESYS runtime system and use any non-CODESYS communication protocol.

The communication is based on existing network protocols such as UDP, CAN, serial etc. They can be located at different levels. The communication protocol within the network is defined in layers, based on the ISO/OSI Reference Model. The following table provides a comparison of the ISO/OSI layers and the components specified for CODESYS:

Layer	ISO/OSI	CODESYS	Description
1	Physical layer	-	Not specified (defined through lower-level communication systems).
2	Data link layer	Block driver	Mapping of the communication to lower-level network protocols. Ensuring data consistency (CRC). Sending of <b>blocks</b> with defined maximum length.
3	Network layer	Router	Sending of packets across several stations, "routing"/multiplexing of packets from several senders/receivers on a single "line"
4	Transport layer	Channel management	Secure, connection-oriented end-to-end communication with packet repetition, timeout monitoring, ...
5,6	Session or presentation layer	-	Not specified
7	Application layer	Application services	Specification of the data exchange format, distribution of requests to service handlers

**Table 1 - Comparison of ISO/OSI and CODESYS model**

The following diagram illustrates the structure of the communication stacks and the communication between different device types.



### 5.1.1 Usage scenarios

In order to be able to work conveniently (Plug&Play), securely and reliably within different project phases, a distinction is made between three operating modes, with associated network behaviour.

- **Commissioning**  
During commissioning the addresses of all nodes are determined dynamically. Manual intervention is reduced to a minimum. During this phase the node addresses may change fundamentally.
- **Normal operation**  
During the transition to normal operation all node addresses are „frozen“, i.e. each node stores its address permanently and automatically reloads it after a restart. Adding/removing nodes in this state has no effect on the addresses of the other nodes. Addresses are only changed if requested explicitly.
- **Maintenance**  
If a programming or service PC is connected during operation for maintenance of individual nodes, this PC will automatically be assigned an address, although it will not be able to change the addresses of the other nodes.  
In the delivery state new nodes are set to commissioning, i.e. the node address is automatically assigned according to the network topology, without affecting the addresses of other nodes in the event of misconfiguration. Once the new node has been implemented correctly, it is set to normal operation, which means its address is fixed.

## 5.2 General

Little-endian is specified as the byte order for all fields in all layers, in order to ensure compatibility between different systems. In individual cases the byte order of the target device may be used (→application layer). This is explicitly mentioned where appropriate. In all other cases applies little-endian byte order **always** applies.

## 5.3 Communication Layers

The components of the communication system are based on the ISO/OSI Layer Model. There are 4 components:

- **Medium and datagram layer** (Layer 2, „block driver“)
 

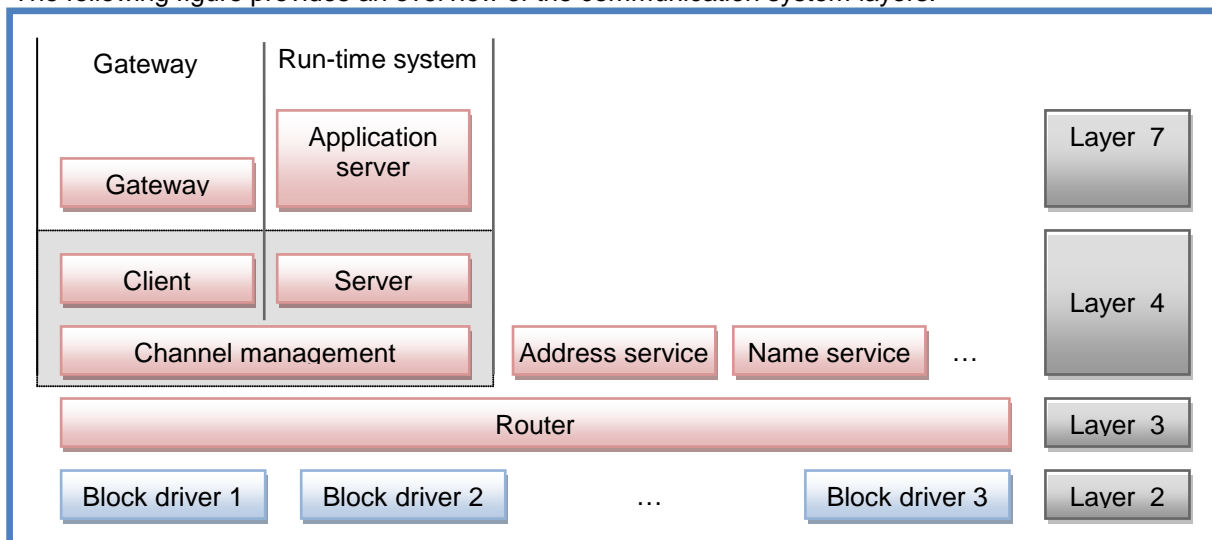
This layer handles the transfer of frames between nodes. Media access is located in this layer. In addition, this layer prevents delivery of faulty frames (bit errors). Frame repetition etc. can be implemented as an option.
- **Network layer** (Layer 3, “router“)
 

This layer handles routing and addressing. There is no error protection in this layer.
- **Messages/protocol layer** (Layer 4, “channel management“)
 

This layer defines a protocol for safe transmission of asynchronous services of any length. This service is asymmetric. It consists of a common part and an associated client or server component.
- **Application layer** (Layer 7)
 

This layer defines services based on Layer 4.

The following figure provides an overview of the communication system layers:



### 5.3.1 Block driver (Layer 2)

Block drivers map the 3S communication model onto the lower-level network (e.g. Ethernet, CAN, serial). Each network type has its own block driver. The block drivers always send blocks with a specified maximum size. The maximum size (512 bytes) is specified by the block driver. If necessary the packet has to be subdivided into suitable pieces and reassembled at the receiving end. The block driver must ensure that only correct blocks, i.e. blocks without transmission errors, are transferred to the higher-level layer (router). If necessary this must be ensured with suitable measures, e.g. CRC. Faulty blocks may be discarded, since layer 2 is not connection-oriented. Block repetition can optionally be implemented for connections that are particularly susceptible to failure, in order to minimize the fault rate in the overall system.

A block driver can manage several physical (e.g. 2 Ethernet cards) or logical connections simultaneously, although each connection has to be registered separately with the higher-level router.

Block drivers are configured on system startup and are available from that time, provided the lower-level network is available. They can only handle primitive **send** and **receive**, i.e. no explicit connection setup or termination. Only one network address is specified as target (see 5.4.4.1), without additional configuration (such as baud rate or start/stop bits).

Communication between two block drivers is on an equal basis, i.e. each of two devices can initiate the communication at any time. If master/slave operation is required for a special communication medium, it must be mapped accordingly. Ideally the connection between the two devices should be permanent.

### 5.3.2 Router (Layer 3)

The router is based on Layer 2 (block driver) and is responsible for sending blocks via any number of intermediate levels or lower-level network types. Since block drivers already abstract from the actual network type, routers only see logical CODESYS nodes.

Each node supporting the CODESYS communication system - and therefore any runtime system - implements the router and can therefore pass on packets itself.

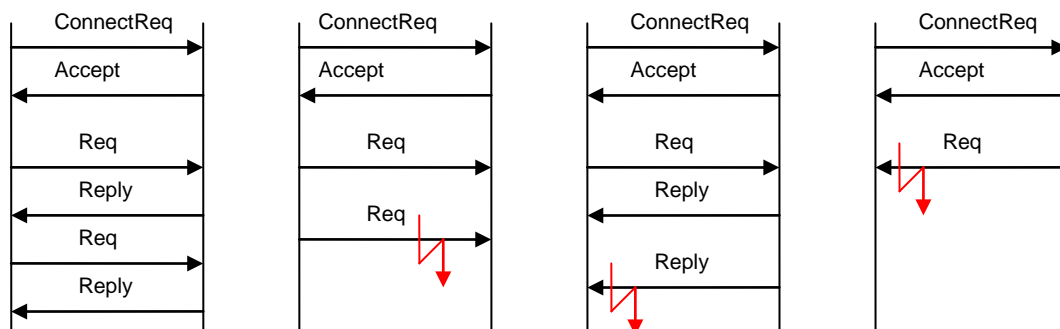
With regard to higher levels, the router offers the option of sending packets to any receiver within a control network or receiving packets for certain services (corresponding to ports). Channel management (Layer 4), which offers a secure connection, is one of the services based on the router. The service number is 64. Further services include automatic address assignment, name resolution, or network variable exchange.

### 5.3.3 Channel management (Layer 4)

Channel management offers a connection-oriented, secure end-to-end transmission between any two nodes in the control network. Packets of any size can be sent. The maximum size only depends on the resources available in the two end nodes. To this end the size of the communication buffer is determined when the connection is set up. Without optimization, two buffers are required on both sides (send buffer and receive buffer). In general a static buffer for a fixed number of maximum simultaneous connections (**channels**) is provided in runtime systems.

Layer 4 deals with subdivision of potentially large packets into suitable blocks for the lower-level layers, packet repetition, and correct reassembly of the packets on the receiver side. In addition, the status of the connection is continuously monitored via keep-alive packets and (dynamic) timeouts, so that any interruption of the connection can already be detected at this level, irrespective of the size and type of the service.

Channel management uses a request reply procedure to avoid requests having to be discarded due to insufficient resources. The node that has initiated the connection is the **master** and therefore the node that may send requests. A new request may only be sent once a reply for the previous request has been received. Similarly, the remote terminal (the **slave**) must send exactly one reply for each request it receives.



This service is basically divided asymmetrically into a client and a server component using a common base component. The client component is always used on the gateway, the server component on a



target device. However, there is no reason why both components should not be implemented on the same node. This node can then execute server and client services simultaneously. This is of interest if two runtime systems are to communicate directly with each other, for example.

### 5.3.4 Application services (Layer 7)

Application services use a format with binary tagging in order to achieve upward and downward compatibility of the services. Each node is identified with a unique number (**tag**) and its length. Together, the tag and the length form the **header** of a node. The node either contains data with a fixed structure or sub nodes, but never both at the same time.

Each Layer 7 packet starts with a general header that defines, among other parameters, the service to be executed via **service group** and **service number**. The service group corresponds to a communication port. A service handler can thus register on a certain service group and will then receive all requests for this service.

If a service requires more space than is offered by the communication buffer for the associated connection, the request has to be split into several service-specific requests (each of which has to be acknowledged with an associated reply).

Each level 7 service handler must be registered at the generic level 7 server (CmpSrv). This can be done in a C component with the following interface function:

```
CAL_ServerRegisterServiceHandler(<ServiceGroup>, <Handler>);
```

To handle level 7 services in IEC, there is a different way. For this, the component must register a callback handler at the event manager. So CmpEventManager.library must be added. Additionally the CmpSrv.library must be added to see, which event parameters are transmitted.

Example:

Declaration:

```
Result : RTS_IEC_RESULT;
hEvent : RTS_IEC_EVENT;
```

Implementation (first cycle in a program or FB\_Init method in a function block):

```
hEvent := EventOpen(EVTPARAMID_CmpSrv, CMPID_CmpSrv, Result);
Result := EventRegisterCallbackFunction(hEvent, ADR(<Handler>));
```

After that, all services with the specified service group are generating an event, at which the IEC event handler is called, e.g.:

Declaration:

```
FUNCTION EventCallbackFunction : UDINT
VAR_INPUT
    pEventParam : POINTER TO EventParam;
END_VAR
VAR
    pServiceParam : POINTER TO EVTPARAM_CmpSrv;
END_VAR
```

Implementation:

```
pServiceParam := pEventParam^.pParameter;
```

With pServiceParam, all parameters of a standard service handler are accessible!

The feature of the event manager provides to register a callback method of a function block too. So level 7 services can be handled in an encapsulated function block.

## 5.4 Network Topology and Addressing

The aim is to create a system that largely configures itself (address assignment), transparently supports any communication media, and can route packets between different networks. The routing mechanism should be simple enough such that any node in the network, i.e. even nodes with low resources, can reroute packets. In particular this means that large routing tables, complex calculations, or requests at runtime should be avoided.

### 5.4.1 Topology

A control network should generally be configured hierarchically, i.e. each node has one parent node and any number of children. Cycles are not permitted, i.e. a control network has a tree structure.

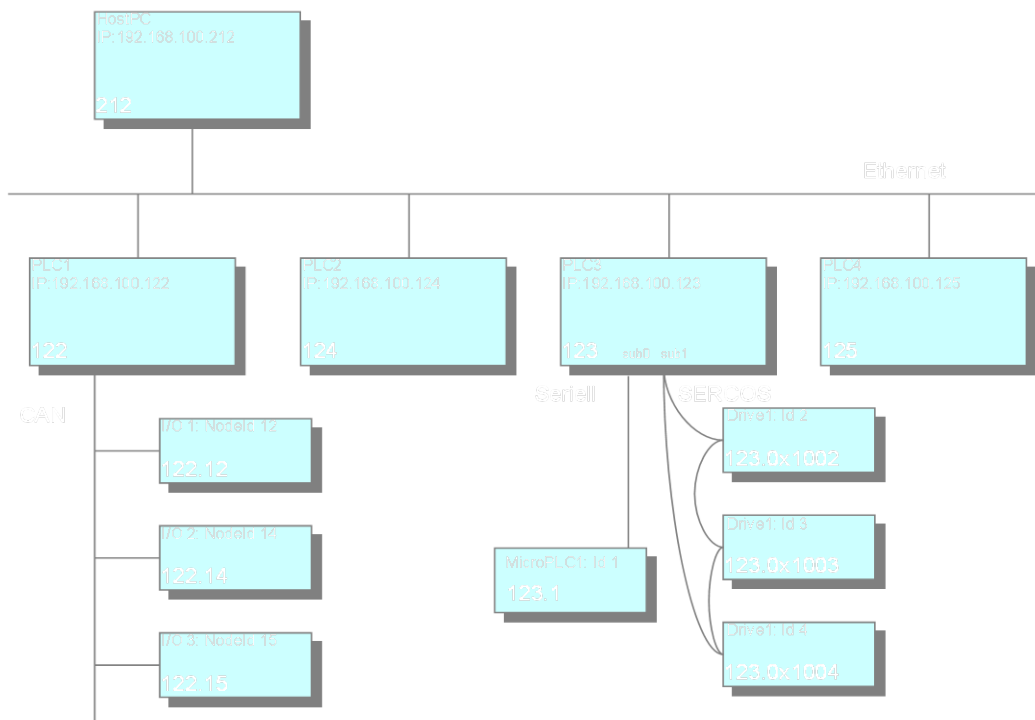
Parent-child relationships arise from the specification of network segments. A network segment corresponds to a local Ethernet or a serial point-to-point connection, for example. A distinction is made between the main network (**main net**) and the sub-networks (**subnet**). Each node may have several main nets as well as each main net may have several subnets. Each main net acts as a parent for the attached subnets.

If the same network segment was simultaneously defined as a subnet of several nodes, the network would have several parents. This is an invalid configuration, since each network segment can only have one parent.

A node without parent is referred to as **top-level node**.

### 5.4.2 Addressing and routing

Addresses map the topology of a control network to unique addresses.



Node addresses are structured hierarchically. Each network connection is allocated a **local address** by the relevant block driver through which the node is unambiguously identified in the respective network. The node address is now formed by first appending the index of the subnet (in the parent) in which the node is located then subsequently the local address of the main network connection to the

address of the parent. A node without main network is a **top-level** node with address 0. A node with a main network that does not contain a parent is assigned its local address in the main network.

Due to the address structure the routing algorithm can be kept relatively lean. No routing tables are required. The only requirement is local information about the own address and the address of the parent node:

- Target address identical to own address? → Current node is receiver
- Target address starts with own address? → Packet is intended for a child or descendant of the node. Forward to associated child node.
- Otherwise: Packet is not a descendant of the current node. → Forward to own parent.

Relative addresses are a special feature. They do not contain the node number of the receiver node, but directly describe the path from the sender to the receiver. The principle is similar to a relative path in the file system: The address is comprised of the number of steps the packet has to move "up", i.e. to the next respective parent, and the subsequent path down to the target node.

The advantage of a relative address is that two nodes within the same sub tree can continue to communicate if the whole subtree is moved to another position within the overall control network. While the absolute node addresses change during such a move, the relative addresses are preserved.

#### 5.4.2.1 Parallel routing

Parallel routing represents a direct communication mechanism between two router instances on the same node. It is used when there are at least two networks on the same node that should not end up in a main net - subnet relationship (E.g. a PC with two network cards on independent Ethernet segments).

The "ReceiverAdr" (see chapter 5.5) of an incoming packet is screened by the receiving router instance for a special marker identifying a parallel router on the same node. If this marker exists, both "ReceiverAdr" and "SenderAdr" are updated by the receiving router:

- The existing marker will be removed from the incoming "ReceiverAdr".
- A new marker will be added to the "SenderAdr" for identifying the current router.

If the new "ReceiverAdr" is identical to the address of the parallel router, the acting node is the correct recipient and the packet will be handled locally. If not, the packet will directly be sent to the other network segment to be handled there by a different node.

#### 5.4.3 Address determination

In order to be able form its own address, each node must know the address of its parent node or detect that it is a top-level node. To this end it sends an address determination message as a broadcast to its main network during bootup. The parent node responds with an address notification. The node then passes on the modified address to its subnet. Until it receives a response to the address determination message the node considers itself to be a top-level node, although it will continue to try and detect a parent node at suitable intervals.

Address determination can be executed at bootup or when requested by the programming PC.

Once the address of a node is frozen, no further address determination is required. An address notification by the parent node with an address that differs from the stored address should be regarded as a fault.

#### 5.4.4 Address structure

##### 5.4.4.1 Network addresses

Network addresses represent a mapping of the addresses of a network type (e.g. IP addresses) to a logical address within a control network. This mapping is handled by the respective block driver. Within an Ethernet with Class C IP addresses the first 3 bytes of the IP address are identical for all network devices. The last 8 bits of the IP address therefore suffice as network address, since they enable unambiguous mapping between the two addresses at the block driver.

A node has separate network addresses for each network connection. Different network connections may have the same network address, since this address only has to be unique locally for each network connection.

**Terminology:** In general, the network address of node without statement of the network connection refers to the network address in the main network.

The length of a network address is specified in bits and can be chosen by the block driver as required. Within a network segment the same length must be used for all nodes. A network address is represented as an array of bytes with the following coding:

- Length of the network address:  $n$  bits
- Required bytes:  $b = (n + 7) \text{ DIV } 8$
- The  $(n \text{ MOD } 8)$  lowest-order bits of the first byte and all remaining  $(n \text{ DIV } 8)$  bytes are used for the network address.

Example for network address coding:

- Length: 11 bits
- Address: 111 1000 1100

Byte	0								1							
Bit	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
									1	0	0	0	1	1	0	0

Reserved (0)

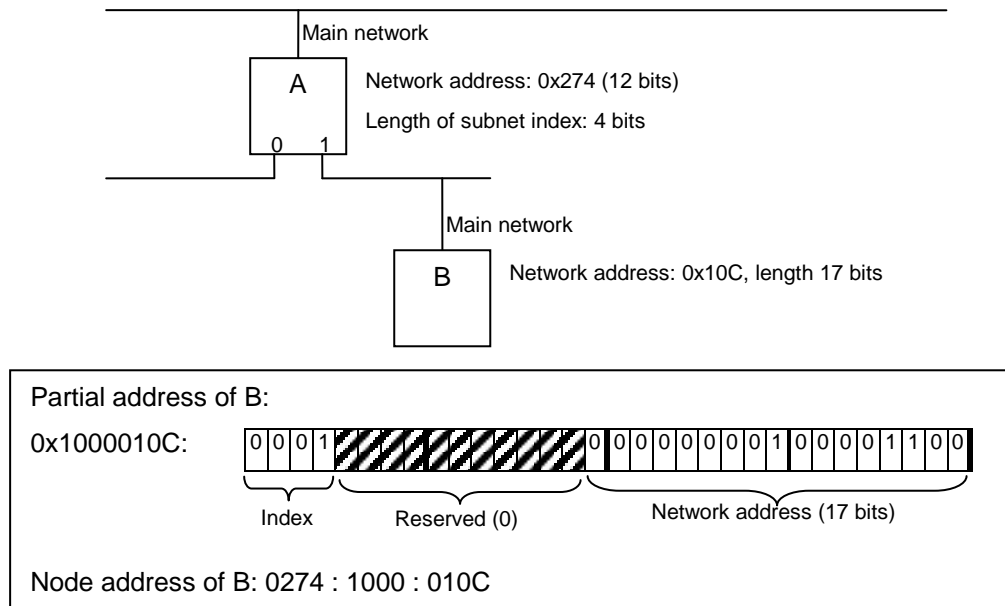
#### 5.4.4.2 Node addresses

The node address indicates the absolute address of a node within a control network and is therefore unique in the whole “tree”. It consists of up to 15 **address components**, each consisting of two bytes. The lower a node is located within the network hierarchy, the longer its address.

The node address is comprised of the **partial addresses** of all predecessors of the node and the node itself. Each partial address consists of one or several address components. The length is therefore always a multiple of two. The partial address of a node is formed from the network address of the node in its main network and the **subnet index** of the main network in the parent node. The bits required for the subnet index are determined by the router of the parent node. Filler bits are inserted between the subnet index and the network address in order to ensure that the length of the partial address is a multiple of 2 bytes.

Special cases:

- **Node has no main network:** This means there is no subnet index nor a network address in the main network. In this case the address is set to 0x0000.
- **Node with main network but without parent:** In this case a subnet index with 0 bit length is assumed. The partial address corresponds to the network address, supplemented with filler bits if required.



The node address representation is always hexadecimal. The individual address components (two bytes in each case) are separated by a “:” (colon). The bytes within a component appear sequentially without separator (see example above). Since this represents a byte array and not a 16-bit value, the components are **not** displayed in little-endian format. For manually entered addresses missing digits in an address component are filled with leading zeros from the left: “274” = “0274”. To improve readability the output should always **include** the leading zeros.

#### 5.4.4.3 Absolute and relative addresses

Communication between two nodes can be based on relative or absolute addresses. Absolute addresses are identical to node addresses. Relative addresses specify a path from the sender to the receiver. They consist of an address offset and a descending path to the receiver.

The (negative) address offset describes the number of address components that a packet has to be handed upwards in the tree before it can be handed down again from a common parent. Since nodes can use partial addresses consisting of more than one address component, the number of parent nodes to be passed is always = the address offset. This means that the demarcation between parent nodes is no longer unambiguous, which is why the common initial part of the addresses of the communication partners is used as parent address. Each address component is counted as an upward step, irrespective of the actual parent nodes. Any errors introduced by these assumptions can be detected by the respective parent node and must be handled correctly by the node.

On arrival at the common parent the relative path (an array of address components) is then followed downwards in the normal way.

Formal: The node address of the receiver is formed by removing the last AddressOffset components from the node address of the sender and appending the relative path to the remaining address.

Example:

Node A: a.bc.d.ef.g

Node B: a.bc.i.j.kl.m

→ Address of the lowest common parent: a.bc

→ relative address from A to B: -4/i.j.kl.m

To ensure that the routing works correctly, the relative address must be adjusted with each pass through an intermediate node. It is sufficient to adjust the address offset. This is **always done by the parent node**: If a node receives a packet from one of its subnets, the address offset is increased by the length of the address component of this subnet. If the new address offset is < 0 the packet must be forwarded to the parent node. If the address offset >= 0 the packet must be forwarded to the child node whose local address is located at the position described by the address offset within the relative address. First, the address offset must be increased by the length of the local address of the child node to ensure that the node sees a correct address.

A special case is created by a situation in which the above described error occurs when the common parent is determined. In this case the address offset at the “real” common parent is negative, but the magnitude is greater than the length of the partial address of the subnet from which the packet originates. The node must detect this case, calculate the local address of the next child node based on the address of the previous node and the length difference, and adapt the address offset such that the next node sees a correct relative address. Here too the address components themselves remain unchanged, only the address offset changes.

#### 5.4.4.4 Broadcast addresses

There are two types of broadcast - global and local. A **global broadcast** is sent to all nodes within a control network. The empty node address (length 0) is reserved for this purpose.

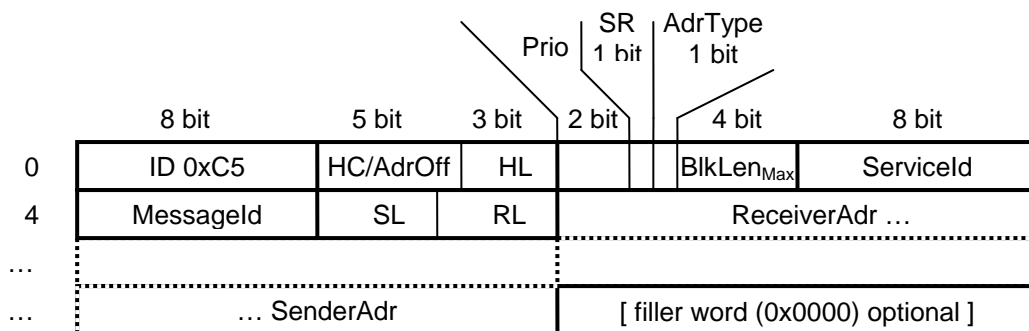
**Local broadcasts** are sent to all devices of a network segment. For this purpose, all bits of the network address are set to 1. This is possible both in relative and in absolute addresses.

A block driver must be able to handle both broadcast addresses, i.e. empty network addresses and network addresses with all bits set to 1 must be interpreted and sent as broadcast.

## 5.5 Router Communication

A router packet consists of a **header** and the actual message (**data**). The length of the header is variable, since node addresses have different lengths. Irrespective of this the data always start at a 4-byte limit within the packet. Correct alignment is therefore always ensured, provided the data are structured accordingly.

The following diagram illustrates the header structure:



- HC/AdrOff: Depending on the hop count addressing type hop count (direct addresses) or the address offset (relative addresses)
- HL: Length of the static header in words. Corresponds to the offset of ReceiverAdr. Currently 3, i.e. 6 bytes.
- AdrType: 0 for direct addresses, 1 for relative
- Prio: Packet priority: 0 low, 1 normal, 2 high, 3 emergency. Packets with higher priority are given preference by routers.
- SR: SignalRouter. Is set by a router to indicate an error (e.g. block size).
- BlkLen<sub>Max</sub>: Field for maximum block length. The maximum block length is calculated as  $(BlkLen_{Max} + 1) * 32$ . (→ 0: 32 bytes, 1: 64 bytes, 2: 96 bytes, ..., 15: 512 bytes)
- ServiceId: A kind of port number for the called service.
- MessageId: Is set by the sender. Is used for identification of a message, detection of duplicates for unsecured services, etc.
- SL: Length of the sender address in words
- RL: Length of the receiver address in words.

- Filler word: 0. Is inserted if SenderAdr does not end at a 4 byte-limit, in order to align the user data with a 4-byte limit.

→ Minimum packet length (both addresses one word, no content): 12 bytes

### 5.5.1 Hop count

This field is initialized to 31 (the maximum number of nodes that a packet has to pass through when the network reaches the maximum depth of 15). Each node that forwards a packet reduces the value of this field by 1. Once the field value has reached 0 the packet can no longer be forwarded. The field also has to be decremented for broadcasts that are forwarded to subnets.

This mechanism offers reliable protection from broadcast floods and infinite packet loops, both for cycles and in the event of misconfiguration.

### 5.5.2 Router signaling

This header flag identifies a message as a router message. If a router is unable to deliver a message due to particular circumstances, it generates a new message that is sent back to the sender of the original message, which sets the router flag. The service ID is taken from the undelivered packet, while bit 0 is toggled. A request thus becomes a response and vice versa (see 3.6.2). With separate client/server implementation of a service this ensures that the sending component receives the message back.

A router message contains the complete header of the discarded packet. It also contains an ErrorId of the router and further, error-dependent information. The original sender then forwards the packet to the corresponding service as usual. Via a "MessageId" header field, which is generated by the service when a message is sent, the returned message can be assigned to a current request. The router can first analyze the error itself and adapt internal information as required (e.g. the maximum block length for a particular transmission route).

If a message with active router flag cannot be delivered it is discarded without generating a further router message, in order to avoid endless message floods.

Possible router messages:

BLOCKLEN\_OVERRUN: Maximum block length for a transmission route exceeded.

INVALID\_SUBNET: The subnet for the next node does not exist.

INVALID\_NODE: Node does not exist (not detectable for all block drivers).

CONGESTION: Next transmission route blocked, but return channel is available.

INVALID\_SERVICE: The specified ServiceId is not used on the target computer.

### 5.5.3 Variable maximum block length for a transmission route

Not all block drivers can send up to 512 bytes (the maximum size of a router packet) in once piece. With many transmission protocols 512 bytes no longer fit into 1, 2, .. frames, which means that for each packet an additional, almost empty frame would have to be generated and sent, resulting in useless overhead. This problem may arise for any fixed block length. Shorter block lengths may therefore be selected, depending on the transmission route (in steps of 32 bytes).

A field in the Level 3 header is reserved for the maximum block length of the transmission route. In the absence of further information this field is initialized with 512 bytes. Each router checks whether the next transmission route has a smaller block length (information from block driver) and updates the field as required. The header of the receiver then contains the smallest maximum block length for the whole communication path. If a packet cannot be forwarded due to excessive block length, it is discarded and the problem is reported to the sender via the signaling mechanism (→ 5.5.2).

Router components should remember the values determined in this way, at least for the last communication partners. Since the first packets between two end points are generally relatively small packets (e.g. connection setup, network scan, ...) no packets should have to be discarded due to this mechanism during normal operation.

### 5.5.4 Multiple router instances

Routers are Instantiable, i.e. for each device several router instances can exist simultaneously. Each router exclusively manages part of the block drivers, i.e. there is no joint block driver access. Each router instance has its own main network configuration, and therefore has its own address and forms its own (partial) network. There is no provision for direct routing between router instances. Consequently, a router instance always has to be specified in order to be able to send network packets. Router instances are given a name for identification and are addressed via an Id.

In order to keep the configuration simple, a default router is available that can be used for standard communication. Further routers are available for redundant network connections, service channels etc.

The default configuration of a router comprises a main network connection but no subnets. Subnets have to be configured explicitly in order to avoid problems.

## 5.6 Layer 3 Services

In addition to the Layer 4 handler various other services are based on Layer 3 that are addressed via protocol numbers (**ServiceId** field in the router header). Each of these services uses two protocol numbers - one for the requesting the service, the second one for responses. The two numbers differ in the lowest-order bit: for requests this bit is set, for responses the bit is deleted.

When defining such a service certain specifications should be followed. A service packet always consists of a fixed header and user data. In order to ensure expandability, the first two bytes specify the length of the header (= user data offset). Extensions must be designed such that older services can handle more recent packets and vice versa.

In order to enable communication between different systems, the byte order is always little-endian. In addition, all fields are based on natural alignment. Structures, particularly user data, always start at 4-byte boundaries (filler byte must used if necessary).

The following service groups are defined:

Req/Reply	Associated service
1 / 2	Address service
3 / 4	Name service (name resolution, network scan, ...)
5 / 6	Network variables
63 / 64	Channel management (Layer 4)

## 5.7 Gateway and client

As a minimum, a gateway consists of the standard communication stack and an additional gateway component. This makes the gateway into a network node. In principle, each node can be extended to a gateway by integrating this component. A runtime system can also take on the role of a gateway. Via a special gateway protocol this component provides external clients with access to nodes within the network, in which case communication is via the gateway instead of the client. The gateway has no other special role within the network. A network can therefore have any number of gateways, and the gateway can be positioned anywhere in the network. Two positions are particularly suitable in order to avoid unnecessary bandwidth problems:

- As top-level node: In general this is where the network with the highest bandwidth is located. In addition, access to all branches involves the same level of complexity.
- Near the target system: This solution is particularly suitable if the gateway is mainly intended for a particular subnet that may be connected to the rest of the network via low (available) bandwidth.

If more clients can communicate with a target system than the number of channels the system can make available on Layer 4, the gateway can multiplex the requests for different clients on one channel. To this end the gateway opens a single channel for all clients and sends requests to the clients in turn, waits for the reply, sends the next request, etc. While this technique slows down communication of the



individual clients to some extent, for sporadic requests it uses significantly less resources in the target system.

Common clients are tools for programming (e.g. CODESYS itself), visualization, remote diagnostics, etc. The main feature of a client is that it is generally only connected to the network on a temporary basis. In addition, a client is always the active communication partner, i.e. a node will usually not establish communication with, but only respond to requests from a client.

The communication between client and gateway is based on **communication drivers**. In contrast to communication between block drivers this communication is connection-oriented. In addition there is no (direct) limitation of the maximum packet sizes. However, since the main part of the communication is forwarding of Layer 7 packets, the packet size is effectively limited by the negotiated size of the communication buffer plus administrative data.

Gateway requests include the following components:

- Network scan: From the scan the gateway generates a request for the name service and supplies all the nodes available in the network as response.
- Address services: Including services for disabling automatic address assignment ("freezing" of addresses), initiation of address assignment, etc.
- Layer 4 services: Setup and removal of a channel, sending of packets via this channel. By default the content and structure of these packets (e.g. monitoring request) is not known to the gateway. It merely forwards these packets to the associated receiver without interpreting them.
- Diagnostics and administration of the gateway itself.
- ...

These services can be extended as required.

A basic client implementation is available for clients (*GWClient*), that covers communication with the gateway, integration of the communication drivers, etc. Specific clients should use this library in order to avoid getting bogged down in details of the communication with the gateway.

## 5.8 Implementation Aids

### 5.8.1 Implementation of own block driver

Block drivers are structured symmetrically, i.e. usually there is no distinction between a client or a server. If associated system components for abstraction of operating system and hardware details are used, an implementation that can be used on all platforms is generally sufficient.

An overview of the implementation of a block driver is provided below. For an actual implementation the comments provided in the *ltf* files must be taken into account, particularly with regard to data consistency and buffer administration.

### 5.8.2 Interface

A block driver is an ordinary component within the CODESYS runtime system. It has no dedicated component manager interface. Instead it registers each network interface it manages (e.g. each Ethernet card) with the router component using the *RouterRegisterDevice* function. The associated function prototype is shown below (from *CRouterltf.h*):

```
RTS_RESULT CDECL RouterRegisterNetworkInterface(NETWORKINTERFACEINFO
    *pInterfaceInfo, RTS_HANDLE * phSubnet);

typedef struct
{
    PFBDSEND pfBDSend;
    /* pointer to the blockdrivers send method */
    RTS_HANDLE hInterface;
    /* Interfacehandle within the block driver. This handle is
    passed to all calls to the block driver */
}
```

```

int nMaxBlockSize;
    /* The maximum size of a block that may be sent over this
       device. */
int nNetworkAddressBitSize;
    /* Number of bits occupied by an address of this driver */
int bServiceChannel;
    /* If TRUE, this device provides a service channel. */
NETWORKADDRESS addrDevice;
    /* address of the device within it's subnet (CAN-Node ID, etc.)
       */
char szName[MAX_INTERFACE_NAME];
    /* human readable name of the device. Must be unique. */
    /* Could be something like "eth0" or "Ethernetcard #1" */
} NETWORKINTERFACEINFO;

typedef RTS_RESULT( *PFBDSEND) (RTS_HANDLE hInterface, NETWORKADDRESS
    addrReceiver, PROTOCOL_DATA_UNIT pduData);

```

- *pfBDSend* is a function that is called by the router in order to forward a block (*pduData*) to the next relevant node (*addrReceiver*).
- *hInterface* identifies the relevant network interface in the block driver, if the driver manages several interfaces simultaneously. The Id is assigned by the block driver and is transferred whenever *pfBDSend* is called.
- *nMaxBlockSize* is the maximum size of a block that can be sent via this interface.
- *nNetworkAddressBitSize* is the number of bits required for a network address on this interface.
- *addrDevice* is the local address of the network interface.
- *szName* is the name of the interface as displayed to the system user. This name is also used to configure the main network and the subnets.
- *phSubnet* is assigned by the router and identifies the interface in the router. When the block driver receives a block and forwards it to the router it must also forward this Id (see below).

When the block driver receives a correct block on one of its network interfaces, it notifies the router via the *RouterHandleData* function. The function prototype is (again from *CRouterIrf.h*):

```

RTS_RESULT CDECL RouterHandleData(RTS_HANDLE hSubnet, NETWORKADDRESS
    sender, PROTOCOL_DATA_UNIT pduData, int bIsBroadcast);

```

- *hSubnet* must be the Id assigned by the router on interface registration.
- *sender* is the network address of the sender.
- *pduData* contains the received block
- *bIsBroadcast* should set to 1, if the received block was sent to a broadcast address.

### 5.8.3 Addressing

The block driver has as many bits available for network addresses as it specified on registration with the router. A network address in which all bits are set to 1 is reserved as a broadcast address. The same applies to a network address of length 0. Blocks for these addresses therefore have to be distributed to all subnet devices.

An example: Addressing in the UDP block driver

An IP address consists of four components <a>.<b>.<c>.<d>. In a Class C network the first three components are identical for all devices. Therefore only the last byte (d) is used for the network address, i.e. 8 bits are sufficient. Blocks for 0xFF are sent to address <a>.<b>.<c>.FF (local broadcast within the network).

### 5.8.4 General implementation procedure

A simple block driver simply responds to associated system hooks and polls its network interfaces at regular intervals. Faster response times can be achieved if the block driver responds directly to events in its network hardware. This aspect is not covered in this documentation. As reference implementation we recommend BlkDrvUdp (supplied with the runtime system).

Once addressing has been specified (see 5.8.3), a send function must be implemented based on the *PFBDSEND* function prototype. It should return the following values:

- **ERR\_OK** if the block could be sent or at least copied to an internal buffer
- **ERR\_NOBUFFER** if the block could not be sent immediately and no internal buffer is available for intermediate data storage. During the next cycle the router will try sending the block again.
- **ERR\_FAILED** if an error, such as an invalid address, for example, permanently prevents a block being sent. In this case the router will discard the block.

The block driver should initialise its network interfaces in the *CH\_Init* hook and register them with the router.

In the *CH\_CommCycle* hook the block driver should continue resending blocks that were not sent completely and check whether a new block was received. If a block was received it must be forwarded to the router via the *RouterHandleData* function.

### 5.8.5 Synchronisation

In principle the send function of the block driver can be called from different threads. It is therefore important to ensure adequate and correct synchronisation.

Critical code components should be secured through suitable semaphore or similar mechanisms. It is particularly important to ensure that a call of *RouterHandleData* can initiate another send operation. Deadlocks must be avoided. Before a *RouterHandleData* call all semaphores should therefore be enabled if possible.

In single-tasking systems the semaphore mechanisms of the system libraries have been route-optimized and are therefore irrelevant for performance considerations.

## 5.9 Implementation of Own Communication Driver

In contrast to block drivers communication drivers are structured asymmetrically, i.e. there is a clear distinction between a client (on the client side) and a server (on the gateway side). The transfer perspective is also different from that of block drivers: While block drivers invariably send or receive whole blocks or nothing, communication drivers provide an “endless” data stream. Communication drivers therefore do not deliver received data directly to the gateway/client. Instead they report receipt of the data, which can then be retrieved by the higher-level layer in freely selectable portions as required.

The fundamental principle of the send and receive functions is the same on both sides. However, the client must be able to actively establish a connection to the server, while server has to respond to incoming client connections. In addition the client must support a plug-in mechanism for generic configuration of the connection.

Both sides make a secure **stream** available. Data may be sent and delivered in portions of any size. The only requirements are that the data must arrive at the receiver correctly, fully and in the right order. If necessary the communication driver must deal with packet repetition, checksums etc. If data are not retrieved fast enough from the driver, suitable flow control mechanisms should be provided to avoid data loss. If data have to be discarded or cannot be transferred correctly, the connection must be terminated since it violates the stream principle. In addition the communication driver should be able to detect connection interruptions independently.

A normal TCP connection precisely meets the requirements of communication drivers and can therefore be regarded as reference.

### 5.9.1 Communication driver for the gateway

The communication driver for TCP, *CmpGWCommDrvTcp*, should be regarded as reference implementation.

In its *CGateway.h* interface the gateway component provides four functions for operation by a communication driver:

- **GWRegisterCommDrv**  
For registration of a communication driver with the gateway.
- **GWClientConnect**  
Notifies the gateway about the fact that a new connection with a client was established.
- **GWClientDisconnect**  
Notifies the gateway about the fact that a connection to a client was terminated.
- **GWConnectionReady**  
This callback notifies the gateway about the fact that new data can be received via a connection or now further data can be sent.

These functions are described in more detail below.

```
int CDECL GWRegisterCommDrv (COMMDRVINFO *pInfo,
    unsigned long *pdwDriverId);
typedef struct
{
    PFCOMMDRVSEND    pfSend;
    PFCOMMDRVRECEIVE pfReceive;
    PFCOMMDRVCLOSE   pfClose;
} COMMDRVINFO;

typedef int (CDECL *PFCOMMDRVSEND)(unsigned long dwConnHandle,
    PROTOCOL_DATA_UNIT data,
    unsigned long *pdwSent);
typedef int (CDECL *PFCOMMDRVRECEIVE)(unsigned long dwConnHandle,
    PROTOCOL_DATA_UNIT *pData);
typedef int (CDECL *PFCOMMDRVCLOSE)(unsigned long dwConnHandle);
```

This function is used by a communication driver for registering at a gateway. In *pInfo* the driver passes on three function pointers for sending and receiving data and for terminating an existing connection. *pdwDriverId* is set by the gateway and must be passed to the gateway with all further calls as driver identification.

- **PFCOMMDRVSEND:**  
Sends data (*data*) via an existing connection (*dwConnHandle*). *pdwSent* must be set by the communication driver to the number of bytes that were actually sent (or copied to the internal send buffer). If not all bytes could be sent, the gateway will resent the unsent data during the next cycle.
- **PFCOMMDRVRECEIVE**  
Reads data for a connection (*dwConnHandle*) from the receive buffer of the communication driver. During a call *pData->ulCount* contains the maximum number of data to be read. Existing data have to be copied to *pData->pBuffer*, and *pData->ulCount* has to be set to the number of actually read data.
- **PFCOMMDRVCLOSE**  
Closes a connection (*dwConnHandle*). *GWClientDisconnect* must not be called.

**For all three functions the following applies:**

**The function must not block and must not call gateways functions (either directly or indirectly), because this may lead to deadlocks that cannot be rectified!**

```
int CDECL GWClientConnect( unsigned long dwDriverId,
    unsigned long dwConnHandle);
```

This function is called by the communication driver once it has established an incoming connection from a client. *dwDriverId* identifies the communication driver and was returned by the gateway at *GWRegisterCommDrv*. *dwConnHandle* is a driver-specific handle for the new connection and is transferred by the gateway for *send*, *receive* and *close* calls.

```
int CDECL GWClientDisconnect( unsigned long dwDriverId,
                             unsigned long dwConnHandle);
```

This function is called by the communication driver if a connection to a client was cancelled or terminated by the client. *dwDriverId* identifies the communication driver and was returned by the gateway at *GWRegisterCommDrv*. *dwConnHandle* is the driver-specific handle for the terminated connection, which the driver had transferred at *GWClientConnect*.

```
int CDECL GWConnectionReady(unsigned long dwDriverHandle,
                             unsigned long dwConnHandle,
                             int nAction);
```

This function can optionally be called by the communication driver (*dwDriverHandle*) if new data are available for a connection (*dwConnHandle*) or data can be sent. *nAction* can have one of the following values:

- **COMMDRV\_ACTION\_SEND**  
There are free send buffers, i.e. data can be sent again.
- **COMMDRV\_ACTION\_RECEIVE**  
New data are available in the receive buffer.

The same condition should not be signaled more than once per connection as long as the gateway has not responded. After a *COMMDRV\_ACTION\_SEND* for instance, this signal may only be triggered again after a *send* on this connection.

## 5.9.2 Communication driver for the client

The communication driver for TCP, in this case *CmpGWClientCommDrvTcp*, should be regarded as reference implementation.

In its *CGWClientItf.h* interface a client provides two functions for interaction with communication drivers:

- **GWClientRegisterCommDrv**  
To register a communication driver
- **GWClientConnectionReady**  
Notifies the client that new data are available or the driver is ready again to send data.

```
int CDECL GWClientRegisterCommDrv(COMMDRVITF *pItf,
                                  COMMDRVINFO *pDrvInfo,
                                  unsigned long *pdwDriverHandle);

typedef struct
{
    PFCOMMDRVBEGINCONNECT    pfBeginConnect;
    PFCOMMDRVENDCONNECT      pfEndConnect;
    PFCOMMDRVSEND            pfSend;
    PFCOMMDRVRECEIVE        pfReceive;
    PFCOMMDRVCLOSE          pfClose;
}COMMDRVITF;

typedef int (CDECL *PFCOMMDRVSEND)(unsigned long dwConnHandle,
                                   PROTOCOL_DATA_UNIT data,
                                   unsigned long *pdwSent);
```

```

typedef int (CDECL *PFCOMMDRVRECEIVE)(unsigned long dwConnHandle,
                                     PROTOCOL_DATA_UNIT *pData,
                                     unsigned long dwReceive);
typedef int (CDECL *PFCOMMDRVBEGINCONNECT)(PARAMLIST *pParams,
                                           unsigned long *pdwConnHandle,
                                           ASYNCRESET *pAsyncRes);
typedef int (CDECL *PFCOMMDRVENDCONNECT)(ASYNCRESET *pAsyncRes,
                                          unsigned long *pdwConnHandle);
typedef int (CDECL *PFCOMMDRVCLOSE)(unsigned long dwConnHandle);
typedef struct tagASYNCRESET
{
    void *pUser;
    PFASYNCCALLBACK pfCallback;
    unsigned long ulEvent;
    unsigned long ulRequestId;
}ASYNCRESET;
typedef void (STDCALL *PFASYNCCALLBACK)(ASYNCRESET *pAsyncRes);

```

The *GWClientRegisterCommDrv* function registers a communication driver with the client. *pdwDriverHandle* is set by the client and must be passed on by the communication driver to the client with all other calls. *pltf* contains function pointers to the communication driver functions, while *pDrvInfo* contains information about the driver itself and about the parameters required for establishing a connection.

Functions required for *pltf*:

- **PFCOMMDRVSEND**  
Sends data (*data*) via an existing connection (*dwConnHandle*). *pdwSent* must be set by the communication driver to the number of bytes that were actually sent (or copied to the internal send buffer). If not all bytes could be sent, the client will resent the unsent data during the next cycle.
- **PFCOMMDRVRECEIVE**  
Reads data for a connection (*dwConnHandle*) from the receive buffer of the communication driver. During a call *pData->ulCount* contains the maximum number of data to be read. Existing data have to be copied to *pData->pBuffer*, and *pData->ulCount* has to be set to the number of actually read data.
- **PFCOMMDRVBEGINCONNECT**  
This function initiates a connection setup. The parameter *pParams* is described in Section 5.9.2.1, Connection parameters. This function is blocking if *pAsyncRes* == *NULL*. Otherwise this function must be non-blocking. Two cases can be distinguished:
  - o **pAsyncRes != NULL and connection cannot be established immediately**  
The function returns *ERR\_PENDING*, *pdwConnHandle* is not touched. The function initializes the following fields of *pAsyncResult*: *ulEvent* is assigned a handle for a *SysEvent*; *ulRequestId* can be used by the function as required for identifying this asynchronous request.  
The connection now has to be established in the background (e.g. in *CH\_COMMCYCLE* or in a dedicated thread). As soon as the connection has been established or failed the driver sets *ulEvent*. If *pAsyncRes->pfCallback* != *NULL*, the driver calls this callback function.  
The caller of the function can call the function *PFCOMMDRVENDCONNECT* at any time, generally in *pAsyncRes->pfCallback* or after *pAsyncRes->ulEvent* was set, in order to retrieve the result of the connection setup.  
The driver may not remember the pointer to *pAsyncRes*. Instead it usually retains a copy of *\*pAsyncRes*. *pAsyncRes->pUser* is set by the caller and is not touched by the driver.
  - o **Otherwise:**  
*pAsyncRes* is not touched. The function blocks until the connection setup result is known. If successful, *pdwConnHandle* is set to the handle of the new connection, otherwise the function returns the associated error code.

- **PFCOMMDRVENDCONNECT**

This function returns the result of a previous asynchronous call of *PFCOMMDRVBEGINCONNECT*. It must be called exactly once if the asynchronous call returned *ERR\_PENDING*. In all other cases, including multiple calls, an error is returned.

The function blocks until asynchronous connection setup is complete, identified through *pAsyncRes->ulRequestId*. It then returns the result in the same form as *PFCOMMDRVBEGINCONNECT* in the synchronous case.

- **PFCOMMDRVCLOSE**

Closes a connection (*dwConnHandle*).

The data required in *pDrvInfo* are described in 5.9.2.1, Connection parameters.

```
int CDECL GWClientConnectionReady(unsigned long dwDriverHandle,
                                  unsigned long dwConnHandle,
                                  int nAction);
```

This function can optionally be called by the communication driver (*dwDriverHandle*) if new data are available for a connection (*dwConnHandle*) or data can be sent. *nAction* can have one of the following values:

- **COMMDRV\_ACTION\_SEND**

There are free send buffers, i.e. data can be sent again.

- **COMMDRV\_ACTION\_RECEIVE**

New data are available in the receive buffer.

The same condition should not be signaled more than once per connection as long as the client has not responded. After a *COMMDRV\_ACTION\_SEND* for instance, this signal may only be triggered again after a *send* on this connection.

### 5.9.2.1 Connection parameters

A connection to a gateway has to be parameterized in different ways, depending on the communication driver. A TCP connection, for example, requires the IP address and the port of the remote terminal, while a serial connection requires the COM port, baud rate, stop bits etc. The required parameters are therefore specified by the driver itself. Since the driver handle may change depending on the installed drivers, each driver is allocated a unique number that also unambiguously identifies the driver, even on different computers.

The driver therefore describes itself with the parameter *pDrvInfo* on registration. This structure is defined as follows:

```
typedef struct
{
    COMMDRIVERHANDLE hDriver;
    GUID guid;
    wchar_t *pwszName;
    PARAMDEFLIST params;
}COMMDRVINFO;

typedef unsigned long COMMDRIVERHANDLE;

typedef struct
{
    int nNumParams;
    PARAMETERDEFINITION *pParam;
}PARAMDEFLIST;

typedef struct
{
    wchar_t pwszName[MAX_PARAM_NAME+1];
    unsigned long dwParamId;
```

```

    unsigned long dwType; /* PT_xxx */
}PARAMETERDEFINITION;

```

*hDriver* is set by the client. This parameter is independent of the driver. *guid* is a GUID, as generated by the guidgen tool from Microsoft, for example. The GUID must be recreated for each driver. It identifies the driver across different client instances. *pwszName* is the name of the driver. It is displayed in the configuration dialog of the client, for example. *params* defines the parameters required for connection setup for this driver. These parameters are also displayed in the generic configuration dialog of the client. Suitable names should therefore be chosen.

*PARAMDEFLIST* is a list of parameters required for the driver. It includes the number of required parameters (*nNumParams*) and an array of *nNumParams* parameter definitions (*pParam*).

A PARAMETER DEFINITION describes an individual parameter through a name (*pwszName*), Id (*dwParamId*, must be unique within the driver), and type. Types are defined by PT\_xxx (e.g. PT\_CHAR, PT\_INT16, PT\_string, ...) constants in file *CGWClientIf.h*. All common numeric data types and strings are available.

#### An example:

The communication driver for TCP requires two parameters. One *IP address* parameter with Id 0 that either contains the address or the DNS name of the gateway as a string, and one *Port* parameter, a 16-bit value with Id 1 that specifies the port for the gateway. The parameter array for the definition has the following structure:

```

static PARAMETERDEFINITION s_ParamDefinitions[] =
{
    {L"IP-Address", (unsigned long)0, (unsigned long)PT_string},
    {L"Port", 1, PT_uint16}
};

```

From this list the client generates a parameter list (*pParams*) during connection setup (*PFCOMMDRVBEGINCONNECT*) containing specific values for the individual parameters. The values are transferred as pointers to the type corresponding to the parameter definition (*short\**, *int\**, ...). Strings are transferred in the usual way as pointers to the first element of the string, i.e. as *char\**. The parameter list is defined as follows:

```

typedef struct
{
    int nNumParams;
    PARAMETER *pParam;
}PARAMLIST;

typedef struct
{
    unsigned long dwParamId;
    unsigned long type;
    void * pValue;
}PARAMETER;

```

#### An example:

The following code establishes a connection to gateway 192.168.100.70 on port 1217 using the TCP communication driver from the previous example. In the interest of transparency the configuration is static in this case and uses the parameter definition of the communication driver described above. In real applications the list is generated dynamically based on the user inputs and the parameter definition of the driver.

```

int nResult;
unsigned long ulConnHandle;

ASYNCRESULT async = {NULL,NULL,0,0};

```



```

char stAddr[] = "192.168.100.70";
unsigned short usPort = "1217";
PARAMETER params[2] =
{
    {0, PT_string, stAddr}, /* stAddr is already a pointer */
    {1, PT_uint16, &usPort} /* numeric values as pointer on value */
}
PARAMLIST list = {2, params};

nResult = pfCommDrvBeginConnect(&list, &ulConnHandle, &async);
if(nResult == ERR_PENDING)
{
    CAL_SysEventWait(async.ulEvent, -1); /* Wait until finished */
    nResult = pfCommDrvEndConnect(&async, &ulConnHandle);
}

if(nResult == ERR_OK)
{
    /* Send and receive data */
    ...
    pfCommDrvClose(ulConnHandle);
}
...

```

### 5.9.2.2 Implementation of BeginConnect

The following code snippet illustrates the basic structure of the `CommDrvBeginConnect` and `CommDrvEndConnect` functions with correct handling of the `pAsyncResult` parameter. While other implementations are conceivable and indeed often desirable, connection setup is always asynchronous in the example. For the case `pAsyncRes == NULL` an additional short code snippet is included that converts the asynchronous function into a synchronous function through the function `CommDrvBeginConnect` recalling itself recursively and then blocking itself by calling `CommDrvEndConnect`. In the synchronous case `CommCycleHook` may therefore be blocked. In this implementation a separate thread must therefore be used for asynchronous connection setup.

```

int BeginConnect(PARAMLIST *pList, ULONG *pdwHandle, ASYNCRESET *pAsyncRes)
{
    int nRes;
    unsigned long ulReqId;
    if(pAsyncRes == NULL)
    {
        AsyncRes async = {NULL, NULL, 0, 0};
        nRes = BeginConnect(pList, pdwHandle, &async);
        if(nRes == ERR_PENDING)
            nRes = EndConnect(&async, pdwHandle);
        return nRes;
    }

    CheckParameters(pList); // All available? Correct types?

    ulReqId = StartAsyncConnect(pList);
    pAsyncRes->ulRequestId = ulReqId;
    pAsyncRes->ulEvent = SysEventCreate(LongToString(ulReqId));
    AddPendingRequest(*pAsyncRes);

    return ERR_PENDING;
}

int EndConnect(ASYNCRESET *pAsyncRes, ULONG *pdwHandle)
{
    Request req;
    req = GetPendingRequest(pAsyncRes->ulRequestId);
    if(req == NULL)
        return ERR_PARAMETER;
    SysEventWait(req.asyncRes.ulEvent, -1);
}

```

```

    if(req.result == ERR_OK)
        *pdwHandle = req.handle;

    RemovePendingRequest(pAsyncRes->ulRequestId);

    return req.result;
}

/* Callback example. We assume that this function is called by the
network layer, for example, as soon as connection setup was either
completed or has failed.
The function is intended to indicate driver behaviour required for
signaling the end of the asynchronous call to the client application.
*/
void OnConnectFinished(int reqId, int nResult, ulong ulHandle)
{
    Request req;
    req = GetPendingRequest(reqId);
    req.result = nResult;
    req.handle = ulHandle;
    SysEventSet(req.asyncRes.ulEvent);
    if(req.asyncRes.pfCallback != NULL)
        req.asyncRes.pfCallback(&(req.asyncRes));
}

```

## 5.10 Standard block drivers and their network addresses

### 5.10.1 Overview

This chapter uses an example to describe the formation of the network address for network adapters based on different block drivers. For formation of the node address from this network address please refer to chapter 5.4.4 Address structure.

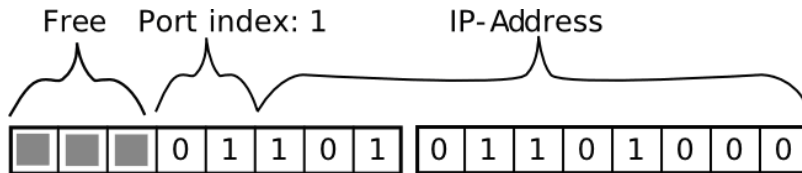
### 5.10.2 UDP block driver

The UDP block driver does not define physical network connections as network connections, but uses each available IP address of a system as a (virtual) network connection. For each such network connection the block driver uses four predefined UDP ports. In the standard configuration they are ports 1740, 1741, 1742 and 1743. The block driver uses precisely one of these ports for sending and receiving data. It either uses the port specified by the configuration or the first free port if no port is configured. The remaining ports are available for further runtime system instances on the same device. The block driver sends broadcasts always to all four ports in the network.

The block driver operates on local network segments that have a common IP network address. The network address is formed from the relevant bits of the IP address (i.e. without the network mask) and the port index.

The diagram illustrates the address formation for the following configuration:

- IP address **172.17.69.104** (= **AC.11.45.68** hexadecimal)
- IP network mask **255.255.248.0** → i.e. only the last 11 bits of the IP address are relevant
- Port **1741**, corresponds to port index **1** (1740 = 0)
- Since 2 bits are required for the port index, this configuration uses 13 bits, i.e. the top 3 bits of the first byte remain unused. They can be used by the router for encoding the subnet ID, for example.



The following source code snippet generates this address from the IP configuration. Unfortunately several special cases have to be covered that make the code a little difficult to read. The function expects the relevant part of the IP address in `dwLocalAddr`, appends the 2-bit port index to `wPortIdxOffset`, and finally converts the required bytes to a byte array in `pnaResult`.

```
static void CreateNetworkAddress(NETWORKADDRESS *pnaResult, RTS_UI32
    dwLocalAddr, int iPortIdx, RTS_UI16 wPortIdxOffset)
/* dwLocalAddr = dwIpAddress & ~dwNetworkMask
 * wPortIdxOffset = <number of non-networkmask bits in the ip-address>
 */
{
    int i;
    RTS_UI16 wLocalAddrLen

    /* wLocalAddrLen = Total number of bytes required for this network
     * address. PortIdxOffset equals the number of bits required for the
     * ipaddress, then adding 2 bits for the port index and finally
     * calculate the number of bytes, rounding up ("+7")
     */
    wLocalAddrLen = (wPortIdxOffset + 2 + 7) / 8;

    /* Caveat:
     * We are shifting the portidx to the left in order to set the
     * matching bits within the dwLocalAddr. If 31 or even all 32 bits
     * of the ipaddress are used for the local address we need some
     * extra treatment, which is done after the for loop below. So
     * everything should work fine.
     * But if the portidx offset is 32 then (at least on x86 with VC6
     * compiler) the left shift is executed as a shift (i % 32)
     * effectively doing nothing, where zero should result.
     * In short: Expected (x << 32) == 0, but what we get is
     * (x << 32) == x.
     * Therefore we exclude that case in the next statement.
     */
    if(wPortIdxOffset < 32)
        dwLocalAddr = dwLocalAddr | (iPortIdx << wPortIdxOffset);

    pnaResult->nLength = wLocalAddrLen;
    for(i=wLocalAddrLen-1; i>=0; i--)
    {
        pnaResult->address[i] = (RTS_UI8)(dwLocalAddr & 0xFF);
        dwLocalAddr = dwLocalAddr >> 8;
    }
    if(wPortIdxOffset > 30)
    {
        pnaResult->address[0] = iPortIdx >> (32 - wPortIdxOffset);
    }
}
```

### 5.10.3 Serial block driver

The serial block driver is called `CmpBlkDrvSimpleCom`. The driver sends the following characters for synchronisation of each data block:

- Block start delimiter: "#<"
- Block start delimiter: "#>"

- Within a block each "#" is expanded to "#\_"

Each block carries a 16 bit crc over the original data, stored in Intel byte order. Thus a block looks like this:

```
<start delimiter>[<data><crc_highbyte><crc_lowbyte>]<end delimiter>
```

Data within [ ] is escaped to the above rule.

## 5.11 Modules

The communication stack is implemented in the following runtime system modules:

Modules	Brief description
CmpRouter	Implements the router layer
CmpBlkDrvUdp	Block driver for communication via UDP
CmpChannelManager	Basic channel management component
CmpChannelClient	Client for the channel level
CmpChannelServer	Server for the channel level
CmpSrv	Application server (Layer 7)
CmpGateway	Gateway component
CmpGWCommDrvTCP	Gateway communication driver for TCP connections
CmpGWClientCommDrvTCP	Client communication driver for TCP connections
CmpAddrSrv	Address assignment
CmpNameServiceClient	Name service (client)
CmpNameServiceServer	Name service (server)

## 5.12 Client API Interfaces

There are three different levels to enter the CODESYS V3 PLC network.

The lowest level is to enter the network in the runtime system itself. This is typically used, if you write your own component and want to communicate with another plc.

The next level is the Gateway Client. This is typically used by a PlugIn in CODESYS or by CODESYS itself.

The highest interface is the PLCHandler. It provides to enter the communication network from any other proprietary client.

### 5.12.1 Channel client (CmpChannelClient)

The channel client component can be used from a component in the runtime system to enter the PLC communication network. The channel client can be used to open the communication to another PLC and to sent native online service though this channel.

### 5.12.2 Gateway client (GwClient)

The Gateway-Client has a native C++ interface, so it can be used from .NET and C++ clients (like PlugIns in CODESYS). This interface provides to open the communication and to sent native online service though this channel. In this layer, no symbolic access to variables is possible! To use symbolic access, you have to use the PLCHandler interface.

### 5.12.3 PLCHandler

This is the highest level API that is provided by 3S.

The PLCHandler is a C++-class which on a comfortable level provides services for the communication between a client (e.g. visualization) and a 3S Automation-Alliance compliant PLC (controller).

The following features and services are available:

- Establishing and terminating the communication with the PLC
- Reading all variables on the PLC
- Cyclic reading of variables' values from the PLC
- Synchronous reading of variables' values from the PLC
- Synchronous writing of variables' values to the PLC
- Possibility of instancing for the purpose of a simultaneous communication with several PLCs
- Automatic reconnecting with the PLC after an break of the connection
- Automatic restart after a program download from CODESYS to the PLC
- Data transfer to and from PLC

Thus the PLCHandler can be used as a basic component for OPC Servers or visualizations. The Handler is delivered as a SDK, i.e. all C++ header files, the static link library (PLCHandler.lib resp. PLCHandlerComplete.lib) and a sample program (main.cpp) are part of the package.

The PLCHandler supports beneath the CODESYS V3 generation runtime systems additionally the CODESYS V2.x generation runtime systems! So you only have to handle one interface for all runtime systems in the field.

## 6 Device- / I/O Configuration

As I/O configuration it is called the complete set of mechanisms, to support all hardware-devices and I/O<sup>2</sup> systems that are connected to a controller (the runtime system). This includes the description of the hardware hierarchy, description and parameterization of each device and the support of each device with a driver (I/O-driver).

One of the main aims of the I/O-configuration in CODESYS V3 is internal standardization of the different device types, so that different field bus nodes, internal I/Os, extension modules, drives, etc. are basically treated in the same way. This means that each device is described with the identical description structure. So a driver can interpret principally every device description.

The second main issue is that the I/O-configuration is created as an IEC data-structure and is downloaded to the runtime system with the IEC application that includes the I/O-configuration.

If you want to access the I/O-configuration from another IEC application, this application must be a child application of the application with the I/O-configuration.

In the following chapters, all themes to access devices and IOs are described in detail.

### 6.1 Graphical Configuration

The IO-configuration in CODESYS is organized as a tree of devices.

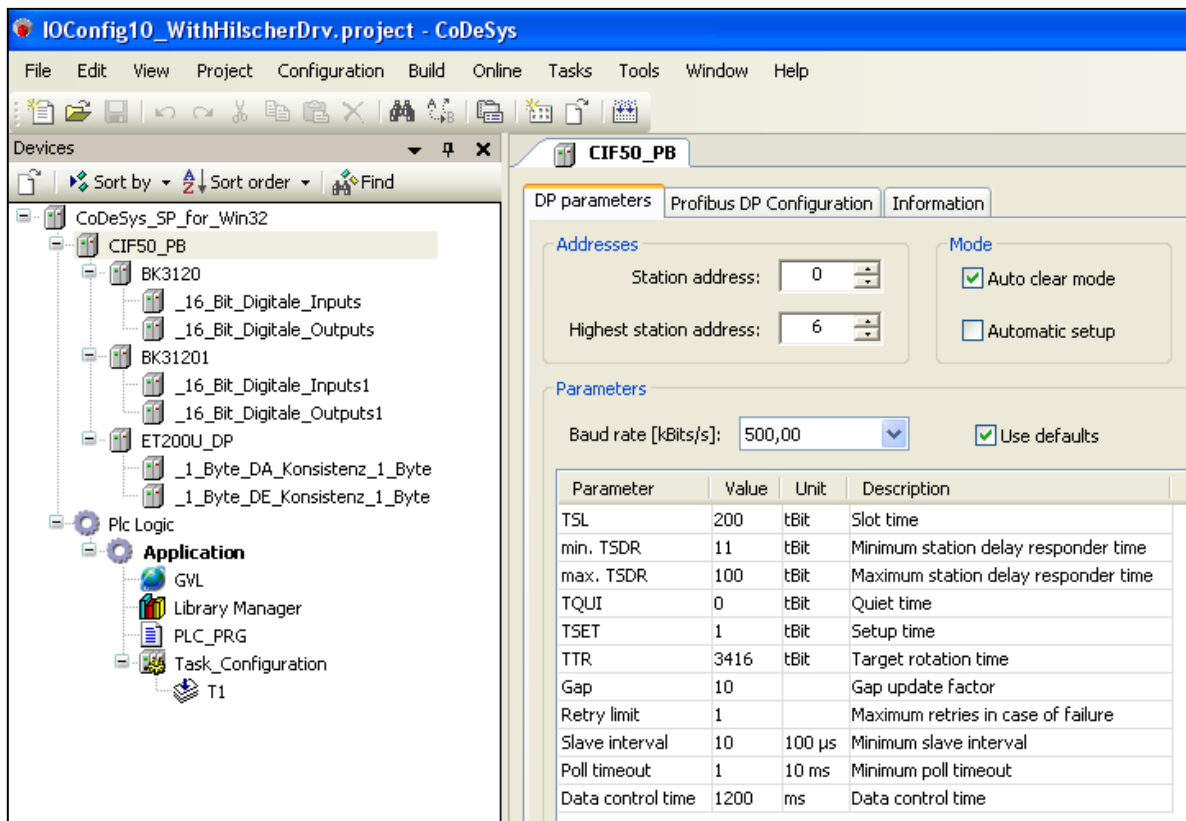


Figure 5: IO-configuration in CODESYS

The base node for the devices typically is a PLC (the runtime system; sometimes called target). In the example above it is a windows runtime system with the name CODESYS\_SP\_for\_Win32.

A PLC node specifies typically, which kind of devices can be appended under this node. For example on a PC you can append (and physically plugin) PCI cards under this node. These appended devices correspond to the hardware that is plugged and attached to the PLC.

<sup>2</sup> I/O: Input- and Output-Signals (Digital- or Analog), e.g. Sensor Data

In the example above, a PCI Hilscher Profibus Master Card is appended under the PLC. Under the Profibus Master there are Profibus Slaves to configure the complete field bus. This configuration represents the physical structure of a hardware that is connected and attached on the PLC.

## 6.2 Devices

*Devices* represent, strictly speaking, hardware units that are relevant for the controller configuration. They may be programmable devices (PLC), local I/O units, extension modules, field bus nodes, or bus coupler modules. In a wider sense the term may also include functional hardware units that are selectable or configurable by the user but have no direct hardware equivalent. Unless otherwise specified, in the following sections the term *device* is used in this wider sense. With regard to the device description and the interface both aspects are also treated equally.

Devices may be linked hierarchically. The possible combinations are defined in the respective description files. The link points of the devices are referred to as *Connectors*. Connectors can either take a *Parent* role or a *Child* role. Depending on the configuration, between 1 and *n* devices with suitable child connectors can be attached to Parent connectors.

In diagrams the following symbols are used for illustration purposes: a device is represented by a large rectangle. Smaller rectangles to the left of the device indicate child connectors, i.e. points where the device can be attached to a parent node. The same type of rectangle on the right-hand side represents parent connectors, i.e. possible connection points for children.

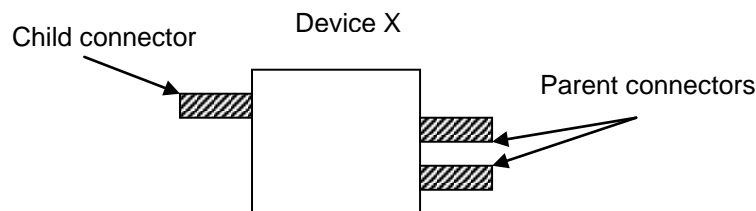


Figure 6: Logical structure of one device

## 6.3 Device Descriptions

To use a device in the device configuration tree in CODESYS, the device must be completely described in XML (device description file). This includes the description of device parameters and connectors. The device description file contains all information for:

- Getting inserted the device object at the correct position in the I/O-configuration tree
- Getting displayed and editable the device parameters in the user interface
- Optionally getting specified an I/O driver that operates the device

See in the following general information on connectors (chap. 6.3.1), parameters (chap. 6.3.2) and IO-Mapping (chap. 6.3.3)

See Chap. 6.4 for information on particular device description file entries.

### 6.3.1 Connectors

Connectors form the connections between devices and have their own configuration data in the form of a parameter set. So a device is fully described with a single child connector, one or more parent connectors and a set of so-called parameters for each connector. Parameters hold the information to configure a connector (e.g. baud rate of a Profibus master connector). An I/O-channel is described as a parameter too.

Connectors can only be connected with each other if they have the same interface. Interfaces are identified via a string. A manufacturer abbreviation used as a prefix for this string defines a namespace and ensures uniqueness of the interface names across manufacturers. The “Common” namespace contains generally used interfaces specified by 3S, e.g.:

“Common.PCI”: Standard PCI connection

“Common.DP”: Profibus

“Common.CANopen”: CANopen network

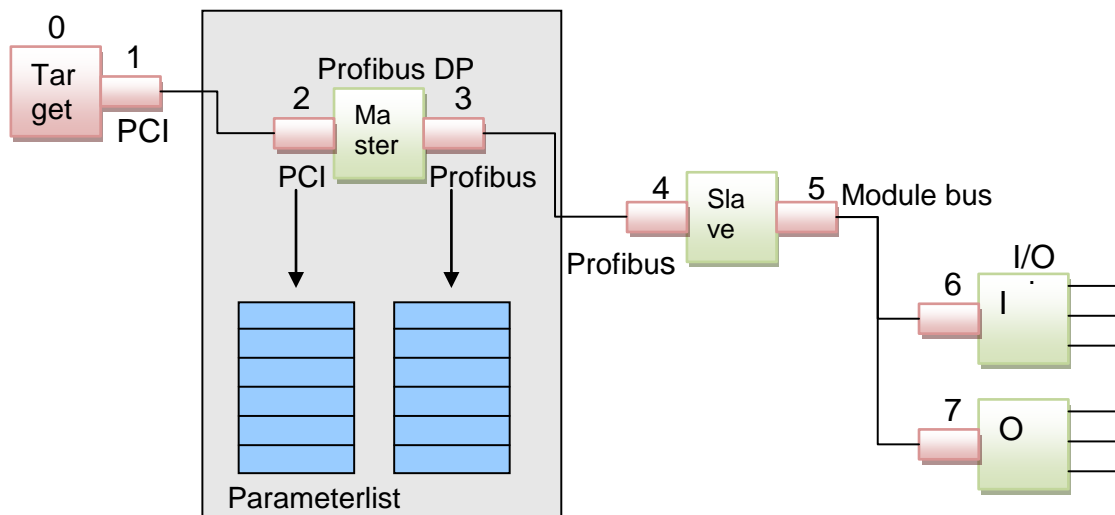
Device manufacturers can define own (internal and external) interfaces within their namespace.

The interface initially only defines the type of devices that can be connected to a connector. More control over the number of possible children and the representation is provided through the definition of *adapters* within a parent connector. Three types of adapters are available:

- **Fixed:** Specifies that a certain child is always present. This is used, for example, for hard-wired I/Os that the user cannot install or remove. The user cannot remove these devices.
- **Slot:** Defines a slot as used in expandable controllers, for example. The number of slots is fixed, while modules can be replaced or (optional) removed altogether, leaving an empty slot.
- **Var:** Defines an adapter to which any number of children (up to a specified upper limit) can be connected and disconnected. This corresponds to the model of field bus to which new devices can be added.

Each type has its own *module Id* so that an I/O driver can recognize the connector type in the runtime system. The parameter profile for a certain module Id is predefined. The module Id must not be confused with the interface type (see above). While in many cases there may be a unique assignment between the module Id and the interface (e.g. for “Common.DP” the parent connector has module Id 32 and the child connector has Id 33), additional interfaces can be defined that are mapped to the same module Ids. In this case the driver treats these devices in the same way, although they cannot be combined with each other. This may be useful if a different physical interface is used internally for a standard bus, for example (e.g. a special PCI connection). In this case only devices with this special interface can be connected, even though it makes no difference for the driver.

The next diagram shows a PLC node (target) with a Profibus master, one modular slave with one input and one output module as it is configured in a simple I/O-configuration:



**Figure 7: Profibus configuration**

In the figure above, all connectors are marked with a red color. Above the connectors you see the logical number. The numbers are assigned from the top to the down position in the configuration tree.

As you can see, for example the Profibus master device is completely described with its child connector (PCI), with its parent connector (Profibus) and with a parameter list for each connector.

The I/O-configuration is stored as one single list. The index in the list is the logical position in the tree. To hold the tree information, every child connector contains a pointer to its father connector. So the tree information can be restored out of the information in the list.

Internally, every connector is stored in the same structure (this is one entry in the list):



Element	IEC Data type
Connector Type:	DWORD
#define CT_PROGRAMMABLE	0x1000
#define CT_SAFETY	0x1002
#define CT_DRIVE	0x1003
#define CT_PARAMETRIZABLE	0x1004
#define CT_HMI	0x1005
#define CT_SOFTMOTION_CONTROLLER_3S	0x1006
#define CT_GATEWAY_3S	0x1007
#define CT_CAN_MASTER	0x0010
#define CT_CAN_SLAVE	0x0011
#define CT_CAN_DEVICE	0x0012
#define CT_CAN_MODULE	0x0013
#define CT_J1939_MANAGER	0x0018
#define CT_J1939_ECU	0x0019
#define CT_PROFIBUS_MASTER	0x0020
#define CT_PROFIBUS_SLAVE	0x0021
#define CT_PROFIBUS_DEVICE	0x0022
#define CT_PROFIBUS_MOD_MASTER	0x0023
#define CT_PROFIBUS_MOD_SLAVE	0x0024
#define CT_DEVICENET_CANBUS	0x002F
#define CT_DEVICENET_MASTER	0x0030
#define CT_DEVICENET_SLAVE	0x0031
#define CT_DEVICENET_DEVICE	0x0032
#define CT_ETHERCAT_MASTER	0x0040
#define CT_ETHERCAT_SLAVE	0x0041
#define CT_ETHERCAT_DEVICE	0x0042
#define CT_ETHERCAT_MODULE_PARENT_CONNECTOR	0x0043
#define CT_ETHERCAT_KBUS_MODULE	0x0044
#define CT_SERCOSIII_MASTER	0x0046
#define CT_SERCOSIII_SLAVE	0x0047
#define CT_SERCOSIII_MODULE	0x0048
#define CT_SERCOSIII_SAFETY_MODULE	0x0049
#define CT_SERCOSIII_SLAVE_CONNECTOR_TO_MODULE	0x004A
#define CT_PROFINET_IO_MASTER	0x0050
#define CT_PROFINET_IO_SLAVE	0x0051
#define CT_PROFINET_IO_MODULE	0x0052
#define CT_PROFINET_IO_DEVICE	0x0053
#define CT_PROFINET_IO_SUBMODULE	0x0054
#define CT_MODBUS_TCP_MASTER	0x0058
#define CT_MODBUS_TCP_SLAVE	0x0059
#define CT_MODBUS_TCP_SLAVE_DEVICE	0x0073
#define CT_MODBUS_SERIAL_MASTER	0x005A
#define CT_MODBUS_SERIAL_SLAVE_TO_MASTER	0x005B
#define CT_MODBUS_SERIAL_PORT	0x005C
#define CT_MODBUS_SERIAL_MASTER_TO_PORT	0x005D
#define CT_ETHERNET_IP_SCANNER	0x0064
#define CT_ETHERNET_IP_REMOTE_ADAPTER	0x0065

Element	IEC Data type
#define CT_ETHERNET_IP_MODULE	0x0066
#define CT_ETHERNET_IP_LOCAL_ADAPTER	0x0078
#define CT_ETHERNET_ADAPTER	0x006E
#define CT_ASI_MASTER	0x0082
#define CT_ASI_SLAVE	0x0083
#define CT_SOFTIO_MASTER	0x0094
#define CT_SOFTIO_SLAVE	0x0095
#define CT_GENERIC_LOGICAL_DEVICE	0x0096
#define CT_GENERATED_LOGICAL_DEVICE	0x0097
#define CT_LOGICAL_GVL_DEVICE	0x0098
#define CT_IOLINK_MASTER	0x00A0
#define CT_IOLINK_DEVICE_V101	0x00A2
#define CT_IOLINK_DEVICE_V11	0x00A3
#define CT_IOLINK_STANDARD_INOUT	0x00B0
#define CT_SIL2_UNSAFE_BRIDGE_MASTER	0x00C0
#define CT_SIL2_UNSAFE_BRIDGE_SLAVE	0x00C1
#define CT_PCI_MASTER	0x0100
#define CT_PCI_SLAVE	0x0101
#define CT_IEC61850_SERVER	0x0200
#define CT_IEC61850_SERVER_PARENT	0x0201
#define CT_IEC61850_CONTROL_BLOCKS	0x0202 /*..0x021F */
#define CT_IEC61850_LOGICAL_DEVICE	0x0230 /*..0x023F */
#define CT_IEC61850_LOGICAL_NODE	0x0240 /*..0x024F */
#define CT_IEC61850_DATA_SETS_NODE	0x0250
#define CT_IEC61850_REPORT_CONTROL_BLOCKS_NODE	0x0260
#define CT_IEC61850_LOG_CONTROL_BLOCKS_NODE	0x270
#define CT_IEC61850_GOOSE_CONTROL_BLOCKS_NODE	0x280
#define CT_IEC61850_SAMPLED_VALUE_CONTROL_NODE	0x290
#define CT_SAFETYSP_IO_MASTER	0x301
#define CT_SAFETYSP_IP_SLAVE	0x302
#define CT_SOFTMOTION_ALLGEMEIN	0x400
#define CT_SOFTMOTION_POSCONTROL	0x401
#define CT_SoftMotion_CAN	0x402
#define CT_SOFTMOTION_ETHERCAT	0x403
#define CT_SOFTMOTION_SERCOSIII	0x404
#define CT_SOFTMOTION_FREE_ENCODER	0x480
#define CT_SOFTMOTION_FIX_ENCODER	0x481
#define CT_SOFTMOTION_ENCODER_CAN	0x482
#define CT_SOFTMOTION_ENCODER_ETHERCAT	0x483
#define CT_SOFTMOTION_LOGICAL_AXIS	0x4e0
#define CT_SOFTMOTION_DRIVEPOOL	0x4ff
#define CT_SOFTVISION_CAMERA_DEVICE	0x800
#define CT_USB_GAME_CONTROLLER	0x900
#define CT_FDT_Communication_DTM	0xFD7
#define CT_FDT_Gateway_DTM	0xFD8

Element	IEC Data type
#define CT_FDT_Device_DTM                    0xFD9	
#define CT_OEM_START                        0x8000	
#define CT_OEM_END                         0x8FFF	
Diagnostic Flags:	DWORD
#define CF_ENABLE                            0x0001	
#define CF_DRIVER_AVAILABLE                0x0010	
#define CF_CONNECTOR_FOUND                0x0020	
#define CF_CONNECTOR_CONFIGURED         0x0040	
#define CF_CONNECTOR_ACTIVE              0x0080	
#define CF_CONNECTOR_BUS_ERROR           0x0100	
#define CF_CONNECTOR_ERROR               0x0200	
#define CF_CONNECTOR_DIAGNOSTIC_AVAILABLE 0x0400	
#define CF_CONNECTOR_PASSIVE             0x0800	
I/O-Driver handle, that operates this connector	DWORD
Number of parameters	DWORD
Pointer to the parameter list	DWORD
Pointer to the father node	DWORD

The complete I/O-configuration is generated as a list of connectors. For the example shown in the figure above an I/O-configuration as follows will be generated.

Nr	Connector	Father
0	Target (PLC)	--
1	PCI Master	0
2	PCI Slave	1
3	Profibus Master	2
4	Profibus Slave	3
5	Modulbus Master	4
6	Modulbus Slave, Input	5
7	Modulbus Slave, Output	5

This structure will be downloaded exactly in such a list of connectors to the runtime system. This list then can be examined by an I/O-driver.

### 6.3.2 Parameters

The whole configuration of a device is based on *parameter lists*. Parameter lists correspond to object dictionaries as used in various field buses. Parameters are identified via a 32-bit number, the *parameter Id*. The structure and the meaning of a parameter with a certain Id are described separately for each device type in the form of a profile (profiles exist for Profibus masters and Profibus slaves, for example).

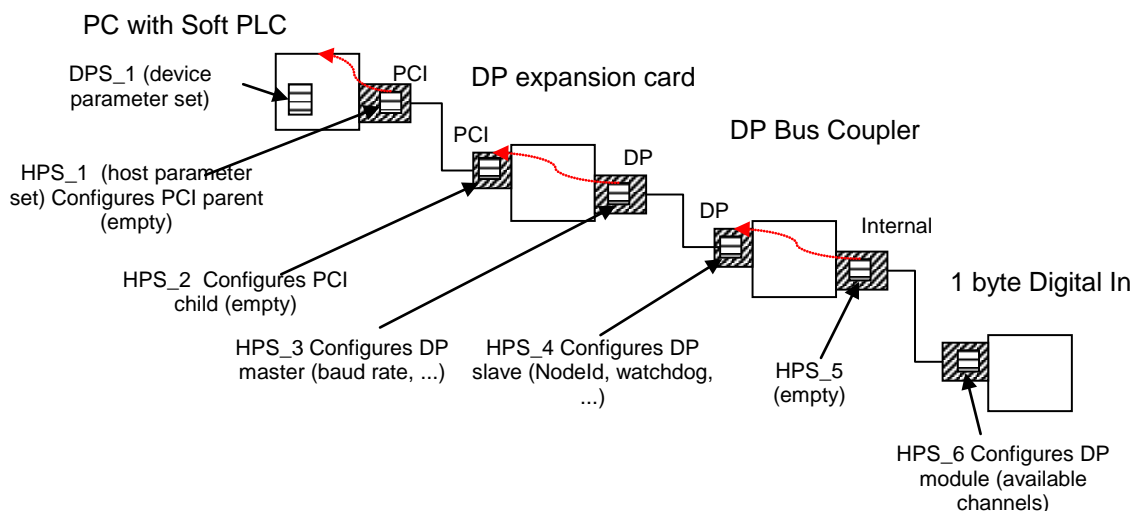
Process data (I/O channels) are also defined as parameters. They are identified separately as input or output. These parameters can be displayed in the process map and updated based on the bus cycle, even if the mapped variable is not used directly in any task (attribute “alwaysmapping” in the device description of the parameter).

A device can have any number of connections, so that it can be operated in a Profibus or a CANopen network, for example. In each case the same device is parameterized, although the configuration will differ significantly (in one case via the CANopen object dictionary; in the other as Profibus slave parameter). *Each connector has its own parameter list for parameterizing the device from the perspective of this connector, i.e. as CANopen device or as Profibus slave. In this way several similar master connections that are used simultaneously (e.g. 2 Profibus strands) can be configured separately (different baud rate etc.). These parameter lists are usually not required on the device (e.g. the field bus slave), which usually has no CODESYS access. Instead they are stored in a CODESYS-configurable device (host), where an I/O driver converts the parameter lists to the fieldbus-specific format and then configures the device via the fieldbus. Parameter lists in connectors are therefore referred to as host parameter set<sup>3</sup>, because they are loaded to the responsible host.*

In contrast, a CODESYS-configurable device can define a *device parameter set*<sup>4</sup>. This parameter list is directly transferred to the device and is independent of the connectors used.

In many cases host parameter sets have to be “handed” upwards through several devices to ensure they end up at the right parent. For child connectors the parameter set is always transferred to the parent by default. Ambiguity can arise with parent connectors, where the parameter set may be required on the device itself (for CODESYS-configurable devices) or may have to be handed upwards via a child connector. In order to ensure that CODESYS finds the right path, parent connectors specify via which child connector the parameter set should be transported, or whether it should be displayed directly in the device (→ *host path*). For identification purposes all connectors therefore have a unique (within the device) Id > 0.

The following example illustrates this. It shows a configuration with a PC-based Soft PLC, with a Profibus card connected to the PCI bus, to which a bus coupler with several modules is connected.



**Figure 8: Parameter set in a typical Profibus configuration**

<sup>3</sup> engl. Hostparameterset

<sup>4</sup> engl. Deviceparameterset

When the configuration is loaded to the Soft PLC it receives its own device parameter set (DPS\_1) and then all host parameter sets for connectors that have a host path to this device (i.e. all host parameter sets shown in the diagram): HPS\_1 as parent of HPS\_2, below HPS\_3, ... down to HPS\_6. If an "intelligent" device below the PLC had its own device parameter set defined (e.g. the Bus Coupler), it would **not** appear on the PLC. It would only be used if the Bus Coupler was CODESYS-configurable and therefore had its own (bus-independent) configuration.

A parameter is described in the following structure:

Element	IEC Datatype
Parameter ID	DWORD
Value or pointer to value	DWORD
Datatype (Enum)	WORD
Length in Bits	WORD
Flags: #define PVF_FUNCTION      0x0001 #define PVF_POINTER      0x0002 #define PVF_VALUE        0x0004 #define PVF_READ         0x0010 #define PVF_WRITE        0x0020	DWORD
Driver specific	DWORD

Master specific slave parameters:

A fieldbus master can define parameters that should be created within its slaves additionally to the default parameters. For this purpose the attribute "*createInChildConnector*" is to be added to the respective parameter definition in the device description of the master. The parameter then will be created also in every child device. Example:

```
<Parameter ParameterId="12345678" type="std:BOOL" >
- <!--
16#30010000 // ETC_MASTER_USELRW
-->
<Attributes offlineaccess="readwrite" download="true" functional="false"
createInChildConnector="true" />
<Default>true</Default>
<Name name="local:NONE">testparameter</Name>
<Description name="local:NONE">">Testparameter for child</Description>
</Parameter>
```

See also chap. 6.4.4.6 for device description entries concerning parameters.

### 6.3.3 I/O mapping

As described in the chapter before, input and output channels are described in form of parameters. Where an input value must be copied from the physical device into the application and where to copy an output value from the application to the device, is called the *I/O-Mapping*. The I/O-Mapping can be specified in CODESYS by writing IEC variables to each configured input and output channel. Here a new variable can be generated or an existing variable can be mapped to the corresponding channel.

---

**Note:** Strings cannot be mapped on channels!

---

Each mapping entry is stored in the following structure:

Element	IEC Datatype
Pointer to Parameter Description	POINTER
Pointer to IEC-address	POINTER
Parameter Bit-Offset	WORD
IEC Bit-Offset	WORD
Size in Bits	WORD
BaseTypeInfoInformation	WORD
Driver specific	DWORD

This mapping information is downloaded to the runtime system with the I/O-configuration (connector list and parameter list). The mapping list is used to copy the channel values at every IEC task cycle from the devices into the IEC application (inputs) and from the IEC application to the devices (outputs) via the I/O-drivers.

The basetype information parameter contains information, which is needed to perform a correct swapping action. This must be done by the driver, because the runtime system does not know the byte order of the fieldbus. The io driver knows the byte order of the fieldbus and the byte order of the target system.

In the lower byte the type class is set. The type classes are defined in the IBase library. With the base class information you can calculate the swapping size e.g. WORD base class has got the swapping size 2. This information is important, because the parameter size in bits can be different to the swapping size e.g. in array or structs, the sizes are different.

In the high byte only two bits are used:

Bit 0x0100 : The swapping information in the parameter is valid. If this bit is set to 0, the BaseTypeInfoInformation is not set.

Bit 0x8000: Swapping is enabled or disabled. If this bit is set to 0. The BaseTypeInfoInformation is valid, but the parameter shouldn't be swapped at all. If it is set to 1, the parameter must be swapped.

## 6.4 Device Description Files

Devices are described in XML files with extension `.devdesc.xml`. They must follow the `DeviceDescription-1.0.xsd` scheme (see `devicedescription_xsd.pdf`). A description file may describe a generally available *device* and any number of *modules* that are only available in the context of this device.

Like all valid XML documents the description file contains a root node, in this case "DeviceDescription", which has no further attribute. Below this node there may be the following general sections:

```
<DeviceDescription>
  <Types></Types>
  <Strings></Strings>
  <Files></Files>
  <Device>
    <DeviceIdentification></DeviceIdentification>
    <DeviceInfo></DeviceInfo>
    <DriverInfo></DriverInfo>
    <Connector></Connector>
    <Functional></Functional>
    <ExtendedSettings></ExtendedSettings>
  </Device>
</DeviceDescription>
```

```

    <Modules>
      <Module>
        <DeviceInfo></DeviceInfo>
        <Connector></Connector>
      </Module>
    </Modules>
  </DeviceDescription>

```

Additionally there might be subsections for defining the device resp. host parameters (<deviceparameterset>, <hostparameterset>), for the library (<requiredlib>) handling, for the child objects handling (<functional>), and for “Extended Settings” like the target settings.

**Note:** For complete documentation on the particular device description elements see the xsd schema provided by `devicedescription_xsd.pdf`.

## 6.4.1 Defining types

The Types section contains type definitions for [bitfields](#), [ranges](#), [arrays](#) resp. [simple structures](#):

### 6.4.1.1 Bitfields

Example:

```

<Types namespace="localTypes">
  <BitFieldType basetype="std:BYTE" name="TBitFieldByte">
    <Component identifier="Bit0" type="std:BOOL">
      <Default>FALSE</Default>
      <VisibleName name="local:Bit0">Bit0</VisibleName>
    </Component>
    <Component identifier="Bit1" type="std:BOOL">
      <Default>FALSE</Default>
      <VisibleName name="local:Bit1">Bit1</VisibleName>
    </Component>
  </BitFieldType>
</Types>

```

This example defines two bits. The base type is byte. The I/O mapping tab in CODESYS will show two bit addresses and 1 byte address.

```

<Parameter ParameterId="33554433" type="localTypes:TBitFieldByte">
  <Attributes channel="input" download="true" offlineaccess="readwrite" />
  <Default />
  <Name name="local:Input1_1">Digital Input</Name>
</Parameter>

```

For defined parameters the type must be the namespace of the type section and name of the specified defined type.

### 6.4.1.2 Range types

The range type defines a subrange of a given base type. For example UDINT has a normal range from 0 to 4294967296. With the range type the maximum values are limited.

Example:

```

<RangeType basetype="std:UDINT" name="TRange">
  <Min>16#0</Min>
  <Max>16#1FFFFFFF</Max>
  <Default>0</Default>
</RangeType>

```

### 6.4.1.3 Array types

Parameters can also be defined as arrays with “std:ARRAY[0..10] OF DWORD”. In the I/O configuration tab of the programming system however only one line will be available and for this reason only one variable could be mapped to the complete array. So it is a better solution to define the special array type. Up to three dimensions are possible and in the mapping tab for each element a separate line will be shown.

Example:

```
<ArrayType name="TestArray" basetype="std:DWORD">
  <FirstDimension>
    <LowerBorder>0</LowerBorder>
    <UpperBorder>10</UpperBorder>
  </FirstDimension>
</ArrayType>
<Parameter ParameterId="30" type="localTypes:TestArray">
  <Attributes offlineaccess="readwrite" download="true" />
  <Default>12</Default>
  <Default>23</Default>
  <Default>34</Default>
  <Default>35</Default>
  <Name name="local:NONE">Array</Name>
</Parameter>
```

A new parameter with the defined array type is created and also default values are set to the first four elements.

#### 6.4.1.4 Simple structures

The structure type could define structures like in IEC:

Example:

```
TYPE TFMMU:
  STRUCT
    GlobStartAdr:DWORD;
    Length:UINT;
  END_STRUCT
END_TYPE
```

```
<StructType name="TFMMU">
  <Component identifier="globstartadr" type="std:DWORD">
    <Default>8000</Default>
    <VisibleName name="local:globstartadr">Global Start Address</VisibleName>
  </Component>
  <Component identifier="length" type="std:uint">
    <Default>1</Default>
    <VisibleName name="local:length">Length</VisibleName>
  </Component>
</StructType>
<Parameter ParameterId="32" type="localTypes:TFMMU">
  <Attributes offlineaccess="readwrite" download="true" />
  <Name name="local:NONE">Structure</Name>
</Parameter>
```

With this example a parameter with a structure is added. In the generic device configuration view all sub elements will be shown in separate lines.

#### 6.4.2 Defining strings for localization

In order to provide device specific strings in multiple languages, e.g. names and descriptions of parameters, the respective language entries can be added to the device description file:

Example:

```
<ParameterSection>
  <Name name="local:GeneralParameters">General Parameters</Name>
  <Parameter ParameterId="65792" type="std:BOOL">
    <Attributes download="true" offlineaccess="read" />
    <Default>FALSE</Default>
```



```

    <Name name="local:SlaveOptional">Slave optional</Name>
    <Description name="local:SlaveOptional">Slave optional</Description>
  </Parameter>
</ParameterSection>
<Strings namespace="local">
  <Language lang="en">
    <String identifier="SlaveOptional">Slave optional (en)</String>
  </Language>
  <Language lang="de">
    <String identifier="SlaveOptional">Slave optional (de)</String>
  </Language>
</Strings>

```

In this example a parameter section "General Parameters" is created and one configuration parameter is added. The name attribute for the section name, parameter name and description is linking to a string table for localization. Two languages are added in preparation for getting translated. The name attribute "local:SlaveOptional" is composed of the namespace "local" and the string identifier "SlaveOptional". So only the value for the string element has to be translated. The value of Name or Description will be the default string if the currently selected language in CODESYS is not found in the string table.

### 6.4.3 Defining files and adding icons and images

Like strings also files can be localized. This is done in a special "files" section.

Example:

```

<Files namespace="local">
  <Language lang="en">
    <File fileref="local" identifier="PCICard">
      <LocalFile>PCICard.ico</LocalFile>
    </File>
  </Language>
</Files>

```

The file will be used as an icon in the device tree. This is defined in the DeviceInfo element.

Example:

```

<DeviceInfo>
  <Name name="local:TypeName">SercosIII Master</Name>
  <Description name="local:typedescription">Sercos III Master</Description>
  <Vendor name="localStrings:_3S">3S - Smart Software Solutions GmbH</Vendor>
  <OrderNumber>1</OrderNumber>
  <Icon name="local:PCICard">PCICard.ico</Icon>
</DeviceInfo>

```

With the "Icon" element the icon file is linked to the device. The value PCICard.ico is the default value for the case that the language cannot be found. The "name" attribute is a combination of the namespace and identifier of the file section.

### 6.4.4 Defining the device itself (identification, connectors, driver, parameters)

The following chapters describe sub-sections of section <Device> within the device description file:

- Device, general settings (6.4.4.1)
- DeviceIdentification (6.4.4.2)
- DeviceInfo (6.4.4.3)
- Driverinfo (including libraries and function block handling, 6.4.4.4)
- Connector (6.4.4.5)

- Functional (for child objects, 6.4.4.7)
- ParameterSection, HostParameterSet, DeviceParameterSet, Parameter (6.4.4.6)
- Compatible Versions (6.4.4.8)

#### 6.4.4.1 Device

Attributes for general settings for the device might be assigned to the <Device> tag.

Example:

```
<Device showParamsInDevDescOrder="true">
```

XML-Tag	Description
showParamsInDevDescOrder	Boolean; If TRUE, the generic device configuration view and I/O mapping view will show the parameters in the same order as they are listed in the device description. If FALSE (default) the parameters are listed according to their IDs.

#### 6.4.4.2 Device identification

This section below a <device> specifies the device identification. The identification is checked for consistency when a connection with a runtime system is established. For this reason these entries must match the corresponding values in the runtime system.

Example:

```
<DeviceIdentification>
  <Type>4096</Type>
  <Id>0000 0001</Id>
  <Version>3.0.2.0</Version>
</DeviceIdentification>
```

Type	Device type: 0000 (0x0000)      special type: empty slot 4096 (0x1000)      programmable devices 4097 (0x1001)      3S special devices (OfflineVisuClient or similar) 4098 (0x1002)      safety controllers 4099 (0x1003)      drives 5000 (0x1004)      parameterizable devices  ID number: High word: VendorId (allocated to OEMs by 3S). 3S = 0x0000  Low word: OEM-based controller Id (may be a consecutive number)
Version	Version of the controller or the target description. The version is verified as follows when a connection with the controller is established.  <b>Version:                    4.3.2.1</b> Main version:            4 Subversion:              3 Service pack version:   2 Patch version:          1

- a) Different main versions of the device description and the runtime system are generally not compatible and cannot be used together
- b) While different subversion are compatible in principle, a warning is issued because changes in the runtime system may lead to semantic changes or changes in behaviour
- c) Different service pack numbers are compatible, since only new interfaces were added, for example. Any changes in existing interfaces would lead to incompatibility!
- d) Different patch numbers are compatible, since only changes in the target description were made, without affecting the controller

This numbering scheme must be taken into account for all future changes in the target description or the runtime system.

In the runtime system the DeviceIdentification values are set in the SysTarget component. This component offers interfaces for accessing the DeviceIdentifications.

Alternatively, the values may be defined in the general header file of runtime system sysdefines.h. In this case the associated values are automatically set during compilation of the SysTarget component. This has the advantage that the target component only has to be parameterized, not replaced.

The defines for the file sysdefines.h are listed below:

```
#define SYSTARGET_DEVICE_TYPE SYSTARGET_TYPE_PROGRAMMABLE

#define SYSTARGET_VENDOR_ID      0x0000
#define SYSTARGET_DEVICE_ID     0x0001

#define SYSTARGET_DEVICE_VERSION 0x03000301
```

#### 6.4.4.3 Device info

In this section, below a <device>, some general information and descriptions of the device can be specified.

Example:

```
<DeviceInfo>
<Name name="local:typename">CODESYS Control for Win32</Name>
  <Description name="local:typedescription">A CODESYS V3 Soft PLC for
Win32</Description>
  <Vendor name="local:3S">3S - Smart Software Solutions GmbH</Vendor>
  <OrderNumber>xxx</OrderNumber>
</DeviceInfo>
```

XML-Tag	Description
Name	Name of the device that appears in the Properties dialogue for the device in the CODESYS
Description	Description of the device; this is also displayed in the Properties dialogue
Vendor	Manufacturer name
OrderNumber	Product number under which the device can be ordered from the manufacturer

**Note:** For complete documentation on the particular device description elements see the xsd schema provided by [devicedescription\\_xsd.pdf](#).

These values are only used for display in the device tree.

During online scanning of devices the device name and the manufacturer's name are displayed. The values are read online from the run-time system, rather than being taken from the device description. In the runtime system the values can be set through the following defines in the file sysdefines.h:

```
#define SYSTARGET_DEVICE_NAME      "CODESYS Control for Win32"
#define SYSTARGET_VENDOR_NAME     "3S - Smart Software Solutions GmbH"
```

#### 6.4.4.4 Driver info

A device is typically operated by a driver. In this section, below a <device>, some general information for the driver can be specified.

Example:

```
<DriverInfo needsBusCycle="false" UpdateIosInStop="true"
  StopResetBehaviour="SetToDefault">
  <RequiredLib libname="IoStandard" vendor="System" version="3.1.2.0"
    identifier="iostandardlib" />
</DriverInfo>
```

XML-Tag	Description
needsBusCycle	Specifies, if the device needs a separate bus cycle. If this attribute is TRUE, the corresponding IO-driver get a cyclic call to is IoDrvStartBusCycle() interface method. The context of this call can be specified by the user (see chapter 7.2.2).
UpdateIosInStop	If TRUE, the option "Update IO while in stop" in dialog "PLC settings" of the device editor will be activated.
StopResetBehaviour	Defines the default selection for option "Update IO while in stop" in dialog "PLC settings" of the device editor.
RequiredLib	Defines a library that is added when a device is added in the project. The IO standard is required by the IO configuration and must be compatible with the controller implementation. An IoStandard.library for the device must therefore be specified here Instead of a fix library name a library placeholder (attribute "placeholderlib") can be specified here. This will effect that, when the device is included to the project, each the currently available corresponding customer-specific library will be included.
defaultBusCycleTask	Defines the default bus cycle task; example: defaultBusCycleTask="MainTask"  <b>Note:</b> This setting overwrites a possibly available "useSlowestTask" resp. the basic default setting (=task with the shortest cycle time will be used as bus cycle task).
useSlowestTask	If TRUE, the task with the longest cycle time will be used as bus cycle task. Otherwise the task with the shortest cycle time or – if defined – the "defaultBusCycleTask" (see above) will be used.
RequiredCyclicTask	Specifies a cyclic task which shall automatically be added to the Task Configuration, when the device is inserted in the device tree. The task is defined by the attributes "taskname" and "priority", see example below.
taskpou	Setting "taskpou" allows to specify a POU, which should be added to the configuration of the above defined task. Regard that this only means, that the POU name will be entered in the task configuration, not however that the POU itself will be created in the project !  Example: <RequiredCyclicTask taskname="Ethercat" priority="2" cycletime="t#6ms"> <taskpou>Test_Pou1</taskpou>

XML-Tag	Description
	<taskpou>Test_Pou2</taskpou> </RequiredCyclicTask>
RequiredExternalEventTask  taskpou	Corresponds to the above described "RequiredCyclicTask" setting; for automatically adding an external event task to the task configuration when inserting the device  Example:  <RequiredExternalEventTask taskname="EthercatEvent" priority="2"> <taskpou>Test_Pou3</taskpou> <taskpou>Test_Pou4</taskpou> </RequiredExternalEventTask>

**Note:** For complete documentation on the particular device description elements see the xsd schema provided by devicedescription\_xsd.pdf.

**Note:** If libraries, having the same library name and vendor name, are requested by a father object as well as by a child object (a child object is an object indented below the father object in the device tree), only that library version will be included which is requested by the father object.

#### 6.4.4.4.1 Adding libraries and function blocks

Example:

```
<DriverInfo needsBusCycle="false">
  <RequiredLib identifier="IoDrvSercos3" libname="IoDrvSercos3"
placeholderlib="IoDrvSercos3" vendor="3S - Smart Software Solutions GmbH"
version="3.1.2.0">
    <FBInstance basename="$(DeviceName)" fbname="Sercos3Slave">
      <Initialize methodName="Initialize" />
    </FBInstance>
  </RequiredLib>
</DriverInfo>
```

With the DriverInfo and RequiredLib elements it is possible to get libraries added to the project automatically as soon as the user adds a specific device in the configuration tree. The library is selected by libname, vendor and version and must match with the information from the library file.

The identifier element selects the namespace as it is shown in the library manager of the project.

With the element placeholderlib an unique name is defined. The placeholder itself is defined in the device description of the PLC in the target settings section and it will override the libname, vendor and version attribute. Optionally an attribute "loadAsSystemLibrary=true|false" can be added: If it is "false", the library will be inserted as a "normal" library (black writing). If it is "true" (default) or missing, the library will be inserted as a "system library" (grey writing).

With this placeholder mechanism it is possible to select a specific library version by only changing the PLC device description. All other description files (master, slaves) do not need to be changed.

The FBInstance entry effects that automatically a function block will be created for each device. The basename \$(DeviceName) will be replaced by the actual device name in the device tree. Also combinations like \$(DeviceName)\_abc are possible. If the device is renamed in the device tree for example to "Drive1" then the function block will be named Drive1\_abc.

With "Initialize" a special method could be declared, which will be called automatically when downloading the project.

The declaration for the Initialize method must be exactly as follows:

```
METHOD Initialize : UDINT
VAR_INPUT
  wModuleType : UINT;
  dwInstance : UDINT;
  pConnector : POINTER TO IoConfigConnector;
```

[END\\_VAR](#)

For complete documentation see the information given by the [xsd schema](#).

#### 6.4.4.5 Defining connectors

In this section, below a <device>, a connector for the device can be specified. See chap. 6.3.1 for general information on connectors.

See in the following a description on the following cases:

- [Slave with 1 connector](#)
- [Master with 2 connectors](#)
- [Multiple parent connectors](#)

##### 6.4.4.5.1 Slave with 1 connector

Slaves with no module typically have only one connector. In the device description there is only one connector with role "child".

Example (EtherCAT):

```
<Connector connectorId="1" explicit="false" hostpath="-1" interface="Common.Ethercat"
moduleType="65" role="child">
  InterfaceName name="local:EtherCAT">EtherCAT</InterfaceName>
  Slot allowEmpty="false" count="1" />
</Connector>
```

For such devices the connector is a slot with allowEmpty false and a count of 1.

The connectorId is 1 as it has only one connector. The hostpath for a child is not important but is included for completeness. The moduleType 65 defines a EtherCAT slave. All numbers below 32768 are maintained by 3S-Smart Software Solutions GmbH. The interface "Common.Ethercat" must match with the parent connector of the master. The part "Common" is used by 3S-Smart Software Solutions GmbH. Customers use their own unique identification here.

##### 6.4.4.5.2 Master with 2 connectors

A master device description typically has 2 connectors. One of them is a child connector for the PLC and the other is designated for the slave devices.

Example (EtherCAT master):

```
<Connector connectorId="1" explicit="false" hostpath="-1" interface="Common.PCI"
moduleType="257" role="child">
  <InterfaceName name="local:PCI_Interface">PCI-Bus</InterfaceName>
  <Slot allowEmpty="false" count="1" />
</Connector>
<Connector connectorId="2" explicit="false" hostpath="1" interface="Common.Ethercat"
moduleType="64" role="parent">
  <InterfaceName name="local:EtherCAT_Interface">EtherCAT</InterfaceName>
  <Var max="125"/>
</Connector>
```

Here the child connector with connectorId 1 is for slaves. It has a slot type with count 1 and allowEmpty set to false.

The second connector is a parent connector with connectorId 2 as it is the second one. Hostpath is 1 because the connectorId of the child connector is 1. This controls the link between parent and child connectors.

The Var type allows 125 devices to be added to the master.

##### 6.4.4.5.3 Multiple parent connectors

Example (Sercos 3 slave with modules):

First there is a connector to the master:

```
<Connector connectorId="0" explicit="false" hostpath="-1"
interface="Common.SercosIIIMaster" moduleType="71" role="child">
  InterfaceName name="local:SERCOSIII">SERCOS III</InterfaceName>
  Slot allowEmpty="false" count="1" />
</Connector>
```

A fixed module, defined in the same device description, will automatically be inserted below the slave when the slave gets added to the configuration:

```
<Connector connectorId="1" explicit="false" hostpath="0"
interface="Common.SercosIIISlave" moduleType="74" role="parent">
  <InterfaceName name="local:SercosIIIModule_23003886">SERCOS III
Module</InterfaceName>
  <Fixed>
    <Module>
      <LocalModuleId>23003886</LocalModuleId>
    </Module>
  </Fixed>
</Connector>
```

A second parent connector is used for the modules:

```
<Connector connectorId="2" explicit="false" hostpath="0"
interface="Common.SercosIIISlave" moduleType="74" role="parent">
  <InterfaceName name="local:SercosIIIModule">SERCOS III Module</InterfaceName>
  <Var max="255" />
</Connector>
```

The connectorId for the child is 0 and therefore the hostpath for both parent connectors also is 0.

XML Tag	Description
moduleType	Connector type
interface	Interface type; defines under which interface type this device can be added (role="child") or which interface can logically be added (role="child")
role	Specifies the purpose of the interface type
explicit	Specifies, whether the connector should appear as a separate object in the device tree (i.e. the connector is visible to the user as an object).
alwaysmapping	If this attribute is set to "true", all in- and output variables will be updated automatically in each task cycle regardless if they are used or not.
hideInStatusPage	If set to true (default false and optional) then no status information will be shown for this connector in the device editor.
updateAllowed	If FALSE, the device cannot be updated in the projects device tree (only to be used for child connectors!).

**Note:** For complete documentation on the particular device description elements see the xsd schema provided by devicedescription\_xsd.pdf.

#### 6.4.4.6 Defining parameters and parameter sections

See chap. 6.3.2 for general information on parameters in the device description.

Parameters and parameter sections can be defined inside a parameter set.

Parameter lists in [connectors](#) (see chap. 6.4.4.5) are referred to as **HostParameterSet**, because they are loaded to the responsible host. See below for an example. In contrast a [CODESYS-configurable PLC device with local I/Os](#) (see chap. 6.4.4) can define a **DeviceParameterSet** (to be defined within the <device> section). This parameter list is directly transferred to the device and is independent of the connectors used. However, in case of devices with connectors like master or slaves the DeviceParameterSet will be ignored by CODESYS.

**Parameter sections** just serve for grouping parameters for organizational purposes.

Example of a parameter section:

```
<ParameterSection>
  <Name name="local:GeneralParameters">General Parameters</Name>
  <Parameter ParameterId="65792" type="std:BOOL">
    <Attributes download="true" offlineaccess="read" />
    <Default>FALSE</Default>
    <Name name="local:SlaveOptional">Slave optional</Name>
    <Description name="local:SlaveOptional">Slave optional</Description>
  </Parameter>
</ParameterSection>
```

In this example a parameter section "General Parameters" is created and one configuration parameter is added. The **name** attribute for the section name, parameter name and description is linking to a string table for localization (see also chap. 6.4.2). The value of Name or Description will be the default string if the currently selected language in CODESYS is not found in the string table.

The **parameter id** is a unique number within the device description and is depending on the module type of the connector (values <32768 are maintained by 3S).

In order to get parameters accessible by the "intellisense" function in the programming system, and by its **full namespace**, as from compiler version >= 3.5.1.0 the attribute "functional" can be used in the parameter description. It effects that the CODESYS compiler instead of a parameter description at every read access will create some external function calls supported by a runtime system interface. **Regard** that this functionality is only possible for simple datatypes. Structures and Arrays are not supported. External functions will be created by the compiler, which are supported by an interface in the runtime system.

Example:

```
<Parameter ParameterId="1234" type="std:REAL">
  <Attributes channel="none" download="false" functional="true"
offlineaccess="readwrite" onlineaccess="readwrite" />
  <Default>111</Default>
  <Name name="local:Id1234">Velocity</Name>
  <Description name="local:Id1234.Desc">Test parameter</Description>
</Parameter>
```

The parameter can be accessed now by its namespace, e.g. if this parameter is a member of the device "Drive":

```
Drive.Velocity := 1.23;
```

Only simple datatypes are possible (BOOL, BYTE, WORD, DWORD, SINT, INT, DINT, LWORD, LINT, REAL, LREAL). ENUMs are possible because they are handled as INT. Structures and arrays are not supported.

The compiler generates instead of a parameter description at every read access the following external function call:

Only for BOOL types

```
RTS_IEC_BOOL ParamGetBit(RTS_IEC_DWORD dwParameterId, DWORD dwBitNr);
```

For all other simple datatypes. XXX is the IEC datatype like WORD

```
RTS_IEC_XXX ParamGetXXX(RTS_IEC_DWORD dwParameterId);
```



The ID of the parameter will be transmitted in dwParameterId.

For every write access on a parameter one of the following external functions is called:

Only for BOOL types

```
VOID ParamSetBit(RTS_IEC_DWORD dwParameterId, DWORD dwBitNr, BOOL bValue);
```

For all other simple datatypes separated for 4 and 8 Byte values

```
VOID ParamSet4Byte(RTS_IEC_DWORD dwParameterId, DWORD dwValue);
```

```
VOID ParamSet8Byte(RTS_IEC_DWORD dwParameterId, LWORD lwValue);
```

With version 3.5 SP3 it is also possible to use the enumerations as the enum value instead of the integer value.

Also structured parameters are possible when using parameter sections.

Example if parameter is stored in a section:

```
<ParameterSection>
  <Name name="local:SectionName">Section</Name>
  <ParameterSection>
    <Name name="local:SubSectionName">SubSection</Name>
  </ParameterSection>
</ParameterSection>
```

Then the access will be Drive.Section.SubSection.Velocity := 1.23;

Example for a **host parameter set** (sub-section below <connector>):

```
<HostParameterSet>
  <Parameter ParameterId="0" type="std:WORD">
    <Attributes channel="none" download="true" functional="false" offlineaccess="read"
onlineaccess="read" />
    <Default>0</Default>
    <Name name="local:Id0">NumberOfInputs</Name>
    <Description name="local:Id0.Desc">Number of input channels</Description>
  </Parameter>
  ...
  <Parameter ParameterId="458759" type="std:BOOL">
    <Attributes channel="none" download="false" functional="false" offlineaccess="read"
onlineaccess="none" />
    <Default>TRUE</Default>
    <Name name="local:Id458759">AutoClearSupported</Name>
    <Description name="local:Id458759.Desc">auto-clear supported by
master</Description>
  </Parameter>
  ...
</HostParameterSet>
```

XML Tag	Description
ParameterId	Identification number of the parameter; see 6.3.2 For defining the NetX configuration dialog in the device editor of the programming system use IDs "1879048194" and "1879048195" as shown in the example below.
AutoClearSupported	If TRUE

**Note:** For complete documentation on the particular device description elements see the xsd schema provided by devicedescription\_xsd.pdf.

Special Parameter entries for NetX configuration dialog:

The following entries below a <HostParameterSet> of a <Connector> enable the user to select the NetX chip and the communication channel on this chip via the "NetX Configuration" dialog for a NetX Master in the device editor of the programming system:

Example:

1. The following entry establishes a field named "Slot" in the configuration dialog and the selection list next to it will offer the available slots, i.e. the available NetX cards as defined by an enumeration "eSlot":

```
<Parameter ParameterId="1879048194" type="local:eSlot">
  <Attributes channel="none" download="true" functional="false"
offlineaccess="readwrite" onlineaccess="read" />
  <Default>0</Default>
  <Name name="local:Id1879048194">Slot</Name>
  <Description name="local:1879048194.Desc">Slot of the NetX chip</Description>
</Parameter>
```

2. The following entry establishes a field named "NetX Com Channel" in the configuration dialog and the selection list next to it will offer the available communication channels on the NetX card as defined by an enumeration "NetXComChannel":

```
<Parameter ParameterId="1879048195" type="local:NetXComChannel">
  <Attributes channel="none" download="true" functional="false"
offlineaccess="readwrite" onlineaccess="read" />
  <Default>0</Default>
  <Name name="local:Id1879048195">NetX Com Channel</Name>
  <Description name="local:1879048195.Desc">NetX Communication channel</Description>
</Parameter>
```

#### Parameters in a Device Parameter Set for defining input and output channels:

A device parameter set (Note: only usable with a CODESYS-configurable PLC device with local I/Os !) can contain fix settings for input and output channels:

Example:

```
<DeviceParameterSet fixedInputAddress="%IB10" fixedOutputAddress="%QB10">
  <Parameter ParameterId="1" type="std:BYTE">
    <Attributes channel="output" download="true" offlineaccess="readwrite"
noManualAddress="false"/>
    <Default>0</Default>
    <Name name="local:None">Output</Name>
    <Description name="local:None">Output</Description>
  </Parameter>
  <Parameter ParameterId="2" type="std:BYTE">
    <Attributes channel="input" download="true" offlineaccess="readwrite"
noManualAddress="true"/>
    <Default>0</Default>
    <Name name="local:None">Input</Name>
    <Description name="local:None">Input</Description>
  </Parameter>
</DeviceParameterSet>
```

XML-Tag	Description
fixedInputAddress	Start address of the first input channel
fixedOutputAddress	Start address of the first output channel

XML-Tag	Description
ParameterId	Identification number of the parameter; see 6.3.2
type	datatype of the parameter
Attribute channel	Channel type of the parameter. Possible values are "output" and "input"
Attribute download	Set this (optional) attribute, if no i/o configuration should be downloaded to the device (ie. The device does not support the IoManager)
Attribute offlineaccess	Boolean value. If set, secure online mode is by default activated for the device. Secure online mode means that the user will have to confirm most of the online commands before they become effective.
Attribute noManualAddress	If this attribute is set to TRUE, no manual change of the address of the input or output channel is possible

#### 6.4.4.7 Functional, defining child objects

In this section below a <device> any child objects of the device can be described. A child object is an object indented below the father object in the device tree.

Example:

```
<Functional>
  <ChildObject>
    <ObjectGuid>8cee4e-ac7a-4fbd-9415-bfb2d98668ab</ObjectGuid>
    <ObjectName>Plc Logic</ObjectName>
  </ChildObject>
  <Attribute name="StdCommunicationLink">True</Attribute>
  <Attribute name="NoIoDownload">False</Attribute>
  <Attribute name="SecureOnlineMode">False</Attribute>
</Functional>
```

XML-Tag	Description
ObjectGuid	Typeguid of the factory, that is capable of creating the object.
ObjectName	The name of the object as it should appear in the device tree.
Attribute "StdCommunicationLink"	Boolean value. Must be set to true, if the device has a CODESYS communication stack.
Attribute "NoIoDownload"	Set this (optional) attribute, if no i/o configuration should be downloaded to the device (ie. The device does not support the IoManager)
Attribute "SecureOnlineMode"	Boolean value. If set, secure online mode is by default activated for the device. Secure online mode means that the user will have to confirm most of the online commands before they become effective.

**Note:** For complete documentation on the particular device description elements see the xsd schema provided by `devicedescription_xsd.pdf`.

#### 6.4.4.8 Compatible Versions

Here you can define, to which older versions the current device version is compatible. When the CODESYS user opens a project using one of the here defined device versions, he will be prompted to update to the new version.

Examples:

```
<CompatibleVersions>
  <Version>3.4.2.0</Version>
</CompatibleVersions>
<CompatibleVersions>
  <Version>3.4.2.10</Version>
</CompatibleVersions>
```

or

```
<CompatibleVersions>
  <Version>3.4.*.*</Version>
</CompatibleVersions>
```

Wildcards can be used. Example: "3.4.\*.\*" means that the current version is compatible to all versions of 3.4. When using a V3.3 device however in this case no update proposal will be made.

### 6.4.5 Target description

The term "target" always refers to the whole controller or PLC. The PLC node in the I/O-configuration is defined by the target description, specifying all controller properties such as:

- which processor is used by the controller and therefore which code generator should be used in CODESYS
- which compiler settings should be used for the CODESYS compiler
- what types and number of IEC tasks are possible on the controller
- limit and default values for memory areas of the IEC program (e.g. retain, code, data)

The target settings are also described in an XML file following the DeviceDescription-1.0.xsd scheme, although there is a node **<ExtendedSettings>** that differs from this scheme since it only applies to the target device descriptions.

These target settings also may define whether a setting is visible in the user interface of the programming system and whether it is editable there (property "access").

The individual sections of the **<ExtendedSettings>** node are described in the following chapters.

#### 6.4.5.1 Target settings

This sub-section of **<ExtendedSettings>** describes the target settings of the **<device>**.

```
<ts:TargetSettings xmlns:ts="http://www.3s-software.com/schemas/TargetSettings-0.1.xsd">
</ts:TargetSettings>
```

##### 6.4.5.1.1 Runtime features

Sub-section "runtime\_features" of **<ts:TargetSettings...>** describes specific properties of the runtime system.

```
<ts:section name="runtime_features">
  <ts:setting name="cycle_control_in_iec" type="boolean" access="edit">
    <ts:value>1</ts:value>
  </ts:setting>
  <ts:section name="boot_project">
    <ts:setting name="boot_project_on_download_tag" type="boolean"
access="edit"><ts:value>1</ts:value>
    ...
  </ts:section>
</ts:section>
```

XML Tag	Description
<b>ts:section "runtime_features"</b>	Specific properties of the runtime system
ts:setting "cycle_control_in_iec"	1=Task FB is always called, irrespective of the application status. The task FB uses this switch to check the status of the task and no longer calls the

XML Tag	Description
(type="boolean")	task POU's in stop state, for example 0=CmpApp in the runtime system must evaluate the status of an application and call the task FB accordingly
ts:setting "cycle_control_version_2"	1= 1, __sys_rts__cycle__2() is called in the runtime instead of __sys_rts__cycle(). The runtime calls the taskcycle code independently of whether the task is stopped or not and the generated code controls which parts have to be executed (eg. update I/O's but don't execute the Task POU's) This increases performance, because no strings but only a handle to the corresponding task is transmitted to this function.
ts:setting "dynamic_memory_supported" (type="boolean")	1=The "Dynamic memory settings" tab in the Application properties is available 0=The "Dynamic memory settings" tab is hidden
ts:setting "support_user_check_functions" (type="boolean")	1=The menu command for adding "POUs for implicit checks" to an application is available in the programming system 0=The user cannot add own check functions, because these are provided by a hidden library
ts:setting "optimized_online_change" (type="Boolean")	1=online change (o.c.) with limited jitter by an optimized o.c. procedure in the runtime system (as from V3.4.3.0); o.c. code is divided to three POU's containing the following: <ul style="list-style-type: none"> <li>- initialization code without any side effects on the iec task (can be done in communication task)</li> <li>- copy code for already initialized data (can be repeated)</li> <li>- online change code for the rest (must run without interrupt)</li> </ul> Some examples for online change events, which will now produce no jitter: <ul style="list-style-type: none"> <li>- change of an existing array size (even really big ones)</li> <li>- add new data of any size</li> <li>- call new functions (a complete new call tree will have no effect on the running iec-tasks)</li> </ul> <b>Note however:</b> If a function block interface is changed, the jitter will depend on the initialization code of this function block, and of its derived function blocks. 0=online change with jitter as done up to V3.4.3.0; this setting might be used, if jitter does not matter and you prefer to use the established "old" online change procedure, e.g. in order to avoid problems due to required function block initializations
ts:section "boot_project"	<b>subsection</b> of section "runtime_features" for boot application settings; defines whether the "Properties/Boot applications settings" for an application object are available at all in the programming system and which are the default settings

XML Tag	Description
ts:setting "boot_project_on_download_tag" (type="boolean")	1=The "Boot application settings" are available in the application properties dialog 0=The settings are switched off and not available in the dialog
ts:setting "support_boot_project_on_online_change" (type="boolean")	0= setting "Implicit boot application on online change" is switched off and not available in the dialog (1=available is default)
<b>ts:setting "boot_project_on_download_default" (type="boolean")</b>	<b>1=setting "Implicit boot application on download" is activated per default; i.e. at each download automatically a boot application is created on the target</b> <b>0=setting is deactivated</b>
ts:setting "boot_project_on_online_change_default" (type="boolean")	1= setting "Implicit boot application on online change" is activated per default; i.e. at each online change automatically a boot application is created on the target 0=setting is deactivated
ts:setting "remind_boot_project_default" (type="boolean")	1= setting "Remind boot application on project close" is activated per default 0=setting is deactivated

The following settings (type="boolean") serve to control the visibility of commands of the categories "OnlineCommands" and "Breakpoints". TRUE -> command is available

XML Tag	Description
ts:setting "only_explicit_features_supported"	This setting acts as an overall switch. If "TRUE", only the commands explicitly enabled by the following settings will be available. If "FALSE", all features will be supported by default and only the explicitly disabled ones will no be available.
ts:setting "source_download_allowed"	Source download Source download to connected device
ts:setting "online_change_supported"	Online Change for active application
ts:setting "boot_application_supported"	Create boot application
ts:setting "force_variables_supported"	Force values Unforce values Release ForceList Add All Forces to WatchList
ts:setting "write_variables_supported"	Write values
ts:setting "connect_device_supported"	Connect to Disconnect from Reset origin device
ts:setting "file_transfer_supported"	File download File upload
ts:setting "core_application_handling_supported"	Download active application Login active application

XML Tag	Description
d"  : This setting will be needed for "tiny" devices that do not have a runtime system.	Logout active application Reset active application Reset warm active application Reset cold active application Reset origin active application Start active application Stop active application Simulation active application Single Cycle active application
ts:setting "breakpoints_supported"	New Breakpoint Run to Cursor Set next statement Show current statement Step Into Step Out Step Over Toggle Breakpoint Flow Control (Note that this feature can be disabled by the following C compiler switch in the runtime in the CmpAppBP component: "APPBP_DISABLE_FLOWCONTROL")
ts:setting "conditional_breakpoints_supported"	Used in the Breakpoints.BreakpointDialog to hide/show the tab "Condition". (Type: bool)
ts:setting "max_number_of_apps"	Used in the ApplicationObject-Plugin to check the maximum numbers of allowed applications. If the number of applications reached the specified limit, the command to add another application will be invisible. A setting of "-1" indicates, that there is no upper limit for the number of applications. (Type: int)

#### 6.4.5.1.2 Memory layout

This sub-section of <ts:TargetSettings...> describes the memory layout of the controller. It contains all information necessary for generating code for the compiler.

**Note:** With compiler version >= 3.4.2.0 all memory settings of a parent application are also set on the child application.

```
<ts:section name="memory-layout">
  <ts:setting name="memory-size" type="integer" access="visible">
    <ts:value>1280</ts:value>
  </ts:setting>
  <ts:setting name="input-size" type="integer" access="visible">
    <ts:value>1280</ts:value>
  </ts:setting>
  <ts:setting name="output-size" type="integer" access="visible">
    <ts:value>1280</ts:value>
  </ts:setting>
  <ts:setting name="retain-size" type="integer" access="visible">
    <ts:value>1280</ts:value>
  </ts:setting>
  <ts:setting name="retain-in-own-segment" type="boolean" access="visible">
    <ts:value>1</ts:value>
  </ts:setting>
  <ts:setting name="pack-mode" type="integer" access="visible">
    <ts:value>8</ts:value>
  </ts:setting>
</ts:section>
```

```

<ts:setting name="stack-alignment" type="integer" access="visible">
  <ts:value>4</ts:value>
</ts:setting>
<ts:setting name="code-segment-size" type="integer" access="visible">
  <ts:value>65536</ts:value>
</ts:setting>
<ts:setting name="data-segment-size" type="integer" access="visible">
  <ts:value>65536</ts:value>
</ts:setting>
<ts:setting name="code-segment-prolog-size" type="integer" access="visible">
  <ts:value>16</ts:value>
</ts:setting>
<ts:setting name="additional-areas" type="boolean" access="visible">
  <ts:value>1</ts:value>
</ts:setting>
<ts:setting name="max-stack-size" type="integer" access="visible">
  <ts:value>40000</ts:value>
</ts:setting>
<ts:setting name="max-stack-size-external-call" type="integer" access="visible">
  <ts:value>3000</ts:value>
</ts:setting>
<ts:section name="areas">
  <ts:setting name="number" type="integer" access="visible">
    <ts:value>1</ts:value>
  </ts:setting>

  <ts:section name="area_0">
    <ts:setting name="area_flags" type="integer" access="visible">
      <ts:value>0x10</ts:value>
    </ts:setting>
    <ts:setting name="flags" type="integer" access="visible">
      <ts:value>0xffBf</ts:value>
    </ts:setting>
    <ts:setting name="minimal-area-size" type="integer" access="visible">
      <ts:value>262144</ts:value>
    </ts:setting>
    <ts:setting name="maximal-area-size" type="integer" access="visible">
      <ts:value>262144</ts:value>
    </ts:setting>
    <ts:setting name="allocation-plus-in-percent" type="integer" access="visible">
      <ts:value>0</ts:value>
    </ts:setting>
    <ts:setting name="start-address" type="integer" access="visible">
      <ts:value>0x34020218</ts:value>
    </ts:setting>
  </ts:section>
</ts:section>
</ts:section>

```

XML Tag	Description
<b>ts:section "memory-layout"</b>	Contains information about the runtime system memory layout
ts:setting "memory-size"	Size of the memory segment in bytes (%M.-Addresses)
ts:setting "input-size"	Size of the input segment of the process map (inputs, %I-Addresses)
ts:setting "output-size"	Size of the output segment of the process map (outputs, %Q-Addresses)



XML Tag	Description
ts:setting "retain-size"	Size of the retain segment  IMPORTANT: If the retains are stored in an SRAM or NVRAM in the controller, a value that is 24 bytes less than the value actually available in the controller must be specified here! This is used for saving an ID in the retain memory through which consistency of the retain data with respect to the IEC application can be ensured.
ts:setting "retain-in-own-segment"	1= retain data are stored in their own segment in the controller 0= retain data are stored in the same segment as all volatile data of an IEC application (Default is 1)
<b>as from V3.3!:</b> ts:setting "dynamic-retain"  ts:setting "dynamic-persistent"	1= If you define a retain area with dynamic size (area-flag 4), the size of the retain (persistent) area will be calculated dynamically and allocated according to the actual need. If no retain (persistent) data is needed, no area will be allocated during download.  <b>Note:</b> "retain-in-own-segment" must be 1. "retain-size" must be 0.  If retain (persistent) data is newly needed for download or online change, the size of the area will be calculated once according to the settings in device description (min. area size, allocation plus in percent). For subsequent online changes no additional areas will be allocated, so the memory might be not sufficient even if a full new download would work.
ts:setting "constants-in-own-segment"  type="boolean"	1= constants of type userdef, array or string will be located in a defined code area.  <b>Notes:</b>  1. only constant initial values are allowed for those variables  2. the initial value of these variables will be downloaded as an array of bytes together with the program code  3. there is no code for initialization of these variables, i.e. a reset will not change these variables after they e.g. have been changed by wrong usage of pointers.  4. The setting with compiler version >=3.4.4.10 effects that all variables that are initialized with attribute 'blob_init_const' get allocated to the area with the constant section flag.  Example: With the following memory-layout in the target settings, all constant arrays will be located to area 1 (same as code), all non-constant variables will be located to area 0: <pre>&lt;ts:section name="memory-layout"&gt;   [...]   &lt;ts:setting name="constants-in-own-segment" type="boolean" access="visible"&gt;     &lt;ts:value&gt;true&lt;/ts:value&gt;   &lt;/ts:setting&gt;   &lt;ts:section name="areas"&gt;     &lt;ts:setting name="number" type="integer" access="visible"&gt;       &lt;ts:value&gt;4&lt;/ts:value&gt;     &lt;/ts:setting&gt;     &lt;ts:section name="area_0"&gt;       &lt;!-- Global data memory --&gt;       &lt;ts:setting name="flags" type="integer" access="visible"&gt;         &lt;ts:value&gt;0xfe9d&lt;/ts:value&gt;       &lt;/ts:setting&gt;       &lt;ts:setting name="minimal-area-size" type="integer" access="visible"&gt;         &lt;ts:value&gt;0x100000&lt;/ts:value&gt;       &lt;/ts:setting&gt;       &lt;ts:setting name="allocation-plus-in-percent" type="integer"         access="visible"&gt;         &lt;ts:value&gt;30&lt;/ts:value&gt;       &lt;/ts:setting&gt;     &lt;/ts:section&gt;     &lt;ts:section name="area_1"&gt;       &lt;!-- Global code and constant memory --&gt;</pre>

XML Tag	Description
	<pre> &lt;ts:setting name="flags" type="integer" access="visible"&gt;   &lt;ts:value&gt;0x0042&lt;/ts:value&gt; &lt;/ts:setting&gt; &lt;ts:setting name="minimal-area-size" type="integer" access="visible"&gt;   &lt;ts:value&gt;0x100000&lt;/ts:value&gt; &lt;/ts:setting&gt; &lt;ts:setting name="allocation-plus-in-percent" type="integer" access="visible"&gt;   &lt;ts:value&gt;30&lt;/ts:value&gt; &lt;/ts:setting&gt; &lt;/ts:section&gt; &lt;ts:section name="area_2"&gt; &lt;!-- RETAIN memory --&gt; &lt;ts:setting name="flags" type="integer" access="visible"&gt;   &lt;ts:value&gt;0x0020&lt;/ts:value&gt; &lt;/ts:setting&gt; &lt;ts:setting name="minimal-area-size" type="integer" access="visible"&gt;   &lt;ts:value&gt;0x1000&lt;/ts:value&gt; &lt;/ts:setting&gt; &lt;ts:setting name="allocation-plus-in-percent" type="integer" access="visible"&gt;   &lt;ts:value&gt;20&lt;/ts:value&gt; &lt;/ts:setting&gt; &lt;/ts:section&gt; &lt;ts:section name="area_3"&gt; &lt;!-- PERSISTENT RETAIN memory --&gt; &lt;ts:setting name="flags" type="integer" access="visible"&gt;   &lt;ts:value&gt;0x0120&lt;/ts:value&gt; &lt;/ts:setting&gt; &lt;ts:setting name="minimal-area-size" type="integer" access="visible"&gt;   &lt;ts:value&gt;0x1000&lt;/ts:value&gt; &lt;/ts:setting&gt; &lt;ts:setting name="allocation-plus-in-percent" type="integer" access="visible"&gt;   &lt;ts:value&gt;20&lt;/ts:value&gt; &lt;/ts:setting&gt; &lt;/ts:section&gt; &lt;/ts:section&gt; &lt;/ts:section&gt; </pre>
<p>ts:setting "online-change-in-own-segment" type="boolean"</p>	<p>1= Online Change (o.c.) code is created as an application in a free code area; during first download two code areas get allocated; at each o.c. the code is allocated to the currently free area. For initialization all function pointers must be rewritten - even the pointers of functions, the code of which has not been changed – because those get shifted to another location. However: Writing these pointers has not to be protected from interrupts; so the part to be protected during initialization does not change, which allows to keep the jitter small (precondition: writing of the pointer must be atomic!).</p> <div style="border: 1px solid black; padding: 5px; background-color: #f0f0f0;"> <p><b>Note:</b> This setting should be combined with the optimized online change mechanism (see setting “optimized_online_change”, chap. 6.4.5.1.1); otherwise there could be jitter effects, because all function pointers have to be reassigned after o.c. This can take some time, in combination with the setting "optimized_online_change" all non-changed functions will be initialized in the communication task without effect on the iec-tasks.</p> </div> <p>Although the double memory of the maximum application size is necessary, this option has two advantages:</p> <ul style="list-style-type: none"> <li>- No extra boot project required</li> <li>- No fragmentation</li> <li>- CRC over code and constants possible even after online change</li> </ul> <p>0= default: Online Change will take place in the same code area</p>
<p>ts:setting "pack-mode"</p>	<p>4= structures are created 4-byte-aligned, i.e.: BYTE on BYTE limits</p>

tech\_doc\_e.doc / V1.2

XML Tag	Description
	<p>WORD on WORD limits  DWORD on DWORD limits  LINT on DWORD limits</p> <p>8= structures are created 8-byte-aligned, i.e.:</p> <p>BYTE on BYTE limits  WORD on WORD limits  DWORD on DWORD limits  LINT on LINT limits</p> <p>[Default=8]</p> <p>Within a structure, bytes are aligned to byte limits, words to word limits etc.  On the other hand a variable of type of a structure will be aligned to the limit of the biggest non-structured data type which is contained.</p>
ts:setting "stack-alignment"	Alignment of the stack during function calls [Default=4]
ts:setting "code-segment-size"	Use this setting if your controller has segmented code. A POU may not grow larger than one code segment, and is never placed over segment borders. No entry or value 0xffffffff is interpreted as « no segmentation ».
ts:setting "data-segment-size"	Use this setting if your controller has segmented data. One variable may not grow larger than one data segment, and one variable is never placed over segment borders. No entry or value 0xffffffff is interpreted as « no segmentation ».
ts:setting "byte-addressing"	<p>TRUE: The compiler and the devices will use byte-addressing mode (ADR(%IW1)=ADR(%IB1))</p> <p>FALSE (default): Word addressing mode is used: (ADR(%IW1)=ADR(%IB2))</p>
ts:setting "bit-word-addressing"	<p>TRUE: Word addressing mode is used on bits; ADR(%IW1)=ADR(%IX1);</p> <p>FALSE (default): byte addressing mode is used: ADR(%IW1)=ADR(%IX2)</p> <hr/> <p><b>Note:</b> Setting "bit-byte-addressing" is obsolete.</p>
ts:setting "additional-areas"	TRUE: additional areas will be used for online change, if code or data exceeds the memory in the specified areas.
ts:setting "code-segment-prolog-size"	<p>If this value is != 0, the specified size at the beginning of each code area is not used for allocation of IEC-Code. Thus the runtime system is free to add specific code in this place.</p> <p>At the moment this setting is only used for ARM-Targets, in this case the prolog contains a jump to the breakpoint handling routine in the runtime system. The code patched for Breakpoints is a jump to offset 0 in the area. Therefore we can avoid Exception handling for handling breakpoints.</p>
ts:setting "address-assignment-guid"	Specifies the GUID which determines the address assignment strategy. The GUID is defined in the plug-in.
<b>as from V3.5 SP 2!:</b> ts:section "static-area"	New possibility to define an area for both: retain and persistent data. If the memory-management contains a static-area, this area will replace any area containing Retain or Persistent data. The only settings that are possible for this area is size and address. The area will then allocate memory for persistent variables and retain variables. Persistent Variables are allocated with rising addresses

XML Tag	Description
	<p>and start at 0. Retain variables are allocated with falling addresses starting at Area-SIZE.</p> <p>Since this setting is evaluated from codesys-versions <math>\geq</math> 3.5.2.0, you should keep the old area definitions in the areas section for compatibility with previous codesys versions. In the order of areas, the static area will be inserted last!</p> <p>Example:</p> <pre data-bbox="600 495 1374 757">&lt;ts:section name="static-area"&gt; &lt;ts:setting name="size" type="integer" access="visible"&gt;   &lt;ts:value&gt;0x10000&lt;/ts:value&gt; &lt;/ts:setting&gt; &lt;ts:setting name="start-address" type="integer" access="visible"&gt;   &lt;ts:value&gt;0xba5eadd&lt;/ts:value&gt; &lt;/ts:setting&gt; &lt;/ts:section&gt;</pre> <p>You should use this possibility, if the device is not suitable for handling several applications with retain variables. Then you can set the size to the maximum possible amount of retain data (minus 24 bytes -&gt; see note for static-area\size).</p>
<p><b>as from V3.5 SP 2!:</b> ts:setting "static-area\size"</p>	<p>This is the size of the static area, this size will be allocated on the controller for each application, despite the actual need of the application. If the application defines more retain and persistent data than available, the compiler will generate an error.</p> <p><b>IMPORTANT:</b> If retains are stored in an SRAM or NVRAM in the controller, a value that is 24 bytes less than the value actually available in the controller must be specified here! This is used for saving an ID in the retain memory through which consistency of the retain data with respect to the IEC application can be ensured.</p>
<p><b>as from V3.5 SP 2!:</b> ts:setting "static-area\start-address"</p>	<p>The start-address of the static memory, if applicable. Do not define this setting, if the address is not the same on each device at any time. Without a start-address, relocation information will be generated along with the code, to relocate address accesses on the controller after download.</p>
<p>ts:section "areas"</p>	<p>Contains information about what areas to allocate on the runtime system.</p>
<p>ts:section "area_&lt;num&gt;"</p>	<p>Contains information about the area with index &lt;num&gt;.</p>
<p>ts:setting "area_flags"</p>	<p>What kind of area is to allocate. There are two possibilities:</p> <pre data-bbox="600 1648 1358 1962">// this is the setting for an area that is // allocated dynamically on the runtime system // its size is typically not fixed. This is the // default DynamicSize = 0x4, // setting for an area of fixed size, and with // fixed address. Use this setting only if you // have a reserved area for your data and the // code should not be relocated. This setting // does not work with multiple applications! Fixed = 0x10,</pre>
<p>ts:setting "area\flags"</p>	<p>This setting defines what kind of data is located in this area. The following values are possible:</p>

XML Tag	Description
	Data = 0x1, // "normal" data Constant = 0x2, // constant data (not used at the moment) Input = 0x4, // input segment (%I) Output = 0x8, // output segment (%Q) Memory = 0x10, // memory segment (%M) Retain = 0x20, // retain segment Code = 0x40, // code Persistent = 0x100, // persistent data
ts:setting "area/minimal-area-size"	The minimal size of the area, despite of the needed memory. (additional size will be used for online changes).  <b>IMPORTANT:</b> If this area has the Retain flag and if retains are stored in an SRAM or NVRAM in the controller, a value that is 24 bytes less than the value actually available in the controller must be specified here! This is used for saving an ID in the retain memory through which consistency of the retain data with respect to the IEC application can be ensured. For persistent variables, the actual available size for the program is 44 Bytes less than the specified size, these bytes are used for identification information!
ts:setting "area/maximal-area-size"	The maximal size of the area. If more space is required and no other area can be used, an error will be displayed
ts:setting "area/allocation-plus-in-percent"	The size allocated for the area will be this percentage larger than required for the data.  Thus, the calculation of the area size is this: TestSize := neededsize + neededsize * allocation-plus/100; Size := MIN(MAX(minimalsize, TestSize), maximalsize)  Any additional size in the area will be used for online changes, as long as possible.
ts:setting "area/start-address"	<b>Only used for Areas with Flag "FixedSize"!</b> The start address for the area as used on the runtime system. If the area contains data (variables), any reference in code to this will not be relocated on the runtime system, but will contain the correct address.
ts:setting "minimal-structure-granularity" type:integer	Special setting for Tricore  Tricore EABI documentation: Padding must be applied to the end of a union or structure to make its size a multiple of the alignment. ... To facilitate copy operations, any structure larger than 1 Byte must have a minimum 2 Byte alignment, even if its only members are byte elements  i.e: set minimal-structure-granularity to 2 in this case  Default: -1 (value to ignore)
ts:setting "max-stack-size" type:integer	Enables the check of the used stack at compile time. The maximum number of bytes available on the runtime system.
ts:setting "max-stack-size-external-call" type:integer	Maximum number of bytes of stack required by external function calls. This value is considered for calculating the used stack for every IEC-POU that calls an external function.

#### 6.4.5.1.2.1 Some Use cases of memory layout settings

In the following we have listed a number of problems the average editor of a device description will face. Of course there might as many memory layouts as there are target devices, but some settings are very common for classes of devices.

Runtime system with lot of RAM, areas can be allocated dynamically:

You will define one Area containing every kind of Data and Code (maybe except RETAIN or PERSISTENT see 3.). The size should not be limited. Minimal size should be rather big (e.g: the CODESYS Control device defines 1048576 bytes) to prevent memory fragmentation. A typical value for allocation-plus-in-percent is 30.

Example CODESYS Control device description:

```
<ts:section name="memory-layout">
  <ts:setting name="memory-size" type="integer" access="visible">
    <ts:value>1280</ts:value>
  </ts:setting>
  <ts:setting name="input-size" type="integer" access="visible">
    <ts:value>1280</ts:value>
  </ts:setting>
  <ts:setting name="output-size" type="integer" access="visible">
    <ts:value>1280</ts:value>
  </ts:setting>
  <ts:setting name="retain-size" type="integer" access="visible">
    <ts:value>1280</ts:value>
  </ts:setting>
  <ts:setting name="retain-in-own-segment" type="boolean" access="visible">
    <ts:value>1</ts:value>
  </ts:setting>
  <ts:setting name="pack-mode" type="integer" access="visible">
    <ts:value>8</ts:value>
  </ts:setting>
  <ts:setting name="stack-alignment" type="integer" access="visible">
    <ts:value>4</ts:value>
  </ts:setting>
  <ts:setting name="minimal-area-size" type="integer" access="visible">
    <ts:value>1048576</ts:value>
  </ts:setting>
  <ts:setting name="allocation-plus-in-percent" type="integer" access="visible">
    <ts:value>30</ts:value>
  </ts:setting>
  <ts:section name="areas">
    <ts:setting name="number" type="integer" access="visible">
      <ts:value>2</ts:value>
    </ts:setting>
    <ts:section name="area_0">
      <!-- Global and RETAIN memory -->
      <ts:setting name="flags" type="integer" access="visible">
        <ts:value>0xfeff</ts:value>
      </ts:setting>
      <ts:setting name="minimal-area-size" type="integer" access="visible">
        <ts:value>0x100000</ts:value>
      </ts:setting>
      <ts:setting name="allocation-plus-in-percent" type="integer" access="visible">
        <ts:value>30</ts:value>
      </ts:setting>
    </ts:section>
    <ts:section name="area_1">
      <!-- PERSISTENT RETAIN memory -->
      <ts:setting name="flags" type="integer" access="visible">
        <ts:value>0x120</ts:value>
      </ts:setting>
      <ts:setting name="minimal-area-size" type="integer" access="visible">
        <ts:value>0x10000</ts:value>
      </ts:setting>
    </ts:section>
  </ts:section>
</ts:section>
```

```

</ts:setting>
<ts:setting name="allocation-plus-in-percent" type="integer" access="visible">
  <ts:value>0</ts:value>
</ts:setting>
</ts:section>
</ts:section>

```

1. Small system with a limited RAM, Code in Flash:  
Define two areas (plus retain or persistent):  
One area containing all normal data with a fixed size at a fixed address, to avoid relocation in the runtime system:

```

area_0.flags := 0xffbf; // all except 0x40
area_0.area_flags := 0x10; // fixed size
area_0.minimal-area-size := <size>;
area_0.maximal-area-size := <size>;
area_0.allocation-plus-in-percent := 0;
area_0.start-address := <address of area pointer in runtime system>;

```

One area containing only code.

```

area_1.flags := 0x40; // only 0x40
area_1.minimal-area-size := <size>;
area_1.maximal-area-size := <size>;
area_1.allocation-plus-in-percent := 0;

```

The number of applications must be limited to 1 for this runtime system, otherwise two applications would share the same memory. Additional areas should also be set false.

2. Usage of RETAIN:  
Retain is handled like input or output. The retain data is allocated as one segment in an appropriate area. If you want to define one area containing only retain data, you should choose the following settings:

```

retain-size : <size>
retain-in-own-segment : 1 // this is also the default
area_x/flags : 0x20
area_x/ minimal-area-size : <size_x>
area_x/ maximal-area-size : <size_x>
area_x/allocation-plus-in-percent : 0

```

Then area\_x will contain exactly one segment for the retain data of the required size. In case of the settings of the CODESYS Control device description example shown above only one segment of retain data will be mapped in the normal data area.

#### Example of SysMemAllocArea in SysMem<OS>.c:

This is a very simple example of the function SysMemAllocArea. It assumes that 2 areas are defined in the devdesc file: One for retain data (flags: 0x20), and one for all the rest (code, data, inputs, outputs) (flags: 0xFFDF). It returns pointers to these 2 areas. The retain area is located in some SRAM, the other area is static memory.

Please note that this example only covers the memory of one application. If a second application would be downloaded to this controller, the same addresses would be assigned, and this would lead to some memory mismatch between the applications.

```

#define RETAIN_SIZE 0x1000 /* Size of retain data */
#define MEMORY_SIZE 0x100000 /* Static memory for code and data */
char pMem[MEMORY_SIZE];

```

```

void* CDECL SysMemAllocArea(char *pszComponentName, unsigned short
usType, unsigned long ulSize, RTS_RESULT *pResult)
{
    if (usType == 0x20) /* Retain */
    {
        /* Check size */
        if (ulSize > RETAIN_SIZE)
        {
            if (pResult != NULL)
                *pResult = ERR_NOMEMORY;
            return NULL;
        }
        /* Return pointer to SRAM or NVRAM */
        if (pResult != NULL)
            *pResult = ERR_OK;
        return (void*)0x12345678; /* Pointer to SRAM or NVRAM
*/
    }
    else /* all other data */
    {
        /* Check size */
        if (ulSize > MEMORY_SIZE)
        {
            if (pResult != NULL)
                *pResult = ERR_NOMEMORY;
            return NULL;
        }
        /* Return pointer to SRAM or NVRAM */
        if (pResult != NULL)
            *pResult = ERR_OK;
        return (void*)pMem; /* Pointer to static buffer */
    }
}

```

### 3. Usage of PERSISTENT:

At the moment, each application can only contain **one** list of persistent data. This list can be either PERSISTENT or RETAIN PERSISTENT. **The persistent area can't contain anything else than persistent data!** The persistent area contains a header with a checksum for the current list, and the length of the current list. We upload this information at login and if both values did not change we estimate the list to be the same. All other persistent variables are located one after the other - according to the pack mode - in the persistent area. An example for a persistent area shows the CODESYS Control example above. Note that the RETAIN PERSISTENT area will not contain any retain data.

#### 6.4.5.1.2.2 Child applications

Child applications have access to any data and code of their parent applications. A child application cannot handle own input, output, memory, retain or persistent data, so the area for any child application is created implicitly and not due to target settings.

- With compiler version < 3.4.2.0 child applications have one area with the following settings:

```

Child_area.flags := 0x43; // code and all data
Child_area.minimal-area-size := 0x100000;
Child_area.allocation-plus-in-percent := 100;

```

Additional areas needed for online change will have the same settings.

- With compiler version >= 3.4.2.0 all memory settings of the parent application automatically will also be set on the child application.

Additional areas needed for online change will have the same settings.

#### 6.4.5.1.3 Online



```

<ts:section name="online">
  <ts:setting name="customizedonlinemgr" type="boolean" access="edit">
    <ts:value>1</ts:value>
  </ts:setting>
  <ts:setting name="customized-fileupload" type="string" access="edit">
    <ts:value>4B84C13B-D82E-461c-9A3E-47E2AFAD3A2D</ts:value>
  </ts:setting>
  <ts:setting name="customized-filedownload" type="string" access="edit">
    <ts:value>46F722D1-A347-44ca-B1D6-55A3CE398497</ts:value>
  </ts:setting>
</ts:section>

```

XML-Tag	Description
ts:section "online"	Online section
ts:setting "customizedonlinemgr"	type="boolean" If TRUE, a customer specific OnlineMgr will be used for diagnosis.
ts:setting "customized-fileupload"	type="string" This entry serves to implement an „own“ file upload. The value is the typeguid of the implemented interface ( IFileUpload ).
ts:setting "customized-filedownload"	type="string" This entry serves to implement an „own“ file download. The value is the typeguid of the implemented interface ( IFileDownload ).

#### 6.4.5.1.4 Task configuration

This sub-section of <ts:TargetSettings...> describes the task configuration options for the <device>.

Example:

```

<ts:section name="taskconfiguration">
  <ts:setting name="supportmicroseconds" type="boolean" access="readonly">
    <ts:value>1</ts:value>
  </ts:setting>
  <ts:setting name="supportfreewheeling" type="boolean" access="readonly">
    <ts:value>1</ts:value>
  </ts:setting>
  <ts:setting name="supportinterval" type="boolean" access="readonly">
    <ts:value>1</ts:value>
  </ts:setting>
  <ts:setting name="supportevent" type="boolean" access="readonly">
    <ts:value>0</ts:value>
  </ts:setting>
  <ts:setting name="supportexternal" type="boolean" access="readonly">
    <ts:value>0</ts:value>
  </ts:setting>
  <ts:setting name="supportextendedwatchdog" type="boolean" access="readonly">
    <ts:value>1</ts:value>
  </ts:setting>
  <ts:setting name="defaulttaskpriority" type="integer" access="readonly">
    <ts:value>1</ts:value>
  </ts:setting>
  <ts:setting name="mintaskpriority" type="integer" access="readonly">
    <ts:value>0</ts:value>
  </ts:setting>
  <ts:setting name="maxtaskpriority" type="integer" access="readonly">

```

```

    <ts:value>31</ts:value>
  </ts:setting>
  <ts:setting name="maxnumoftasks" type="integer" access="readonly">
    <ts:value>100</ts:value>
  </ts:setting>
  <ts:setting name="maxeventtasks" type="integer" access="readonly">
    <ts:value>3</ts:value>
  </ts:setting>
  <ts:setting name="maxintervaltasks" type="integer" access="readonly">
    <ts:value>3</ts:value>
  </ts:setting>
  <ts:setting name="maxexternalevents" type="integer" access="readonly">
    <ts:value>3</ts:value>
  </ts:setting>
  <ts:setting name="maxfreetasks" type="integer" access="readonly">
    <ts:value>3</ts:value>
  </ts:setting>
  <ts:setting name="cycletimedefault" type="string" access="readonly">
    <ts:value>t#20ms</ts:value>
  </ts:setting>
  <ts:setting name="cycletimemin_us" type=" integer " access="readonly">
    <ts:value>0</ts:value>
  </ts:setting>
  <ts:setting name="cycletimemax_us" type=" integer " access="readonly">
    <ts:value>0x7FFFFFFF</ts:value>
  </ts:setting>
  <ts:setting name="watchdogtimemax_us" type=" integer " access="readonly">
    <ts:value>400</ts:value>
  </ts:setting>
  <ts:setting name="maxwatchdogsensitivity" type=" integer " access="readonly">
    <ts:value>10</ts:value>
  </ts:setting>
  <ts:setting name="systemtick" type="string" access="visible">
    <ts:value>LTIME#250us</ts:value>
  </ts:setting>
  <ts:setting name="externalevents" type="cdata" access="hide">
  </ts:setting>
  <ts:setting name="systemevents" type="cdata" access="readonly">
    <![CDATA[]]>
  </ts:setting>

  <ts:section name="priorityinfo">
    <ts:setting name="priority-1" type="string" access="visible">
      <ts:value>realtime priority</ts:value>
    </ts:setting>
    <ts:setting name="priority-2" type="string" access="visible">
      <ts:value>medium priority</ts:value>
    </ts:setting>
  </ts:section>

  <ts:section name="applicationtasks">
    <ts:section name="dataserver">
      ... see below for detailed example and description
    </ts:section>
  </ts:section>
</ts:section>

```

#### 6.4.5.1.4.1 Application tasks

For Visualization and DataServer the programming system automatically creates tasks, per default with the lowest possible priorities. To overwrite the defaults of **priority and cycle time** for these tasks, the

following entries in sections **"dataserver"** res. **"visualization"** might be used, which must be positioned within a section **"applicationtasks"** and this again within section **"taskconfiguration"**:

Example:

```
<ts:section name="applicationtasks">
  <ts:section name="dataserver">
    <ts:setting name="defaulttaskpriority" type="integer" access="visible">
      <ts:value>25</ts:value>
    </ts:setting>
    <ts:setting name="cycletimedefault" type="string" access="visible">
      <ts:value>t#200ms</ts:value>
    </ts:setting>
  </ts:section>
  <ts:section name="visualization">
    <ts:setting name="defaulttaskpriority" type="integer" access="visible">
      <ts:value>24</ts:value>
    </ts:setting>
    <ts:setting name="cycletimedefault" type="string" access="visible">
      <ts:value>t#150ms</ts:value>
    </ts:setting>
  </ts:section>
</ts:section>
```

XML Tag	Description	Default
<b>ts:section "taskconfiguration"</b>	Task configuration section	
ts:setting "supportmicroseconds"	1= IEC task cycles in the microsecond range are supported. 0= IEC task cycles in the microsecond range are NOT supported. Only tasks in the millisecond range can be projected.	1
ts:setting "supportfreewheeling"	1= free-running IEC tasks are supported 0= free-running IEC tasks are NOT supported	0
ts:setting "supportinterval"	1= cyclical IEC tasks are supported 0= cyclical IEC tasks are NOT supported	0
ts:setting "supportevent"	1= event-driven IEC tasks are supported 0= event-driven IEC tasks are NOT supported	0
ts:setting "supportexternal"	1= external event-driven IEC tasks are supported 0= external event-driven IEC tasks are NOT supported	0
ts:setting "supportextendedwatchdog"	1= watchdog monitoring of IEC tasks is supported 0= watchdog monitoring of IEC tasks is NOT supported	0
ts:setting "defaulttaskpriority"	Standard priority of an IEC task	1
ts:setting "mintaskpriority"	Minimum priority for an IEC task (maximum task priority)	0
ts:setting "maxtaskpriority"	Maximum priority for an IEC task (minimum task priority)	31
ts:setting	Maximum number of IEC tasks	0

XML Tag	Description	Default
"maxnumoftasks"		
ts:setting "maxeventtasks"	Maximum number of event-driven IEC tasks	
ts:setting "maxintervaltasks"	Maximum number of cyclical IEC tasks	
ts:setting "maxexternalevents"	Maximum number of external event-driven IEC tasks	
ts:setting "maxfreetasks"	Maximum number of free-running IEC tasks	
ts:setting "cycletimedefault"	Default values for the cycle time	10
ts:setting "cycletimemin_us"	Minimum length of time interval; a value underrunning the minimum length within the task configuration will lead to compile errors	0
ts:setting "cycletimemax_us"	Maximum length of time interval; a value exceeding the maximum length within the task configuration will lead to compile errors	0x7FFFFFFF
ts:setting " watchdogtimemax_us "	Maximum watchdog time; a value exceeding the maximum length within the task configuration will lead to compile errors	40000000
ts:setting maxwatchdogsensitivity "	Maximum watchdog sensitivity; a value exceeding the maximum will lead to compile errors	10
ts:setting "systemtick"	String value; The cycle time must be a integral multiple of the systemtick; example:  <pre>&lt;ts:setting name="systemtick" type="string" access="visible"&gt;   &lt;ts:value&gt;LTIME#250us&lt;/ts:value&gt; &lt;/ts:setting&gt;</pre>	
ts:setting "externalevents"	List of names of supported external events	
ts:setting "supportprofiling"	1=The Profiling functionality is supported, the resp. dialogs are available in the Task-Editor	
ts:setting "watchdog-enabled"	1= IEC Task Watchdog is enabled	
ts:setting "default-watchdog-time"	String, defining the default watchdog time	
ts:setting "default-watchdog-sensitivity"	String, defining the default watchdog sensitivity	
<b>ts:section</b> <b>"systemevents"</b>	<u>subsection within section "taskconfiguration"</u> here the particular events must be specified which should be available in the Event Configuration of the programming system (Task Configuration dialog); an event is uniquely specified by component-id, event-id, parameter-id and parameter-version	
ts:setting "library"	String: name of a library containing general functions needed for event handling; the library will be included automatically in the	

XML Tag	Description	Default
	<p>project when using the event function; example:</p> <pre>&lt;/ts:setting&gt;  &lt;ts:setting name="library" type="string" access="visible"&gt; &lt;ts:value&gt;CmpApp, 3.5.0.0 (System)&lt;/ts:value&gt; &lt;/ts:setting&gt;</pre>	
<b>ts:section "systemevent"</b>	<b>subsection within section "systemevents" for the settings concerning system events; an event is uniquely specified by component-id, event-id, parameter-id and parameter-version;</b>	
ts:setting "eventname"	String, name for event (can be defined here as desired)	
ts:setting "description"	String, short description for event (can be defined here as desired)	
ts:setting "component-id"	Integer, ID of the event as defined in the library specified below by setting "library"; example:	
	<pre>&lt;ts:setting name="component-id" type="integer" access="visible"&gt; &lt;ts:value&gt;2&lt;/ts:value&gt;</pre>	
ts:setting "event-id"	Integer, ID of the event as defined in the library specified below by setting "library"; example:	
	<pre>&lt;ts:setting name="component-id" type="integer" access="visible"&gt; &lt;ts:value&gt;2&lt;/ts:value&gt;</pre>	
ts:setting "parameter-id"	Integer, ID of the event parameter, as defined in the library specified by setting "library" (see below)	
ts:setting "parameter-version"	Integer, version of the event parameter, as defined in the library specified below by setting "library"	
ts:setting "parameter-struct"	String: name of the parameter structure as defined in the library specified below by setting "library"; example:	
	<pre>&lt;ts:setting name="parameter-struct" type="string" access="visible"&gt;&lt;ts:value&gt;CmpApp.EVTPARAM _CmpAppStop&lt;/ts:value&gt; &lt;/ts:setting&gt;</pre>	
ts:setting "library"	String: name of a library containing the event function; this library will be included automatically in the project when using the event function; example:	
	<pre>&lt;/ts:setting&gt;  &lt;ts:setting name="library" type="string" access="visible"&gt; &lt;ts:value&gt;CmpApp, 3.5.0.0 (System)&lt;/ts:value&gt; &lt;/ts:setting&gt;</pre>	
<b>ts:section "applicationtasks"</b>	<b>subsection with section "taskconfiguration" for setting the priority and cycletime of</b>	

XML Tag	Description	Default
	Visualizaton resp. DataServer tasks	
<b>ts:section "dataserver"</b>	<u>subsection within section "applicationtasks"</u> for the dataserver task settings	
<b>ts:section "visualization"</b>	<u>subsection within section "applicationtasks"</u> for the visualization task settings	
ts:setting "defaulttaskpriority"	priority of dataserver task, value e.g. "25"	lowest possible
ts:setting "cycletimedefault"	cycle time of data server task in time format, value e.g. t#200ms	
ts:section "priorityinfo"	In this section the meaning of the task priorities can be described. An information icon shows the priority information in the tooltip of the configuration window	

#### 6.4.5.1.5 Network variables

Network variables can be used for data exchange between two or several PLCs, currently possible for UDP networks. The values of the variables are exchanged automatically via broadcast telegrams. These services are not confirmed by the protocol, which means that it is not checked, whether a message is received by the addressee. The exchange of network variables is a 1 (sender) to n (recipients) – connection. Network Variables must be defined in fix variables lists as well in the sender as in the receiver devices and their values will be transmitted via broadcasting.

In sub-section "**networkvariables**" of <ts:TargetSettings...> the following settings concerning the support of this functionality can be defined:

Example:

```
<ts:section name="networkvariables">
  <ts:section name="protocols">
    <ts:setting name="numofprotocols" type="integer" access="visible">
      <ts:value>1</ts:value>
    </ts:setting>
    <ts:section name="protocol1">
      <ts:setting name="protocolname" type="string" access="visible">
        <ts:value>UDP</ts:value>
      </ts:setting>
      <ts:setting name="library" type="string" access="visible">
        <ts:value>NetVarUdp</ts:value>
      </ts:setting>
      <ts:setting name="libraryversion" type="string" access="visible">
        <ts:value>*</ts:value>
      </ts:setting>
      <ts:setting name="packetsize" type="integer" access="visible">
        <ts:value>256</ts:value>
      </ts:setting>
      <ts:setting name="max-num-sender" type="integer" access="visible">
        <ts:value>2</ts:value>
      </ts:setting>
      <ts:setting name="max-num-receiver" type="integer" access="visible">
        <ts:value>2</ts:value>
      </ts:setting>
      <ts:setting name="max-gvl-size" type="integer" access="visible">
        <ts:value>5</ts:value>
      </ts:setting>
      <ts:setting name="min-interval" type="string" access="visible">
        <ts:value>T#50ms</ts:value>
      </ts:setting>
    </ts:section>
  </ts:section>
```

```

    <ts:setting name="max-interval" type="string" access="visible">
      <ts:value>T#100ms</ts:value>
    </ts:setting>
  </ts:section>
</ts:section>
</ts:section>

```

<b>ts:section</b> <b>"networkvariables"</b>	Network Variables section	
<b>ts:section</b> <b>"protocols"</b>	Network Protocols section	-
<b>ts:setting</b> <b>"numofprotocols"</b>	Number of supported protocols (currently 1, because only UDP is supported)	-
<b>ts:section</b> <b>"protocol&lt;n&gt;"</b>	Protocol section	-
ts:setting "protocolname"	Name of the protocol	
ts:setting "library"	Name of needed library (must be installed)	-
ts:setting "libraryversion"	Version of above specified library Note: This must not be set to a fix version, but for the newest (-> it must be set to "*"). All other settings may lead to problems with the current project.	-
ts:setting "packetsize"	Size of packets of the resp. protocol (UDP: 256)	-
ts:setting "max-num-sender"	Maximal number of GVLs of an application that can be send	
ts:setting "max-num-receiver"	Maximal number of GVLs of an application that can be received	
ts:setting "max-gvl-size"	Maximal number of bytes that can be send with one network variable list (limits the number of variables)	
ts:setting "min-interval"	The minimum transmission time that can be configured	
ts:setting "max-interval"	The maximum transmission time that ca be configured	

#### 6.4.5.1.6 Code generator

In sub-section "codegenerator" of <ts:TargetSettings...> all code generator options for the target processor are set.

Example:

```

<ts:section name="codegenerator">
  <ts:setting name="CPU" type="codegenerators" access="edit">
    <ts:value>Intel X86</ts:value>
  </ts:setting>
  <ts:setting name="Floating Point Unit" type="boolean" access="visible">
    <ts:value>1</ts:value>
  </ts:setting>
  <ts:setting name="Breakpoint Size" type="integer" access="visible">

```

```

    <ts:value>5</ts:value>
  </ts:setting>
  <ts:setting name="external-linkage" type="string" access="visible">
    <ts:value>MUL,DIV</ts:value>
  </ts:setting>
  <ts:setting name="codegenerator-guid" type="string" access="visible">
    <ts:value>{C746DD9C-011F-4ab9-A555-EB14C69F50CF}</ts:value>
  </ts:setting>
  <ts:setting name="codegenerator-version-constraint" type="string" access="visible">
    <ts:value>newest</ts:value>
  </ts:setting>
</ts:section>

```

XML Tag	Description
ts:section "codegenerator"	Codegenerator section
ts:setting "CPU"	Processor type
ts:setting "Floating Point Unit"	1=Floating point unit present 0=FPU missing [Default=1]
ts:setting "Breakpoint Size"	Size of the breakpoint OpCode in bytes: Intel X86= 5
ts:setting "external-linkage"	The functions listed will be implemented by external code (actually valid for MUL, DIV for NIOS only)
ts:setting "codegenerator-guid"	Internal CODESYS GUID: Intel X86= {C746DD9C-011F-4ab9-A555-EB14C69F50CF}
ts:setting "codegenerator- version-constraint"	Version limitation of the code generator: [Default=newest]
ts:setting maximum_num_applications	Maximum number of applications, which can be used in the project without getting a warning at compile time. Also implicitly generated applications will be regarded, like e.g. symbol application. You might use this setting to control the number of applications for download to the target system. Note however that the applications for a certain target may be distributed over several projects, so that a download might fail even if there has been no warning at compilation of a particular project.
ts:setting "retain-in-cycle"	Default: 0, type="boolean" If the value for this setting is "1" <b>and compiler version is &gt;=3.2.2.20</b> , then all retain parameters will be handled in the following way: - <u>All</u> retain variables (not only that for functionblocks) are located in the standard data memory - Additional code is generated for all retain variable (especially for all retains in FB-instances) to copy all retains at the end of a cycle to the retain area. - After booting with a project the retain variables are initialized with the data in retain memory. As a consequence, it is now also possible to declare variables on direct output addresses as RETAIN variables.  <b>Note:</b> This behavior may have effects on code size and performance.



XML Tag	Description
ts:setting "check-multiple-task-output-write"	<p>Default: 0, type="boolean"</p> <p>If the value for this setting is "1" <b>and compiler version is &gt;=3.2.2.40</b> then it will be checked, whether two tasks are accessing the same output. An error message will be generated in the following cases:</p> <ul style="list-style-type: none"> <li>- a POU called in multiple tasks writes to an output</li> <li>- two POUs write to the same output and are called in multiple tasks</li> <li>- two POUs write to different bits of the same byte and are called in multiple tasks</li> </ul> <p>This is valid for variables mapped in the device configuration („new variable“) or via an AT-declaration, and for direct access on IOs in the implementation code (%QX := ...).</p>
ts:setting "reserved-registers"	<p>type="string"</p> <p>The specified register will be reserved. Value e.g. "14" in order to reserve register R14.</p>
ts:setting "Motorola Byte Order"	<p>type="boolean"</p> <p>If TRUE, Motorola Byte Order will be used (Default for PowerPC code generators!), otherwise Intel byte order is used (default for the other code generators and when used in other scopes, e.g. for network variables)</p>
ts:setting "hexfile"	<p>type="boolean"</p> <p>If TRUE, the command "Generate hex file for active application" will be available in the Build menu of the programming system. For this At least 8 byte of code segment prolog size is required:</p> <pre>&lt;section name="memory-layout"&gt;   &lt;setting name="code-segment-prolog-size" type="integer"   access="visible"&gt;     &lt;value&gt;8&lt;/value&gt;</pre>
ts:setting "lreal-data-type" ts:setting "lint-data-types"	<p>type="boolean"</p> <p>If FALSE, the resp. 64bit datatypes are not valid; example:</p> <pre>&lt;ts:setting name="lint-data-types" type="boolean" access="visible"&gt;   &lt;ts:value&gt;0&lt;/ts:value&gt; &lt;/ts:setting&gt; &lt;ts:setting name="lreal-data-type" type="boolean" access="visible"&gt;   &lt;ts:value&gt;0&lt;/ts:value&gt; &lt;/ts:setting&gt;</pre>
ts: setting "do-persistent-code"	<p>Type="boolean" (default: 1)</p> <p>In CODESYS V3 persistent variables are handled differently to V2.3. If there are persistent variables defined in a function block, but their instance paths are not mapped in the list of persistent variables, then warnings resp. error messages will be created. If do-persistent-code is set to "1", then these messages will be suppressed. This enables customers to implement an own persistent handling.</p>
ts: setting "link-all-globalvariables"	<p>Type="boolean" (default: 1)</p> <p>If 1 (default): All global variables of a project get downloaded, even if no one is used. This e.g. is needed for variables that are only used by the HMI but not by the PLC.</p>

XML Tag	Description
	If 0: A list of global variables only get downloaded, if at least one is used
ts: setting "multithreading"	Type="boolean" (default: 1) If 1 (default): The CODESYS compiler uses multiple threads to increase the compile speed. If 0: The CODESYS compiler uses just one single thread to build. <b>Note:</b> Because of timing effects, the result of a multithreading build might be different across single compiles.
ts: generate_direct_calls	Type="Boolean" (default: 0) Compilerversion >= 3.5.1.0 Programm and function calls are done direct and not via a function pointer. Online change must be switched off in the target settings. Implemented for C166, ARM, MIPS and CortexM3
ts: c-calling-convention	Type:"Boolean" (default: 0) Compilerversion >= 3.5.1.0 External function calls are performd according the C calling conventions. Implemented for ARM (intel byte order) without FPU under CE and for x86 under Windows

## Additional settings for TI-DSP Codegenerator:

XML Tag	Description
ts: setting "dp-register-addressing"	Type="boolean" (default: 0) For compiler versions >=3.5.1.0 If 0 (default): 32 bit addressing (standard) If 1: DP registers is used for global data access. This reduces the code size for global data access. 22 bit addressing. Preconditions: 1. all area sections must have the following settings for fix location: <pre>&lt;ts:setting name="area_flags" type="integer" access="visible"&gt;   &lt;ts:value&gt;0x10&lt;/ts:value&gt; &lt;/ts:setting&gt; &lt;ts:setting name="start-address" type="integer" access="visible"&gt;   &lt;ts:value&gt;0xA0000&lt;/ts:value&gt; &lt;/ts:setting&gt;</pre>

## Additional settings for PowerPC Codegenerator:

XML Tag	Description
ts: setting "single-precision-floating-point-apu" ts: setting "double-precision-floating-point-apu" ts: setting "Vector Unit"	Type="boolean" (default: 0) Settings to support special floating point units
ts: misaligned-access	Type="Boolean" (default: 1) Memcpy is performed 4Byte-wise. On some platform, this misalignment may lead to alignment exceptions

Additional settings for 386 Codegenerator:

XML Tag	Description
ts: setting "sse2-unit"	Compilerversion >= 3.5.2.0 Type="Boolean" (default: false) SSE-2 unit is used for REAL/LREAL operations

Additional settings for x86-64 Codegenerator:

XML Tag	Description
ts: setting "operating-system"	Type="string" (default: Empty string for Windows OS) "Linux": To consider special rules for the Linux operating system (e.g.: calling conventions)
ts: setting "sse2-unit"	Compilerversion >= 3.5.2.0 Type="Boolean" (default: true) SSE-2 unit is used for REAL/LREAL operations

Additional settings for the ARM and x86 Codegenerator:

XML Tag	Description
ts: setting "c-calling-convention"	Type="string" (default: use 3S calling convention) "CDECL": To generate a C-compatible call interface for external calls. Implemented for ARM with intel byte order without FPU running with Windows CE and for x86 with Windows

Additional settings for SH Codegenerator:

XML Tag	Description
ts: setting "CPU"	Type="string" (default: "SH-3") Possible values: "SH-2", "SH-2A", "SH-3", "SH-4"
ts: setting "rts-globaldatapointer-area"	Type="integer" (default: -1 = do not use) Global data in the specified area is accessed via register 10 holding the base address initialised by the runtime system.

### 6.4.5.1.7 Device configuration

In sub-section "deviceconfiguration" of <ts:TargetSettings...> settings for the device configuration via the CODESYS device editor can be defined.

Example:

```
<ts:section name="deviceconfiguration">
  <ts:setting name="mapping-changeable" type="boolean" access="visible">
    <ts:value>false</ts:value>
  </ts:setting>
  <ts:setting name="simulation-disabled" type="boolean" access="visible">
    <ts:value>>true</ts:value>
  </ts:setting>
```

XML Tag	Description
ts:setting "mapping-changeable"	True or 1: if mapping could be changed between creating new variable and map to existing variable is possible. (Default, if section "target settings" is not available in the device description) False: Only creating new variable is possible (behaviour like CODESYS V2.x)
ts:setting "simulation-disabled"	True or 1: Entry "Simulation <device>, which usually is available in the Online menu, if the currently connected device is a PLC, will be not available False: The menu entry is visible (Default, if section "target settings" is not available in the device description)
ts:setting "update-only-device-version"	type: boolean; True or 1: Update of the device update is only possible with different versions of exactly this device. False: Also updating to a different device is possible (e.g. changing from EtherCAT to CAN)
ts:setting "updateAllToEqualVersion"	type: boolean; True or 1: The device gets updated recursively. All devices below the plc get updated to the same version as the plc. For each device a message gets added, whether the update was successful or not.
Ts:setting "Motorola-bitfields"	Type: boolean; TRUE or 1: Bytes in all bitfields of this device are organized according Motorola byte order
ts:setting "Union-root-editable"	Type: boolean; TRUE or 1: Also the root element of union elements, which are defined as IO channels in the IO mapping dialog of the device editor, can be mapped (default); if FALSE, only the subelements are mappable
ts:setting "enableAdditionalParameters"	Type: boolean; TRUE or 1: extended online config mode: If already an application is on the PLC, the user gets a dialog, where he can choose between connecting to the PLC via "Parameter mode" and via "Application mode". Parameter mode: The project structure on the PLC will be compared with that in the project, and if there is no mismatch, a connection to the PLC will be established. In the generic parameter editor dialog the parameters can be read and written. Applications keep untouched. Application mode: The "online configuration mode" will be established, i.e. an implicit application will be generated on the device for test purposes. No real application program must be downloaded.

XML Tag	Description
	FALSE or 0: No "Parameter mode" available (like if you choose "online config mode" when there is no application on the PLC)
ts:setting "skipAdditionalParametersForEmptyConnectors"	Type: Boolean; True or 1: for connectors with no parameters there will be no additional parameters if setting "enableAdditionalParameters" is set. This will reduce the amount of download parameters and therefore the size of the plc boot project.  False or 0: Behaviour as with old versions. For all connectors the additional parameters are created.
ts:setting "Multiple-mappable-allowed"	type: boolean; TRUE or 1: It is allowed to map a subelements of a bitfields type even when the basic element is mapped (multiple mapping). However a warning will appear while compiling. Default: 0  Example: <ts:setting name="Multiple-mappable-allowed" type="boolean" access="edit"> <ts:value>1</ts:value> </ts:setting>
ts:setting "createBitChannels"	type: boolean; TRUE or 1: All integer datatypes in the device configuration description (example "std:DINT") will be created with bit channels below the standard data type. Can be used in device descriptions of bus systems which are not supporting bit channels.  <b>Note:</b> If used in the device configuration description of the PLC, all devices below will be affected.  <b>Note:</b> The setting works only if bit channels are also supported by the IO driver, in IoDrvReadInputs resp. IoDrvWriteOutputs. Otherwise wrong channels or none might be updated !  Default: 0  Example: <ts:section name="deviceconfiguration"> <ts:setting name="createBitChannels" type="boolean" access="visible"> <ts:value>1</ts:value> </ts:setting> </ts:section>
ts:setting "ShowMultipleTaskMappingsAsError"	Type: Boolean; True  The compiler will show an error if same inputs or outputs are used in more than one task. Example if %QB0 is used in task 1 and 2 then the error message is shown when generating code.  False: No error is show as with older versions.  Default: false
ts:setting "Basetype-mappable"	Type: Boolean, true or 1 (default) In the io mapping all types (unions, structs) are mappable.  False or 0: The io variables without an iec base type (for example root of structs) are not mappable in the io mapping editor. A message is shown if the user tries to set the io mapping.
ts:setting "Bitfield-mappable"	Type: Boolean, true or 1 (default) In the io mapping the base and all bits of a bitfield type could be mapped.  False or 0: The base element of a bitfield is not mappable. Only the underlying bits could be mapped. A message is shown if the user

XML Tag	Description
	tries to set the io mapping.
ts:setting "MapAlwaysIecAddresses"	Type: Boolean, true or 1: IoDrvReadInputs and IoDrvWriteOutputs are always copying to input %I or output %Q space. The mapping to the existing variables is done in functions afterreadinputs or beforewriteoutputs. SysMem library must be included in the library manager because if a structured parameter is mapped SysMemCpy is used to transfer the data from % to the existing variable. If the setting is enabled IO drives could optimize the copying of data in IoDrvReadInputs and IoDrvWriteOutputs by copying the complete data with one SysMemCpy because the IO-driver must not handle each mapping entry separately. False or 0 (default): Standard behavior as in old versions.
ts:setting "disableChildApp"	For Safety child mapping application Type: Boolean, true or 1: No child mapping application is created False or 0 (default): Child mapping application is created for safety plc device to copy all data from physical devices to logical devices.
ts:setting "NoIoInstance"	Type: Boolean, true or 1. Function block instances are hidden for the input assistant (attribute 'hide') is set to all instances for devices. False or 0 (default): Function block instances are shown for input assistant.

#### 6.4.5.1.8 Library management

In sub-section "library-management" of <ts:TargetSettings...> the settings for the handling of libraries are defined.

Example:

```
<ts:section name="library-management">
  <ts:section name="placeholder-libraries">
    <ts:setting name="IoStandard" type="string" access="visible">
      <ts:value>IoStandard, 3.1.1.0 (System)</ts:value>
    </ts:setting>
  </ts:section>
</ts:section>
```

##### 6.4.5.1.8.1 Placeholder Libraries

In this sub-section of "library-management" the replacement of libraries is set. The name sets the library name to replace and the value defines the replacement name.

Example: in a library a placeholder library is used. The name in the library is IoStandard. In the target settings the IoStandard is replaced by IoStandard, 3.1.10 (System). In the library manager this replaced library is automatically included.

Example:

```
<ts:section name="placeholder-libraries">
  <ts:setting name="IoStandard" type="string" access="visible">
    <ts:value>IoStandard, 3.1.1.0 (System)</ts:value>
  </ts:setting>
</ts:section>
```

#### 6.4.5.1.8.2 Placeholderlib, for replacing 3S-libraries by customer-specific libraries

Instead of the name of a default 3S-library to be automatically included with the device, a library placeholder can be specified in the Driver Info section (see chap. 6.4.4.4). When the device is added to the project, the currently corresponding customer specific library will be included.

#### 6.4.5.1.8.3 Exclude library category

Example:

```
<ts:section name="library-manager-filter">
  <ts:section name="filter-entry">
    <ts:setting name="hide-category" type="string" access="visible">
      <ts:value>2A47D467-E781-4f73-A35C-01B9D33D28D2</ts:value>
    </ts:setting>
    <ts:setting name="show-category" type="string" access="visible">
      <ts:value>C179664B-4223-4f78-B898-054073C60FB3</ts:value>
    </ts:setting>
    <ts:setting name="show-library" type="string" access="visible">
      <ts:value>CAA CanMiniDriver SJA1000 (PLCWinRTE VV3), 3.0.0.2 (3S - Smart Software
Solutions GmbH)</ts:value>
    </ts:setting>
  </ts:section>
</ts:section>
```

The example excludes the category "PLCWinRTE VV3" with the GUID 2A47D467-E781-4f73-A35C-01B9D33D28D2 (defined in the file LibraryCategoryBase.libcat.xml).

It will show the complete category CAA with the GUID C179664B-4223-4f78-B898-054073C60FB3

Also with show-library only specific libraries could be shown. The values have to be set in the same way as for placeholder libraries. It is a string of the name, version and vendor name.

XML Tag	Description
ts:setting "hide-category"	The guid of the library category to disable in the library manager. It is possible to disable several categories by adding multiple "filter-entry" sections.
Ts:setting "show-library"	The name of the library containing the library name, the version and the vendor. A wildcard for the version is also possible (*)
ts:setting "show-category"	With this setting it is possible to show a complete sub category in the disabled parent category.

#### 6.4.5.1.9 Visualization

Subsection "visualization" of <ts:TargetSettings...> is available for device-specific visualization settings.

Example:

```
<ts:section name="visualization">
  <ts:section name="targetsupport">
    <ts:setting name="webvisualization" type="boolean" access="visible">
      <ts:value>1</ts:value>
    </ts:setting>
    <ts:setting name="webvisualization_client" type="boolean" access="visible">
      <ts:value>0</ts:value>
    </ts:setting>
    <ts:setting name="integratedwebserver" type="" boolean="" access="visible">
      <ts:value>1</ts:value>
    </ts:setting>
  </ts:section>
</ts:section>
```

```

    <ts:setting name="webvisualization_insertbydefault" type="boolean"
access="visible">
    <ts:value>1</ts:value>
</ts:setting>
    <ts:setting name="targetvisualization_insertbydefault" type="boolean"
access="visible">
    <ts:value>1</ts:value>
</ts:setting>
    <ts:setting name="transferfilestoplc" type="boolean" access="visible">
    <!--If this value is set to true, then the visualization files get re-process to
the plc !>
    <ts:value>1</ts:value>
</ts:setting>
</ts:section>
<ts:section name="keyboardusage">
    <ts:setting name="availablemodifiers" type="string" access="visible">
    <ts:value>SHIFT,CTRL,ALT</ts:value>
</ts:setting>
    <ts:setting name="basekeys" type="string" access="visible">
    <ts:value>default</ts:value>
</ts:setting>
    <ts:section name="additionalkeys">
    <ts:setting name="EMERGENCY_STOP" type="integer" access="visible">
    <ts:value>300</ts:value>
</ts:setting>
    <ts:setting name="MACHINE_SHUTDOWN" type="integer" access="visible">
    <ts:value>301</ts:value>
</ts:setting>
</ts:section>
    <ts:section name="notavailablekeys">
    <ts:setting name="BACKSPACE" type="string" access="visible">
    <ts:value/>
</ts:setting>
</ts:section>
</ts:section>

<ts:section name="TargetConstraints">
    <ts:section name="TargetFonts">
    <ts:section name="Arial CE">
    <ts:setting name="WindowsMatchingFont" type="string"
access="visible">
    <ts:value>Arial</ts:value>
</ts:setting>
    <ts:setting name="Sizes" type="string" access="visible">
    <ts:value>10,12,14</ts:value>
</ts:setting>
    <ts:setting name="Styles" type="string" access="visible">
    <ts:value>default,bold</ts:value>
</ts:setting>
</ts:section>
</ts:section>
    <ts:section name="TargetColors">
    <ts:setting name="Black" type="uint" access="visible">
    <ts:value>0xff000000</ts:value>
</ts:setting>
    <ts:setting name="White" type="uint" access="visible">
    <ts:value>0xffffffff</ts:value>
</ts:setting>
    <ts:setting name="Green" type="uint" access="visible">
    <ts:value>0xff00ff00</ts:value>
</ts:setting>
    <ts:setting name="Red" type="uint" access="visible">
    <ts:value>0xffff0000</ts:value>

```



```

</ts:setting>
<ts:setting name="Blue" type="uint" access="visible">
  <ts:value>0xff0000ff</ts:value>
</ts:setting>
</ts:section>
<ts:section name="TargetVisualElements">
  <ts:setting name="NotAvailableEleme"ts" type="string" access="visible">
    <ts:value>Trace|Table</ts:value>
  </ts:setting>
  <ts:setting name="AvailableEleme"ts" type="string" access="visible">
    <ts:value>Rectangle|Button|Frame|Polygon</ts:value>
  </ts:setting>
</ts:section>
<ts:setting name="TargetImageFormats" type="string" access="visible">
  <ts:value>*.bmp|*.jpg</ts:value>
</ts:setting>
<ts:setting name="MaxNumOfVisualizations" type="integer" access="visible">
  <ts:value>500</ts:value>
</ts:setting>
<ts:setting name="MaxNumOfElementsPerVisualizat"on" type="integer"
access="visible">
  <ts:value>200</ts:value>
</ts:setting>
</ts:section>
</ts:section>

```

Section “**targetsupport**”, subsection of “visualization”:

The following settings define the file transfer mode:

XML Tag	Description
ts:setting "webvisualization"	1=Web-Visualization is supported (type: bool)
ts:setting "integratedwebserver"	1=Web Server integrated in runtime system is available (type: bool)
ts:setting "webvisualization_insertbydefault"	1=A “WebVisualization” object is automatically inserted in the project tree below the Visualization Manager object (type: bool)
ts:setting "targetvisualization_insertbydefault"	1=A “TargetVisualization” object is automatically inserted in the project tree below the Visualization Manager object (type: bool)
ts:setting "transferfilestoplc"	1=The visualization files get transferred to the PLC (type: bool)
ts:setting "webvisualization_client"	1=The runtime requires the Web-Visualization (client), that doesn't need a visualization in the PLC (the Web- Visualization is completely calculated in the Web- Visualization applet). If this value is set to false, then the Web-Visualization is only interpreting paint command from the visualization running on the PLC.
ts:setting "supportlocalvisualizationfiles"	1=File Transfer settings can be made in the Visualization Manager dialog in the programming system; Note: activated for CODESYS Control Win V3 and CODESYS Control RTE V3 !  0=No File Transfer, settings available; default

**Section "keyboardusage", subsection of "visualization":**

Defines which keys and key modifiers are supported for the hotkey configuration in visualizations in CODESYS (if not available, see below "Standard Keys").

XML Tag	Description
ts:setting "availablemodifiers"	Defines exactly those key modifiers that are available on the device. The value must be a comma separated list of the generally available modifiers; possible values: "SHIFT", "CTRL" and "ALT". If this restricting setting is not available, all modifiers will be treated as if they were available. (type: string)
ts:setting "basekeys"	Allows the usage of a predefined set of keys. If no basekeys setting is given, all available keys must be defined in the section "additionalkeys". Possible values: "default": the keys defined in table "Standard Keys" (see below) are available (type: string)

**Section "additionalkeys", sub-section of "keyboardusage"**

Contains definitions of keys that are available for the keyboard usage in addition to the optional set of "basekeys"

XML Tag	Description
ts:setting "key name"	Defines a newly available key; the name must be a valid canonical desired key name. Value = key code; Example: ts:setting "EMERGENCY_STOP"; value: "300" (type: int)

**Section „notavailablekeys”, subsection of "keyboardusage"**

Allows to remove some of the keys that are previously defined (probably using the "basekeys") because they are not available on the current platform.

XML Tag	Description
ts:setting "<name of key> "	Removes a key that was previously defined. The value of such a key is the canonical name of a previously defined key. Example: ts:setting <BACKSPACE> Removing keys, that are not defined, results in a "no-operation". In fact, it does nothing at all. The value of this setting is ignored at the moment. (type: string)

If the device description contains no setting concerning key codes, this means that the following keys are supported by default:

**Standard Keys:**

Canonical Name	Key Code	Description
,A' b,s ,Z'	0x41-0x5A	letter keys
0 to 9	0x30-0x39	number keys
NUM0 to NUM9	0x60-0x69	number keys in number pad
F1 to F1	0x70-0x7B	function keys

Canonical Name	Key Code	Description
BACKSPACE	0x08	
TAB	0x09	
RETURN	0x0D	
PAUSE	0x13	
SPACE	0x20	
END	0x23	
HOME	0x24	
LEFT	0x25	
UP	0x26	
RIGHT	0x27	
DOWN	0x28	
PRINT	0x2A	
INSERT	0x2D	
DELETE	0x2E	
MULTIPLY	0x6A	*
ADDITION	0x6B	+
SUBTRACT	0x6D	-
COMMA	0x6E	,
DIVIDE	0x6F	/

An optional localization of the key names can be defined in a string table for namespace „keyboardusage“. See chap. 6.4.2 on localization strings.

See in the following a [localization example](#) for the keys EMERGENCY\_STOP and MACHINE\_STOPDOWN, which are assumed to be defined in the „additionalkeys“ section (see above):

```
<Strings namespace="keyboardusage">
  <Language lang="de">
    <String identifier="EMERGENCY_STOP">
      NOT AUS
    </String>
    <String identifier="MACHINE_SHUTDOWN">
      Maschine ausschalten
    </String>
  </Language>
  <Language lang="fr">
    <String identifier="EMERGENCY_STOP">
      Arrêt d'urgence
    </String>
    <String identifier="MACHINE_SHUTDOWN">
      Eteindre la machine
    </String>
  </Language>
  <Language lang="en">
```

```

<String identifier="EMERGENCY_STOP">
  Emergency Stop)
</String>
<String identifier="MACHINE_SHUTDOWN">
  Shut down machine
</String>
</Language>
</Strings>

```

**Section "TargetConstraints", subsection of "visualization":**

The settings within this section define restrictions valid for all visualizations inserted in the device tree below the device. There are no restrictions for any visualizations in the POU's tree! If visualizations from the POU's tree, which are not matching the restrictions given by the device, are used, compile errors will be issued.

XML Tag	Description
ts:setting "TargetImageFormats"	" " -separated list of supported image formats; Example: <ts:value>*.bmp *.jpg</ts:value> (type: string)
ts:setting "MaxNumOfVisualizations"	maximum number of visualizations which can be inserted below the device (type: int)
ts:setting "MaxNumOfElementsPerVisualization"	maximum number of visualization elements within a visualization (type: int)
ts:setting "SupportGradientFill"	If this setting is available, "gradient fill" for colors will be supported by the device (note: currently available for Windows, Linux, not however WinCE). If the setting is missing, compile error messages will appear with visualization projects containing gradient fill definitions.  Example: <ts:setting name="SupportGradientFill" type="boolean" access="visible"> <ts:value>1</ts:value> </ts:setting>

**Section "TargetFonts", subsection of section "TargetConstraints"**

**Section "<fontname>", subsection of section "TargetFonts"**

*fontname:* e.g. „Arial\_ CE“;

Only the fonts defined here will be available in the configuration dialogs of a visualization element. This restriction might be reasonable if projects are created for usage in a Target-Visualization.

If this section is not available, the default font selection will be available.

If no font styles are specified, the style options in the font dialog will be disabled.

XML Tag	Description
ts:setting "WindowsMatchingFont"	optional: if available, this windows matching font will be used for the display within the editor view; Example: <ts:value>Arial</ts:value> (type: string)
ts:setting "Sizes"	comma-separated list of possible font sizes; e.g. <ts:value>10,12,14</ts:value>

XML Tag	Description
	(type: string)
ts:setting "Styles"	comma-separated list of available font styles; possible values: default, bold, bold_cursive) Example: <ts:value>default,bold</ts:value> (type: string)

**Section "TargetColors", subsection of section "TargetConstraints"**

These settings restrict the color selection within the configuration dialogs of a visualization element. This might be reasonable if projects are created for usage in a Target-Visualization. For each color an own setting must be entered. If no color restrictions are defined, the default color selection dialog will be provided.

XML Tag	Description
ts:setting "<color>"	<i>color</i> ="Black", "Green" etc.; the value must specify the color id (type: uint);  Example: <ts:setting name="Black" example: type="uint" access="visible"> <ts:value>0xff000000</ts:value> </ts:setting>

**Section "TargetVisualElements", subsection of section "TargetConstraints"**

These settings define which visualization elements are available in the visualization editor toolbox. Either you specify a "|" -separated list of „AvailableElements“ or a list of „NotAvailableElements“.  
If no list is specified, all elements will be supported.

XML Tag	Description
ts:setting "AvailableElements"  resp. ts:setting „NotAvailableElements"	" " -separated list of elements which should be available resp. not available; possible values: element names as used in the visualization editor element toolbox; Example: <ts:value>Rectangle Circle</ts:value> (type: string)

**6.4.5.1.10 Online Manager**

Subsection "onlinemanager" of <ts:TargetSettings...>

Example:

```
<ts:section name="onlinemanager">
  <ts:setting name="communication-buffer-size" type="integer" access="edit">
    <!-- Communication buffer size. Is used to create an offline boot project (no
compact download!), where this is the size of the largest service. -->
    <ts:value>100000</ts:value>
  </ts:setting>
</ts:section>
```

XML Tag	Description
ts:setting "communication-buffer-size"	type: integer (bytes)  The setting defines the size for the layer 7 communication buffer. This is important for the handling of boot projects on small devices.  For device description versions < V3.5.0.0: If the setting is not available, CODESYS will create only one layer 7 service

XML Tag	Description
	<p>for every download regardless of the size of the boot project; so large boot projects possibly cannot be loaded to the target.</p> <p>For device description versions <math>\geq</math> V3.5.0.0: Default is 65536 (0x10000) which should work for most runtimes.</p> <p>If the setting is missing, an error will be reported when trying to create an offline boot application.</p>

#### 6.4.5.1.11 Recipe manager

In sub-section "RecipeManager" of <ts:TargetSettings...> device-specific settings as well as the default values for the Recipe Manager dialog can be defined.

Example:

```
<ts:section name="RecipeManager">
  <ts:setting name="StorageType" type="integer" access="visible">
    --- This value sets the default storage type -->
    <ts:value>1</ts:value>
  </ts:setting>
  <ts:setting name="FileExtension" type="string" access="visible">
    --- This value sets the default file extension -->
    <ts:value>.txtrecipe</ts:value>
  </ts:setting>
  <ts:setting name="FilePath" type="string" access="visible">
    --- This value sets the default file path -->
    <ts:value></ts:value>
  </ts:setting>
  <ts:setting name="TokenSeparator" type="string" access="visible">
    --- This value sets the default token separator -->
    <ts:value>:=</ts:value>
  </ts:setting>
  <ts:setting name="SelectedColumns" type="string" access="visible">
    --- This value sets the default selected columns -->
    <ts:value>0|1|2|3</ts:value>
  </ts:setting>
  <ts:setting name="SaveAsDefault" type="boolean" access="visible">
    --- This value can be used to disable the save as default button -->
    <ts:value>1</ts:value>
  </ts:setting>
  <ts:setting name="AutoSaveRecipes" type="boolean" access="visible">
    --- This value sets the default for the auto save recipes mode -->
    <ts:value>1</ts:value>
  </ts:setting>
</ts:section>
```

XML Tag	Description
ts:setting "StorageType"	Integer defining the type of storage; possible values: 0=Binary, 1=Textual
ts:setting "FileExtension"	String defining the default extension of the storage file; e.g. ".txtrecipe"
ts:setting "FilePath"	String defining the default storage location; e.g. "D:\rec_stor"
ts:setting "TokenSeparator"	String defining the default separator in case of storing to a textual file; possible values: Tab, Semicolon, Comma, Space, :=,

XML Tag	Description
ts:setting "SelectedColumns"	String defining the default list of "selected columns"; possible values: " " -separated list of the column numbers 0,1,2,3; e.g. if the available columns are Type, Name, Minimal Value, Maximal Value, then value "0 1 2 3" will effect that these four columns per default get entered as "selected columns"
ts:setting "SaveAsDefault"	Boolean; defines whether the "Save as default" button is enabled (1) or disabled (0)
ts:setting "AutoSaveRecipes"	Boolean; defines whether option "Save changes to recipes automatically" per default is activated (1) or deactivated (0)

#### 6.4.5.1.12 Symbolconfiguration

In sub-section "SymbolConfiguration" of <ts:TargetSettings...> device-specific settings concerning the memory for the Symbolconfiguration, which is handled as a "sub-application" can be defined.

Example:

```
<ts:section name="symbolconfiguration">
  <ts:setting name="max-area-size" type="integer" access="visible">
    <ts:value>10000</ts:value>
  </ts:setting>
  <ts:setting name="generate_as_separate_application" type="boolean" access="exit">
    <ts:value>0</ts:value>
  </ts:setting>
</ts:section>
```

XML Tag	Description
ts:setting "max-area-size"	Integer defining the maximum size [bytes] of the memory area for the symbolconfiguration
ts:setting "generate_as_separate_application"	Boolean: 0 = Symbolconfiguration is generated in father application 1 = Symbolconfiguration is generated as a separate child application [Default]

#### 6.4.5.1.13 Trace

IEC-Trace:

In sub-section "Trace" of <ts:TargetSettings...> device-specific settings concerning the memory for the Trace Configuration, which is handled as a "sub-application" can be defined.

**Note:** This setting affect only the internal IEC-Trace!

XML Tag	Description
ts:setting "max-area-size"	Integer defining the maximum size [bytes] of the memory area for the trace configuration

Example:

```
<ts:section name="trace">
  <ts:setting name="max-area-size" type="integer" access="visible">
    <ts:value>10000</ts:value>
  </ts:setting>
</ts:section>
```

### TraceManager:

With V3.4.2.0 the new TraceManager is available. For this new feature, the new CmpTraceMgr component in the runtime system is necessary. To enable this TraceManager in CODESYS, the following setting is necessary:

XML Tag	Description
ts:setting "tracemanager"	Bool value to enable (=true/1) the trace manager

Example:

```
<ts:section name="trace">
  <ts:setting name="tracemanager" type="boolean" access="visible">
    <!-- With this option, the new tracemanager is activated in CODESYS.
         With this option, the CmpTraceMgr component is needed in the runtime system! -->
    <ts:value>1</ts:value>
  </ts:setting>
</ts:section>
```

#### 6.4.5.1.14 Object Type Restrictions

You can add a sub-section "object-type-restrictions" of <ts:TargetSettings...> where you explicitly define the types of objects which should be available when creating the application in the programming system.(if the section is missing, there will be no restriction on the availability of object types):

XML Tag	Description
ts:setting "type-guid"	String, specifying the type GUID of the object type
ts:setting "interface"	String, specifying the assembly-qualified name of the interface

There can be arbitrarily many "type-guid" and "interface" settings. If the section is present, only objects which match at least one of the criteria can be added to any subobject of the device. If the criteria is not matched, the object cannot be created, pasted, imported, dropped, etc.

Note: If an "Update Device" operation is performed, object types that are becoming unsupported will not be deleted. Instead, a corresponding warning message will be displayed in the Message View.

Example: The following target section restricts the possible set of object types to POUs and actions:

```
<ts:section name="object-type-restrictions">
  <ts:setting name="type-guid" type="string" access="visible">
    <ts:value>{6F9DAC99-8DE1-4efc-8465-68AC443B7D08}</ts:value>
  </ts:setting>
  <ts:setting name="interface" type="string" access="visible">
    <ts:value>_3S.CoDeSys.ActionObject.IActionObject,ActionObject</ts:value>
  </ts:setting>
</ts:section>
```

### 6.4.6 Custom tags

Custom tags can be found in many places within the description file. They can be used to store customer-specific information. They are not transferred to the target system, although they can be evaluated via in the programming system via Plugins. A custom tag may have any number of sub-nodes. In order to keep the description file validatable, all nodes used should be defined in a scheme, and this scheme should be referenced.



### 6.4.7 Strings

The description files offer a mechanism for localized text display. The texts can be created in the form of tables and referenced via an identifier. Each table is identified via a unique *namespace*. The tables are separated by different namespaces in order to prevent overlaps of identifiers in different tables. Texts can be stored in each table in all supported languages. The *Strings* section is currently the only table defined in the description file. Further tables (e.g. import of general texts) will be supported in the future.

Within the description file texts are referenced via nodes of type *StringRef*. The node reference has the form "<namespace>:<identifier>". The content of a *StringRef* node is used as default value if no entry is available in a suitable language and should therefore be written in English.

Each *language* node contains all texts for a language. The language identifier has the form <language code>-<country code> ("en-us", "de-au", ...). The default language is English ("en-en"). Under this node there is a *String* node for each stored text. The *name* attribute matches the text identifier and must not contain a colon.

### 6.4.8 Types

The type system of the description files is based on the IEC type system. However, not all IEC types are currently available. A number of IEC types are made available as standard data types. More complex types such as structures (*StructType*), enumerations (*EnumType*), range types (*RangeType*) and bit fields (*BitfieldType*) can be defined in this section.

Types are always referenced via a namespace and the type name as follows: "<namespace> : <name>". All basic types are located in the "std" namespace. The following basic types are available:

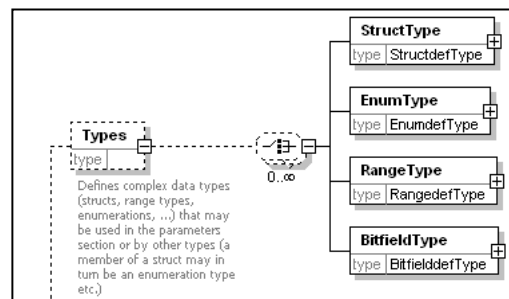


Figure 9: The "Types" node

std:BOOL <sup>5</sup>	Truth value (TRUE or FALSE)	std:REAL	Floating-point numbers, 32 bits
std:BYTE	8 bits, logic	std:LREAL	Floating-point numbers, 64 bits
std:SINT	8 bits, numeric	std:STRING	Character string, 80 characters max.
std:USINT	8 bits, numeric, unsigned	std:BIT	Single bit (for inputs/outputs).
std:WORD	16 bits, logic		
std:INT	16 bits, numeric		
std:UINT	16 bits, numeric, unsigned		
...	...		
std:ULINT	64 bits, numeric, unsigned		

The *Types* node has an attribute:

- *namespace*: This name defines the namespace for all types that were defined under this node.

*Structures represent a composition of several elements, as is common practice in programming languages.* These elements may be based on standard types, or further types may have been defined. Recursion of structures (directly or indirect) is not permitted.

The *StructType* has a *name* attribute. This name is used for referencing the type. Within the type any number of *Component* nodes can be defined, which define the individual elements of the structure. Each component has two attributes:

<sup>5</sup> Important: BOOL is internally mapped to a full byte and has the size 8 bits. This type is unsuitable for input/output channels representing an individual bit. In this case the data type std:BIT should be used.

Identifier: Corresponds to the variable names of the structure member in IEC. It is used in CODESYS for accessing this structure member.

The nodes under a component describe this in more detail:

- *Default* contains the *DefaultValue* for this structure component. Depending on the type of the component the content is made up as follows (as at any point where a *ValueType* is expected)
  1. *Simple type* (int, string, bool, ...): The associated value as string.
  2. *Enum*: The identifier of the value used.
  3. *Complex (composite) type*, for example a further structure:  
For each subvalue to be set an element node containing the value of the element (complex if required).
- *VisibleName* contains the component name to be displayed at the interface (a more user-friendly representation of the identifier). For the *VisibleName* any characters are permitted (including spaces and special characters). The content can be localized (→StringRef).
- *Unit*: an optional unit designation for the value used. This value can be localized.
- *Description* of the node (can be localized).
- *Custom* enables customer-specific content to be stored that is not interpreted by the system (see *CustomType* section). A PlugIn can access these data during runtime. There is no transfer to the target system.

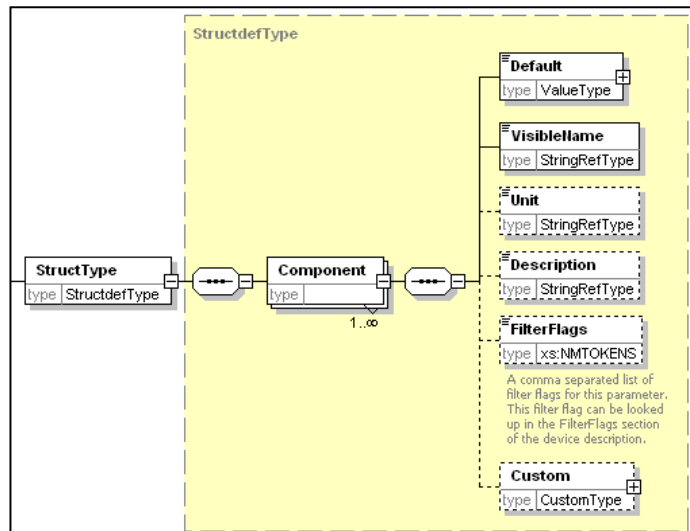


Figure 10: Structure definition

## 6.5 Device administration

The descriptions of all devices to be used in CODESYS V3 must be installed with CODESYS. The description files are stored in an internal “database” referred to as *Device Repository*. The repository can handle different versions of a device description at the same time. To this end it is necessary to use versioning for description files, i.e. a version number is incremented internally if the description changes. Installation of a modified description file with the same version number will overwrite an existing version.

The repository expects the device description in the form of an XML file. These are described in the following chapter. Import filters are made available for alternative description files (e.g. fieldbus-specific: EDS, GSD, ...). The description files are converted to the standard description (as far as possible and meaningful). For specialized editors (e.g. a Profibus configurator) the original file is generally also stored in the repository so that it can be accessed. Both files are copied into the repository, i.e. changes in the original file have no effect until it is reimport–d - here too it is important to increment the version number.

To access a device description it must be referenced uniquely. A combination of three values is used for this purpose:

- *Type* (16 bit unsigned): Device type  
The possible types are defined by 3S. The type also determines the format of the next two values. A list of defined types can be found in the Appendix.  
Ids from 0x8000 can be used for manufacturer-specific modules that use internal or non-standardized busses.  
Examples:
  - 0x20: Profibus master

- 0x21: Profibus slave (imported from GSD file)
- 0x1000: CODESYS-programmable device (described as XML file).
- *Id (string)*: Uniquely identifies the device description within a type. Device descriptions with different types can use the same Id. The definition of the Id as a string offers a certain degree of flexibility regarding the format required for importing alternative description files. For all types  $\geq 0x1000$  the format is defined as follows:

```
####<space>####
```

with “#” representing exactly one hexadecimal digit. The first four digits form a manufacturer Id that is allocated by 3S. It is the same Id that is also used in the runtime system, for example. The second group of four digits can be assigned freely by the manufacturer (it should be different for each device).

Example: manufacturer ID is 0x1f3, internal device Id is 0x23, in which case the device Id would be “01F3 0023”

- *Version (string)*: Indicates the version of a device description. Here too the string enables a variable format for importing alternative description formats. For all types  $\geq 0x1000$  the format is specified as follows:

```
n.n.n.n
```

with “n” representing any decimal number, e.g. “1.0.17.3”.

Within a device description it is possible to describe further modules that are only used in the context of this device description. In principle they are described and used in the same way as a device, although they can only be used as a child of the main device or another module defined in the main device. Modules are identified through the device identification together with an additional module Id.

- *Module Id (string)*: Identifies a module within a device description. The format is not free.

## 6.6 Save and Restore Changed IO Configuration Parameters

With the following setting for the CmpIoMgr component in the configuration file of the runtime, you can activate to store all changed parameter values in the file "IoConfig.ar":

```
[ CmpIoMgr ]
StoreChangedParameters=1
```

This file is written always if a parameter has been changed online or via IoMgrWriteParameter(). All changed parameters are restored at the next bootup, right after the bootproject with the IO-Configuration is loaded.

## 7 I/O Drivers

An I/O driver is a component that operates and supports a specific hardware or device. Such an I/O driver typically has to perform the following functions:

- Physical access to the device
- Detection and initialization of the device
- Data access to and data exchange with the device
- Support of diagnostic information of the device
- Optional functions like scanning sub devices, parameter access, etc.

### 7.1 Concept

In CODESYS V3, an I/O driver can be implemented in ANSI-C/C++ and also in IEC! This sounds strange, because in IEC you typically have no access to the hardware. But in CODESYS V3, the hardware access can be realized by so called System-Libraries. These libraries offer the possibility to access for example a PCI bus (SysPCI.library), physical memory (SysShm.library) or device ports (SysPort.library).

An I/O driver will always be called by the runtime system in the appropriate moments. These moments are for example:

- At download of an application to read in I/O-configuration (see previous chapter)
- At the beginning of a task to read in input channels
- At the end of a task to write out output channels
- At sending an online service to read or write an I/O driver parameter

To enable this, the I/O driver has to implement one mandatory interface (IBase) and can implement some optional interfaces (dependent of the supported features). These interfaces are declared in the next chapter. The interface functions are typically called by the so called I/O-manager. This component manages all I/O-drivers independent of their implementation. An I/O driver in IEC will be represented by a wrapper implemented in C/C++ against the I/O-manager, so from this point of view every I/O driver looks the same.

Each physical I/O device is typically supported by one *instance* of an I/O driver.

In the case of a C/C++-driver, the driver first has to (auto) detect its supported cards at startup of the runtime system and has to create one instance for each detected device.

In the case of an IEC-driver, the driver is automatically instantiated by CODESYS, if a device is appended in the graphical PLC-configuration. An auto detection of the device must be done at download time of the application and the physical device must be assigned to each instance of the driver afterwards.

The instantiation and assignment of a physical device to an I/O driver instance is described in chapter 7.5 in detail.

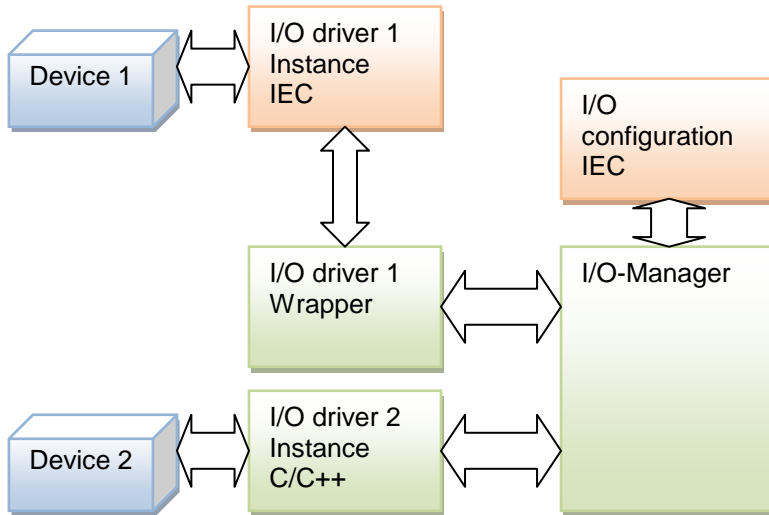


Figure 11: Architecture calling I/O-drivers

To simplify start writing an I/O-driver, we provide a toolkit with the source code frames of I/O-drivers and some description files (XML). We provide an I/O driver frame in C and in IEC. You can find the source code and the description files typically under `$(Components)\IoDriver\IoDrvTemplate`.

In the CODESYS V3 I/O-concept, there are 4 different layers of functionality that the I/O-concept based on (see figure below).

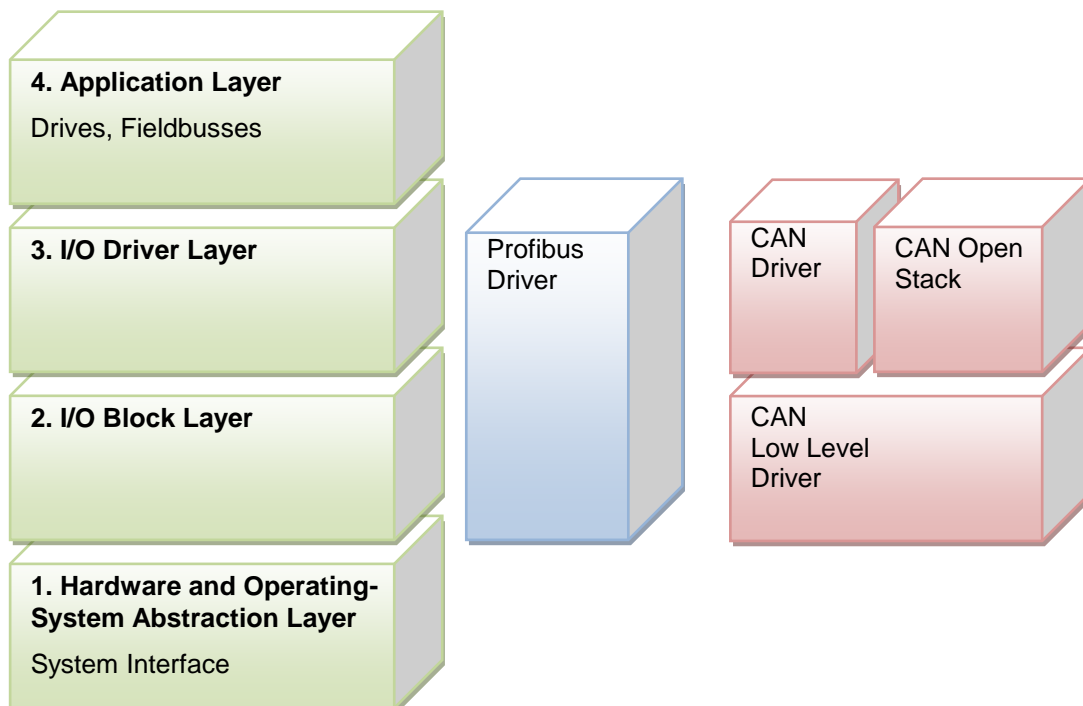


Figure 12: I/O model in CODESYS V3

In the figure above, you can see the four layers at the left side. The base level is the hardware- and operating system abstraction layer. This layer provides a standard interface for every hardware access (e.g. Shared-Memory, PCI, Interrupt-Handling, etc.).

## 7.2 Main I/O Driver Interfaces

To understand how an I/O driver works internally, we need to take a look at the interfaces that an I/O driver has to implement.

If the driver is written in C/C++ you can find the available interfaces in the directory components. The interface header files always begin with CmpIoDrv and ends (like every interface) with Itf.h:

CmpIoDrv...Itf.h

If you intend to write an I/O driver in IEC, you can find the interface in the corresponding libraries with the following structure:

IloDrv... .library

### 7.2.1 IBase

The IBase interface is the mandatory interface that every I/O driver has to implement! It Contains the following functions:

IBase (C/C++, the Parameter pIBase is only needed for C-drivers):

1. **void \*QueryInterface (IBase \*pIBase, ITFID ItfId, RTS\_RESULT \*pResult)**
2. **int AddRef (IBase \*pIBase)**
3. **int Release (IBase \*pIBase)**

IBase (IEC):

1. **POintER TO BYTE QueryInterface (ITFID ItfId, POintER TO UDint \*pResult)**
2. **Dint AddRef()**
3. **Dint Release()**

With the QueryInterface() function, an I/O driver can be requested for an interface, that the I/O driver implements. The corresponding interface pointer is returned by this function.

An interface is always specified and can be requested by an ID. To save memory, the ID is only a 32-Bit number and no GUID. But this causes to manage all interface IDs. The ID consists of the high word vendor Id and the low word interface Id. So every vendor can create its own interfaces.

A list of all available interface IDs that are used in the runtime system can be found in the header file CmplItf.h or for IEC-driver in the corresponding interface libraries.

The AddRef() function is always called implicitly, if a QueryInterface call was successful to increase a reference counter of this object.

The Release() can be called to release an interface pointer, that was provided by QueryInterface. This function decrements the reference counter of an object.

If the reference counter is 0, the object will be deleted.

### 7.2.2 ICmpIoDrv

The most important interface, that an I/O driver can implement, is the ICmpIoDrv interface. It contains the following functions for different issues:

Identification functions:

```
typedef struct tagIoDrvInfo
```

```
{
```

```
    RTS_IEC_WORD wId;                // Index of the instance
    RTS_IEC_WORD wModuleType;        // Supported module type
    RTS_IEC_DWORD hSpecific;         // Specific handle
    RTS_IEC_string szDriverName[32]; // driver name
    RTS_IEC_string szVendorName[32]; // vendor name
    RTS_IEC_string szDeviceName[32]; // device name
    RTS_IEC_string szFirmwareVersion[64]; // Firmware version
    RTS_IEC_DWORD dwVersion;         // Version of the driver
```

```
} IoDrvInfo;
```

### **RTS\_RESULT IoDrvGetInfo(IoDrvInfo \*\*ppIoDrv):**

With this function, some generic information can be requested like driver name (see structure above), device name that is supported by the driver, vendor name and firmware number (if the supported device has an own firmware on it like the hilscher cards).

### **RTS\_RESULT IoDrvIdentify(IoConfigConnector \*pConnector):**

By calling this function, the I/O driver should identify its device, like blinking some LEDs, stopping the bus, etc. It is planned to call this function by a menu action on the device in the plc-configuration to physically identify this device. This could be very useful, if there are several identical cards plugged in the plc and the assignment in the plc-configuration in CODESYS is unclear.

Configuration: (called during application download)

### **RTS\_RESULT IoDrvUpdateConfiguration(IoConfigConnector \*pConnectorList, int nCount):**

This function is called at download of the application that contains the I/O-configuration. Each driver instance gets the complete list of connectors!

The first thing that must be done in this function is to detect the connector that is supported by the I/O-driver. For this, the I/O driver can request the Io-manager for the first connector with the specified type Id, like:

```
pConnector = CAL_IoMgrConfigGetFirstConnector(pConnectorList, &nCount, 0x0020);
```

Here, the first connector with the Id 0x0020 (=CT\_PROFIBUS\_MASTER) is searched. See chapter 6.3.1 for detailed information.

If the first connector with the matching type is found (pConnector is unequal 0), it must be checked:

- if it is the correct supported device
- if the connector is not supported already by a previous instance

To check, if it is the correct device, typically some additional parameters are used to detect this like vendor name, device name or specific device id.

To check if the connector is free and can be used and it is not occupied by another instance, therefore the connector entry hIoDrv must be checked for 0 or -1. In both cases, the I/O-connector is free and can be used. To occupy the connector, the driver has to write its handle into the connector.

So typical sequence of IoDrvUpdateConfiguration looks like in C:

```
IBase *pIBase;
IoConfigConnector *pConnector = CAL_IoMgrConfigGetFirstConnector(pConnectorList, &nCount,
CT_PROFIBUS_MASTER);
```

```
while (pConnector != NULL)
{
```

```
    IoDrvInfo *pInfo;
    IoConfigParameter *pParameter;
    char *pszVendorName = NULL;
    char *pszDeviceName = NULL;
```

```
    IoDrvGetInfo(hIoDrv, &pInfo);
```

```
    pParameter = CAL_IoMgrConfigGetParameter(pConnector, 393218);
    if (pParameter != NULL && pParameter->dwFlags & PVF_POINTER)
        pszVendorName = (char *)pParameter->dwValue;
    pParameter = CAL_IoMgrConfigGetParameter(pConnector, 393219);
    if (pParameter != NULL && pParameter->dwFlags & PVF_POINTER)
        pszDeviceName = (char *)pParameter->dwValue;
```

```

if (pConnector->hIoDrv == 0 &&
    pszVendorName != NULL && strcmp(pszVendorName, pInfo->szVendorName) == 0 &&
    pszDeviceName != NULL && strcmp(pszDeviceName, pInfo->szDeviceName) == 0)
{
    pConnector->hIoDrv = (RTS_IEC_DWORD)pIBase;
}

```

In IEC you can find the appropriate sequence in the template driver. It looks quite the same.

After detecting the right connector, the next step in the function `IoDrvUpdateConfiguration` is to configure the physical device with the connector parameters and optional to detect all slaves (if is a fieldbus master).

To detect the slaves, the I/O-manager provides some interface functions too:

```
pChild = CAL_IoMgrConfigGetFirstChild(pConnectorList, &nCount, pConnectorFather);
```

With this function, the first child of the father connector `pConnectorFather` was returned.

The next child (slave) can be requested by:

```
pChild = CAL_IoMgrConfigGetNextChild(pChild, &nCount, pConnectorFather);
```

**ATTENTION:** The driver must register its instance at each supported connector (also PCI connectors, slaves, etc.)! This must be done in the `hIoDrv` component of the corresponding connector, like:

```
pChild->hIoDrv = (RTS_IEC_DWORD)pIBase;
```

#### **RTS\_RESULT IoDrvUpdateMapping(IoConfigTaskMap \*pTaskMapList, int nCount):**

The driver is called with the so called task map list. A Task map contains the following information:

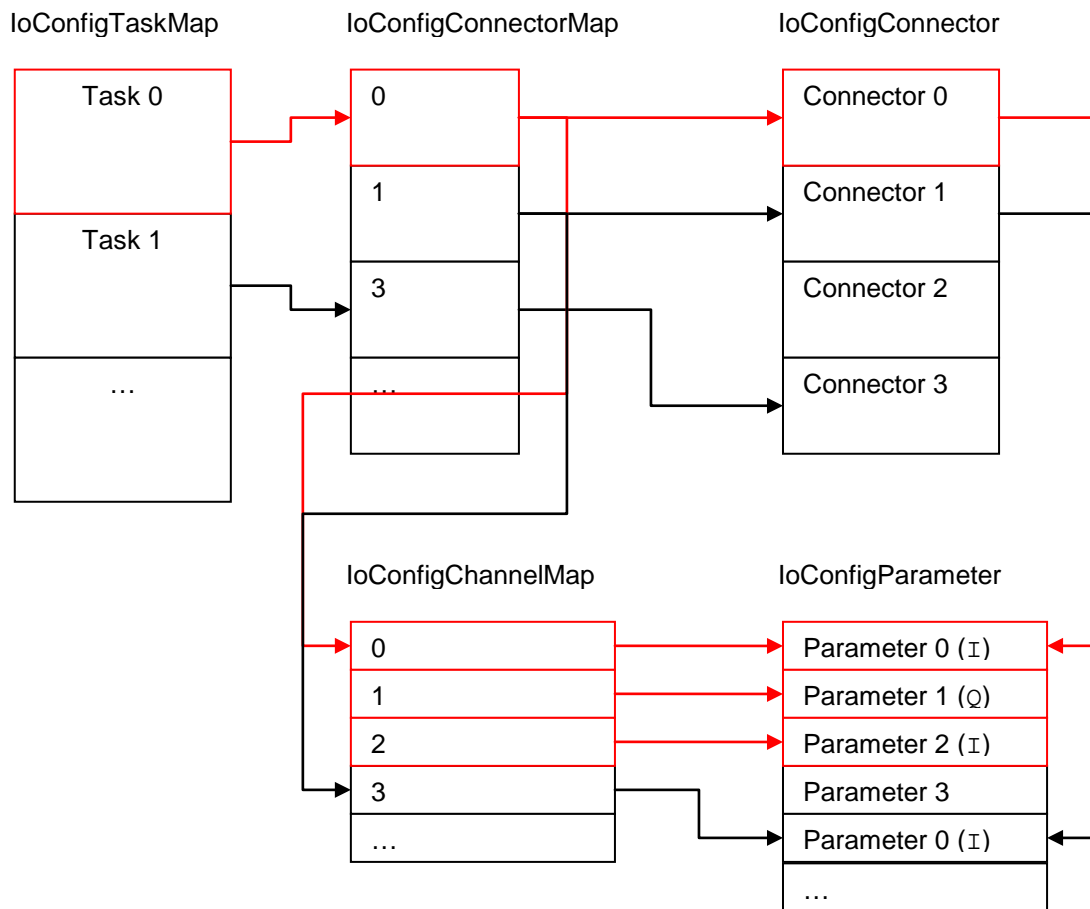
Element	IEC Data type
Task ID	DWORD
Type (Input or Output)	WORD
Number of connector maps	WORD
Pointer to connector map list	POintER

A connector map list contains the following information:

Element	IEC Data type
Pointer to a connector	DWORD
Number of channel maps	WORD
Pointer to channel map list	POINTER



The complete I/O-mapping structure is shown in the following picture:



**Figure 13: Mapping structure**

These are the missing bricks to understand the I/O-mapping.

The task map contains all mapping information for each task, that means, all I/O-channel that are use by a task. For each task you got one entry for all inputs and one entry for all outputs.

The task map contains a list of connectors maps, that means on which connectors the I/O-channels are residing.

And at least, the connector map entry contains a list of channel maps, which includes the real mapping information, where to copy the inputs to which offset on the device and where to copy the outputs from the device to which offset in the application.

In the mapping table, the I/O driver can sort or rearrange entries to optimize later cyclic access. E.g. several byte channels can be collected to one byte stream, to use one memcpy at the cyclic update.

#### Cyclic Calls:

#### **RTS\_RESULT IoDrvReadInputs(IoConfigConnectorMap \*pConnectorMapList, int nCount):**

This interface function is called at the beginning of a task to read in all referred input values of this driver. Only one call is done for each task.

#### **RTS\_RESULT IoDrvWriteOutputs(IoConfigConnectorMap \*pConnectorMapList, int nCount):**

This interface function is called at the end of a task to write out all referred output values of this driver. Only one call is done for each task.

#### **RTS\_RESULT IoDrvStartBusCycle(IoConfigConnector \*pConnector):**

This function is used to trigger a bus cycle (if necessary on the device). This can be specified in the device description as followed, if a bus cycle is necessary (see chapter 6.4.4.4):

```
<DriverInfo needsBusCycle="true">
...
</DriverInfo>
```

In which context this function is called can be specified in the IO-configuration.

On the device dialog in the register card "PLC settings", you can specify a dedicated "bus cycle task". If no task is specified here, the task with the shortest cycle time is used out of the task configuration. With the attribute "useSlowestTask" you can specify in your device description, that the slowest task does the bus cycle (see chapter 6.4.4.4).

On the device (e.g. Master), there is a register card "Mapping", where you can specify an optional bus cycle task. If no task is specified here, the configuration of the device is used (see above) as default.

### **RTS\_RESULT IoDrvWatchdogTrigger(loConfigConnector \*pConnector):**

This function is called cyclically to retrigger a watchdog on the device. The cycle time must be calculated in the I/O-driver.

#### Scanning sub devices/modules:

### **RTS\_RESULT IoDrvScanModules(loConfigConnector \*pConnector, loConfigConnector \*\*ppConnectorList, int \*pnCount):**

This function is called to scan sub devices. This can be used to scan physically available slaves of fieldbus master, that are connected to one fieldbus.

It is necessary to enable the scan mechanism in the device description file. This is done by an additional xml element in the <DriverInfo> section.

Example:

```
<DriverInfo>
  <Scan supported="true" identify="true"></Scan>
</DriverInfo>
```

The scan function is enabled for the device and the command is enabled in the context menu if the device is selected in the device tree.

The identify attribute enables the call of IoDrvIdentify to identify a scanned device. In most cases a LED is blinking to show the user the selected device. It is helpful for bus systems without DIP switches for address setting like sercos or ProfiNet.

In the method IoDrvScanModules the connected devices have to be returned.

With versions before V3.5 SP2 the method is called only once and therefore all devices must be scanned at once.

With version from V3.5 SP2 it is possible to return ERR\_PENDING. In that case the method IoDrvScanModules is called again and the method could return just the found number of devices or 0 if the stack needs additional time or calls to collect the available devices.

The programming and runtime system must support this and it will lower the required memory for the scan function.

If flag ConnectorOptions.CO\_SCAN\_PENDING\_SUPPORTED is set in pConnector^.wOptions then it is possible to use the pending functionality.

Descriptions of parameters:

loConfigConnector \*pConnector

It contains the connector for the device (for example master) and the parameters for starting the scan function.

loConfigConnector \*\*ppConnectorList

Inside the method this parameter has to be set to the memory containing a list of connectors for all devices found. If CO\_SCAN\_PENDING\_SUPPORTED is not available the memory has to be

allocated dynamically and freed in FB\_Exit or before the next scan call.  
If CO\_SCAN\_PENDING\_SUPPORTED is available the method could return a pointer to one instance of IoConfigConnector.

```
int *pnCount
```

The method has to set the number of connectors (=devices + sub devices) stored in the ppConnectorList.

Devices and sub devices could be returned in the connector list. The sub devices must be directly copied to the memory behind the devices then the scan mechanism could automatically assign the sub devices.

Memory content for IoConfigConnector

```
Device 1
Sub device 1.1
Sub device 1.2
Device 2
Sub device 2.1
```

Example in IEC working with all versions:

Member variables for function block:

```
m_pScanConnector: POINTER TO IoConfigConnector; // only necessary to free allocated
// memory
m_diScannedSlaves: DINT;
```

Declaration:

```
METHOD IoDrvScanModules : UDINT
VAR_INPUT
  pConnector : POINTER TO IoConfigConnector;
  ppConnectorList : POINTER TO POINTER TO IoConfigConnector;
  pnCount : POINTER TO DINT;
END_VAR
VAR
  pSlaveConnector: POINTER TO IoConfigConnector;
  pSlaveParameters: POINTER TO IoConfigParameter;
  diCount: DINT;
  dwParamCount : DWORD;
  stComponent : STRING := 'Test';
  wSlaves: WORD;
  wLen: WORD;
  bFailed: BOOL;
  dwVendorID: DWORD := 0;
  dwDeviceId: DWORD := 0;
  dwRevision : DWORD := 0;
  udiResult: UINT;
  stModuleID: STRING;
  wConnectorCount: WORD;
  bScanWithPending : BOOL;
END_VAR
```

Implementation:

```
IoDrvScanModules_Count := IoDrvScanModules_Count + 1;
// Counter for debugging. Shows that IoDrvScanModules is called
{IF defined (variable:ConnectorOptions)}
  bScanWithPending := pConnector^.wOptions = ConnectorOptions.CO_SCAN_PENDING_SUPPORTED;
  // Check for version V3.5 SP2. If flag is set the return value ERR_PENDING could be
  used.
{END_IF}
IF m_pScanConnector <> 0 AND m_diScannedSlaves > 0 THEN
  // free memory from the last scan
  pSlaveConnector := m_pScanConnector;
  FOR diCount := 1 TO m_diScannedSlaves DO
    pSlaveParameters := pSlaveConnector^.pParameterList;
    FOR dwParamCount := 1 TO pSlaveConnector^.dwNumOfParameters DO
      // Free the memory for the parameters
      IF (pSlaveParameters^.dwFlags AND 16#2) = 16#2 THEN
        // dwValue is pointer
        IF pSlaveParameters^.dwValue <> 0 THEN
          SysMemFreeData(stComponent,pSlaveParameters^.dwValue);
        END_IF
      END_IF
    END_FOR
    pSlaveParameters := pSlaveParameters + SIZEOF(IoConfigParameter);
  END_FOR
  // Free the parameter list
```

```

        SysMemFreeData(stComponent,pSlaveConnector^.pParameterList);
        pSlaveConnector := pSlaveConnector + SIZEOF(IoConfigConnector);
    END_FOR
    // Free the connectors
    SysMemFreeData(stComponent,m_pScanConnector);
    m_pScanConnector := 0; // Mark memory as freed
    m_diScannedSlaves:=0;
    IF bScanWithPending THEN
        // Scan return ERR_PENDING to free the allocated memory
        // now return ERR_OK to finish the scan process
        IoDrvScanModules := Errors.ERR_OK;
    END_IF
END_IF
// to-do: get the number of slaves
wSlaves := 1;
// Example for one device
// Number of Slaves is now known -> allocate memory
pSlaveConnector^ :=
SysMemAllocData(stComponent,wSlaves*SIZEOF(IoConfigConnector),ADR(udiResult));
// For old version allocate the necessary memory for all device.
// With V3.5 SP2 it is possible to return only one device for each call of
IoDrvScanModules
// Therefore it is not necessary to allocate memory dynamically. it could be also a
member variable of the function block
IF ppConnectorList = 0 THEN
    // Not enough memory
    IoDrvScanModules := Errors.ERR_FAILED;
    RETURN;
END_IF
ppConnectorList^ := pSlaveConnector;
// Set the return value of the method to the IoConfigConnector memory.
// Store the memory pointer and size for freeing the memory after scan
m_pScanConnector := pSlaveConnector;
m_diScannedSlaves := wSlaves;
wConnectorCount := 0;
FOR diCount := 1 TO WORD_TO_DINT(wSlaves) DO
    pSlaveParameters := SysMemAllocData(stComponent, 4 *
SIZEOF(IoConfigParameter),ADR(udiResult));
    // At least 4 parameters have to be returned for each connector.
    IF pSlaveParameters <> 0 THEN
        bFailed := FALSE;
        // to-do: get information from device, vendor id, product, revision etc.
        // anything that is needed to find the matching device description in the
        // repository
        IF NOT bFailed THEN
            // device information successfully read
            pSlaveConnector^.wType := 32768; // DeviceID of device as in device
            //Description <DeviceIdentification><Type>
            pSlaveConnector^.dwNumOfParameters := 4; // 4 parameters minimum
            pSlaveConnector^.pParameterList := pSlaveParameters
            // store the parameters vendor id
            pSlaveParameters^.dwParameterId := 1; // Vendor ID is always 1
            pSlaveParameters^.dwValue := dwVendorID;
            pSlaveParameters^.wLen := 32; // Bitlength
            pSlaveParameters^.wType := TypeClass.TYPE_DWORD;
            pSlaveParameters^.dwFlags := 16#34; // Value is a direct value
            // next parameter device id
            pSlaveParameters := pSlaveParameters + SIZEOF(IoConfigParameter);
            pSlaveParameters^.dwParameterId := 2; // Product ID is always 2
            pSlaveParameters^.dwValue := dwDeviceID;
            pSlaveParameters^.wLen := 32; // Bitlength
            pSlaveParameters^.wType := TypeClass.TYPE_DWORD;
            pSlaveParameters^.dwFlags := 16#34; // Value is a direct value
            // next parameter revision
            pSlaveParameters := pSlaveParameters + SIZEOF(IoConfigParameter);
            pSlaveParameters^.dwParameterId := 3; // Revision ID is always 3
            pSlaveParameters^.dwValue := dwRevision;
            pSlaveParameters^.wLen := 32; // Bitlength
            pSlaveParameters^.wType := TypeClass.TYPE_DWORD;
            pSlaveParameters^.dwFlags := 16#34; // Value is a direct value

            // next parameter devicestring for search of corresponding device in the
            // repository
            stModuleID := '0000 0001';
            // It is the same string as the <DeviceIdentification><ID> element.
            wLen := INT_TO_WORD(len(stModuleID))+1;
            pSlaveParameters := pSlaveParameters + SIZEOF(IoConfigParameter);
            pSlaveParameters^.dwValue :=
                SysMemAllocData(stComponent,wLen,ADR(udiResult));
            IF pSlaveParameters^.dwValue <> 0 THEN
                pSlaveParameters^.dwParameterId := 4; // Device ID is always 4
                SysMemCpy(pSlaveParameters^.dwValue,ADR(stModuleID),wLen);
            END_IF
        END_IF
    END_IF
END_FOR

```

```

        pSlaveParameters^.wLen := wLen * 8; // Bitlength
        pSlaveParameters^.wType := TypeClass.TYPE_STRING; // type string
        pSlaveParameters^.dwFlags := 16#32; // Pointer to data
    END_IF
    pSlaveConnector := pSlaveConnector + SIZEOF(IoConfigConnector);
    wConnectorCount := wConnectorCount + 1;
END_IF
END_IF
END_FOR
pnCount^ := wConnectorCount;
// Set the number of devices successfully found
IF bScanWithPending THEN
    // Pending is supported then just return the found number of devices and set
    ERR_PENDING
    // IoDrvScanModules is called again to free the allocated memory.
    IoDrvScanModules := Errors.ERR_PENDING;
ELSE
    IoDrvScanModules := Errors.ERR_OK;
    // Pending is not available. Return all found devices. Allocated memory will be freed
    either in the next scan call or must be
    // freed in FB_Exit to prevent a memory loss.
END_IF

```

If only V3.5SP2 and later should be supported then this could be used:

**Member variables for function block**

```

m_ScanConnector: IoConfigConnector;
m_aSlaveParameters: ARRAY[0..3] OF IoConfigParameter;
m_stDeviceId : STRING;

```

In the implementation there is no need to dynamically allocate or free memory. The IoDrvScanModules just returns always 1 found device or 0 if nothing found.

```

m_ScanConnector.pParameterList := ADR(m_aSlaveParameters[0]);
m_aSlaveParameters[3].dwValue := ADR(stDeviceId);
ppConnectorList^ := ADR(m_ScanConnector);
pnCount^ := 1;

```

The method has to return Errors.ERR\_OK if all devices are done.

Information to the parameters:

**Fixed parameter ids returned by IoDrvScanModules**

Parameter ID	Type	Description	Mandatory/Optional
1	DWORD	Vendor ID	M
2	DWORD	Product number	M
3	DWORD	Revision	M
4	STRING	<DeviceDescription><ID>	M
5	DWORD	Slot index for slot devices (0 first slot)	O
6	STRING	Reserved for special data types	O
7	BOOL	Used for IoMgrIdentify True, if identify enabled.	O
8	WORD	ModuleTypeCode Used for module type code of connector to find the correct connector if more than one connector is possible to add the devices	O

**Additional parameters:**

It is possible to add additional parameters starting with ID 10. For example the station name or node id could be passed to the device scan dialog. Additional parameters will be shown in an extra column. If the parameter ID is also available in the device description then the parameter values will automatically copied to the devices after inserting the devices. The access rights of the parameter are set to readwrite then the column is editable.

Diagnostic information:**RTS\_RESULT IoDrvGetModuleDiagnosis(IoConfigConnector \*pConnector):**

With this function, device specific diagnostic information are stored in the connector (diagnostic flags).

**7.2.3 ICmploDrvParameter**

This interface is used to get access to the system parameter of a device.

**RTS\_RESULT IoDrvReadParameter(IoConfigConnector \*pConnector, IoConfigParameter \*pParameter, void \*pData, unsigned long ulBitSize, unsigned long ulBitOffset):**

With this function, the I/O-manager reads the value of a device parameter. This function is typically called, if an online-service with a parameter read request is sent to the I/O-manager.

**RTS\_RESULT IoDrvWriteParameter(IoConfigConnector \*pConnector, IoConfigParameter \*pParameter, void \*pData, unsigned long ulBitSize, unsigned long ulBitOffset):**

With this function, the I/O-manager writes the value of a device parameter. This function is typically called, if an online-service with a parameter write request is sent to the I/O-manager.

**7.3 Optional Interfaces**

There are several specific optional interfaces that an I/O driver can implement. These interfaces can be device or bus specific. Optional interfaces are for example:

ICmploDrvDPV1: Interface for the DPV1 Profibus protocol

ICmploDrvBusControl: Interface to control a fieldbus (Start, Stop, Reset)

ICmploDrvProfibus: Interface for Profibus Sync/Freeze feature

**7.4 I/O Manager**

The I/O-Manager is the central component for managing all I/O-drivers and providing the access to the I/O-configuration. The name of the component is CmploMgr and the interface is named ICmploMgrItf.

IO data consistency is realized in the generic part of the CmploMgr component. Thus it is not necessary that each IO driver separately takes care of this. If there is an active IO data exchange for reading inputs, writing outputs, or starting bus cycle on an IO-driver instance, each part will be protected against getting called from any other. This protection is done by a processor atomic bit operation and not via a semaphore and thus is a non-blocking operation.

An I/O driver must register its instance after creating at the I/O-Manager. Therefore, two functions of the interface must be used:

**RTS\_HANDLE IoMgrRegisterInstance(IBase \*pIBase, , RTS\_RESULT \*pResult):**

This function should be used to register your I/O driver instance. Therefore the IBase pointer must be transmitted to the I/O-Manager.

A handle to the internal management entry is returned. This handle can be used to remove the instance from the I/O-Manager.

**RTS\_RESULT IoMgrUnregisterInstance(IBase \*pIBase):**

This function is used to deregister the I/O driver instance from the I/O-Manager.

**7.5 Access to the I/O Configuration**

The I/O-Configuration is downloaded to the runtime system as an initialized IEC data structure list.

Typically the I/O driver is called from the I/O-manager if a new I/O-configuration is downloaded. But during the IoDrvUpdateConfiguration() call, the I/O driver must search e.g. for a special connector or

special parameter of a connector. Therefore the I/O-manager provides the following functions to get access to the I/O-configuration:

**IoConfigConnector\* IoMgrConfigGetFirstConnector(IoConfigConnector \*pConnectorList, int \*pnCount, unsigned short wType):**

Get the first connector in the connector list.

**IoConfigConnector\* IoMgrConfigGetNextConnector(IoConfigConnector \*pConnectorList, int \*pnCount, unsigned short wType):**

Get the next connector of the connector list.

**IoConfigConnector\* IoMgrConfigGetFirstChild(IoConfigConnector \*pConnectorList, int \*pnCount, IoConfigConnector \*pFather):**

Get the first child of a specified connector.

**IoConfigConnector\* IoMgrConfigGetNextChild(IoConfigConnector \*pConnectorList, int \*pnCount, IoConfigConnector \*pFather):**

Get the next child of a specified connector.

The following functions provide access to parameters of specified connectors:

**IoConfigParameter\* IoMgrConfigGetParameter(IoConfigConnector \*pConnector, unsigned long dwParameterId):**

Get the parameter specified by the parameter Id.

**unsigned long IoMgrConfigGetParameterValueDword(IoConfigParameter \*pParameter, RTS\_RESULT \*pResult):**

Get a DWORD parameter specified by Id.

**unsigned short IoMgrConfigGetParameterValueWord(IoConfigParameter \*pParameter, RTS\_RESULT \*pResult):**

Get a WORD parameter specified by Id.

**unsigned char IoMgrConfigGetParameterValueByte(IoConfigParameter \*pParameter, RTS\_RESULT \*pResult):**

Get a BYTE parameter specified by Id.

**void \* IoMgrConfigGetParameterValuePointer(IoConfigParameter \*pParameter, RTS\_RESULT \*pResult):**

Get a POintER parameter specified by ID.

## 7.6 I/O Drivers in C/C++

An I/O driver that is written in C or C++ looks similar to an I/O driver written in IEC except the start up- and shutdown phase. This difference is explained in this chapter.

An I/O driver in C/C++ must register its component interface at the component manager at start up of the runtime system (see chapter 2.9) in the ComponentEntry function. This is identical for all component written in C or C++.

After that, the typical start up sequence with the corresponding hooks is called by the component manager. In the CH\_INIT hook, the I/O driver should detect its supported devices and should create one instance for each device with its local CreateInstance() function. After that, the I/O driver should register these instances at the I/O-manager or the Component-Manager. In the example code of the IoDrvTemplate, this looks like:

```
case CH_INIT:
{
```

```

int iInstance = 0;
RTS_RESULT Result;
IBase *pIBase = (IBase *)CAL_IoDrvCreate(0, CLASSId_CIoDrvTemplate, iInstance, &Result);
CAL_CMRegisterInstance(CLASSId_CIoDrvTemplate, iInstance, s_pIBase);

```

## 7.7 I/O Drivers in IEC

The instance of an I/O driver written in IEC is created by CODESYS. This instance is created at the moment, a device is added in the graphical configuration, that has a reference to this IEC driver in the device description. In the device description typically the library and the function block with the driver is specified.

The initialization and detection of the devices is typically done in the FB\_Init method of the driver FB or a specific init function, that can be specified in the device description too (3S drivers uses mostly the Initialize() method).

For the autodetection, a special static FB can be used in the driver FB, so the detection of all card must be done only once for all driver instance, e.g.:

```

VAR_STAT
    s_HilscherCardMgr : HilscherCardMgr;
END_VAR

```

HilscherCardMgr is here a special FB, that detects all cards in its FB\_Init method.

The implementation of the interfaces and the access to the I/O-configuration are equivalent for the driver in IEC as it was described in the chapters before for the C/C++-drivers.

## 7.8 Diagnostic Information

The diagnostic information consists of two different parts:

1. Bit-Field in every connector for the **general diagnostic information**
2. **Extended diagnostic** information of a device with detailed information

The Bit-Field can be used e.g. for the online-diagnostic information in the device tree to see, if one device has an error or has some information for the user.

The extended diagnostic information can be used to display some device specific information about an error or a special detected state.

These parameters are explained now in detail.

### 7.8.1 General diagnostic information bit-field

The actual state of a device can be published in a bit-field, that is a generic part of each connector. This is a bit-field (32-Bit) with the following meaning:

<b>Bitname</b>	<b>Bit-Value</b>	<b>Is set by</b>	<b>Description</b>
CF_ENABLE	0x0001	IO-Config	Connector enabled in the IO-configuration
CF_DRIVER_AVAILABLE	0x0010	CmpIoMgr	A driver has registered to this device, so the device will be supported by a driver.
CF_CONNECTOR_FOUND	0x0020	IoDriver	Connector found (device detected)
CF_CONNECTOR_CONFIGURED	0x0040	IoDriver	Connector configured





CF_CONNECTOR_ACTIVE	0x0080	IoDriver	Connector active (bus is active)
CF_CONNECTOR_BUS_ERROR	0x0100	IoDriver	Bus error
CF_CONNECTOR_ERROR	0x0200	IoDriver	General error
CF_CONNECTOR_DIAGNOSTIC_AVAILABLE	0x0400	IoDriver	Extended diagnostic information available

The Connector flags must always be set out of the IO-driver with the functions IoMgrConfigSetDiagnosis() and IoMgrConfigResetDiagnosis() of the IO-manager to set and reset single diagnostic bits.

If CF\_CONNECTOR\_DIAGNOSTIC\_AVAILABLE is set, two diagnostic parameters out of the device description are used to transmit further extended information.










This bitfields is monitored by CODESYS, if the target is online.

If everything is OK (the bits CF\_ENABLE, CF\_DRIVER\_AVAILABLE, CF\_CONNECTOR\_FOUND, CF\_CONNECTOR\_CONFIGURED, CF\_CONNECTOR\_ACTIVE are set), the following Icon is displayed at the corresponding device (connector): 

If something is wrong (one of the bits CF\_CONNECTOR\_BUS\_ERROR, CF\_CONNECTOR\_ERROR or CF\_CONNECTOR\_DIAGNOSTIC\_AVAILABLE is set or one of the other bits is not set), the following Icon is displayed at the corresponding device (connector): 

After a new initialization of the fieldbus (new download, reset of application) the state of a fieldbus (CF\_CONNECTOR\_ACTIVE flag is set or not) depends on whether the fieldbus is already running or not. This means that it depends on the fieldbus, if the connector has got a green (fieldbus already running) or an red (fieldbus not running) icon after a new initialization.

The table shows the icons of some field busses after a new initialization:

Fieldbus	Icon after new Initialization	Description
3S CANopen stack		Initialization is done in the first cycles of the application.
3S Ethercat stack		Initialization is done in the first cycles of the application.
3S Modbus TCP		Initialization is done in the first cycles of the application.
3S Modbus Serial		Initialization is done in the first cycles of the application.
3S SERCOS stack		Initialization is done in the first cycles of the application.
Hilscher CIF Profibus		Fieldbus is started at the end of IoUpdateConfiguration.
Hilscher CIFX Profibus		Fieldbus is started at the end of IoUpdateConfiguration.
Hilscher CIFX EthernetIP		Fieldbus is started at the end of IoUpdateConfiguration.
Hilscher CIFX Profinet		Fieldbus is started at the end of IoUpdateConfiguration.

## 7.8.2 Extended diagnostic parameter

The parameter with the attribute "diag" can be a structure, string or whatever. This contains the extended diagnostic information of this device, if the corresponding device dialog with the register card "Status" is opened in CODESYS. In this situation, a IoDrvReadParameter() /IoDrvReadParameterById() is called of the corresponding driver of the device.

The parameter can be specified in the device description as followed:

```
<Parameter ParameterId="327936" type="local:TSlaveDiag">
  <Attributes channel="diag" download="true" functional="false"
offlineaccess="read" onlineaccess="read" />
  <Default>
```



	For example, if a slave informs about the activation of the watchdog is a information, that can be implicitly acknowledged.
R06	The relevant extended diagnostic information must be available, until this parameter is acknowledged by the user (or the program!) with the extended diagnostic acknowledge parameter!

## 7.9 IO Consistency

At the IO-update (reading inputs and writing outputs) all IO-channels must be transmitted task-consistent. That means only if a task has finished its IO-update completely, the IO-channels are allowed to transmit to the periphery!

An IO-Update must reach the following 3 different constraints:

1. Consistency
2. Jitter
3. Latency

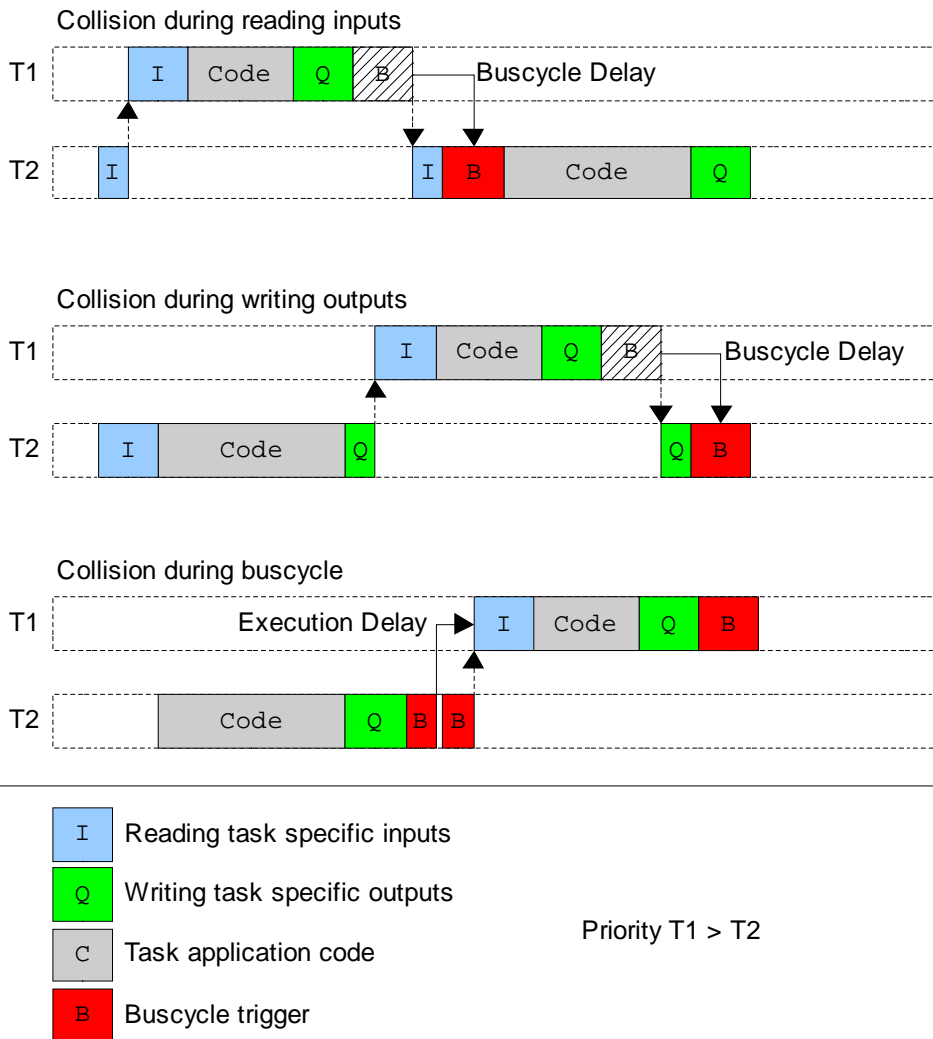
Task consistency must always be maximum! That means, IOs must be updated by a task contiguous. This means not, that a task cannot be interrupted during an IO update by another task.

Jitter means the time shift between the calculated start time of a task and its real start time. This jitter should be minimum to avoid a bad real time influence.

Latency is the time shift between the IOs are updated until they are updated at the hardware. This should be minimum too, to have a maximum real time behaviour.

To reach these 3 goals, the following mechanism are used in the runtime system:

- Bits are written in an atomic way, so Inputs and outputs must not be pre-processed
- Bus cycle must not be executed during IOs update of another task to realize consistency! To avoid jitter, the bus cycle is moved or delayed when it can be done. Only a latency will occur in this situation.
- A collision during a bus cycle can only be resolved by a delay of the interrupt task. This leads to a small jitter, but no latency. This is the only situation that is actually not handled in the runtime!



### 7.9.1 Consistency in the IO Driver

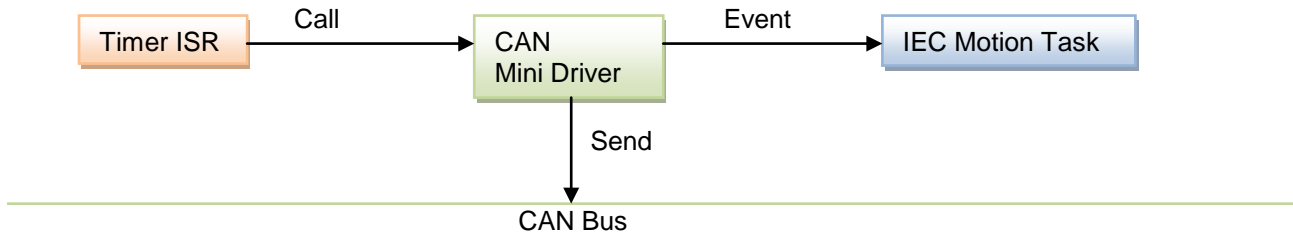
The IO-consistency is realized by the IO-manager only. But to realize this consistency with a maximum performance, the following conditions must be fulfilled with support by the IO-driver:

1. Placeholder of IoStandard.library V3.1.3.2 must be integrated in the target description
2. IoDriver must set the DRVPROP\_CONSISTENCY property flag for the Io-Manager (with the interface function IoMgrSetDriverProperties())
3. SysCpuTestAndSet / SysCpuTestAndReset must be used for each bit input and output access in the IoDriver!

## 7.10 External CAN Sync

By default, CAN Sync packets are sent cyclically from the IEC bus cycle task. This means that, under bad circumstances, the CAN Sync packet will have a jitter which is equal to the jitter of this cyclically called task.

In order to generate a more accurate sync signal, it is possible to generate those packets externally from a hardware timer and to trigger the CAN task from this timer.



### 7.10.1 CAN L2 API

When sync is enabled in a CAN project, the 3S CANOpen stack calls tries to enable the external sync mechanism of the CAN driver. This will be successful if the driver implements the following two functions of the CAN L2 driver API:

- CMD\_SetBlock()
- CMD\_SetCycle()

The function CMD\_SetBlock() is called to pass the sync packet down to the driver. CMD\_SetBlock() is getting a block handle from a block which contains the sync packet. This function takes control over the passed block handle. This control is given back to the calling function at the next call of CMD\_SetBlock().

The function CMD\_SetCycle() is called by the CAN L2 to define the cycle time in which a sync packet will be generated. This time value is given in microseconds and should be used to program a hardware timer.

### 7.10.2 Timer ISR

Within the timer ISR, the CAN driver needs to send the packet to the CAN. After a send interrupt has signaled that the sync packet was send, the driver needs to send an event to the motion task.



#### CAN Timer ISR example:

```

s_pClonedSyncBlock = CAL_CL2_MsgClone( CanNet, pSavedSyncBlock, &error );
if(error == CL2_NO_ERROR) {
    CMD_Send( CanNet, s_pClonedSyncBlock, 0, 0);
}
  
```

**CAN Send ISR:**

```

if(s_pSyncBlockCloned == hBlock) {
    /* Wake up the Motion Task */
    if((s_hEventCanSync != RTS_INVALID_HANDLE)) {
        CAL_SysEventSet(s_hEventCanSync);
    }
}
}

```

**7.10.3 Motion Cycle Time**

When you are using a target with external sync, you are creating a motion task which is triggered by an external event. But this also means that you won't define a cycle time for this task. This time will be implicitly defined with the external sync period.

But, because the SoftMotion stack needs to know the cycle time of the task on which it is running, you need to set this time within your CAN driver manually. This can be done directly in the function `CMD_SetCycle()`, because in this function you get the sync period.

To get the task handle of the motion task, as well as the sync event, you need to register on the event „TaskCreateDone“ of „CmpSchedule“. You should search for your event name and save a handle to this event, as well as to the task handle, in a static variable of the driver.

**Example of `CMD_SetCycle()`:**

```

/* set cycle time of IEC task */
if(s_hTaskCanSyncInfo != NULL)
    s_hTaskCanSyncInfo->tInterval = dwCycle;
/* program hardware timer */
...

```

**7.11 Byte order specific data handling in IO driver**

When developing an IO driver for CODESYS Control, the byte order of the target platform must be regarded. This chapter describes how to create a portable IO driver which could be executed on both Motorola- and Intel-byte order platforms.

**7.11.1 Bits handling in BYTE/WORD/DWORD**

One of the basic tasks during driver development is to control separate bits. For example this could be the handling of digital inputs/outputs or handling of control bits in the registers of your peripheral device.

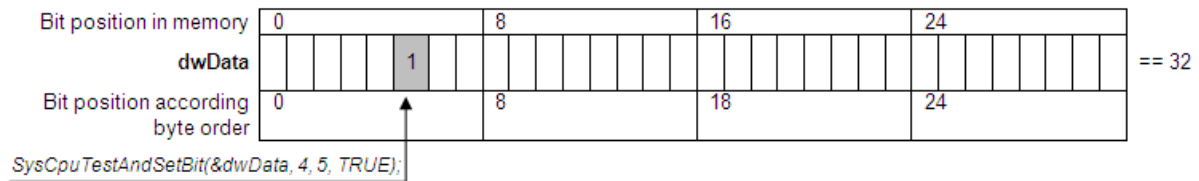
The CODESYS runtime system provides several functions for bit handling to the drivers' developer.

The function `SysCpuTestAndSetBit()` sets (`bSet = 1`) or resets (`bSet = 0`) a specified bit in the bit string with length `nLen`. Pointers to data must correspond to the type which is specified by `nLen`. The function will return `ERR_OK` if after the operation the bit has a changed value. The usage of this function is thread- and interrupt safe. If you want to set- or reset a bit within a `WORD` or `DWORD`, the behaviour of the function depends on the byte order of the platform. Let's consider how bits are handled on Motorola and Intel byte order platforms.

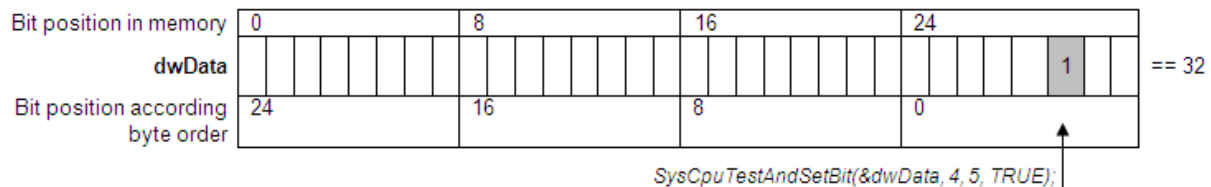
For example the following call is executed

```
SysCpuTestAndSetBit(&dwData, 4, 5, TRUE);
```

Platform with Intel-byte order:



Platform with Motorola-byte order:



As shown in the picture above, the function sets the bits within the DWORD according to the byte order of the platform. That means that the value of the data accessed by pAddress is independent of the byte order of the platform, but the appropriate, different bit number is set in memory. In this example on both kinds of platforms the value of the accessed data is equal to 32. But while bit number 5 is set on Intel byte order, we are setting bit 29 on Motorola byte order.

If the function `SysCpuTestAndSetBit()` is used for exchanging data between several software components of the firmware (for example for tasks synchronization), the position of the bits in memory controlled by the function doesn't matter because all intercommunicating parts have the same byte order and correspondingly access the same memory by the same bit number.

To set/reset a specific bit in the bit string independently of byte order you can use the following fact. The behaviour of the function is independent of the platform if `nLen = 1`, e.g. it the function accesses bit in byte. So the following code will handle the same bit in memory on Motorola and Intel byte order platform:

```

unsigned int BitOffsetInDword; unsigned int BitOffsetInByte;

ByteOffset = BitOffsetInDword / 8;
BitOffsetInByte = BitOffsetInDword % 8;
pbyData = &dwData;
pbyData = pbyData + ByteOffset;
SysCpuTestAndSetBit(pbyData, 1, BitOffsetInByte, TRUE);

```

This code for example could be used for handling a special bit in the control register of any peripheral device.

#### **RTS\_RESULT SysCpuTestAndSet(unsigned long\* pul,int iBit):**

This function is used for testing and setting a bit in a DWORD.

#### **RTS\_RESULT SysCpuTestAndReset(unsigned long\* pul,int iBit):**

This function is used for testing and setting a bit in a DWORD.

All these functions are available in C and IEC.

**ATTENTION:** Please always use a pointer to correct the data type in these functions. Functions `SysCpuTestAndSet/ SysCpuTestAndReset` may never be used for controlling a bit in a Byte, even if the bit number was always expected to be less than 8. Such code will be executed without any

problems on an Intel CPU but it will cause a damage of the memory because the number of the bit will not fit into the byte specified by the pointer. When you are using the newer interface SysCpuTestAndSetBit() you will not encounter these problems, because the pointer now points to a BYTE and not to a DWORD anymore.

### 7.11.2 Helper functions for I/O update

One of the main tasks during I/O driver development in the runtime system is writing I/O update functions (see chap. 7.2.2) which are responsible for coping I/Os values from/to the hardware (bus). Target and bus could have different byte order. For example in case of Modbus the I/Os are transmitted in Motorola byte order but the target might have Intel byte order. To simplify the handling of IOs the runtime system provides several helper functions to the driver developer:

- IoMgrCopyInputLE( )
- IoMgrCopyInputBE( )
- IoMgrCopyOutputLE( )
- IoMgrCopyOutputBE( )

Postfix LE of the helper functions means that the bus has **Little-Endian** (Intel) byte order and accordingly BE means **Big-Endian** (Motorola) byte order. The byte order of the target platform is detected automatically.

### 7.11.3 Representation of bit-fields in IO configuration

The values of I/Os can be monitored and controlled in the I/O configuration of a device. The standard possibility to represent digital IOs is using bit fields (see chap. 6.4.1.1). If the target has Intel byte order the bit channels will be displayed in the following way:

Words	%QW4	WORD	1		
Bit 0	%QX4.0	BOOL	TRUE		Bit 0
Bit 1	%QX4.1	BOOL	FALSE		Bit1
Bit 2	%QX4.2	BOOL	FALSE		Bit 2
Bit 3	%QX4.3	BOOL	FALSE		Bit 3
Bit 4	%QX4.4	BOOL	FALSE		Bit 4
Bit 5	%QX4.5	BOOL	FALSE		Bit 5
Bit 6	%QX4.6	BOOL	FALSE		Bit 6
Bit 7	%QX4.7	BOOL	FALSE		Bit 7
Bit 8	%QX5.0	BOOL	FALSE		Bit 8
Bit 9	%QX5.1	BOOL	FALSE		Bit 9
Bit 10	%QX5.2	BOOL	FALSE		Bit 10
Bit 11	%QX5.3	BOOL	FALSE		Bit 11
Bit 12	%QX5.4	BOOL	FALSE		Bit 12
Bit 13	%QX5.5	BOOL	FALSE		Bit 13
Bit 14	%QX5.6	BOOL	FALSE		Bit 14
Bit 15	%QX5.7	BOOL	FALSE		Bit 15

If the target has Motorola byte order, the bytes inside the word will be swapped. Therefore the Bytes within the Word will be swapped to the target byte order.



Variable	Mapping	Channel	Address	Type	Default Value	Current Value	Prepared Value	Unit	Description
out			%QW2	WORD		1			
Bit0		Bit0	%QX4.0	BOOL	FALSE	FALSE			
Bit1		Bit1	%QX4.1	BOOL	FALSE	FALSE			
Bit2		Bit2	%QX4.2	BOOL	FALSE	FALSE			
Bit3		Bit3	%QX4.3	BOOL	FALSE	FALSE			
Bit4		Bit4	%QX4.4	BOOL	FALSE	FALSE			
Bit5		Bit5	%QX4.5	BOOL	FALSE	FALSE			
Bit6		Bit6	%QX4.6	BOOL	FALSE	FALSE			
Bit7		Bit7	%QX4.7	BOOL	FALSE	FALSE			
Bit8		Bit8	%QX5.0	BOOL	FALSE	TRUE			
Bit9		Bit9	%QX5.1	BOOL	FALSE	FALSE			
Bit10		Bit10	%QX5.2	BOOL	FALSE	FALSE			
Bit11		Bit11	%QX5.3	BOOL	FALSE	FALSE			
Bit12		Bit12	%QX5.4	BOOL	FALSE	FALSE			
Bit13		Bit13	%QX5.5	BOOL	FALSE	FALSE			
Bit14		Bit14	%QX5.6	BOOL	FALSE	FALSE			
Bit15		Bit15	%QX5.7	BOOL	FALSE	FALSE			

CODESYS provides a possibility to change the representation of a bit field and swap bytes correspondingly to the type used in the IO configuration. For this purpose please define a setting in the device description of the PLC as described in chap 0. This helps to make the representation more user friendly and independent from a target byte order.

Variable	Mapping	Channel	Address	Type	Default Value	Current Value	Prepared Value	Unit	Description
out			%QW2	WORD		1			
Bit0		Bit0	%QX5.0	BOOL	FALSE	TRUE			
Bit1		Bit1	%QX5.1	BOOL	FALSE	FALSE			
Bit2		Bit2	%QX5.2	BOOL	FALSE	FALSE			
Bit3		Bit3	%QX5.3	BOOL	FALSE	FALSE			
Bit4		Bit4	%QX5.4	BOOL	FALSE	FALSE			
Bit5		Bit5	%QX5.5	BOOL	FALSE	FALSE			
Bit6		Bit6	%QX5.6	BOOL	FALSE	FALSE			
Bit7		Bit7	%QX5.7	BOOL	FALSE	FALSE			
Bit8		Bit8	%QX4.0	BOOL	FALSE	FALSE			
Bit9		Bit9	%QX4.1	BOOL	FALSE	FALSE			
Bit10		Bit10	%QX4.2	BOOL	FALSE	FALSE			
Bit11		Bit11	%QX4.3	BOOL	FALSE	FALSE			
Bit12		Bit12	%QX4.4	BOOL	FALSE	FALSE			
Bit13		Bit13	%QX4.5	BOOL	FALSE	FALSE			
Bit14		Bit14	%QX4.6	BOOL	FALSE	FALSE			
Bit15		Bit15	%QX4.7	BOOL	FALSE	FALSE			

## 8 Symbolic IEC Variable Access

In many systems where a controller or plc is running, there is the requirement of accessing the variables of the IEC application to display in a HMI, a SCADA system or to provide these variables by an OPC server.

The access to these variables must be always symbolic, that means via the names as they are declared in the IEC application.

Symbols are transferred to the controller in the form of a (automatically generated) child-application, when the father-application (application, the symbols of which are exported) gets downloaded.

In CODESYS V3, the symbolic access was completely redesigned. The following chapters give you some basic information to understand this new architecture.

### 8.1 Architecture

The symbolic IEC variable access based on the resolution of variable names in the runtime system! The resolution is done via a special child IEC application, that contains all symbolic information with the variables of the father application. The following figure shows this issue.

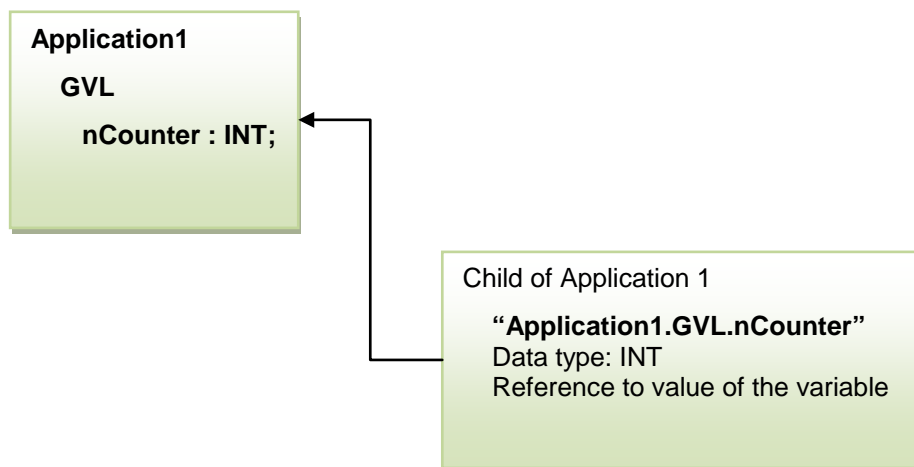


Figure 13: Schema with the symbolic application

For each variable in the father application, there will be generated an FB with the name, data type, access rights and the reference to the value of the variable.

To access this symbolic information, the runtime system provides the component CmplecVarAccess.

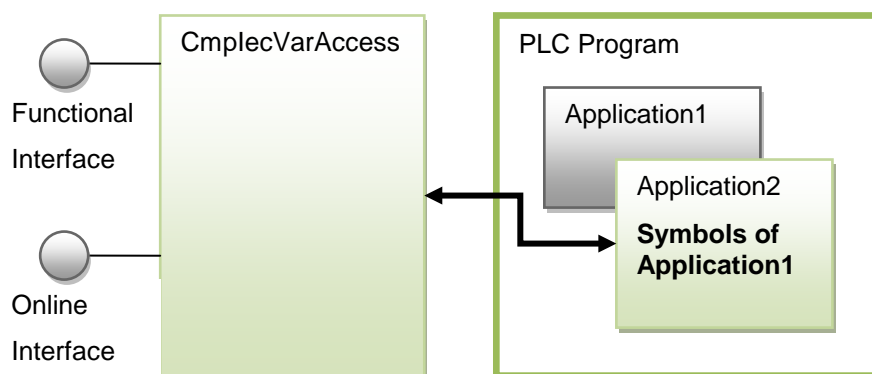


Figure 14: CmplecVarAccess interface

## 8.2 Database of Symbolic Information

To specify, which variables are exported to the symbolic application, actually there is only the possibility to specify an attribute at the declaration of the POUs.

For example the variable nCounter in the GVL can be exported with the following attribute:

```
{attribute 'symbol':='readwrite'}
VAR_GLOBAL
    nCounter : int;
END_VAR
```

In the future it is planned, that this configuration can be done in a graphical editor.

The symbolic information is generated as functions blocks in IEC in an own child application under the father application, which variables should be exported.

A *branch node* FB is here generated for each application or POUs. A *leaf node* FB is generated for each variable. Each FB has here three references:

1. Reference to the father node
2. Reference to the first brother node
3. Reference to the first child node

With these references, a complete hierarchic tree with the symbol information is generated in CODESYS.

A branch node contains the following information:

1. Name of the node
2. References to father, brother and child node

A leaf node contains the following information:

1. Name of the node (variable name)
2. References to father, brother and child node
3. Data type of the variable
4. Access rights (read and/or write access possible)
5. Pointer to the variable value

For a better understanding, we make a small example. An application (Application1) exports a variable from its global variable list (GVL) and the variables of the main program (PLC\_PRG).

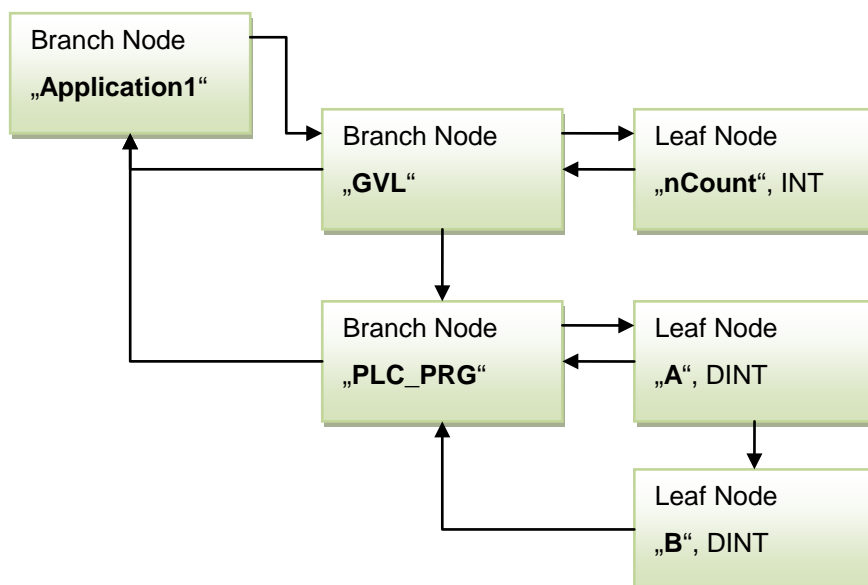


Figure 15: Example of a tree with symbolic nodes

The Application1 has declared a GVL:

```
{attribute 'symbol':='readwrite'}  
VAR_GLOBAL  
    nCounter : int;  
END_VAR
```

Additionally Application1 has declared a POU:

```
{ attribute 'symbol':='readwrite'}  
PROGRAM PLC_PRG  
VAR  
    A : Dint;  
    B : Dint;  
END_VAR
```

As you can see, the description of the symbolic information is strong hierarchic and avoids duplicated names. So the description is compressed maximal.

Arrays are stored compressed respectively collapsed, that means for each array it is generated only one FB for the complete array. The addresses for the index access are calculated internally.

## 8.3 Variable Access Interfaces

The symbolic application with all the symbolic FBs can be accessed in the runtime system after downloading the applications. The access can be done with the CmplecVarAccess component. This component provides a functional interface that can be used by other component in the runtime system.

### 8.3.1 Functional interface

The functional interface provides access routines to browse hierarchically through the complete variable tree with all variables that are exported. Additionally there are routines to arrange a set of variables to one list and to use this list afterwards to read or write all variables in the list with only one function call

### 8.3.2 Online interface

The variable access interface can also be used online. Here a client can send service to browse through the variables, to register variable lists and to read and write variables.

## 8.4 Data Consistency

The variable access component has one feature, to optional read a list of variables consistent to the IEC tasks. This functionality can be used with the VLF\_CONSISTENT option flag at creating a variable list.

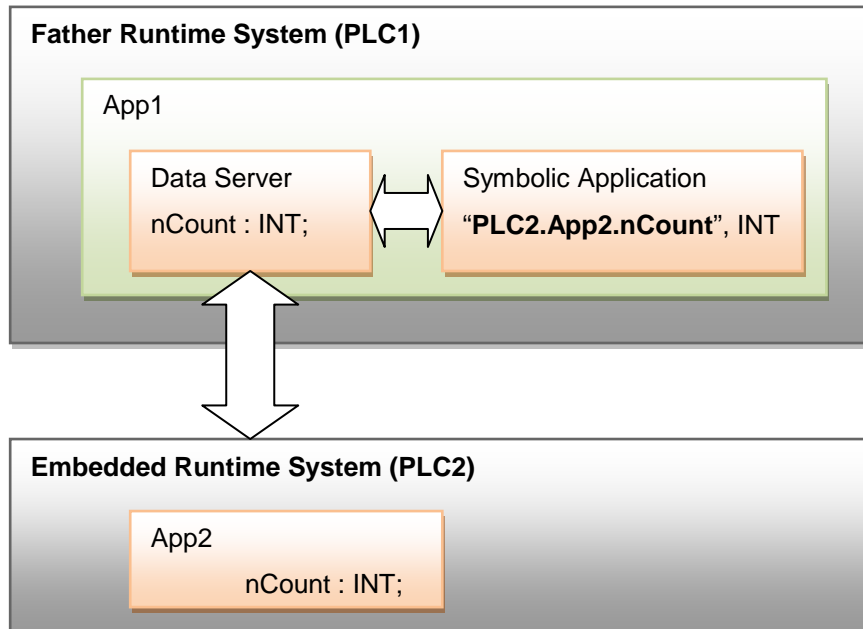
## 8.5 Behaviour at Download/Online Change

If something changed in the father and an online change or download is done in the father, the symbolic client application is deleted and must be downloaded separately. This will be solved in the future, that an online change or download of the father will automatically update the symbolic client application with a download or online change too.

## 8.6 Usage on Small Embedded Systems

On small embedded system, there is no resource to store a symbolic application additionally on the target. The symbolic application can become much larger than the application itself.

For this small targets, there is needed a father runtime system, that hold this symbolic application as a representative. This father runtime updates the variables from the child runtime with the so called *data server*. A data server consists of generated IEC code, that uses the online communication to get the values of the variables. A separate symbolic application in the father allows the symbolic access to these variables. The next figure shows this architecture.



**Figure 16: Architecture of symbolic access on embedded systems**

A client connects typically to the target that it would like to use symbolic access. In the case of an embedded runtime, the client connects to the target a request the symbolic information. Because the symbolic information is not available on the target, the target returns the network address of the father that hold the symbolic information. So every target knows about the position in the network of the symbolic information. If the symbolic information resides on the target itself, then the target returns the first symbolic request from the client with the symbolic information.

## 9 Customer Adaptations and Expansions

There are several different possibilities to extend and adapt the runtime system to your own needs.

You can use the provided and existing components and can arrange them to a set of components with the functionality you need. This issue is called *scalability*. For some components, you have the choice under several different implementations with different functionality and resource requirements.

As the second possibility, you can configure these provided standard components to your needs. Here you can specify the behaviour and the resource needs of the runtime system. This issue is called *configurability*.

The third way is write your own component to replace an existing component or to add this component additionally to the runtime system. This is called *extensibility*.

Each of these issues will be described in detail in the following chapters.

### 9.1 Configuration

The runtime system is configured at compile time via pre-processor defines. The following adjustments can be made via the defines:

- Link type of the runtime system (static, dynamic, mixed or C++)
- Choice of components
- Static configuration of components

The required defines can be set in the workspace itself or in the file "sysdefines.h".

#### 9.1.1 Link type of the runtime system

The runtime system can be linked in several ways. The runtime system can be configured in static or dynamic mode. Static configuration is implemented at compile time and cannot be modified afterwards. In contrast to static configuration, dynamic configuration is implemented at runtime and can be modified at each restart.

Because the M4 mechanism is used for creating the header files and macros are used for calling functions, no code changes are required for switching between different types.

1. **Static:**

If the "STATIC\_LINK" define is set, all components are linked statically and the result is an executable file. It is then no longer possible to load components such as certain IO drivers. All components to be used must be available at compile time. If a component is changed the whole runtime system has to be rebuilt.

This link type is particularly useful for embedded systems that offer no option for reloading modules dynamically.

2. **„Mixed“:**

If the "MIXED\_LINK" define is set, core components are statically linked to an executable file, while further modules can be dynamically reloaded as required. Which files are linked statically is specified in the make file or in the workspace.

In this way it is possible to make a basic system available while dynamically loading certain optional components such as IO drivers.

3. **C++:**

If the "CPLUSPLUS" define is set, the result is a runtime system in C++.

#### 4. Dynamic:

Without one of the defines mentioned above, the runtime system is linked dynamically. The executable file only contains the component manager. All components are dynamically reloaded.

The result is a system with maximum flexibility. Only components that are actually required have to be loaded. If a component changes only an individual module has to be replaced, not the whole runtime system.

### 9.1.2 Choice of components

The choice of components may differ from system to system and can be adapted individually. This is a significant advantage of strict separation of modules in individual components.

Once components have been selected the component manager must be notified. In mixed and statically linked runtime systems a preselection is made in the make file or in the workspace, although an explicit list of components to be loaded still has to be transferred to the components manager. In this context it is important that all statically linked components are included in the load list of the component manager.

The load list of the component manager consists of three different lists. Each component only has to be listed in one of the three lists in order to be loaded.

#### 1. List of system components:

The component manager defines certain components as system components required for its own tasks. These components have to be included in each runtime system.

- CmpMgr
- SysMem
- SysOut
- SysFile
- SysSem
- SysTime
- CmpLog
- CmpSettings
- CmpMemPool
- CmpChecksum

In the interest of uniform logging the component manager itself is also treated as a component.

#### 2. Static list:

For mixed and statically linked runtime systems it is useful to specify a static list of components. This list should include all components that were statically linked. The static list is transferred to the CMLnit function, which is called from the main routine as standard.

An example:

```

StaticComponent s_ComponentList[] = {
    {"CmpApp", CmpApp__Entry},
    {"CmpBlkDrvUdp", CmpBlkDrvUdp__Entry},
    .
    .
    {"SysTime", SysTime__Entry},
    {"", NULL}
};
int main(int __argc, char *__argv[])
{
    .
    .
    CMLnit(s_szComponentFile, s_ComponentList);
    .
    .

```

### 3. Dynamic list:

Lastly, the list of components to be loaded later can be read via the settings component. This is particularly useful for modules to be dynamically reloaded.

Example: configuration file (CFG-file):

```
[ComponentManager]
Component . 1=SysTask
Component . 2=SysSem
Component . 3=SysEvent
Component . 4=SysCom
Component . 5=SysSocket
Component . 6=SysOut
. . .
```

### 9.1.3 Static configuration of components

Each component can have specific pre-processor defines for different component parameters (e.g. buffer sizes). The defines are assigned a default value in the interface header file.

```
#ifndef APPL_NUM_OF_STATIC_APPLS
#define APPL_NUM_OF_STATIC_APPLS      8
#endif
```

Defines can be changed at compile time by assigning a new value in file sysdefines.h.

```
#define APPL_NUM_OF_STATIC_APPLS      4
```

A description of the defines for a particular component can be found in the interface header file of the component.

### 9.1.4 Dynamic configuration (CmpSettings)

Depending on the system the setting component may use a different backend for reading the setting from a medium. The following backends are currently available:

- Configuration via an INI file
- Configuration via the registry
- Static configuration with compiled values (mainly for embedded applications)
- In addition to static configuration there is the option to read in a dynamic configuration via the setting component. This can be modified with each restart.

Users always have the option of implementing their own backend for supporting an additional medium.

Since the INI file is the most frequently used backend, the following examples are based on the INI notation.

The whole configuration is subdivided into blocks. Each block is allocated to a component. The block contains the settings for the component. The following notation is used throughout:

```
[Component]
<Key>=<Value>
```

A list of runtime system settings is shown below, arranged by components. All settings and their description can also be found in the interface header file of a component.

- Component Manager

```
[ComponentManager]
Component . 1=SysTask
Component . 2=SysSem
. . .
List of components to be load.
```



- Router

```
[CmpRouter]
NumRouters=1 //Number of routers
0.MainNet=ether 0 //Main net, where the first
//router is inserted
0.Subnet.0.Interface=COM<1> //First subnet of first
//router; here "Com1"
```

- Block Driver UDP

```
[CmpBlkDrvUdp]
itf.0.ipaddress=192.168.100.27 //fix IP address of network
//interface
itf.0.name=main //Interface name
itf.0.networkmask=255.255.255.0 //Interface subnet mask
itf.0.portindex=1 //Fix port for CODESYS
//addresses 0-3 (1740-1743)
itf.1.ipaddress=192.168.0.1 //Example of second
interface
itf.1.name=vxwin
itf.1.networkmask=255.255.255.0
```

- Block Driver Seriell

```
[CmpBlkDrvSimpleCom]
Baudrate=57600 //Baudrate of serial
interface
ComPort=1 //Port number of interface
HalfDuplexAutoNegotiate=1 //RS485 half duplex mode
EnableRtsToggleHandshake=1 //Currently only used to set
the RTS_CONTROL_TOGGLE
handshake on MS Windows
based systems. This is
needed for some externals
RS232/RS485 adapters to
switch the data direction on
the (half-duplex) line.v
EnableAutoAddressing //Enable the auto addressing
feature
```

- Application

```
[CmpApp]
CreateBootprojectOnDownload=0 //Creation of a bootproject
//at download
StoreBootprojectOnlyOnDownload=0 //Creation of a bootproject
//only at download
PersistentForce=0 //Persistent Forcing
Application.1=Application //Name of first boot
//application
Application.2=BootApp2 //Name of second boot
//application
RetainType.Applications= Cyclic //Retain type:
//-Cyclic
//-OnPowerfail
//-InSRAM
```

- Scheduler

```
[CmpSchedule]
EnableLogger=1 //Activation of a logger
//for the scheduler
MaxProcessorLoad=80 //Maximum load of CPU
Timeslicing.Mode=Internal //Timeslicing Mode:
//NONE: no timeslicing
//intERNAL: internal
//timeslicing
```

```

//EXTERNAL: external
//timeslicing
Timelicing.PlcSlicePercent=80 //time slice of PLC in
//percents
Timelicing.PlcSliceUs=4000 //time slice of PLC in Us
Timelicing.StartOnProcessorLoad=1 //Timeslicing depending on
//maximum processor load
SchedulerPriority=5 //Priority of scheduler
SchedulerInterval=1000 //Interval of schedulers in
//Us

```

- **Logger**

```

[CompLog]
Logger.0.Name=StdLogger //Name of first Logger
Logger.0.Enable=1 //activation of logger
Logger.0.MaxEntries=1000 //Maximum number of log
//entries in internal queue
Logger.0.MaxFileSize=5000 //Maximum file size
Logger.0.MaxFiles=3 //Number of log files
Logger.0.Backend.0.ClassId=0x010B //ClassId of first backend
Logger.0.Backend.1.ClassId=0x0104 //ClassId of second backend
//Ids read from CompItf.h
Logger.1.Name=CommLog //Name of second logger
Logger.1.Enable=1 //etc.

```

- **Retain**

```

[CompRetain]
Retain.SRAM.Address=0x10000000 //Physical start address
SRAM
Retain.SRAM.Size //Größe SRAM

```

In addition to general settings, the components of the system adaptation interface may also have system-dependent settings. These settings are identified by separate abbreviations in the key

```

[component]
<shortcut of the customization>.<Key>=<Value>

```

- **Files**

```

[SysFile]
FilePath.1=./Boot, *.app, *.ap_ //files with extension app
//or ap_ are stored in
//folder ./Boot
FilePath.2=./var, *.ret, *.frc //files (*.ret, *.frc) are
//stored in ./var

```

- **Timer**

```

[SysTimer]
VxWorks.TimerSource=Auxiliary //Timersource of scheduler
//Auxiliary: Auxiliary-Clock
//System: System-Clock
//default: high prior task

```

- **Interrupt**

```

[SysInt]
WinCE.UseIRQSysIntrMapping=1 //Mapping of an IRQ on a
//system IRQ

```

- **Shared Memory**

```

[SysShm]

```

```

WinCE.MapPhysical=1           //setting flag MapPhysical
WinCE.DividePhysicalAddressBy=256 //calculation of physical
                                //address

```

### 9.1.5 Typical configurations of the CODESYS Control WinV3 runtime system

There are the following typical configurations of the CODESYS Control Win V3 runtime system:

- Single tasking systems (embedded)
- Timer based systems.
- Multi tasking systems (full)

#### 9.1.5.1 Embedded Runtime System

An “embedded runtime” we call a system that typically has only one main loop for the execution of all tasks in the runtime system (communication, IEC tasks, etc.). For this kind of runtime system, a minimal system is sufficient.

For this type of runtime system, the configuration can be the following:

```

[ComponentManager]
Component.1=CmpIecTask
Component.2=CmpMgr
Component.3=CmpEventManager
Component.4=SysMem
Component.5=CmpSettings
Component.6=CmpChannelServer
Component.7=SysExcept
Component.8=CmpMemPool
Component.9=CmpChecksum
Component.10=CmpAddrSrv
Component.11=SysFile
Component.12=SysTime
Component.13=CmpMonitor
Component.14=CmpBinTagUtil
Component.15=SysShm
Component.16=CmpLog
Component.17=SysSocket
Component.18=CmpChannelMgr
Component.19=CmpRouter
Component.20=CmpSrv
Component.21=SysCpuHandling
Component.22=CmpRetain
Component.23=CmpCommunicationLib
Component.24=CmpScheduleEmbedded
Component.25=CmpBlkDrvUdp
Component.26=CmpApp

```

#### 9.1.5.2 Timer runtime system

The timer runtime uses processor timers to operate separate cyclic tasks. This is typically used on small embedded controllers with several timers. The behaviour is much better than the embedded runtime, because for example, the communication has no influence on some higher priority tasks (IEC tasks), because the communication is done in the main (background) loop and the higher priority tasks are executed in the timer interrupts.

For this type of runtime system, the configuration can be the following:

```

[ComponentManager]
Component.1=CmpIecTask
Component.2=SysEvent
Component.3=CmpMgr
Component.4=CmpEventManager
Component.5=SysMem
Component.6=CmpSettings
Component.7=CmpChannelServer

```

```

Component.8=CmpScheduleTimer
Component.9=SysExcept
Component.10=SysTimer
Component.11=CmpMemPool
Component.12=CmpChecksum
Component.13=CmpAddrSrvc
Component.14=SysFile
Component.15=SysTime
Component.16=CmpMonitor
Component.17=CmpBinTagUtil
Component.18=SysShm
Component.19=CmpLog
Component.20=SysSocket
Component.21=CmpChannelMgr
Component.22=CmpRouter
Component.23=CmpSrv
Component.24=SysCpuHandling
Component.25=CmpRetain
Component.26=CmpCommunicationLib
Component.27=CmpBlkDrvUdp
Component.28=CmpApp

```

### 9.1.5.3 Full runtime system

As a full runtime we specified a system that is based on a preemptive multitasking operating system. Here the various operations can be executed by tasks with different priorities. So a very fine adjustable system behaviour can be reached. A full runtime contains typically all of the available features of a CODESYS runtime.

For this type of runtime system, the configuration can be the following:

```

[ComponentManager]
Component.1=CmpIecTask
Component.2=SysEvent
Component.3=CmpMgr
Component.4=CmpEventMgr
Component.5=SysMem
Component.6=CmpSettings
Component.7=CmpChannelServer
Component.8=SysExcept
Component.9=CmpMemPool
Component.10=CmpChecksum
Component.11=CmpAddrSrvc
Component.12=SysFile
Component.13=CmpSchedule
Component.14=SysTime
Component.15=CmpMonitor
Component.16=CmpBinTagUtil
Component.17=SysShm
Component.18=CmpLog
Component.19=SysSocket
Component.20=CmpChannelMgr
Component.21=SysTask
Component.22=CmpRouter
Component.23=CmpSrv
Component.24=SysCpuHandling
Component.25=CmpRetain
Component.26=CmpCommunicationLib
Component.27=CmpBlkDrvUdp
Component.28=CmpApp

```

### 9.1.5.4 Gateway runtime system

For this type of implicit runtime system, the configuration can be the following:

```

[ComponentManager]
Component.1=SysTime
Component.2=SysTask
Component.3=SysEvent

```

```

Component.4=SysSem
Component.5=SysCom
Component.6=SysSocket
Component.7=SysExcept
Component.8=SysShm
Component.9=SysSemProcess
Component.10=SysCpuHandling
Component.11=SysInt
Component.12=CmpRouter
Component.13=CmpChannelMgr
Component.14=CmpChannelClient
Component.15=CmpBlkDrvUdp
Component.16=CmpAddrSrvc
Component.17=CmpGateway
Component.18=CmpGwCommDrvTcp
Component.19=CmpNameServiceClient
Component.20=CmpCommunicationLib
Component.21=CmpBlkDrvShm

```

```

[CmpRouter]
EnableParallelRouting=1
0.MainNet=ether x
0.NumSubNets=1
0.SubNet.0.Interface=BlkDrvShm

```

```

[CmpGwCommDrvTcp]
ListenPort=1217

```

### 9.1.5.5 Visualization runtime systems (target visualization CODESYS HMI)

For the Target-Visualization and the special target visualization CODESYS HMI the position and size parameters of the window as well as the update rate of the Target-Visualization can be configured in the runtime system ini-file in section CmpTargetVisu.

Example:

```

[CmpTargetVisu]

Application.Updaterate_ms=200
Application.WindowPositionX=50
Application.WindowPositionY=50
Application.WindowSizeWidth=200
Application.WindowSizeHeight=200
Application.WindowType=0

App2.WindowPositionX=300
App2.WindowPositionY=300
App2.WindowSizeWidth=200
App2.WindowSizeHeight=200
App2.WindowType=1

```

### 9.1.6 Create your own configuration with the RtsConfigurator

For your system, you may want to define a specific set of components, including standard components and own components such as IO drivers (see below).

As there are a lot of dependencies between the components, it may not be easy to find a valid combination of components. To solve this problem, you can use the tool "RtsConfigurator". This windows tool checks the dependencies between the components according the m4 files. It helps you to select the components you need.

If you have a valid configuration, it can create the following output files for you:

- A list of used components
- A list of used C-Files
- A template for a makefile

- A \*.cfg file for the component manager
- A \*.h file with definitions needed for you main\*.c file (COMPO\_INIT\_DECL and COMPO\_INIT\_LIST)
- A reference documentation for the selected components

Please check the RTSc configurator.chm file for further details.

## 9.2 Implementing own components

The architecture of runtime system V3 is based on components, i.e. each related functional area is represented with an independent component. The runtime system can be expanded with further components at any time.

The CmpTemplate component can be used as a template for a new component. It contains all required files and the whole generic part.

### 9.2.1 Global include files

Each runtime system component requires global information that is contained in general header files:

Cmpltf.h	Definitions of the component manager functions required for exchanging function pointers:
CmpStd.h	Definitions and global “#include” instructions. Operating system-specific header files are also defined here. All components link this file as the first header.
CmpErrors.h	Standardised definition of error return values for the API functions

These files are located in the root directory (“\”) of the runtime system.

### 9.2.2 Include files of the components

Each component must define at least two header files. The interface file contains the interface for the component the dependency file contains the dependencies of the component.

These two header files are generically generated from the associated description files via the M4 mechanism. Any changes must always be implemented in the description files (m4 files). Manual modification of the header files is not provided for and not advisable.

#### 9.2.2.1 Interface file

The interface file CmpXXXItf.h is generated from file CmpXXXItf.m4. It contains the interface for the component. The interface files of the core components are stored in directory /Components. Customer-specific interface files should be stored in the component directory under /Customer Components/<Customer>/<Component>.

The file contains

- Defines and structures defined by the component.
- Definition of the interface functions and the required macros (USE\_, EXT\_, ...)

The interface file is not only included by the respective component, but also by all other components requiring access to the functions of this interface.

#### 9.2.2.2 Dependency file

The dependency file CmpXXXDep.h is generated from file CmpXXXDep.m4. This file contains all dependencies for the component. The dependency file of the core components is stored in the component directory /Components/CmpXXX. Customer-specific dependency files are also stored in the component directory.

The file contains:

- Required include statements

- **EXPORT\_STMT**: Summary of all EXP\_ defines, i.e. all functions exported by the component.
- **IMPORT\_STMT**: Summary of all required GET\_ defines, i.e. importing of functions required by the component.
- **USE\_STMT**: Summary of all required USE\_ defines, i.e. definition of the function pointers of the functions required by the component.

The dependency file is only included by the respective component, not by other components.

### 9.2.3 Generation of include files

Functions cannot be called directly, because the runtime system is linked in different ways with the same sources and can be compiled for C or C++. This is why different macros are used.

Since it would be impossible to create header files with the required macros by hand, we use the GNU M4 macro pre-processor, which generates comprehensive header files from simple description files.

For each component the developer creates an interface file and a dependency description file. Both files are available as input files for the GNU M4 macro pre-processor. In a prebuild step from each file an include file is generated, which is integrated in the respective C files. As an alternative to a prebuild step simple batch files can be used for the compile process. These batch files are included in the toolkit sources and can be used as templates for own files.

Since the files are analyzed by a macro pre-processor, all content that cannot be interpreted is taken over unchanged – e.g. comments, definitions of constants, structs, etc.

The parameters for the macro calls should always be enclosed in quotation marks, i.e. “” and ‘’.

Comments for functions or for the component follow a certain scheme so that they can be interpreted by the configuration tool:

A function comment directly precedes the function definition. As usual in Java, it starts with the “/\*\*” character and ends with “\*/”. The configuration tool ignores new comment lines starting with a single “\*”. The comment itself is created in XML, using the following tags:

- **<description>**: Contains a description of the component/function.
- **<author>**: optional
- **<copyright>**: a copyright note
- **<version>**: Version number

The following tags are defined specially for functions:

- **<param name="paramName" type={"IN" | "OUT" | "INOUT"}>**: a parameter description
- **<returns>**: Description of the return value, for most components a list of possible error values.

### 9.2.4 Source code file

#### 9.2.4.1 General interface

Each component contains the following functions: ComponentEntry(), ExportFunctions(), ImportFunctions(), and HookFunction(). The following examples are again taken from the CmpTemplate component.

- **ComponentEntry()** is the central entry function for each component. It is called by the component manager when the system starts up (see also Chapter 2.1.2 “Operating principle or the component manager”). The function is furnished with a structure for exchanging function pointers between the component manager and the component. The component receives access functions from the component manager and returns pointers for the 3 functions ExportFunctions(), ImportFunctions(), and HookFunction().  
The function should be implemented as follows:

```

DLL_DECL int CDECL ComponentEntry(INIT_STRUCT *pInitStruct)
/* Used to exchange function pointers between component manager and components.
Called at startup for each component.
pInitStruct:           IN    Pointer to structure with:
pfExportFunctions      OUT   Pointer to function that exports
                             component functions
pfImportFunctions      OUT   Pointer to function that imports
                             functions from other components
pfGetVersion           OUT   Pointer to function to get component
                             version
pfRegisterAPI          IN    Pointer to component manager function to
                             register a api function
pfGetAPI               IN    Pointer to component manager function to
                             get a api function
pfCallHook             IN    Pointer to component manager function to
                             call a hook function
Return                 ERR_OK if library could be initialized, else
                             error code
*/
{
    pInitStruct->CmpId = COMPONENT_ID;
    pInitStruct->pfExportFunctions = ExportFunctions;
    pInitStruct->pfImportFunctions = ImportFunctions;
    pInitStruct->pfGetVersion = CmpGetVersion;
    pInitStruct->pfHookFunction = HookFunction;
    pInitStruct->pfCreateInstance = CreateInstance;
    pInitStruct->pfDeleteInstance = DeleteInstance;

    s_pfRegisterAPI = pInitStruct->pfCMRegisterAPI;
    s_pfGetAPI = pInitStruct->pfCMGetAPI;
    s_pfCallHook = pInitStruct->pfCMCallHook;
    s_pfCreateInstance = pInitStruct->pfCMCreateInstance;
    return ERR_OK;
}

```

- ExportFunctions() is also called when the system starts up. It uses the EXP\_xxx macros for notifying the component manager of exported functions. Using the EXPORT\_STMT macro from the interface file the function can be implemented as follows:

```

static int CDECL ExportFunctions(void)
/* Export function pointers as api */
{
    /* Macro to export functions */
    EXPORT_STMT;
    return ERR_OK;
}

```

- ImportFunctions() is also called when the system starts up. It uses the GET\_xxx macros for obtaining required function pointers from the component manager. Using the IMPORT\_STMT macro from the interface file the function can be implemented as follows:

```

static int CDECL ImportFunctions(void)
/* Get function pointers of other components */
{
    /* Macro to import functions */
    IMPORT_STMT;
    return ERR_OK;
}

```



}

- The component manager calls the HookFunction() when a hook occurs. The function enables suitable responses to hooks, e.g. calling of an imported function with the CAL\_xxx macro:

```
/* Example for a Hook function */
static int CDECL HookFunction(unsigned long ulHook, unsigned long ulParam1,
unsigned long ulParam2)
{
    switch (ulHook)
    {
        case CH_INIT_SYSTEM:
            break;
        case CH_INIT:
            break;
        case CH_INIT_DONE:
            break;
        /* Cyclic */
        case CH_COMM_CYCLE:
            break;
        case CH_EXIT_COMM:
            break;

        case CH_EXIT_TASKS:
            break;
        case CH_PRE_EXIT:
            break;
        case CH_EXIT:
            break;
        case CH_EXIT_SYSTEM:
            break;
        default:
            break;
    }
    return 0;
}
```

#### 9.2.4.2 Component specific implementation

In addition to the general interface, which is largely identical for all components, each component also has a component-specific implementation. This is where functions must be implemented that are exported based on the interface description. In addition, purely internal functions can be implemented. While the general interface deals with pure component management, the actual functionality of the component is implemented here.

## 9.3 Implementation Notes

### 9.3.1 Error returning

Each function either returns an error code or a handle.

An error code is of type `RTS_RESULT` (int). The error codes are defined in `CmpErrors.h`. If no error occurred `ERR_OK = 0` is returned.

In the event of an error the handle of type `RTS_HANDLE` is `RTS_INVALID_HANDLE` (`==1`). If this is the case, an error code is returned via a pointer.

Since the pointer test for error code `NULL` is required at many points, the error code can be allocated with the following macro. This macro contains an automatic test for `NOT NULL`.

```
RTS_RESULT* pResult;
RTS_SETRESULT(pResult, ERR_OK);
```

### 9.3.2 Memory

The runtime system should not require dynamic memory management. Even so, an “unlimited” number of tasks should be possible.

The following procedure is used to achieve this:

A component intended for managing a variable number of structures (e.g. tasks, applications, communication connections) uses a static memory for a fixed number of units. The component creates a `MemoryPool` using the `CmpMemPool` component. The `MemoryPool` contains a defined number of blocks with a certain size. Once all blocks are assigned, each memory pool can be extended dynamically.

The whole memory management is handled via the interface of the `CmpMemPool` component.

### 9.3.3 Allocation of IDs

#### 9.3.3.1 Vendor ID

The Vendor ID is allocated by 3S for each manufacturer. It uniquely identifies the manufacturer. In the customer-specific component the following define must be set.

```
#define CMP_VENDORID          <VendorID>          /*16 bit*/
```

**Please note:** all necessary defines can be found in `Cmpltf.h`

#### 9.3.3.2 Component ID

Each component requires a unique, 32 bits long component ID. This ID is generated from the Vendor ID and a component-specific ID. The high word of the component ID corresponds to the Vendor ID, the low word to the component-specific ID.

If a manufacturer re-implements a core component, only the high word changes (i.e. the Vendor ID), but not the low word.

If a completely new component is implemented, the manufacturer must allocate a new component-specific ID. The range between `0x2000` and `0xFFFF` is available for this purpose.

```
#define CMP_Template          <ID>                /*16 bit*/
```

The range between `0x2000` and `0x3FFF` is available for this purpose.

```
/* OEM specific components */
#define CMPID_CmpStartOEM          0x00002000
#define CMPID_CmpEndOEM            0x00003FFF
```

The component ID uniquely identifies the manufacturer who implemented the component and indicates whether or not it is a core component.

### 9.3.3.3 Interface ID

The interface ID uniquely identifies an interface in the runtime system. Each core component in the runtime system implements at least one interface. Manufacturers can replace any core component with their own, as long as it implements the same interface.

Please note, different components may implement the same interface. Examples of these components are different IO and block drivers.

If a completely new component is implemented, the manufacturer must allocate a new interface ID. The range between 0x2000 and 0x3FFF is available for this purpose.

```
/* OEM specific start id */
#define ITFID_ICmpStartOEM          0x00002000
#define ITFID_ICmpEndOEM           0x00003FFF
```

### 9.3.3.4 Class ID

The class ID uniquely identifies the different classes in the C++ runtime system.

If a completely new component is implemented, the manufacturer must allocate a new class ID. The range between 0x2000 and 0x3FFF is available for this purpose.

```
/* OEM specific start id */
#define CLASSID_CCmpStartOEM       0x00002000
#define CLASSID_CCmpEndOEM        0x00003FFF
```

## 9.3.4 Importing of functions

Functions for other components must first be imported before they can be used. All functions are imported via the GET\_xxx macro in the ImportFunctions() function. The component first has to be notified of the functions via the USE\_xxx macro. The IMPORT\_STMT macro from the dependency file consolidates all GET\_xxx macros of the component. The USE\_STMT macro from the dependency file consolidates all USE\_xxx macros of the component.

The complete import generically handled via the general component interface. When new components are developed, only functions to be imported have to be included in the dependency description file (m4). The m4 mechanism will then correctly generate the import statement and function calls.

## 9.3.5 Calling of imported functions

Within a component imported functions are called with the CAL-xxx macro. Since functions of optional components do not necessarily have to exist in the runtime system, before an imported function is called its existence should be verified. The CHK\_xxx macro is used for this purpose. A call looks as follows:

```
. . .
If (CHK_xxxFunc)
{
    Result = CAL_xxxFunc();
}
. . .
```

## 9.3.6 Exporting of functions

The component manager is notified of exported functions via the EXP\_xxx macro in function ExportFunctions(). The EXPORT\_STMT macro from the dependency file consolidates all EXP\_xxx macros of the component.

Just like the import, the export of functions is handled by the general component interface. The statements are generated by the m4 mechanism. The functions to be exported must be specified in the interface description file.

### 9.3.7 Linkage with the runtime system

The newly created component can be used in the runtime system provided it implements the general interface, has its own IDs, and all required header files have been generated. Two steps are required for this:

#### 1. The component must be linked.

The way in which the runtime system is linked is specified during configuration of the runtime system (Chapter 3.1). Three options are available: static, dynamic, or mixed.

- If the runtime system was linked **statically**, the new component must be included in the workspace or the makefile, and the whole runtime system must be recompiled.

In addition, an entry for this component must be added to the MainLoadComponent function when linking statically

- If the runtime system was linked **dynamically**, the new component must be available as a reloadable module.
- If the runtime system was linked in **mixed** mode, the component can either be linked statically to the runtime system, or it can be available as a reloadable module.





























#### 2. The component manager must be notified of the component.

As soon as the component was linked or is available as a module, the component manager must be notified of the new component. This can be done in different ways (see Chapter 3.1.2):

- If the component is statically linked with the runtime system, it is useful to include the new component in the static list of components.
- If the new component is not to be included in the static list, it must be listed in the dynamic list.

### 9.3.8 Order of the INIT Hooks

Chapter 3.1.1 already explained the default actions which are executed in the INIT Hooks of the core runtime components. Beside this, there are some general rules for every hook, which need to be taken into account when writing your own components.

	System Komponenten aufrufen	Standard Komponenten aufrufen	Ressourcen anlegen (Events, Semaphoren, ...)	Ressourcen öffnen (Service Handler, Events, ...)
CH_INIT_SYSTEM				
CH_INIT_SYSTEM2				
CH_INIT				
CH_INIT2				
CH_INIT3				
CH_INIT_TASKS				
CH_INIT_COMM				

## 9.4 Libraries

In CODESYS V3 we no longer distinguish between internal and external libraries. Each library can contain functions and blocks that are either implemented internally (in IEC) or externally (in ANSI-C or C++).

External implementation is based on a component of the runtime system, internal implementation is based directly on the CODESYS library.

### 9.4.1 Creating a library in CODESYS

For an implementer a CODESYS V3 library is not much different to a normal project. Declare your function, your method or your function block via the POUs tab in the way you are used to from IEC. If you now save the library and add it to your library repository you can use it in any project.

If you add the library in a library manager you will find that only the POUs are visible under the POU tab; all entries under the Devices tab are hidden. This feature is very useful for implementing test or sample code within the library. This code is not visible if the library is used in another project. The test or sample code can be executed by opening the library as a project.

Each POU within a library can be implemented in IEC (i.e. internally) or in the runtime system (i.e. externally). By default it is assumed that the POU is implemented in IEC. To implement the POU in the runtime system activate the option 'External Implementation' under 'Properties' in the Build tab of the respective POU.

During compilation the system does not use the IEC code of the function, but instead the name of the function is entered in a list that is downloaded during the download. This function is linked if a function with the right name is found in the RTS.

External POUs have to be treated exactly like internal POUs. All inputs and outputs must be defined. All local variables must be defined in function blocks. Local variables must not be defined in methods or functions. VAR\_STAT variables cannot be used in the RTS!

### 9.4.2 Implementation of external POUs in the runtime system

#### 9.4.2.1 Declaration in the runtime system

In principle external implementation in the RTS imitates the parameter transfer used for IEC functions in CODESYS. Each external function has exactly one parameter. This parameter is a pointer to a structure in which the parameters defined in the IEC programming system are listed. The parameterization details differ for functions, methods and function blocks.

In the RTS, no more external reference table is necessary! To export the external library functions for the plc program, only the flag for external libraries must be declared in the interface m4-file (see chapter 2.8.1).

#### 9.4.2.2 Declaration of functions

The following declarations have to be implemented in the .m4 file of your component:

- Declare a structure `<function name>_struct` with elements in the following order:
  - All INPUT variables and all IN\_OUT variables in the order defined in IEC.
  - All OUTPUT variables in the defined order.
  - The implicit OUTPUT with the name of the function comes last.
- Define a function as follows:

```
DEF_API(`void', `CD'CL', `

```

The "1" after the transfer parameter list indicates that this function can be called from IEC. In the runtime system a function pointer to this function is registered in the component manager.

#### Example:

##### 1. IEC:

```
FUNCTION MyExternalFunction : Dint
VAR_INPUT
    p1 : Dint;
```

```

END_VAR
VAR
END_VAR

```

## 2. M4:

```

typedef struct
{
    RTS_I32 p1;
    RTS_I32 Result;
} myexternalfunction_struct;

DEF_API(`v'id', `CD'CL', `myexternalfunction',
`myexternalfunction_struct 'p)',1,0)

```

## 3. Associated header:

```

typedef struct
{
    RTS_I32 p1;
    RTS_I32 Result;
} myexternalfunction_struct;

void CDECL CDECL_EXT myexternalfunction(myexternalfunction_struct *p);

```

**9.4.2.3 Declaration of function blocks**

Internally a function block is called like a method. For each function block a “\_\_Main” method is created implicitly that is called when the function block is called.

In addition to the Main method there are three predefined initialization methods:

1. FB\_Init: This method is called during initialization of the FB.

```

METHOD FB_Init : bool
VAR_INPUT
    bInitRetains: bool;
    bInCopyCode: bool;
END_VAR

```

2. FB\_Exit: This method is called when the application is terminated.

```

METHOD FB_Exit : bool
VAR_INPUT
    bInCopyCode: bool;
END_VAR

```

3. FB\_Reinit: This method is called during an Online Change involving a change in data layout. It may be used to reassign saved data pointers if the data have moved due to the online change.

```

METHOD FB_Reinit : bool
VAR_INPUT
END_VAR

```

Initialization methods are called in the following situations:

```

Download FB_Exit (bInCopyCode := FALSE);
FB_Init (bInitRetains := TRUE; bInCopyCode := FALSE);

```

**1. Online Change:**

Wenn der FB vom Datenlayout geändert wurde:

```

FB_Exit (bInCopyCode := TRUE);
FB_Init (bInitRetains := FALSE; bInCopyCode := TRUE);
For all other FBs:
FB_Reinit ();

```

```

Reset FB_Exit (bInCopyCode := FALSE);
FB_Init (bInitRetains := FALSE; bInCopyCode := FALSE);

```

```

Reset Col FB_Exit (bInCopyCode := FALSE);
FB_Init (bInitRetains := TRUE; bInCopyCode := FALSE);

```

**2. Start PLC and load application:**

```

FB_Init (bInitRetains := FALSE; bInCopyCode := FALSE);

```

**3. Stop PLC and delete application**

```

FB_Exit (bInCopyCode := FALSE);

```

If a function block is marked to be linked externally, this means:

1. The "Main" method is linked externally
2. The "FB\_Init" method is not linked externally (initialization method), unless the method was explicitly entered in the IEC program and is marked to be linked externally.
3. The "FB\_Exit" method is not linked externally (initialization method), unless the method was explicitly entered in the IEC program and is marked to be linked externally.
4. All other methods are not linked externally. Methods to be linked externally must be marked as such.

Since a function block call corresponds to a method call, please refer to the following section regarding the declaration of an implicit method. This section focuses on the structure of a function block.

The instance of a function block contains all function block variables in the declared order plus a pointer to the virtual function table.

This pointer always comes first in the structure. This pointer is only used for late binding. In the following we assume that function calls are statically linked.

Example:

1. IEC:

```

FUNCTION_BLOCK fubextern
VAR_INPUT
    a,b,c: int;
END_VAR
VAR_OUTPUT
    x,y,z : bool;
END_VAR
VAR
    m,n,o : Dint;
END_VAR

```

2. M4:

```

typedef struct
{
    /*pointer to virtual function table */
    void* __VFTABLEPointer;
    /*inputs*/
    short a;
    short b;
    short c;
    /*outputs*/
    char x;
    char y;
    char z;
    /*locals*/
    int m;
    int n;
    int o;
}fubextern;

```

Please note that the structure of the instance is solely determined by the order of the declaration, rather than through affiliation with an input, output, or var block. If the outputs and inputs were interchanged in the IEC declaration, they would also have to be interchanged in C.

#### 9.4.2.3.1 Declaration of methods

Basically, methods are functions with an additional implicit parameter, i.e. a pointer to the function block instance. In principle, the same method names can be used for different function blocks. The following convention is used for the name of method to be linked in the RTS:

`<fbname>__<method name>` (two underscores)

The following declarations have to be made in the RTS:

- Declare a structure `<fbname>` as described in 5.3.2.
- Declare a structure `<fbname>__<methodname>_struct` with elements in the following order:
  1. All INPUT variables and all IN\_OUT variables in the defined order.
  2. A pointer to instance:
  3. `fbname *__INSTANCEPOintER;`
  4. All OUTPUT variables in the defined order.
  5. The implicit OUTPUT with the name of the method comes last.
- Define a function as follows:
 

```
DEF_API(`void', `CD'CL', `<<fbname>__<methodname>',
`(<fbname>__<methodname>_struct*'p)', 1)
```

#### Example:

If the function block defined in 5.3.2 is to be implemented in the RTS, the following code must be defined in the RTS (see also CmpTemplate):

##### 1. IEC:

```
FUNCTION_BLOCK fubextern
VAR_INPUT
  a,b,c: int;
END_VAR
VAR_OUTPUT
  x,y,z : bool;
END_VAR
VAR
  m,n,o : Dint;
END_VAR
```

##### 2. M4:

```
typedef struct
{
    /*pointer to virtual function table */
    void* __VFTABLEPOintER;
    /*inputs*/
    short a;
    short b;
    short c;
    /*outputs*/
    char x;
    char y;
    char z;
    /*locals*/
    int m;
    int n;
    int o;
```



```

}fubextern;

/*Structure definition for main method*/
typedef struct
{
    /*pointer on function block instance*/
    fubextern *__instancepointer;
}fubextern_main_struct;

/*Function definition for main method*/
DEF_API(`void',`CD'CL',`fubextern__main',
`fubextern_main_struct*'p)',1)

```

### 3. Header:

The typedefs are taken over unchanged

```

void CDECL CDECL_EXT myexternalfunctionblock__main
(myexternalfunctionblock_main_struct *p);

```

### 4. Source code:

```

void CDECL myexternalfunctionblock__main(myexternalfunctionblock_main_struct *p)
{
    /*Do anything*/
    return;
}

```

## 9.4.3 Implementation

In the runtime system an IEC function, a FB or a method always becomes a C-function or a C++ method call. Once the function was correctly declared in the m4 file and the interface file was generated, the function can be implemented in the source code file as usual. Further example from the CmpTemplate component.

### 1. IEC:

```

FUNCTION MyExternalFunction : Dint
VAR_INPUT
    p1 : Dint;
END_VAR
VAR
END_VAR

```

### 2. M4:

```

typedef struct
{
    RTS_I32 p1;
    RTS_I32 Result;
} myexternalfunction_struct;

DEF_API(`void',`CD'CL',`myexternalfunction',
`myexternalfunction_struct 'p)',1,0)

```

### 3. Header:

```

typedef struct
{
    RTS_I32 p1;
    RTS_I32 Result;
} myexternalfunction_struct;

void CDECL CDECL_EXT myexternalfunction(myexternalfunction_struct *p);

```

### 4. Source code:

```
void CDECL myexternalfunction(myexternalfunction_struct *p)
{
    p->Result = p->p1 + 1;
}
```

## 9.5 Adaptations to Specific Operating Systems or Processors

The adaptation to operating systems that are not supported by 3S, the adaptation can be done by your own. For this, we provide source code for all System-Components as a template (under the directory `$(Platform)\SysTemplates`).

Because each system-component can be implemented and tested separately, the runtime system should run with no error after complete porting.

The most important component is **SysTime**. The following two functions must be implemented first:

1. `SysTimeGetMs`: With this function, all timeouts are calculated
2. `SysTimeGetUs`: With this function, the scheduler gets its time base to execute the IEC tasks.

Both functions must provide monotonic rising ticks in a millisecond (`SysTimeGetMs`) and a microsecond (`SysTimeGetUs`) resolution.

The second important component is the **SysMem** component. With the component, access to all memory (except static memory) is managed.

## 9.6 Licensing

Some of the products from 3S - Smart Software Solutions GmbH with costs, which are downloaded to and used on the PLC need to be licensed for this PLC.

Examples for licensed products are:

- CANopen Master / -Slave
- Modbus Master / -Slave
- HMI
- Softmotion
- ...

There are a few possible ways to get such a license for your PLC.

### 9.6.1 Derivate based licensing

To be able to use licensed CODESYS products, like fieldbusses and libraries, on your PLC, you need to have a valid license file called `"3S.dat"` for your PLC. This file is (by default) fixely bound to on target derivate. The informations that are checked are the following:

- Operating System
- CPU Type
- Vendor ID
- Target ID
- Device Type

If those values are correct, the licenses which are contained in the license file `"3S.dat"`, can be used on this PLC.

If those values are not correct, none of the licensed products can be used on this PLC.

If the file is available, but doesn't contain the requested license, the corresponding product will run in demo mode. The demo period for most products is around 30min.

If the license file is missing completely there are two ways to react on that:

1. If the compile flag LICENSE\_THROWEXCEPTION is set for the application, the PLC will switch to an exception state after loading this application.
2. If the compile flag LICENSE\_THROWEXCEPTION is not set for the application, the product will not be licensed, an error will be logged but the application will still be able to run.

To set this compiler flag automatically, you may want to add it to your device description to the following section:

```
<DeviceDescription>
<Device>
  <ExtendedSettings>
    <ts:TargetSettings>
      <ts:section name="codegenerator">
        <ts:setting name="compiler-defines">
          <ts:value>LICENSE_THROWEXCEPTION</ts:value>
```

## 10 Coding Guidelines

The runtime system is implemented high portable, whereby some restrictions and specifications may arise during the implementation of its own components or during the modification of existing components.

The most important coding standards are explained briefly below.

### 10.1 General

1. Implementation in standard ANSI-C (no C99 extensions!)
2. No C++ comments in code! Some C-Compilers does not support this.
3. Switch on the highest warning level in the compiler or use strict ANSI-C
4. Every component must implement the standard component interface (see chapter 2.7)

### 10.2 Naming conventions and identifier

1. The names of all components always begin with **Cmp**, e.g.: CmpApp for the application component. The system components always begin with **Sys** and the IO driver always with **IoDrv**.
2. Function names in component interfaces should always have a component prefix attached to make sure they are unique. The prefix should be as short as possible (ideally 3 or 4 characters), e.g.:  
**AppCreate()**; /\* Create component function CmpApp \*/
3. Customer-specific components should always have the company name attached, e.g.:  
CmpApp3S (3S specific components which replace the standard CmpApp component)  
or:  
Cmp3S (3S specific components, which is loaded as an extension of the runtime system)
4. Names of functions, variables and structures in *camel case*:
  - Functions begin with upper case letters
  - Variables with lower case letters
  - Structures begin with upper case letters

External library functions are always written in lower case, to distinguish them from C functions, which usually have the same name.

5. Defines are always written in upper case letters, e.g. #define MAX\_APPLICATIONS 10  
The pre-processor instructions must also have to be written in upper case letters.
6. Function pointers starts with **pf** as prefix
7. Use Hungarian notation , i.e. the data and memory type should be coded in the variable prefix, e.g.:

**u**nsigned **l**ong **u**lNumOfTasks; /\* local variable \*/  
static variables with **s\_** as prefix, e.g. static char s\_cTag;

Pointer with **p** as prefix, e.g. char \*pszName;

Function pointers with **pf** as prefix, e.g. INT\_HANDLER \*pfHandler;

static arrays with **a** as prefix, e.g. static unsigned char s\_abyBuffer[100];

**Note:** Static arrays have the prefix **a** to avoid confusing a variable with a pointer **p**!

Type of data	Prefix	Description
char *	psz	String terminated by zero
Char	sz	Static string terminated by zero
char	c	
unsigned char	by	
short	s	
unsigned short	us	
int	i or n or b	i for index. n for number. b for Bool
unsigned int	ui	
long	l	
unsigned long	ul	
long long	ll	
unsigned long long	ull	
float	f	
double	d	

## 10.3 Data types

The runtime system defines its own typesystem based on ANSI stdtypes.h. The types are defined in CmpStd.h and starts always with RTS\_, e.g. **RTS\_UI32**.

### Rules:

Use always the predefined RTS\_ datatypes in all interfaces!

Additionally you must use it in all implementations, where a fix size is needed (e.g. L7 service in communication)!

For all other internal impleemntations it is recommended, to use this datatypes.

For IEC libraries there are the corresponding IEC type definitions in CmpStd.h:  
e.g. **RTS\_IEC\_BOOL** for an IEC bool value

### RTS datatypes:

The standard RTS\_ datatypes contains the real size Bits, e.g. RTS\_I8 (8 Bits). So the corresponding size is identical on every platform!

There are some special datatypes which usage is explained here:

Datatype	Size in Bits	Usage
<b>RTS_HANDLE</b>	Size to hold always a pointer	For every handle in the runtime system
<b>RTS_RESULT</b>	32	For error code handling
<b>RTS_UINTPTR</b>	Size to hold always the address of a pointer	For pointer arithmetic hold in an unsigned integral type
<b>RTS_INTPTR</b>	Size to hold always the address of a pointer	For pointer arithmetic hold in a signed integral type
<b>RTS_PTRDIFF</b>	Size to hold always a pointer difference for a buffer offset	Signed integral type that can hold an array index
<b>RTS_SIZE</b>	Size to hold always a buffer offset	Unsigned integral type that can hold a buffer offset
<b>RTS_INT</b>	Variant size, typical the platform specific int size	Data type has no constant size, so be careful in sharing shuch datatypes with IEC!
<b>RTS_BOOL</b>	Variant size, typical the platform specific int size	Boolean value (TRUE or FALSE). For best performance use platform specific int. Data type has no constant size, so be

## ANSI-C datatypes:

Datatype	Range		Platform (Bits)			
	Min	Max	16	32	64 <sup>6</sup>	64 <sup>7</sup>
char	-128	127	1	1	1	1
unsigned char	0	255	1	1	1	1
short	-32768	32767	2	2	2	2
unsigned short	0	65535	2	2	2	2
int	-2.147.483.648	2.147.483.647	2	4	4	4
unsigned int	0	4.294.967.295	2	4	4	4
long	-2.147.483.648	2.147.483.647	4	4	8	4
unsigned long	0	4.294.967.295	4	4	8	4
long long	-9.223.372.036.854.755.807	9.223.372.036.854.755.807	8	8	8	8
unsigned long long	0	18.446.744.073.709.551.615	8	8	8	8
float	0	3.40282347E+38 accuracy: 6-digits mantissa: 23-Bits exponent: 8-Bits	4	4	4	4
double	0	1.7976931348623157E+308 accuracy: 15- digits mantissa: 52-Bits exponent: 11-Bits	8	8	8	8
long double	0	3.4E-4932 1.1E+4932 accuracy: 19- digits mantissa: -Bits exponent: -Bits	10	(8) <sup>8</sup> 10	10	(8) <sup>8</sup> 10
void*			2	4	8	8
wchar_t			2	2 <sup>9</sup> 4 <sup>10</sup>	2 <sup>9</sup> 4 <sup>10</sup>	29 4 <sup>10</sup>

<sup>6</sup> In LP64 data model (used in most Unix and Unix like Systems)

<sup>7</sup> In LLP64 data model (used in Windows Systems with Microsoft Visual Studio)

<sup>8</sup> On Windows System, long double is used as alias for double

<sup>9</sup> On Windows Systems

<sup>10</sup> On most Unix and Unix like Systems

## 10.4 Component interfaces and dependencies

1. Calling another component interface function must always be called with the **CAL\_** prefix (see chapter 2.7).  
**NOTE:**  
**The CAL\_ macro must not be used for calling functions within the same component!**
2. Optional dependent interface functions must be checked with the check function **CHK\_Fct** the availability before calling **CAL\_Fct** !
3. Usually all interface (\*If.h) and dependency files (\*Dep.h) should be created from m4 compiler. This enables changes to the component link process to be carried out easily.
4. **CDECL** must precede each function prototype (\*If.h) and also the implementation of each function, e.g.:  

```
int CDECL Test(int i); /* prototype in *.itf.h */
int CDECL Test(int i)
{
    ...
}
```
5. **CDECL\_EXT** is created automatically before each function prototype of an external library function. Is used in all prototypes of external library functions in order to attach huge casts to these functions.

Therefore this macro must also be used for every implementation, thus:

```
int CDECL CDECL_EXT __testlibfct (int i); /* prototype in *.itf.h */
int CDECL CDECL_EXT __testlibfct(int i) {...}
```

6. Only use standard ANSI-C functions and headers (no strcmpi() or similar)!  
Don't use any operating system-specific functions in kernel components!!!  
Always use the Sys-API or CMUtils functions for this purpose.
7. Declare all local functions as static (name collisions in the case of static linking)
8. All interface functions of the components should:
  - usually returns **RTS\_RESULT** as a standard return value
  - always use **RTS\_HANDLE** as handle and **RTS\_INVALID\_HANDLE** as invalid handle value
  - if **RTS\_HANDLE** is returned, then always include **RTS\_RESULT \*pResult** as a parameter in the prototype. Hence the exact failure cause can be determined in case the **RTS\_INVALID\_HANDLE** is returned
9. To specify a task, use always one of the following m4-Macros in the Dep.m4 file:
  - **TASK**(`Name', `Priority):  
This macro is to specify a normal task. Please specify a comment for the documentation:  

```
/**
 * <category>Task</category>
 * <description>
 * Block driver communication task.
 * </description>
 */
TASK(`BlkDrv'dp', `TASKPRIO_HIGH_'ND')
```
  - **TASKPREFIX**(`Name', `Priority):  
Task prefix is used, if specified task name is extended by some additional information (like a channel number, etc.).
  - **TASKPLACEHOLDER**(`Name', `Priority):  
Placeholder of a task that is replace by the configured name (e.g. by the user defined name of an IEC-task).



## 10.5 Startup sequence

- **CH\_INIT:**  
Init your own component local stuff
- **CH\_INIT2:**  
You can call other component interface functions  
An event provider must create here the event object!
- **CH\_INIT3:**  
Further init stuff  
An event consumer can open here the existing event and register its callback routine
- **CH\_INIT\_TASKS:**  
At this hooch you must be aware of multitasking calls!
- **CH\_INIT\_COMM:**  
Here the communication server are started and open the runtime system the the world around

The same in opposite direction for the shutdown sequence.

## 10.6 Alignment

Usually you must ensure that all structures are aligned „naturally“. This means that you must align all byte structure elements to byte boundaries, all 2 byte elements to 2 bytes, all 4 byte elements to 4 bytes, all 8 byte data types to 8 bytes etc.

## 10.7 Use of special macros

1. Avoid asserts in real error conditions. Asserts should only catch error conditions which the developer doesn't await and which cannot occur under normal operation. Those errors are typically caused by corrupted memory or false locking. When defining an assert, the `RTS_ASSERT` macro should be used. This can be defined in the file `sysdefines.h`
2. Prefix `HUGEPTR` macro of all pointer variables or use for copying operations, which address a memory area above 64 kB, e.g.:

```
unsigned char HUGEPTR *pbyBuffer;          /* or */  
memcpy((unsigned char HUGEPTR *)pbyDest, (unsigned char HUGEPTR *)pbySrc, Len);
```

The newly introduced `CMSafeMemCpy()` utility function, which casts the pointer correctly and checks the areas, should be used instead of a `memcpy`.

3. Declare **`USEIMPORT_STMT`** instead of **`USEEXTERN_STMT`** in `Sys___OS` modules

## Appendix A: Migrating from CODESYS Control Version 2 to Version 3

To migrate from CODESYS Runtime System 2.x to V3, you have to do some adaptations. In this chapter you will find some hints to move to CODESYS Control 3.

- **Target-Settings / Device Config Files:** The device configuration files from CODESYS V2.x have completely changed in CODESYS V3. All information about a target and a device are stored in Device-Description XML-files. See chapter 6 for detailed information about the content and structure of a device description.
- **Custom- and IO-Driver Interface:** IO-drivers must be ported to CODESYS Control V3. In CODESYS Control 3 you have the possibility to write an IO-driver classical in ANSI-C, but also in IEC!  
The migration of an IO-driver from CODESYS 2 to 3 has to be done in 3 steps:
  1. Integration of the new component interface
  2. Implementation of the new interfaces (at least IBase and ICmploDrv)
  3. Adaptation to the new IO-configuration structure (connectors and parameters) and the new IO-update
- **Hooks:** Hooks from CODESYS Control 2.x are split into 3 different types of functionalities. Events are used to notify special component about an event in the runtime system. Hooks are only used in CODESYS Control V3 to notify every component about an event, like the start up and shutdown sequences.  
The old hooks that are changing settings and configurations in the runtime system are completely replaced by the settings component. So the behaviour of a runtime is only dependant of a defined set of settings and not, which custom component changes which hook!
- **External Libraries:** In CODESYS Control V3, no external library list must be provided by the component that implements the library functions. You can specify now for each function, if it is exported as an external library or not. The second main difference to CODESYS Control 2.x is that all library functions in CODESYS Control V3 have only one pointer to a parameter structure! This must be changed during migration.
- **Custom services:** The level 7 services have been changed in CODESYS Control V3 to a tagged binary format. So here, all services must be adapted to this new tagged format.
- **PLC Browser:** The PLC-Browser is available in CODESYS Control 3.4 with the component named CmpPlcShell.  
The interface of defining own command has changed. But there is a template component under \$\Templates\CmpPlcShellHandler, that illustrates the handling of commands in own components.  
Additionally to handle commands in a runtime component, in V3 you can handle own commands in IEC (e.g. in Libraries or IO-drivers).
- **RtsSym:** This module is replaced by the CmplecVarAccess interface. The functions are comparable. The only big difference is browsing the list of variable. In CODESYS Control 2 you got one big list with all variables. In CODESYS Control 3 you can browse in a hierarchical order through all symbols.
- **Gateway, ARTI, PLCHandler:** The Gateway and the ARTI interfaces are no longer supported. They must be replaced by the PLCHandler. The PLCHandler provides the possibility, to use connections to CODESYS Control 2.x and V3 runtime versions!  
The GClient interface can be replaced by the GwClient interface to use an entry interface in the CODESYS Control V3 network. But here no symbolic information is available.
- **OPC Server:** We provide an OPC server (CODESYS OPC Server V3), that enables connections to runtime systems generation V2.3 and V3 and is released.

## **Bibliography**

- [1] CODESYSControlV3\_Reference.pdf
- [2] PLCHandler Programming Guide.pdf
- [3] RTSc configurator.chm

**Change History**

Version	Description	Editor	Date
0.1	6.2.2004	TZ	06.02.2004
0.2	Some corrections	BR	03.12.2004
0.3	All chapters rearranged, translated and completed	AH	08.01.2007
0.4	Review and Rework	TZ	09.03.2007
1.0	Formal Review, Rework and Release	MN	09.03.2007
1.1	Chap 7.2.2 Figure 13 added	AH	14.06.2007
1.2	New section for target settings 6.4.2.1.5 and 6.4.2.1.6	StR	15.10.2007
2.0	Formal Review and Rework, Release	MN	18.12.2007
2.1	Note in chap. 6.4.1.3 (#32561); extension in chap.7.4 (#31361), chap. 2.6-2.9 (#32350); 6.4.2.1.2 byte-addr.mode (#32501); chap. 6.4.2.1.4 max_number_of_applications (#32063); chap. 6.3.2 "alwaysmapping" (#32827); chap. 6.4.1.3, 6.4.2.1.6.2 "placeholderlib" (#33161); chap. 6.4.1.3: "UpdateLosInStop", "StopResetBehaviour" (#32806); 6.4.2.1.3 "supportprofiling";	MN	05.02.2008
2.2	Reworked chapter 6.4.2.1.2 Memory Settings (#32258)	BW	05.03.2008
3.0	Release after formal review and rework	MN	11.03.2008
3.1	Chap. 3.7.5 (SysFileFlash) and 3.7.6 (SysFlash) added	TZ/MN	03.04.2008
3.2	Chap. 6.4.2.1.4 added: Network variables; 3.2.4 boot app. for HMI (#33655); 6.4.2.1.2: target setting address-assignment-guid (#32502); 6.4.2.1.6: target setting simulation-disabled (#34079); 9.1.5.4 + 9.1.5.5: cfg-entries Gateway + visu (#32166); general rework of chapters Architecture and Overview on components on functions; 7.8 Diagnostic, 7.9 Consistency; 3.3.4 Bootproject, 3.4 Watchdog handling; 3.9.5 SysFileFlash, 3.8.6 SysFlash; 4 Portings: extended	WH/MN	23.06.2008
3.3	Review and changes in 3.3.4	AH	9.07.2008
4.0	Formal Review and Release	MN	9.07.2008
4.1	Adapted chapter 6.4.2.1.7.3 (Exclude Library Category) to the current implementation (#32665)	KeK	21.07.2008
4.2	Added attribute "SecureOnlineMode" in chapter 6.4.1.5 (Functional)	KeK	31.07.2008
4.3	Formal review and rework	MN	03.09.2008
5.0	Release	MN	03.09.2008
5.1	Chapter 7.8 extended	AH	29.09.2008
5.2	Chapter 7.8 review	PhB	01.10.2008
5.3	Chapter 6.4.2.1.2 improved for retain area size	AH	01.10.2008
5.4	Chapter 6.4.2.1.2 Review	MN	02.10.2008
6.0	Release	MN	02.10.2008
6.1	Modifications according to tracker items 35191, 35756, 34803, 34816, 33661	SE	06.10.2008
6.2	Extension of chapter 3.7.1 according to tracker items	SE	13.10.2008

Version	Description	Editor	Date
	32059, 32631		
6.3	Extension of chapter 6.4.2.1.5 according to #34842	SE	20.10.2008
6.4	Added example of SysMemAllocArea	TZ	22.10.2008
6.5	Additional information in chapter 7.8 added (#33848), Review	AH/AS	13.01.2009
7.0	Formal Review, Release	MN	13.01.2009
7.1	Chapter 7.9 improved	AH	16.01.2009
7.2	Retain behaviour chap. 3.3.5, 6.4.2.1.2, chap. 3.3.5, 6.4.2.1.5 (#36858), 6.4.1.4 (#34273), 6.4.1.3 (#33977), chap. 9.1.4 (#36145), 6.4.2.1.2 (#35384, as from V3.3!), chap. 6.3.2 (#31837); chap. 3.5 extended; Kap.6.6 (#35474)	MN/AH/StR	20.01.2009
8.0	Formal Review, Release	MN	25.02.2009
8.1	chaps. 2.7 (files); 4.1.1 (service, cfg-file); 4.3 (#35276); 5.3.4 (#36006); 6.4 (devdescr) extensions ; 6.4.4.3 (#37173), 6.4.4.4.3 (#36577); 6.4.4.5 (#37338, #35240), 6.4.5.1.1 (#37098, cycle_control_version_2); 6.4.5.1.3 (#33374); 6.4.5.1.4 (networkvars); 6.4.5.1.5 (#37291, #37033), ), 6.4.5.1.8 (#34619), 6.4.5.1.9 (#37750), 6.4.5.1.10, 6.4.5.1.11 (#35258)	MN	16.04.2009
8.2	Chaps. 7.8.1 General Diagnostic Information Bit-Field: Startup behaviour	AK	23.04.2009
9.0	Release	MN	20.05.2009
9.1	Chap. 6.4.4.1 (CDS-11195), 6.4.5.1.2 (3S), chap.3.3.5, (CDS-11353), 6.4.5.1.12 (CDS-10239), 6.4.5.1.9 (CDS-5148)	MN (in coord.with developers)	13.07.2009
10.0	Release	MN	13.07.2009
10.1	Chap. 7.8.3 (CDS-9195)	MN	30.07.2009
10.2	(CDS-10922); 6.4.5.1.3 + 6.4.5.1.3.1 (Task Settings removed); 6.4.5.1.9 (CDS-7580), 6.4.5.1.1 (CDS-10766); 6.4.4.4.1 (CDS-13848)	AK MN	25.08.2009 18.02.2010
10.3	Added 5.4.2.1 Parallel Routing	JS	03.03.2010
10.4	(CDS-7682); "Motorola Byte Order"; "hexfile" (CDS-9291); 4.3.1 (CDS-11578); 6.4.5.1.3 Online section (TargetSettings)	MN	18.03.2010
11.0	Release	MN	19.03.2010
11.1	CDS-15266: 3.6.3 improved 3.7 created Appendix A extended Appendix B updated	AH	01.04.2010
11.2	CDS-15140: 6.4.4.4 needsBusCycle attribute documented 7.2.2 IoDrvStartBusCycle chapter improved	AH	15.04.2010
11.3	CDS-14865: 3.5.3 chapter added for external event tasks	AH	15.04.2010
11.4	CDS-727: 3.3.10 chapter added CDS-17713: 2.4 / 2.5 chapter added CDS-14331: 2.3 chapter added	AH	27.05.2010

Version	Description	Editor	Date
11.5	CDS-14875: 4.3.2 chapter added CDS-11734: 7.10 chapter added	MN	08.06.2010
11.6	CDS-17700: chap. 6.4.5.1.9 webvisualization_client added	StS	01.07.2010
11.7	CDS-78609: chap. 0 extended	MN	07.07.2010
12.0	Release	MN	13.07.2010
12.1	CDS-3547: chap. 3.14 License Check	AS	26.07.2010
12.2	CDS-17525: Rename Product CODESYS Control V3 in CODESYS Control V3 (Second Step)	AH	14.10.2010
12.3	CDS-17763: chap. 7.11 added / formal rework	DPa /MN	25.10.2010
12.4	CDS-17763: chap. 7.11 reviewed/reworked content	IH	26.10.2010
12.5	CDS-19470: chap 0 corrected (Trace)	AH	26.10.2010
12.6	chap. 7.11.1, corrections	DPa/MN	04.11.2010
12.7	CDS-19279, CDS-17449, CDS-17455: chap. 6.4.5.1.1, CDS-18387: chap. 6.4.5.1.4.1, CDS-20139	MN/div	06.12.2010
13.0	Release after formal review	MN	09.12.2010
13.1	CDS-10897: chap. 6.4.4.6, extended; CDS-20407, CDS- 16903: chap. 0	MHa/MN	22.12.2010
13.2	CDS-20109: chap. 6.4.5.1.9	MN	04.01.2011
13.3	note conc. persist.variables: chap. 6.4.5.1.2	MN	05.01.2011
13.4	CDS-19462 link-all-globalvariables CDS-21274 constants-in-own-segment, chap. 6.4.5.1.2 CDS-20984 online-change-in-own-segment, chap. 6.4.5.1.2 CDS-19502 optimized_online_change, chap. 6.4.5.1.1 CDS-20985 minimal-structure-granularity, chap. 6.4.5.1.2	MN	22.03.2011
13.5	Review and Rework chap. 6.4.5.1.1 and 6.4.5.1.2	BW	22.03.2011
13.6	Rework acc. To Review: chap. 6.4.4.6, chap.6.3.3	MN	24.03.2011
14.0	Release	MN	24.03.2011
14.1	3.3.4.2 corrected and 3.3.4.3 added	AH	12.05.2011
14.2	Chapter 1. CODESYS SP removed	AH	21.06.2011
14.3	CDS-22659 "retain-in-cycle"	MN/AH	08.07.2011
15.0	Release after formal review	MN	19.07.2011
15.1	CDS-18873 (chap. 6.4.5.1.5)	MN/WH	20.07.2011
15.2	CDS-20032/16581 (chap. 6.4.5.1.2)	MN/BW	25.07.2011
15.3	CDS-24565 (chap. 3.4)	MN	20.09.2011
15.4	Reviewed chapter 3.4	IH	20.09.2011
15.5	CDS-17996 (chap. 6.4.5.1.2.2)	MN/BW	12.10.2011
15.6	In cooperation with responsible assignee: CDS-24218, CDS-22003 (4.2); CDS-24955 (4.2); CDS-23038 (4.2); CDS-24102 (3.6.3); CDS-25015 (6.4.5.1.9); CDS-17454 (6.4.4.4)	MN/StR	01.12.2011

Version	Description	Editor	Date
15.7	In cooperation with responsible assignee: CDS-22629 (0), CDS-25024 (6.4.5.1.10)	MN/BW	02.12.2011
15.8	In cooperation with responsible assignee: CDS-4613 (0), CDS-24006 (6.4.5.1.2)	MN/StR	05.12.2011
16.0	Release after formal review and rework	MN	05.12.2011
16.1	CDS-27255 (3.3.2)	MN	09.03.2012
16.2	CDS-27255 (own chapter 3.3.9 + extension, review by AH)	MN	12.03.2012
16.3	CDS-21387 (6.4.5.1.4.1 "systemtick"); CDS-19331 (6.4.4.8)	MN	20.03.2012
16.4	CDS-27968	MN	28.03.2012
16.5	CDS-15615 (6.4.5.1.4.1)	MN	09.05.2012
16.6	CDS-24679 (3.15)	AH	15.05.2012
16.7	CDS-25568 (3.10.19.2: Win32.MaxJpegByteArraySize)	MN/AH/PB	21.05.2012
16.8	CDS-22641 (6.4.4.6; attribute "functional")	MN	08.06.2012
17.0	Release after formal review and rework	MaH	20.06.2012
17.1	CDS-13471 ; dp-register-addressing	MaH	20.06.2012
17.2	Little-Endian (Motorola) --> Little-Endian (Intel) (7.11.2)	MaH	23.07.2012
17.3	CDS-8251: (6.3) ConnectorTypes updated	AH	30.07.2012
17.4	CDS-28345: (6.4.8) BOOL instead of BOOLEAN	MN	24.08.2012
17.5	CDS-29303; CODESYS + spellcheck	MN	17.09.2012
17.6	CDS-26177: chap. 6.4.5.1.2	BW	20.09.2012
17.7	CDS-31487: chap.1.1: Security note added	MN	20.11.2012
17.8	CDS-31487: chap.6.4.5.1.14: object types restriction	MN	06.12.2012
18.0	Release after formal review	MN	06.12.2012
18.1	Chapter 9.3.3 improved	AH	18.12.2012
18.2	Rework of chapter 10 (previously Appendix A)	AH	18.12.2012
18.3	Chapter 3.13.1 documented	AH	07.01.2013
18.4	CDS-31915: see 6.4.5.1.7 CDS-30732 CDS-22242 CDS-29870 CDS-25678	StR	07.01.2013
18.5	CDS-29583: chap. 4.2: new setting WinCE.DisableMapPhysicalInVirtualAllocCopyEx	JT	07.01.2013
18.6	CDS-29621 CDS-27403 CDS-29853 CDS-29577 CDS-26286 CDS-18350	WH	10.01.2013
18.7	CDS-29775: see 6.4.5.1.7	StR	14.01.2013
18.8	CDS-22641 (6.4.4.6; attribute "functional")	StR	16.01.2013
18.9	CDS-3001 (7.2.2 ; IoDrvScanModules)	StR	16.01.2013

<b>Version</b>	<b>Description</b>	<b>Editor</b>	<b>Date</b>
18.10	CDS-29422 (see the meaning of task priority)	WH	17.01.2013
18.11	CDS-28218 Added chapter 9.6, describing the licensing process	IH	18.01.2013
18.12	CDS-23172 Added chapter 4.3.1.4, describing static memory areas for VxWorks	IH	18.01.2013
18.13	Added chapter 9.3.8 Changed design of some drawings to match the scheme of the master template of the document.	IH	25.01.2013
18.14	Added setting for multithreading builds to chapter 6.4.5.1.6. Highlighted all XML snippets from chapter 6.4.5.1	IH	25.01.2013
18.15	CDS-29286: Chapter 3.4 updated	AH	29.01.2013
18.16	CDS-21243: Targetsetting CPU for SH-Codegenerator	WH	29.01.2013
18.17	CDS-22621: Compile: Task Stack Overflow is not detected	WH	29.01.2013
18.18	CDS-29833 CDS-29661 CDS-29563 CDS-26822 CDS-26296	WH	30.01.2013
18.19	CDS-24105 CDS-26255	PB	15.02.2013
19.0	Release after formal review and rework	MN	13.03.2013