# Table of Contents

## About the Book .................................................. 13

### Part I General Presentation of Unity Pro ................. 15
  - Introduction ..................................................... 15

### Chapter 1 Presentation ........................................ 17
  - Introduction ..................................................... 17
  - Capabilities of Unity Pro ...................................... 18
  - User Interface .................................................. 21
  - Project Browser ................................................ 23
  - Compatibility Between PLCs and Versions of Unity Pro ... 25
  - Configurator ..................................................... 28
    - Data Editor .................................................... 33
    - Program Editor ............................................... 39
    - Function Block Diagram FBD ................................ 42
    - Ladder Diagram (LD) language .............................. 44
    - General information about SFC sequence language .... 46
    - Instruction List IL ............................................ 50
    - Structured Text ST ............................................ 51
    - PLC Simulator ................................................ 52
    - Export/Import ................................................. 53
    - User Documentation .......................................... 54
    - Debug Services ............................................... 55
    - Diagnostic Viewer ........................................... 62
    - Operator Screen ............................................. 63

### Part II Application structure ................................. 65
  - At a Glance ....................................................... 65

### Chapter 2 Description of the available functions for each type of PLC .......... 67
  - Functionalities available for the different types of PLC ... 67

### Chapter 3 Application program structure .................... 69
  - At a Glance ....................................................... 69
  3.1 Description of tasks and processes ......................... 70
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2</td>
<td>Description of sections and subroutines</td>
<td>76</td>
</tr>
<tr>
<td>3.3</td>
<td>Mono task execution</td>
<td>81</td>
</tr>
<tr>
<td>3.4</td>
<td>Multitasking execution</td>
<td>88</td>
</tr>
<tr>
<td>Chapter 4</td>
<td>Application memory structure</td>
<td>109</td>
</tr>
<tr>
<td>Chapter 5</td>
<td>Operating modes</td>
<td>117</td>
</tr>
<tr>
<td>Chapter 6</td>
<td>System objects</td>
<td>129</td>
</tr>
</tbody>
</table>

**Chapter 4 Application memory structure**

<table>
<thead>
<tr>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>At a Glance</td>
<td>109</td>
</tr>
<tr>
<td>Memory structure of Premium and Atrium PLCs</td>
<td>110</td>
</tr>
<tr>
<td>Detailed description of the memory zones</td>
<td>112</td>
</tr>
<tr>
<td>Memory structure of Quantum PLCs</td>
<td>113</td>
</tr>
<tr>
<td>Detailed description of the memory zones</td>
<td>115</td>
</tr>
</tbody>
</table>

**Chapter 5 Operating modes**

<table>
<thead>
<tr>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>At a Glance</td>
<td>117</td>
</tr>
<tr>
<td>Processing of power outage and restoral</td>
<td>118</td>
</tr>
<tr>
<td>Processing on Cold Start</td>
<td>121</td>
</tr>
<tr>
<td>Processing on warm restart</td>
<td>125</td>
</tr>
<tr>
<td>Automatic start in Run</td>
<td>127</td>
</tr>
<tr>
<td>PLC HALT mode</td>
<td>128</td>
</tr>
</tbody>
</table>

**Chapter 6 System objects**

<table>
<thead>
<tr>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>At a Glance</td>
<td>129</td>
</tr>
<tr>
<td>6.1 System bits</td>
<td>130</td>
</tr>
</tbody>
</table>
At a Glance ............................................................. 196
Overview of data types in binary format .............................. 197
Boolean Types ....................................................... 199
Integer types ....................................................... 204
The Time type ..................................................... 206

8.2 Elementary data types (EDT) in BCD format ................. 207
At a Glance ............................................................. 207
Overview of data types in BCD format .............................. 208
The Date type ....................................................... 210
The Time of Day (TOD) type ..................................... 211
The Date and Time (DT) type ..................................... 212

8.3 Elementary Data Types (EDT) in Real format ............... 214
Presentation of the Real data type ................................ 214

8.4 Elementary Data Types (EDT) in character string format . 217
Overview of data types in character string format ............... 217

8.5 Elementary data types (EDT) in bit string format .......... 220
At a Glance ............................................................. 220
Overview of data types in bit string format ....................... 221
Bit string types ...................................................... 222

8.6 Derived Data Types (DDT/IODDT) ......................... 224
At a Glance ............................................................. 224
Tables ................................................................. 225
Structures ............................................................ 228
Overview of the Derived Data Type family (DDT) ............... 229
DDT: Memory occupancy .......................................... 232
Overview of Input/Output Derived Data Types (IODDT) .... 235

8.7 Function block data types (DFB/EFB) ....................... 237
At a Glance ............................................................. 237
Overview of Function Block data type families ................. 238
Characteristics of Function Block data types (EFB/DFB) .... 240
Characteristics of elements belonging to function blocks .... 242

8.8 Generic Data Types (GDT) ..................................... 245
Overview of Generic Data Types ................................. 245

8.9 Data types belonging to Sequential Function Charts (SFC) 248
Overview of the data types of the Sequential Function Chart family .... 248

8.10 Compatibility between data types ............................ 250
Compatibility between data types ................................ 250

Chapter 9 Data instances .......................................... 255
At a Glance ............................................................. 255
Data Type Instances ............................................... 256
Data instance attributes .......................................... 260
Direct Addressing Data Instances ................................. 263

Chapter 10 Data references ........................................ 271
At a Glance ............................................................. 271
| References to data instances by value | 272 |
| References to data instances by name | 274 |
| References to data instances by address | 277 |
| Data naming rules | 281 |

**Part IV Programming language**  283

| Chapter 11 Function Block language FBD | 285 |
| Introduction | 283 |
| General information about the FBD Function Block language | 286 |
| Elementary Functions, Elementary Function Blocks, Derived Function Blocks and Procedures (FFBs) | 288 |
| Subroutine Calls | 297 |
| Control Elements | 298 |
| Link | 299 |
| Text Object | 301 |
| Execution sequence of the FFBs | 302 |
| Change Execution Sequence | 304 |
| Loop Planning | 309 |

| Chapter 12 Ladder Diagram LD | 311 |
| Introduction | 311 |
| General information about the LD Ladder Diagram language | 312 |
| Contacts | 315 |
| Coils | 317 |
| Elementary Functions, Elementary Function Blocks, Derived Function Blocks and Procedures (FFBs) | 319 |
| Control elements | 329 |
| Operation and compare blocks | 330 |
| Links | 332 |
| Text Object | 335 |
| Edge recognition | 336 |
| Execution Sequence and Signal Flow | 343 |
| Loop Planning | 345 |
| Change Execution Sequence | 347 |

| Chapter 13 SFC Sequence Language | 353 |
| Introduction | 353 |
| 13.1 General information about SFC sequence language | 354 |
| Introduction | 354 |
| General information about SFC sequence language | 355 |
| Link rules | 359 |
| 13.2 Steps and Macro Steps | 360 |
| Introduction | 360 |
| Step | 361 |
Chapter 14 Instruction list IL ................................. 417

14.1 General information about the IL instruction list. 418
Introduction ................................................. 417
Operands ..................................................... 421
Modifier ..................................................... 424
Operators ..................................................... 426
Subroutine Call .............................................. 437
Tags and Jumps .............................................. 438
Comment ..................................................... 440
Chapter 17 Description of User Function Blocks (DFB) .......................... 523
  At a Glance ..................................................................................... 523
  Definition of DFB Function Block Internal Data .................................. 524
  DFB Parameters ............................................................................. 526
  DFB variables ............................................................................... 528
  DFB code section .......................................................................... 529

Chapter 18 User Function Blocks (DFB) instance ............................... 531
  At a Glance ..................................................................................... 531
  Creation of a DFB instance ............................................................. 532
  Execution of a DFB instance ........................................................... 534
  Programming example for a Derived Function Block (DFB) .............. 535

Chapter 19 Use of the DFBs from the different programming languages 539
  At a Glance ..................................................................................... 539
  Rules for using DFBs in a program .................................................. 540
  Use of IODDTs in a DFB ................................................................. 542
  Use of a DFB in a Ladder language program .................................... 546
  Use of a DFB in a Structured Text language program ....................... 548
  Use of a DFB in an Instruction List program .................................... 551
  Use of a DFB in a program in Function Block Diagram language ........ 555

Chapter 20 User diagnostics DFB ......................................................... 557
  At a Glance ..................................................................................... 557
  Presentation of user diagnostic DFBs ............................................... 558
  How to create a user diagnostic DFB ............................................... 559

Appendices ...................................................................................... 563
  Introduction .................................................................................... 563

Appendix A EFB Error Codes and Values ........................................... 565
  Overview ....................................................................................... 565
  Tables of Error Codes for the Base Library ....................................... 566
  Tables of Error Codes for the Diagnostics Library ............................. 568
  Tables of Error Codes for the Communication Library ..................... 569
  Tables of Error Codes for the IO Management Library ..................... 573
  Tables of Error Codes for the CONT_CTL Library ............................ 582
  Tables of Error Codes for the Motion Library ................................... 590
  Tables of Error Codes for the Obsolete Library .................................. 592
  Common Floating Point Errors ....................................................... 599

Appendix B IEC Compliance ............................................................... 601
  At a Glance ..................................................................................... 601
  B.1 General Information regarding IEC 61131-3 ............................... 602
  General information about IEC 61131-3 Compliance .......................... 602
  B.2 IEC Compliance Tables .............................................................. 604
At a Glance .............................................................. 604
Common elements .................................................. 605
IL language elements ............................................... 617
ST language elements ............................................. 619
Common graphical elements ...................................... 620
LD language elements ............................................. 621
Implementation-dependent parameters ......................... 622
Error Conditions ..................................................... 626
B.3 Extensions of IEC 61131-3 ..................................... 628
   Extensions of IEC 61131-3, 2nd Edition .................. 628
B.4 Textual language syntax .................................... 630
   Textual Language Syntax .................................... 630
Glossary .............................................................. 631
Index ................................................................. 651
About the Book

At a Glance

Document Scope
This manual describes the elements necessary for the programming of Premium, Atrium and Quantum PLCs using the Unity Pro programming workshop.

Validity Note
Data and illustrations in this guide are not binding. We reserve the right to modify our products in line with our policy of continuous product development. The information in this document is subject to change without notice and should not be construed as a commitment by Schneider Electric.

Product Related Warnings
Schneider Electric assumes no responsibility for any errors that may appear in this document. If you have any suggestions for improvements or amendments or have found errors in this publication, please notify us.

No part of this document may be reproduced in any form or by any means, electronic or mechanical, including photocopying, without express written permission of Schneider Electric.

All pertinent state, regional, and local safety regulations must be observed when installing and using this product. For reasons of safety and to ensure compliance with documented system data, only the manufacturer should perform repairs to components.

When controllers are used for applications with technical safety requirements, please follow the relevant instructions.

Failure to use Schneider Electric software or approved software with our hardware products may result in injury, harm, or improper operating results.

Failure to observe this product related warning can result in injury or equipment damage.
About the Book

User Comments  We welcome your comments about this document. You can reach us by e-mail at techpub@schneider-electric.com.
General Presentation of Unity Pro

Introduction

Contents of this section
This section describes the general design and behavior of an application created with Unity Pro.

What's in this Part?
This part contains the following chapters:

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Chapter Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Presentation</td>
<td>17</td>
</tr>
</tbody>
</table>
## Presentation

### Introduction

#### Overview
This chapter describes the general design and behavior of a project created with Unity Pro.

#### What's in this Chapter?
This chapter contains the following topics:

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Capabilities of Unity Pro</td>
<td>18</td>
</tr>
<tr>
<td>User Interface</td>
<td>21</td>
</tr>
<tr>
<td>Project Browser</td>
<td>23</td>
</tr>
<tr>
<td>Compatibility Between PLCs and Versions of Unity Pro</td>
<td>25</td>
</tr>
<tr>
<td>Configurator</td>
<td>28</td>
</tr>
<tr>
<td>Data Editor</td>
<td>33</td>
</tr>
<tr>
<td>Program Editor</td>
<td>39</td>
</tr>
<tr>
<td>Function Block Diagram FBD</td>
<td>42</td>
</tr>
<tr>
<td>Ladder Diagram (LD) language</td>
<td>44</td>
</tr>
<tr>
<td>General information about SFC sequence language</td>
<td>46</td>
</tr>
<tr>
<td>Instruction List IL</td>
<td>50</td>
</tr>
<tr>
<td>Structured Text ST</td>
<td>51</td>
</tr>
<tr>
<td>PLC Simulator</td>
<td>52</td>
</tr>
<tr>
<td>Export/Import</td>
<td>53</td>
</tr>
<tr>
<td>User Documentation</td>
<td>54</td>
</tr>
<tr>
<td>Debug Services</td>
<td>55</td>
</tr>
<tr>
<td>Diagnostic Viewer</td>
<td>62</td>
</tr>
<tr>
<td>Operator Screen</td>
<td>63</td>
</tr>
</tbody>
</table>
## Capabilities of Unity Pro

### Hardware platforms

Unity Pro supports the following hardware platforms:
- Quantum
- Premium
- Atrium

### Programming languages

Unity Pro provides the following programming languages for creating the user program:
- Function Block Diagram (FBD)
- Ladder Diagram (LD) language
- Instruction List (IL)
- Structured Text (ST)
- Sequential Control (SFC)

All of these programming languages can be used together in the same project. All these languages conform to IEC 61131-3.

### Block libraries

The blocks that are included in the delivery of Unity Pro extensive block libraries extend from blocks for simple Boolean operations, through blocks for strings and array operations to blocks for controlling complex control loops.

For a better overview the different blocks are arranged in libraries, which are then broken down into families.

The blocks can be used in the programming languages FBD, LD, IL and ST.

### Elements of a program

A program can be constructed from:
- a Mastertask (MAST)
- a Fast task (FAST)
- one to four Help tasks (AUX)
- sections, which are assigned one of the defined tasks
- sections for processing time controlled events (Timerx)
- sections for processing hardware controlled events (EVTx)
- subroutine sections (SR)

### Software packages

The following software packages are available:
- Unity Pro M
- Unity Pro L
- Unity Pro XL
- Unity Developers Edition (UDE)
The following table shows the main characteristics of the individual software packages:

<table>
<thead>
<tr>
<th>Programming languages</th>
<th>Unity Pro M</th>
<th>Unity Pro L</th>
<th>Unity Pro XL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function Block Diagram FBD</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Ladder Diagram (LD) language</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Instruction List IL</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Structured Text ST</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Sequential Language SFC</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Libraries</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard library</td>
</tr>
<tr>
<td>Control library</td>
</tr>
<tr>
<td>Communication library</td>
</tr>
<tr>
<td>Diagnostics library</td>
</tr>
<tr>
<td>I/O Management library</td>
</tr>
<tr>
<td>System library</td>
</tr>
<tr>
<td>Motion control drive library</td>
</tr>
<tr>
<td>TCP Open library</td>
</tr>
<tr>
<td>Obsolete library</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>General information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create and use data structures (DDTs)</td>
</tr>
<tr>
<td>Create and use Derived Function Blocks (DFBs)</td>
</tr>
<tr>
<td>Project browser with structural and/or functional view</td>
</tr>
<tr>
<td>Managing access rights</td>
</tr>
<tr>
<td>Operator screen</td>
</tr>
<tr>
<td>Diagnostic viewer</td>
</tr>
<tr>
<td>System diagnostics</td>
</tr>
<tr>
<td>Project diagnostics</td>
</tr>
<tr>
<td>Application converter</td>
</tr>
<tr>
<td>Concept Converter</td>
</tr>
<tr>
<td>Managing multi-stations</td>
</tr>
</tbody>
</table>
### Naming convention

In the following documentation, "Unity Pro" is used as general term for "Unity Pro M", "Unity Pro L" and "Unity Pro XL".

<table>
<thead>
<tr>
<th>Supported platforms</th>
<th>Unity Pro M</th>
<th>Unity Pro L</th>
<th>Unity Pro XL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Premium</td>
<td>P57 00</td>
<td>P57 10</td>
<td>P57 20</td>
</tr>
<tr>
<td>Quantum</td>
<td>-</td>
<td>140 CPU 311 10 140 CPU 34 12 U/A* 140 CPU 534 14 U/A*</td>
<td>All CPUs</td>
</tr>
<tr>
<td>Atrium</td>
<td>PCI 57 204</td>
<td>All CPUs</td>
<td>All CPUs</td>
</tr>
<tr>
<td>Simulator</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Openess</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Hyperlinks</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Unity Pro Server (for OFS, UDE, UAG)</td>
<td>-</td>
<td>-</td>
<td>+</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Software components contained in the software package</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Documentation as context help and PDF</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>OS Loader tool + HW Firmware</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
</tbody>
</table>

**Legend:**

+ = available

- = not available
User Interface

Overview

The user interface consists of several, configurable windows and toolbars.

User interface:
Legend:

<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Menu bar (See Unity Pro, Operating Modes Manual, Menus)</td>
</tr>
<tr>
<td>2</td>
<td>Toolbar (See Unity Pro, Operating Modes Manual, Toolbar)</td>
</tr>
<tr>
<td>3</td>
<td>Project Browser (See Unity Pro, Operating Modes Manual, Project browser)</td>
</tr>
<tr>
<td>4</td>
<td>Editor window (programming language editors, data editor, etc.)</td>
</tr>
<tr>
<td>5</td>
<td>Register tabs for direct access to the editor window</td>
</tr>
<tr>
<td>6</td>
<td>Information window (See Unity Pro, Operating Modes Manual, Output windows)</td>
</tr>
<tr>
<td></td>
<td>(provides information about errors which have occurred, signal tracking, import functions, etc.)</td>
</tr>
<tr>
<td>7</td>
<td>Status bar (See Unity Pro, Operating Modes Manual, Status bar)</td>
</tr>
</tbody>
</table>
Project Browser

Introduction
The Project Browser displays all project parameters. The view can be shown as structural (topological) and/or functional view.

Structural view:
The project browser offers the following features in the structural view:
- Creation and deletion of elements
- The section symbol shows the section programming language and if it is protected
- View the element properties
- Creation of user directories
- Launching the different editors
- Start the import/export function
**Functional view:** The project browser offers the following features in functional view:
- Creation of functional modules
- Insertion of sections, animation tables etc. using Drag and Drop from the structural view
- Creation of sections
- View the element properties
- Launching the different editors
- The section symbol shows the section programming language and other attributes
Compatibility Between PLCs and Versions of Unity Pro

At a Glance

PLCs that can be programmed with Unity Pro software have operating systems whose version levels may be different. Unity Pro software versions can program certain PLC operating system versions. In this part, we will explain the compatibility rules that must be observed in order to implement automation projects using Unity Pro.

Possible Transfers

This illustration shows the possibility of transferring a project between versions of Unity Pro software and PLC versions.
Impossible Transfers

This illustration shows project transfers that are **NOT possible** between versions of Unity Pro software and PLC versions.

Compatibility of XEF Export Files

This illustration shows the compatibility of projects between versions of Unity Pro software for .XEF export files.
Compatibility of STU Files

This illustration shows the compatibility and incompatibility of projects between versions of Unity Pro software for STU files.

Unity Pro V1.0

Unity Pro 1.0 STU file

OK

Unity Pro V2.0

Unity Pro 2.0 STU file

OK

NOT Available

Note: The STU file must be used as a work file within the same version of Unity Pro. If you wish to switch from one version to another of Unity Pro software, we recommend that you use the XEF file.

STA Files

For versions higher than version 2.0, STA files (archive files available for version 2.0 and higher) can be used to transfer projects between versions of Unity Pro software.
Configurator

The configurator window is split into two windows:
- Catalog window
  A module can be selected from this window and directly inserted in the graphical representation of the PLC configuration using Drag & Drop.
- Graphical representation of the PLC configuration

Representation of the Configurator window:

One of the following shortcut menus is called depending on the position of the mouse pointer:
- Mouse pointer on the background allows among others:
  - Change CPU
  - Selection of different Zoom factors
- Mouse pointer on the module allows among others:
  - Access to editor functions (delete, copy, move)
  - Open the module configuration for defining the module specific parameters
  - Show the I/O properties and the total current
- Mouse pointer on an empty slot allows among others:
  - Insert a module from the catalog
  - Insert a previously copied module including its defined properties
Module configuration

The module configuration window (called via the modules shortcut menu or a double-click on the module) is used to configure the module. This also includes channel selection, selection of functions for the channel selected, assignment of State RAM addresses (only Quantum) etc.

Module configuration window for a Premium I/O module:
Module properties

The module properties window (called via the modules shortcut menu) shows the modules properties such as the power consumption, number of I/O points (only Premium) etc.

The module properties window for the power supply shows the total current of the backplane:

<table>
<thead>
<tr>
<th>Power Supply</th>
<th>EA</th>
</tr>
</thead>
<tbody>
<tr>
<td>5V</td>
<td>0 mA</td>
</tr>
<tr>
<td>24V</td>
<td>30 mA</td>
</tr>
<tr>
<td>24VR</td>
<td>1904 mA</td>
</tr>
</tbody>
</table>

Used
Overloaded
Available

VR: For relay
V: For VDC inputs

OK  Cancel
Network configuration

The network configuration is called via the communications folder.

Network configuration:

- Creation of networks
- Network analysis
- Printout of the network configuration

A window for configuring a network:

![Network Configuration Window]

The network configuration windows allows among others:
- Creation of networks
- Network analysis
- Printout of the network configuration
After configuration the network is assigned a communications module.
Data Editor

Introduction
The data editor offers the following features:
- Declaration of variable instances
- Definition of derived data types (DDTs)
- Instance declaration of elements and derived function blocks (EFBs/DFBs)
- Definition of derived function block (DFBs) parameters

The following functions are available in all tabs of the data editor:
- Copy, Cut, Paste
- Expand/collapse structured data
- Sorting according to Type, Symbol, Address etc.
- Filter
- Inserting, deleting and changing the position of columns
- Drag and Drop between the data editor and the program editors
- Undo the last change
- Export/Import

Variables
The Variables tab is used for declaring variables.

Variables tab:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Address</th>
<th>Value</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Station1</td>
<td>Bool</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Filling_up</td>
<td>Byte</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>End_of_filling</td>
<td>Date</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Initial_mixer</td>
<td>Dint</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Initial_filling</td>
<td>Dint</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Empty_mixer</td>
<td>DWord</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Proportioning</td>
<td>EBoot</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fill_up_A</td>
<td>Bool</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fill_up_B</td>
<td>Bool</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sync_1</td>
<td>Dint</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sync_2</td>
<td>Dint</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fill_up_C</td>
<td>Bool</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Start_mixer</td>
<td>Bool</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The following functions are available:
- Defining a symbol for variables
- Assigning data types
- Own selection dialog box for derived data types
- Assignment of an address
- Automatic symbolization of I/O variables
Assignment of an initial value
Entering a comment
View all properties of a variable in a separate properties dialog box

Hardware dependent data types (IO DDT)

IO DDTs are used to assign the complete I/O structure of a module to an individual variable.

Assignment of IO DDTs:

The following functions are available:

- Complete I/O structures can be assigned with individual variables using IO DDTs
- After entering the variables addresses, all elements of the structure are automatically assigned with the correct input/output bit or word
- Because it is possible to assign addresses later on, standard modules can be simply created whose names are defined at a later date.
- An alias name can be given all elements of an I/O structure.
Derived data types (DDT)

The DDT types tab is used for defining derived data types (DDTs).

A derived data type is the definition of a structure or array from any data type already defined (elementary or derived).

Tab DDT types:

The following functions are available:
- Definition of nested DDTs (max. 8 levels)
- Definition of arrays with up to 6 dimensions
- Assignment of an initial value
- Assignment of an address
- Entering a comment
- Analysis of derived data types
- Assignment of derived data types to a library
- View all properties of a derived data type in a separate properties dialog box
Function blocks

The Function blocks tab is used for the instance declaration of elements and derived function blocks (EFBs/DFBs).

Tab Function blocks:

The following functions are available:
- Display of the function blocks used in the project
- Definition of a symbol for the function blocks used in the project
- Automatic enabling of the defined symbols in the project
- Enter a comment about the function block
- View all parameters (inputs/outputs) of the function block
- Assignment of an initial value to the function block inputs/outputs
**DFB types**

The **DFB types** tab is used for defining derived function block (DFBs) parameters.

The creation of DFB logic is carried out directly in one or more sections of the FBD, LD, IL or ST programming languages.

**Tab DFB types:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Nu</th>
<th>Type</th>
<th>Value</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>valve</td>
<td></td>
<td></td>
<td></td>
<td>&lt;DFB&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bool</td>
<td>Bool</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bool</td>
<td>Bool</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bool</td>
<td>Bool</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bool</td>
<td>Bool</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Int</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Int</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The following functions are available:

- Definition of the DFB name
- Definition of all parameter of the DFB, such as:
  - Inputs
  - Outputs
  - VAR_IN_OUT (combined inputs/outputs)
  - Private variables
  - Public variables
- Assignment of data types to DFB parameters
- Own selection dialog box for derived data types
- Assignment of an initial value
- Nesting DFBs
- Use of several sections in a DFB
- Enter a comment for DFBs and DFB parameters
- Analyze the defined DFBs
- Version management
- Assignment of defined DFBs to a library
**Data usage**

Data types and instances created using the data editor can be inserted (context dependent) in the programming editors.

The following functions are available:
- Access to all programming language editors
- Only compatible data is displayed
- View of the functions, function blocks, procedures and derived data types arranged according to their library affiliation
- Instance declaration during programming is possible

Data selection dialog box:

![Data selection dialog box](image_url)
Program Editor

Introduction

A program can be built from:
- **Tasks**, that are executed cyclically or periodically.
  Tasks are built from:
  - Sections
  - Subroutines
- **Event processing**, that is carried out before all other tasks.
  Event processing is built from:
  - Sections for processing time controlled events
  - Sections for processing hardware controlled events

Example of a Program:
Tasks

Unity Pro supports multiple tasks (Multitasking).

The tasks are executed "parallel" and independently of each other whereby the execution priorities are controlled by the PLC. The tasks can be adjusted to meet various requirements and are therefore a powerful instrument for structuring the project.

A multitask project can be constructed from:

- A Mastertask (MAST)
  The Mastertask is executed cyclically or periodically. It forms the main section of the program and is executed sequentially.
- A Fasttask (FAST)
  The Fasttask is executed periodically. It has a higher priority than the Mastertask. The Fasttask is used for processes that are executed quickly and periodically.
- One to four Helptasks (AUX)
  The Helptasks are executed periodically. They are used for slow processing and have the lowest priority.

The project can also be constructed with a single task. In this case, only the Mastertask is active.

Event processing

Event processing takes place in event sections. Event sections are executed with higher priority than the sections of all other tasks. They are suited to processing that requires very short reaction times after an event is triggered.

The following section types are available for event processing:

- Sections for processing time controlled events (Timerx Section)
- Sections for processing hardware controlled events (Evtx Section)

The following programming languages are supported:

- FBD (Function Block Diagram)
- LD (Ladder Diagram Language)
- IL (Instruction List)
- ST (Structured Text)
Sections

Sections are autonomous program units in which the logic of the project is created. The sections are executed in the order shown in the project browser (structural view). Sections are connected to a task. The same section cannot belong to more than one task at the same time.

The following programming languages are supported:
- FBD (Function Block Diagram)
- LD (Ladder Diagram Language)
- SFC (Sequential Function Chart)
- IL (Instruction List)
- ST (Structured Text)

Subroutine

Subroutines are created as separate units in subroutine sections. Subroutines are called from sections or from another subroutine. Nesting of up to 8 levels is possible. A subroutine cannot call itself (not recursive). Subroutines are assigned a task. The same subroutine cannot be called by different tasks.

The following programming languages are supported:
- FBD (Function Block Diagram)
- LD (Ladder Diagram Language)
- IL (Instruction List)
- ST (Structured Text)
Function Block Diagram FBD

Introduction

The FBD editor is used for graphical function block programming according to IEC 61131-3.

Representation

Representation of an FBD section:

Objects

The objects of the FBD (Function Block Diagram) programming language help to divide a section into a number of:

- Elementary Functions (EFs),
- Elementary Function Blocks (EFBs)
- Derived Function Blocks (DFBs)
- Procedures
- Subroutine calls
- Jumps
- Links
- Actual Parameters
- Text objects to comment on the logic
**Properties**

FBD sections have a grid behind them. A grid unit consists of 10 coordinates. A grid unit is the smallest possible space between 2 objects in an FBD section.

The FBD programming language is not cell oriented but the objects are still aligned with the grid coordinates.

An FBD section has 300 horizontal grid coordinates (=30 grid units) and 230 vertical grid coordinates (=23 grid units).

The program can be entered using the mouse or the keyboard.

---

**Input aids**

The FBD editor offers the following input aids:

- Toolbars for quick and easy access to the desired objects
- Syntax and semantics are checked as the program is being written.
  - Incorrect functions and function blocks are displayed in blue
  - Unknown words (e.g. undeclared variables) or unsuitable data types are marked with a red wavy line
  - Brief description of errors in the Quickinfo (Tooltip)
- Tabular display of FFBs
- Actual parameters can be entered and displayed as symbols or topological addresses
- Different zoom factors
- Tracking of links
- Optimization of link routes
- Display of inspection windows
Ladder Diagram (LD) language

Introduction
The LD editor is used for graphical ladder diagram programming according to IEC 61131-3.

Representation
Representation of an LD section:

```
Variable

Topological Addresses

Function

Right Power Rail

XOR

%IX1.5.6

%IX1.5.6

%IX1.5.6

%IX1.5.5

Execution number

Function

Inspection window

Error

%QX4.2.3

Comment

Error

%QX2

Variable

%Q2

%Q3

%QX1.5.6

%QX1.5.5

Contact

Error

VarX

Ein1 Undefined Symbol 'Varx

ToolTip

Left Power Rail

IN1 OUT

IN2
```
Objects

The objects of the LD programming language help to divide a section into a number of:
- Contacts,
- Coils,
- Elementary Functions (EFs)
- Elementary Function Blocks (EFBs),
- Derived Function Blocks (DFBs)
- Procedures
- Control elements
- Operation and compare blocks which represent an extension to IEC 61131-3
- Subroutine calls
- Jumps
- Links
- Actual Parameters
- Text objects to comment on the logic

Properties

LD sections have a background grid that divides the section into lines and columns.
The LD programming language is cell oriented, i.e. only one object can be placed in each cell.
LD sections can be 11-64 columns and 17-2000 lines in size.
The program can be entered using the mouse or the keyboard.

Input aids

The LD editor offers the following input aids:
- Objects can be selected from the toolbar, the menu or directly using shortcut keys
- Syntax and semantics are checked as the program is being written.
  - Incorrect objects are displayed in blue
  - Unknown words (e.g. undeclared variables) or unsuitable data types are marked with a red wavy line
  - Brief description of errors in the Quickinfo (Tooltip)
- Tabular display of FFBs
- Actual parameters can be entered and displayed as symbols or topological addresses
- Different zoom factors
- Tracking of FFB links
- Optimizing the link routes of FFB links
- Display of inspection windows
# General information about SFC sequence language

## Introduction

The sequence language SFC (Sequential Function Chart), which conforms to IEC 61131-3, is described in this section.

IEC conformity restrictions can be lifted through explicit enable procedures. Features such as multi token, multiple initial steps, jumps to and from parallel strings etc. are then possible.
Representation

Representation of an SFC section:

- **Link**: Connections between steps and sections
- **Initial Step**: Starting point
- **Parallel branch**: Branches that occur simultaneously
- **Step**: Individual steps in the sequence
- **Alternative branch**: Branches that occur when a condition is met
- **Jump**: Transition to a specific step
- **Alternative joint**: Condition-based connection
- **Parallel joint**: Parallel connection
- **Transition condition**: Conditions for branch or joint selection
- **Return Var**: Return to a previous step
- **Macro Step**: Larger block of steps
- **Undefined Symbol 'VarX'**: Error message indicating an undefined symbol
**Objects**

An SFC section provides the following objects for creating a program:
- Steps
- Macro steps (embedded sub-step sequences)
- Transitions (transition conditions)
- Transition sections
- Action sections
- Jumps
- Links
- Alternative sequences
- Parallel sequences
- Text objects to comment on the logic

**Input aids**

The SFC editor offers the following input aids:
- Toolbars for quick and easy access to the desired objects
- Automatic step numbering
- Direct access to actions and transition conditions
- Syntax and semantics are checked as the program is being written.
  - Incorrect objects are displayed in blue
  - Unknown words (e.g. undeclared variables) or unsuitable data types are marked with a red wavy line
  - Brief description of errors in the Quickinfo (Tooltip)
- Different zoom factors
- Show/hide the allocated actions
- Tracking of links
- Optimization of link routes
Step properties

The step properties are defined using a dialog box that offers the following features:

- Definition of initial steps
- Definition of diagnostics times
- Step comments
- Allocation of actions and their identifiers
**Instruction List IL**

**Introduction**

The IL editor is used for instruction list programming according to IEC 61131-3.

**Representation**

Representation of an IL section:

<table>
<thead>
<tr>
<th>Label</th>
<th>Operators</th>
<th>Operands</th>
<th>Comments</th>
<th>Inspection window</th>
</tr>
</thead>
<tbody>
<tr>
<td>START:</td>
<td>LD</td>
<td>VarA</td>
<td>(* Key 1 *)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ANDN</td>
<td>VarB</td>
<td>(* and not key 2 *)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ST</td>
<td>VarC</td>
<td>(* Ventilator On *)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>LD</td>
<td>VarX</td>
<td>(* Undeclared Variable *)</td>
<td>E1061 Undefined Symbol 'VarX</td>
</tr>
</tbody>
</table>

**Objects**

An instruction list is composed of a series of instructions. Each instruction begins on a new line and consists of:

- An operator
- A modifier if required
- One or more operands if required
- A label as a jump target if required
- A comment about the logic if required.

**Input aids**

The IL editor offers the following input aids:

- Syntax and semantics are checked as the program is being written.
- Keywords and comments are displayed in color
- Unknown words (e.g. undeclared variables) or unsuitable data types are marked with a red wavy line
- Brief description of errors in the Quickinfo (Tooltip)
- Tabular display of the functions and function blocks
- Input assistance for functions and function blocks
- Operands can be entered and displayed as symbols or topological addresses
- Display of inspection windows
Structured Text ST

Introduction
The ST editor is used for programming in structured text according to IEC 61131-3.

Representation
Representation of an ST section:

Objects
The ST programming language works with "Expressions". Expressions are constructions consisting of operators and operands that return a value when executed.

Operators are symbols representing the operations to be executed.

Operators are used for operands. Operands are variables, literals, function and function block inputs/outputs etc.

Instructions are used to structure and control the expressions.

Input aids
The ST editor offers the following input aids:

- Syntax and semantics are checked as the program is being written.
- Keywords and comments are displayed in color
- Unknown words (e.g. undeclared variables) or unsuitable data types are marked with a red wavy line
- Brief description of errors in the Quickinfo (Tooltip)
- Tabular display of the functions and function blocks
- Input assistance for functions and function blocks
- Operands can be entered and displayed as symbols or topological addresses
- Display of inspection windows
PLC Simulator

Introduction

The PLC simulator enables error searches to be carried out in the project without being connected to a real PLC.

All project tasks (Mast, Fast, AUX and Event) that run on a real PLC are also available in the Simulator. The difference from a real PLC is the lack of I/O modules and communication networks (such as e.g. ETHWAY, Fipio and Modbus Plus) non-deterministic realtime behavior.

Naturally, all debugging functions, animation functions, breakpoints, forcing variables etc. are available with the PLC simulator.

Representation

Representation of a dialog box:

![Dialog Box Image]

Structure of the Simulator

The simulator controller offers the following views:

- Type of simulated PLC
- Current status of the simulated PLC
- Name of the loaded project
- IP address and DNS name of the host PC for the simulator and all connected Client PCs
- Dialog box for simulating I/O events
- Reset button to reset the simulated PLC (simulated cold restart)
- Power Off/On button to simulate a warm restart
- Shortcut menu (right mouse button) for controlling the Simulator
Export/Import

Introduction
The export and import functions allow you to use existing data in a new project. The XML export/import format makes it possible to provide or accept data from external software.

Export
The following objects can be exported:
- Complete projects, including configuration
- Sections of all programming languages
- Subroutine sections of all programming languages
- Derived function blocks (DFBs)
- Derived data types (DDTs)
- Variable declarations
- Operator Screen
- Configuration (in FNES format)

Import
All objects that can be exported can naturally be imported as well.

There are two types of import:
- **Direct import**
  Imports the object exactly as it was exported.
- **Import with the assistant**
  The assistant allows you to change the variables names, sections or functional modules. The mapping of addresses can also be modified.
User Documentation

Scope of the user documentation:

The following are just some of the services provided for documenting the project:

- Print the entire project (2) or in sections (3)
- Selection between structural and functional view (1)
- Adjustment of the result (footer, general information, etc.)
- Local printing for programming language editors, configurator, etc.
- Special indication (bold) for keywords
- Paper format can be selected
- Print preview (4)
- Documentation save
### Debug Services

#### Searching for errors in the user application

The following are just some of the features provided to optimize debugging in the project:

- Set breakpoints in the programming language editors
- Step by step program execution, including step into, step out and step over
- Call memory for recalling the entire program path
- Control inputs and outputs

#### Online mode

Online mode is when a connection is established between the PC and the PLC. Online mode is used on the PLC for debugging, for animation and for changing the program.

A comparison between the project of the PC and project of the PLC takes place automatically when the connection is established.

This comparison can produce the following results:

- **Different projects on the PC and the PLC**
  
  In this case, online mode is restricted. Only PLC control commands (e.g. start, stop), diagnostic services and variable monitoring are possible. Changes cannot be made to the PLC program logic or configuration. However, the downloading and uploading functions are possible and run in an unrestricted mode (same project on PC and PLC).

- **Same projects on the PC and the PLC**
  
  There are two different possibilities:

  - **ONLINE SAME, GENERATED**
    
    The last project generation on the PC was downloaded to the PLC and no changes were made afterwards, i.e. the projects on the PC and the PLC are absolutely identical.
    
    In this case, all animation functions are available and unrestricted.

  - **ONLINE EQUAL, NOT GENERATED**
    
    The last project generation on the PC was downloaded to the PLC, however changes were made afterwards.
    
    In this case, the animation functions are only available in the unchanged project components.
Different possibilities are provided for the animation of variables:

- **Section animation**
  
  All programming languages (FBD, LD, SFC, IL and ST) can be animated. The variables and connections are animated directly in the section.

```st
TIMER(IN := NOT pulse;
  := #1s; (* Blink timer *)
pulse := TIMER.Q;
(* Count every pulse *)
IF pulse = 1 THEN
  + 1;
END_IF;
(* Animate lights according to counter *)
CASE OF 1: out1 := TRUE;
  2: out2 := TRUE;
ELSE (* All lights are on, switch then off again and start from
      out1 := FALSE;
      out2 := FALSE;
```

```fbd
AND ENO IN1 TRUE
  \( \text{AND} \)
  \( \text{ENO} \)
  \( \text{IN1} \)
  \( \text{TRUE} \)
  \( \%\text{IX1.5.4} \)
  \( \%\text{IX1.5.5} \)
```

```fbd```

```st```

```fbd```
• **Tooltips**

A tooltip with the value of a variable is displayed when the mouse pointer passes over that variable.

```plaintext
TIMER(IN := NOT pulse;
    := t#1s; (* Blink timer *)
pulse := TIMER.Q;

(* Count every pulse *)
IF pulse = 1 THEN
; END_IF;

(* Animate lights according to counter *)
CASE OF
  1: out1 := TRUE;
  2: out2 := TRUE;
ELSE (* All lights are on, switch then off again and start from
     out1 := FALSE;
     out2 := FALSE;
```

• **Inspection window**

An inspection window can be created for any variable. This window displays the value of the variable, the address and any comments (if available). This function is available in all programming languages.

```plaintext
TIMER(IN := NOT pulse;
    := t#1s; (* Blink timer *)
pulse := TIMER.Q;

(* Count every pulse *)
IF pulse = 1 THEN + 1;
END_IF;

(* Animate lights according to counter *)
CASE OF
  1: out1 := TRUE;
  2: out2 := TRUE;
ELSE (* All lights are on, switch then off again and start from
     out1 := FALSE;
     out2 := FALSE;
```

• **Variables window**

This window displays all variables used in the current section.
● **Animation table**

The value of all variables in the project can be displayed, changed or forced in animation tables. Values can be changed individually or simultaneously together.
Watch point

Watch points allow you to view PLC data at the exact moment at which it is created (1) and not only at the end of a cycle.

Animation tables can be synchronized with the watch point (2).

A counter (3) determines how often the watch point has been updated.

ST section with watch point:

```
ST (Section): My_ST [MAST]
if pump_1.start
  then pump_1.cmd: = true;
  else pump_1.cmd: = false; pump_1.speed: = 0;
end_if;
if pump_1.cmd then pump_1.speed: = pump_1.speed + 1; end_if;
if pump_1.speed > 100 then pump_1.speed: = 100; end_if;
(* animation drilling & threadint *)
high_anim: = not jack_1_out and not jack_3_out;
```
Breakpoints allow you to stop processing of the project at any point.

ST section with breakpoint:

```
TIMER(IN := NOT pulse;
  tr1 := t1/1s; (* Blink timer *)
  pulse := TIMER.Q;

  (* Count every pulse *)
  IF pulse = 1 THEN
    count := count + 1;
  END_IF;

  (* Animate lights according to counter *)
  CASE count OF
    1: out1 := TRUE;
    2: out2 := TRUE;
    ELSE (* All lights are on, switch then off again and start from
    out1 := FALSE;
    out2 := FALSE;
```

```
```
**Single step mode**

Single step mode allows you to execute the program step by step. Single step functions are provided if the project was stopped by reaching a breakpoint or if it is already in single step mode.

ST section in single step mode:

The following functions are provided in single step mode:
- Step by step execution of the program
- StepIn (1)
- StepOut
- StepOver
- Show Current Step (2)
- Call memory (3)

When the "step into" function is executed several times, the call memory enables the display of the entire path, starting with the first breakpoint.

**Bookmarks**

Bookmarks allow you to select code sections and easily find them again.
Diagnostic Viewer

**Description**

Unity Pro provides system and project diagnostics. Errors which occur are displayed in a diagnostics window. The section which caused the error can be opened directly from the diagnostics window in order to correct the error.

![Diagnostic Viewer Table]

- **System alarm**: Buffer battery error
- **Faulty device rack**: 0
- **Faulty device slot**: 0

![Diagnostic Viewer Table Details]

- **Acknowledged Buffer battery error**
- **Deleted Buffer battery error**
- **Deleted Index overflow**
Operator Screen

**Introduction**

Operator windows visualize the automation process.

The operator screen editor makes it easy to create, change and manage operator screens.

Operator screens are created and accessed via the project browser.
Operator screen editor

An operator window contains much information (dynamic variables, overviews, written text, etc.) and makes it easy to monitor and change automation variables.

Operator Screen

The operator screen editor offers the following features:

- Extensive visualization functions
- Geometric elements
  - Line, rectangle, ellipse, curve, polygon, bitmap, text
- Control elements
  - Buttons, control box, shifter, screen navigation, hyperlinks, input field, rotating field
- Animation elements
  - Bar chart, trend diagram, dialog, date, disappear, blinking colors, variable animation
- Create a library for managing graphical objects
- Copying objects
- Creating a list of all variables used in the operator screen
- Creating messages to be used in the operator screen
- Direct access from the operator screen to the animation table or the cross reference table for one or more variables
- Tooltips give additional information about the variables
- Managing operator screens in families
- Import/export of individual operator screens or entire families
Application structure

At a Glance

In This Chapter
This section describes the application program and memory structures associated with each type of PLC.

What's in this Part?
This part contains the following chapters:

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Chapter Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Description of the available functions for each type of PLC</td>
<td>67</td>
</tr>
<tr>
<td>3</td>
<td>Application program structure</td>
<td>69</td>
</tr>
<tr>
<td>4</td>
<td>Application memory structure</td>
<td>109</td>
</tr>
<tr>
<td>5</td>
<td>Operating modes</td>
<td>117</td>
</tr>
<tr>
<td>6</td>
<td>System objects</td>
<td>129</td>
</tr>
</tbody>
</table>
## Functionalities available for the different types of PLC

### Programming languages

<table>
<thead>
<tr>
<th>Available languages</th>
<th>Platforms</th>
<th>Premium: TSX</th>
<th>Atrium: TSX</th>
<th>Quantum: 140 CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>P57 1**</td>
<td>P57 2**</td>
<td>P57 3**</td>
<td>PCI 57 204/454</td>
</tr>
<tr>
<td></td>
<td>P57 4**</td>
<td>PCI 57 204/454</td>
<td>PCI 57 204/454</td>
<td>651**</td>
</tr>
<tr>
<td></td>
<td>P57 5**</td>
<td>PCI 57 204/454</td>
<td>PCI 57 204/454</td>
<td>671**</td>
</tr>
<tr>
<td>LD language</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>FBD language</td>
<td>-</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>ST language</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>IL language</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>SFC language</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

**Legend**

- X Available language.
- - Unavailable language.
## PLC functions

### Tasks and processing

<table>
<thead>
<tr>
<th>Platforms</th>
<th>Premium: TSX</th>
<th>Atrium: TSX</th>
<th>Quantum: 140 CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processors</td>
<td>P57 1••</td>
<td>P57 2••</td>
<td>P57 5••</td>
</tr>
<tr>
<td></td>
<td>P57 3••</td>
<td>P57 4••</td>
<td>PCI 57 204/454</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>31••</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>43••</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>53••</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>651••</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>671••</td>
</tr>
<tr>
<td>Master task</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>cyclic or</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>periodic</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Fast task</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>periodic</td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Auxiliary</td>
<td>-</td>
<td>-</td>
<td>4</td>
</tr>
<tr>
<td>tasks</td>
<td></td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>periodic</td>
<td></td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>I/O type</td>
<td>32</td>
<td>64</td>
<td>128</td>
</tr>
<tr>
<td>event</td>
<td>64</td>
<td>64</td>
<td>128</td>
</tr>
<tr>
<td>processing</td>
<td>64</td>
<td>64</td>
<td>128</td>
</tr>
<tr>
<td>Timer type</td>
<td>-</td>
<td>-</td>
<td>32</td>
</tr>
<tr>
<td>event</td>
<td></td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>processing</td>
<td></td>
<td></td>
<td>16</td>
</tr>
<tr>
<td>Total of I/O</td>
<td>32</td>
<td>64</td>
<td>128</td>
</tr>
<tr>
<td>type and</td>
<td>64</td>
<td>64</td>
<td>128</td>
</tr>
<tr>
<td>Timer type</td>
<td>64</td>
<td>64</td>
<td>128</td>
</tr>
<tr>
<td>event</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>processing</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Legend**

- Available tasks or processes. The value gives the maximum number of tasks or processing.
- Unavailable tasks or processing.
This chapter describes the structure and execution of the programs created using the Unity Pro software.

This chapter contains the following sections:

<table>
<thead>
<tr>
<th>Section</th>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Description of tasks and processes</td>
<td>70</td>
</tr>
<tr>
<td>3.2</td>
<td>Description of sections and subroutines</td>
<td>76</td>
</tr>
<tr>
<td>3.3</td>
<td>Mono task execution</td>
<td>81</td>
</tr>
<tr>
<td>3.4</td>
<td>Multitasking execution</td>
<td>88</td>
</tr>
</tbody>
</table>
### 3.1 Description of tasks and processes

#### At a Glance

**Subject of this Section**
This section describes the tasks and processes that comprise the application program.

**What's in this Section?**
This section contains the following topics:

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Presentation of the master task</td>
<td>71</td>
</tr>
<tr>
<td>Presentation of the fast task</td>
<td>72</td>
</tr>
<tr>
<td>Presentation of auxiliary tasks</td>
<td>73</td>
</tr>
<tr>
<td>Overview of event processing</td>
<td>75</td>
</tr>
</tbody>
</table>
Presentation of the master task

General
The master task represents the main task of the application program. It is obligatory and created by default.

Structure
The master task (MAST) is made up of sections and subroutines.
Each section of the master task is programmed in the following languages: LD, FBD, IL, ST or SFC.
The subroutines are programmed in LD, FBD, IL, or ST and are called in the task sections.

Note: SFC can be used only in the master task sections. The number of sections programmed in SFC is unlimited.

Execution
You can choose the type of master task execution:
- cyclic (default selection)
- or periodic (1 to 255ms)

Note: The auxiliary tasks are not allowed if the master task is cyclic.

Control
The master task can be controlled by program, by bits and system words.

<table>
<thead>
<tr>
<th>System objects</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>%SW0</td>
<td>Task period.</td>
</tr>
<tr>
<td>%S30</td>
<td>Master task activation</td>
</tr>
<tr>
<td>%S11</td>
<td>Watchdog error</td>
</tr>
<tr>
<td>%S19</td>
<td>Period overrun.</td>
</tr>
<tr>
<td>%SW30</td>
<td>Execution time (in ms) of the last cycle.</td>
</tr>
<tr>
<td>%SW31</td>
<td>Execution time (in ms) of the longest cycle.</td>
</tr>
<tr>
<td>%SW32</td>
<td>Execution time (in ms) of the shortest cycle.</td>
</tr>
</tbody>
</table>
Presentation of the fast task

General
The fast task is intended for short duration and periodic processing tasks.

Structure
The fast task (FAST) is made up of sections and subroutines.
Each section of the fast task is programmed in one of the following languages: LD, FBD, IL or ST.
SFC language cannot be used in the sections of a fast task.
Subroutines are programmed in LD, FBD, IL, or ST language and are called in the task sections.

Execution
The execution of the fast task is periodic.
It is higher priority than the master task.
The period of the fast task (FAST) is fixed by configuration, from 1 to 255ms.
The executed program must however remain short to avoid the overflow of lower-priority tasks.

Control
The fast task can be controlled by program by bits and system words.

<table>
<thead>
<tr>
<th>System objects</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>%SW1</td>
<td>Task period.</td>
</tr>
<tr>
<td>%S31</td>
<td>Fast task activation.</td>
</tr>
<tr>
<td>%S11</td>
<td>Watchdog error</td>
</tr>
<tr>
<td>%S19</td>
<td>Period overrun.</td>
</tr>
<tr>
<td>%SW33</td>
<td>Execution time (in ms) of the last cycle.</td>
</tr>
<tr>
<td>%SW34</td>
<td>Execution time (in ms) of the longest cycle.</td>
</tr>
<tr>
<td>%SW35</td>
<td>Execution time (in ms) of the shortest cycle.</td>
</tr>
</tbody>
</table>
Presentation of auxiliary tasks

General
The auxiliary tasks are intended for slower processing tasks. These are the least priority tasks.

It is possible to program up to 4 auxiliary tasks (AUX0, AUX1, AUX2 or AUX3) on the Premium TSX P57 5•• and Quantum 140 CPU 6•••• PLCs.

Structure
The auxiliary tasks (AUX) are made up of sections and subroutines.

Each section of the auxiliary task is programmed in one of the following languages: LD, FBD, IL or ST.

The SFC language is not usable in the sections of an auxiliary task.

A maximum of 64 subroutines can be programmed in the LD, FBD, IL or ST language. These are called in the task sections.

Execution
The execution of auxiliary tasks is periodic.

They are the least priority.

The auxiliary task period can be fixed from 10ms to 2.55s.
The auxiliary tasks can be controlled by the program by bits and system words.

<table>
<thead>
<tr>
<th>System objects</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>%SW2</td>
<td>Period of auxiliary task 0.</td>
</tr>
<tr>
<td>%SW3</td>
<td>Period of auxiliary task 1.</td>
</tr>
<tr>
<td>%SW4</td>
<td>Period of auxiliary task 2.</td>
</tr>
<tr>
<td>%SW5</td>
<td>Period of auxiliary task 3.</td>
</tr>
<tr>
<td>%S32</td>
<td>Activation of auxiliary task 0.</td>
</tr>
<tr>
<td>%S33</td>
<td>Activation of auxiliary task 1.</td>
</tr>
<tr>
<td>%S34</td>
<td>Activation of auxiliary task 2.</td>
</tr>
<tr>
<td>%S35</td>
<td>Activation of auxiliary task 3.</td>
</tr>
<tr>
<td>%S11</td>
<td>Watchdog error</td>
</tr>
<tr>
<td>%S19</td>
<td>Period overrun</td>
</tr>
<tr>
<td>%SW36</td>
<td>Execution time (in ms) of the last cycle of auxiliary task 0.</td>
</tr>
<tr>
<td>%SW39</td>
<td>Execution time (in ms) of the last cycle of auxiliary task 1.</td>
</tr>
<tr>
<td>%SW42</td>
<td>Execution time (in ms) of the last cycle of auxiliary task 2.</td>
</tr>
<tr>
<td>%SW45</td>
<td>Execution time (in ms) of the last cycle of auxiliary task 3.</td>
</tr>
<tr>
<td>%SW37</td>
<td>Execution time (in ms) of the longest cycle of auxiliary task 0.</td>
</tr>
<tr>
<td>%SW40</td>
<td>Execution time (in ms) of the longest cycle of auxiliary task 1.</td>
</tr>
<tr>
<td>%SW43</td>
<td>Execution time (in ms) of the longest cycle of auxiliary task 2.</td>
</tr>
<tr>
<td>%SW46</td>
<td>Execution time (in ms) of the longest cycle of auxiliary task 3.</td>
</tr>
<tr>
<td>%SW38</td>
<td>Execution time (in ms) of the shortest cycle of auxiliary task 0.</td>
</tr>
<tr>
<td>%SW41</td>
<td>Execution time (in ms) of the shortest cycle of auxiliary task 1.</td>
</tr>
<tr>
<td>%SW44</td>
<td>Execution time (in ms) of the shortest cycle of auxiliary task 2.</td>
</tr>
<tr>
<td>%SW47</td>
<td>Execution time (in ms) of the shortest cycle of auxiliary task 3.</td>
</tr>
</tbody>
</table>
Overview of event processing

General

Event processing is used to reduce the response time of the application program to events:

- coming from input/output modules,
- from event timers.

These processing tasks are performed with priority over all other tasks. They are therefore suited to processing tasks requiring a very short response time in relation to the event.

The number of event processing tasks (See Tasks and processing, p. 66) that can be programmed depends on the type of processor.

Structure

An event processing task is monosectional, and made up of a single (unconditioned) section.

It is programmed in either LD, FBD, IL or ST language.

Two types of event are offered:

- EVT{i}: for events coming from input/output modules
- TIMER{i}: for events coming from event timers.

Execution

The execution of an event processing task is asynchronous.

The occurrence of an event reroutes the application program to the processing task associated with the input/output channel or event timer which caused the event.

Control

The following system bits and words can be used to control event processing tasks during the execution of the program.

<table>
<thead>
<tr>
<th>System objects</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>%S38</td>
<td>Activation of event processing.</td>
</tr>
<tr>
<td>%S39</td>
<td>Saturation of the event call management stack.</td>
</tr>
<tr>
<td>%SW48</td>
<td>Number of event processing tasks executed.</td>
</tr>
</tbody>
</table>
3.2 Description of sections and subroutines

At a Glance

Aim of this Section
This section describes the sections and the subroutines that make up a task.

What's in this Section?
This section contains the following topics:

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description of sections</td>
<td>77</td>
</tr>
<tr>
<td>Description of SFC sections</td>
<td>79</td>
</tr>
<tr>
<td>Description of subroutines</td>
<td>80</td>
</tr>
</tbody>
</table>
Description of sections

Overview of the sections

Sections are autonomous programming entities. The identification tags of the instruction lines, the contact networks, etc. are specific to each section (no program jump to another section is possible).

These are programmed either in:

- Ladder language (LD),
- Functional block language (FBD),
- Instruction List (IL),
- Structured Text (ST) or
- Sequential Function Charting (SFC).

on condition that the language is accepted in the task.

The sections are executed in the order of their programming in the browser window (structure view).

An execution condition can be associated with one or more sections in the master, fast and auxiliary tasks, but not in the event processing tasks.

The sections are linked to a task. The same section cannot belong simultaneously to several tasks.

Example

The following diagram shows a task structured into sections.

```
- MAST
  Sections
    Airlock
    Oven_1
    Dryer
  SR Sections
```
The following table describes the characteristics of a section.

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>32 characters maximum (accents are possible, but spaces are not allowed).</td>
</tr>
<tr>
<td>Language</td>
<td>LD, FBD, IL, ST or SFC.</td>
</tr>
<tr>
<td>Task or processing</td>
<td>Master, fast, auxiliary, event.</td>
</tr>
<tr>
<td>Condition (optional)</td>
<td>A BOOL or EBOOL type bit variable can be used to condition the execution of the section.</td>
</tr>
<tr>
<td>Comment</td>
<td>256 characters maximum.</td>
</tr>
<tr>
<td>Protection</td>
<td>Write-protection, read/write protection.</td>
</tr>
</tbody>
</table>
Description of SFC sections

General
The sections in Sequential Function Chart language are made up of:
- a main chart programmed in SFC,
- macro steps (MS) programmed in SFC,
- actions and transitions programmed in LD, FBD, ST, or IL.

The SFC sections are programmable only in the master task (see detailed description of SFC sections)

Example
The following diagram gives an example of the structure of an SFC section, and uses the chart to show the macro step calls that are used.

```
  - MAST
    - Sections Folder
      - Airlock
      - Oven_1
    - Sequential processing
      - Chart
        - MS_1_1
        - MS_2_1
        - MS_1_2
      - Unused Macro
        - MS_4_1
        - MS_4_2
      - Actions
      - Transitions
    - Dryer
```
Description of subroutines

Overview of subroutines

Subroutines are programmed as separate entities, either in:

- Ladder language (LD),
- Functional block language (FBD),
- Instruction List (IL),
- Structured Text (ST).

The calls to subroutines are carried out in the sections or from another subroutine. The number of nestings is limited to 8. A subroutine cannot call itself (non recursive).

Subroutines are also linked to a task. The same subroutine cannot be called from several different tasks.

Example

The following diagram shows a task structured into sections and subroutines.

```
- MAST
  - Sections
    - SR Sections
      - Control_1
      - Detection
```

Characteristics of a subroutine

The following table describes the characteristics of a subroutine.

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>32 characters maximum (accents are possible, but spaces are not allowed).</td>
</tr>
<tr>
<td>Language</td>
<td>LD, FBD, IL or ST.</td>
</tr>
<tr>
<td>Task</td>
<td>Master, fast or auxiliary.</td>
</tr>
<tr>
<td>Comment</td>
<td>512 characters maximum.</td>
</tr>
</tbody>
</table>
3.3 Mono task execution

At a Glance

Subject of this Section
This section describes how a mono task application operates.

What's in this Section?
This section contains the following topics:

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description of the master task cycle</td>
<td>82</td>
</tr>
<tr>
<td>Mono task: cyclic execution</td>
<td>84</td>
</tr>
<tr>
<td>Periodic execution</td>
<td>85</td>
</tr>
<tr>
<td>Control of cycle time</td>
<td>86</td>
</tr>
<tr>
<td>Execution of Quantum Sections with Remote Inputs/Outputs</td>
<td>87</td>
</tr>
</tbody>
</table>
Description of the master task cycle

General
The program for a mono task application is associated with a single user task, the master task (See Presentation of the master task, p. 71).
You can choose the type of master task execution:
- cyclic or
- periodic

Illustration
The following illustration shows the operating cycle.

```
  Acquisition of inputs
     ↓
  Program processing
     ↓
  Updating of outputs
```

Description of the different phases
The table below describes the operating phases.

<table>
<thead>
<tr>
<th>Phase</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acquisition of inputs</td>
<td>Writing to memory of the status of the data on the inputs of the discrete and application-specific modules associated with the task, These values can be modified by forcing values.</td>
</tr>
<tr>
<td>Program processing</td>
<td>Execution of application program, written by the user.</td>
</tr>
<tr>
<td>Updating of outputs</td>
<td>Writing of output bits or words to the discrete or application-specific modules associated with the task depending on the state defined by the application. As for the inputs, the values written to the outputs can be modified by forcing values.</td>
</tr>
</tbody>
</table>
Operating mode

**PLC in RUN**, the processor carries out internal processing, input acquisition, processing of the application program and the updating of outputs in that order.

**PLC in STOP**, the processor carries out:
- internal processing,
- input acquisition (1),
- and depending on the chosen configuration:
  - fallback mode: the outputs are set to fallback position.
  - maintain mode: the last value of the outputs is maintained.

(1) for Premium and Atrium PLCs; for Quantum PLCs, input acquisition is inhibited when the PLC is in STOP.

**Note:** During the input acquisition and output update phases, the system also implicitly monitors the PLC (management of system bits and words, updating of current values of the real time clock, updating of status LEDs and LCD screens, detection of changes between RUN/STOP, etc.) and the processing of requests from the terminal (modifications and animation).
Mono task: cyclic execution

General
The master task operates as outlined below. A description is provided of cyclic execution of the master task in mono task operation.

Operation
The following drawing shows the execution phases of the PLC cycle.

![Processing the program](diagram)

- %I Reading of inputs
- %Q Writing of outputs

Description
This type of operation consists of sequencing the task cycles, one after another. After having updated the outputs, the system performs its own specific processing then starts another task cycle, without pausing.

Cycle check
The cycle is checked by the watchdog (See Control of cycle time, p. 86).
Periodic execution

Description
In this operating mode, input acquisition, the processing of the application program and the updating of outputs are all carried out periodically over a defined period of 1 to 255 ms.

At the start of the PLC cycle, a time out whose current value is initialized to the defined period starts the countdown.

The PLC cycle must be completed before this time out expires and launches a new cycle.

Operation
The following diagram shows the execution phases of the PLC cycle.

Operating mode
The processor carries out internal processing, input acquisition, processing of the application program and the updating of outputs in that order.

- If the period is not yet over, the processor completes its operating cycle until the end of the period by performing internal processing.
- If the operating time is longer than that assigned to the period, the PLC signals a period overrun by setting the system bit %S19 of the task to 1. Processing then continues and is executed fully (however, it must not exceed the watchdog time limit). The following cycle is started after the outputs have been implicitly written for the current cycle.

Cycle check
Two checks are carried out:
- period overrun (See Control of cycle time, p. 86),
- by watchdog (See Control of cycle time, p. 86).
## Control of cycle time

### General

The period of master task execution, in cyclic or periodic operation, is controlled by the PLC (watchdog) and must not exceed the value defined in Tmax configuration (250 ms by default, 2.55 s maximum).

### Software watchdog (periodic or cyclic operation)

If watchdog overflow should occur, the application is declared in error, which causes the PLC to stop immediately (HALT state).

The bit %S11 indicates a watchdog overflow. It is set to 1 by the system when the cycle time becomes greater than the watchdog.

The word %SW11 contains the watchdog value in ms. This value is not modifiable by the program.

**Note:**
- The reactivation of the task requires the terminal to be connected in order to analyze the cause of the error, correct it, reinitialize the PLC and switch it to RUN.
- It is not possible to exit HALT by switching to STOP. To do this you must reinitialize the application to ensure consistency of data.

### Control in periodic operation

In periodic operation, an additional control enables a period overrun to be detected. A period overrun does not cause the PLC to stop if it remains less than the watchdog value.

The bit %S19 indicates a period overflow. It is set to 1 by the system, when the cycle time becomes greater than the task period.

The word %SW0 contains the value of the period (in ms). It is initialized on cold restart by the defined value. It can be changed by the user.

### Exploitation of master task execution times

The following system words can be used to obtain information on the cycle time:
- %SW30 contains the execution time of the last cycle,
- %SW31 contains the execution time of the longest cycle,
- %SW32 contains the execution time of the shortest cycle.

**Note:** These different items of information can also be accessed explicitly from the configuration editor.
Execution of Quantum Sections with Remote Inputs/Outputs

General
Quantum PLCs have a specific section management system. It applies to stations with remote inputs/outputs.
This system allows remote inputs/outputs to be updated on sections with optimum response times (without having to wait for the entire task cycle time before updating the inputs/outputs).

Operation
The following diagram shows the execution phases of the sections for an example of a task with 3 sections associated with 3 remote input/output stations.

%Ii Reading of inputs of station No. i
%Qi Reading of outputs of station No. i

<table>
<thead>
<tr>
<th>Phase</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Reading the inputs of the remote input/output station associated with section No. 1.</td>
</tr>
<tr>
<td>2</td>
<td>Processing the program of section 1</td>
</tr>
<tr>
<td>3</td>
<td>Reading the outputs of the remote input/output station associated with section No. 1.</td>
</tr>
<tr>
<td>4</td>
<td>The cycle continues in the same way for the following sections and stations.</td>
</tr>
</tbody>
</table>
3.4 Multitasking execution

At a Glance

Subject of this Section
This section describes how a multitasking application operates.

What's in this Section?
This section contains the following topics:

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multitasking software structure</td>
<td>89</td>
</tr>
<tr>
<td>Sequencing of tasks in a multitasking structure</td>
<td>91</td>
</tr>
<tr>
<td>Task control</td>
<td>93</td>
</tr>
<tr>
<td>Assignment of input/output channels to master, fast and auxiliary tasks</td>
<td>97</td>
</tr>
<tr>
<td>Management of event processing</td>
<td>99</td>
</tr>
<tr>
<td>Execution of TIMER-type event processing</td>
<td>101</td>
</tr>
<tr>
<td>Input/Output exchanges in event processing</td>
<td>105</td>
</tr>
<tr>
<td>How to program event processing</td>
<td>106</td>
</tr>
</tbody>
</table>
Multitasking software structure

Tasks and processing

The task structure of this type of application is as follows:

<table>
<thead>
<tr>
<th>Task/Processing</th>
<th>Designation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Master</td>
<td>MAST</td>
<td>Always present, may be cyclic or periodic.</td>
</tr>
<tr>
<td>Fast</td>
<td>FAST</td>
<td>Optional, always periodic.</td>
</tr>
<tr>
<td>Auxiliary</td>
<td>AUX 0 to 3</td>
<td>Optional and always periodic.</td>
</tr>
<tr>
<td>Event</td>
<td>EVT i and TIMER i (See General, p. 99)</td>
<td>Called by the system when an event occurs on an input/output module or triggered by the event timer. These types of processing are optional and can be used by applications that need to act on inputs/outputs within a short response time.</td>
</tr>
</tbody>
</table>

Illustration

The following diagram shows the tasks in a multitasking structure and their level of priority.

[Diagram showing multitasking tasks and priority]

Description

The master (MAST) task is still the application base. The other tasks differ depending on the type of PLC (See Tasks and processing, p. 68).

Levels of priority are fixed for each task in order to prioritize certain types of processing.

Event processing can be activated asynchronously with respect to periodic tasks by an order generated by external events. It is processed as a priority and requires any processing in progress to be stopped.
Precautions

Multitasks: golden rules

CAUTION

Multitasks: golden rules
The sharing of Inputs/Outputs between different tasks can lead to unforeseen behavior by the application.
We specifically recommend you associate each output or each input to one task only.
Failure to follow this instruction can result in injury or equipment damage.
Sequencing of tasks in a multitasking structure

General
The master task is active by default.
The fast and auxiliary tasks are active by default if they have been programmed.
Event processing is activated when the associated event occurs.

Operation
The table below describes the execution of priority tasks (this operation is also illustrated in the diagram below).

<table>
<thead>
<tr>
<th>Phase</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Occurrence of an event or start of the fast task cycle.</td>
</tr>
<tr>
<td>2</td>
<td>Execution of lower priority tasks in progress stopped.</td>
</tr>
<tr>
<td>3</td>
<td>Execution of the priority task.</td>
</tr>
<tr>
<td>4</td>
<td>The interrupted task takes over again when processing of the priority task is complete.</td>
</tr>
</tbody>
</table>

Description of the task sequence
The following diagram illustrates the task sequence of multitasking processing with a cyclic master task, a fast task with a 20ms period and event processing.

Legend:
I: acquisition of inputs
P: program processing
O: updating of outputs
Task control

The execution of fast and event processing tasks can be controlled by the program using the following system bits:

- %S30 is used to control whether or not the MAST master task is active.
- %S31 is used to control whether or not the FAST task is active.
- %S32 to %S35 are used to control whether or not the auxiliary tasks AUX0 to AUX3 are active.
- %S38 is used to control whether EVTi event processing is active.

Note: The elementary functions MASKEVT and UNMASKEVT also allow the global masking and unmasking of events by the program.
Task control

Cyclic and periodic operation

In multitasking operation, the highest priority task shall be used in periodic mode in order to allow enough time for lower priority tasks to be executed. For this reason, only the task with the lowest priority should be used in cyclic mode. Thus, choosing cyclic operating mode for the master task excludes using auxiliary tasks.

Measurement of task durations

The duration of tasks is continually measured. This measurement represents the duration between the start and the end of execution of the task. This measurement includes the time taken up by tasks of higher priority which may interrupt the execution of the task being measured. The following system words give the current, maximum and minimum cycle times for each task (value in ms)

<table>
<thead>
<tr>
<th>Measurement of times</th>
<th>MAST</th>
<th>FAST</th>
<th>AUX0</th>
<th>AUX1</th>
<th>AUX2</th>
<th>AUX3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Current</td>
<td>%SW30</td>
<td>%SW33</td>
<td>%SW36</td>
<td>%SW39</td>
<td>%SW42</td>
<td>%SW45</td>
</tr>
<tr>
<td>Maximum</td>
<td>%SW31</td>
<td>%SW34</td>
<td>%SW37</td>
<td>%SW40</td>
<td>%SW43</td>
<td>%SW46</td>
</tr>
<tr>
<td>Minimum</td>
<td>%SW32</td>
<td>%SW35</td>
<td>%SW38</td>
<td>%SW41</td>
<td>%SW44</td>
<td>%SW47</td>
</tr>
</tbody>
</table>

Note: The maximum and minimum times are taken from the times measured since the last cold restart.
The task periods are defined in the task properties. They can be modified by the following system words.

<table>
<thead>
<tr>
<th>System words</th>
<th>Task</th>
<th>Values</th>
<th>Default values</th>
<th>Observations</th>
</tr>
</thead>
<tbody>
<tr>
<td>%SW0</td>
<td>MAST</td>
<td>0..255ms</td>
<td>Cyclic</td>
<td>0 = cyclic operation</td>
</tr>
<tr>
<td>%SW1</td>
<td>FAST</td>
<td>1..255ms</td>
<td>5ms</td>
<td></td>
</tr>
<tr>
<td>%SW2</td>
<td>AUX0</td>
<td>10ms..2.55s</td>
<td>100ms</td>
<td>The values of the period are expressed in 10ms.</td>
</tr>
<tr>
<td>%SW3</td>
<td>AUX1</td>
<td>10ms..2.55s</td>
<td>200ms</td>
<td></td>
</tr>
<tr>
<td>%SW4</td>
<td>AUX2</td>
<td>10ms..2.55s</td>
<td>300ms</td>
<td></td>
</tr>
<tr>
<td>%SW5</td>
<td>AUX3</td>
<td>10ms..2.55s</td>
<td>400ms</td>
<td></td>
</tr>
</tbody>
</table>

When the cycle time of the task exceeds the period, the system sets the system bit %S19 of the task to 1 and continues with the following cycle.

**Note:** The values of the periods do not depend on the priority of tasks. It is possible to define the period of a fast task which is larger than the master task.
Watchdog

The execution of each task is controlled by a configurable watchdog by using the task properties.

The following table gives the range of watchdog values for each of the tasks:

<table>
<thead>
<tr>
<th>Tasks</th>
<th>Watchdog values (min...max) (ms)</th>
<th>Default watchdog value (ms)</th>
<th>Associated system word</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAST</td>
<td>10..1500</td>
<td>250</td>
<td>%SW11</td>
</tr>
<tr>
<td>FAST</td>
<td>10..500</td>
<td>100</td>
<td>-</td>
</tr>
<tr>
<td>AUX0</td>
<td>100..5000</td>
<td>2000</td>
<td>-</td>
</tr>
<tr>
<td>AUX1</td>
<td>100..5000</td>
<td>2000</td>
<td>-</td>
</tr>
<tr>
<td>AUX2</td>
<td>100..5000</td>
<td>2000</td>
<td>-</td>
</tr>
<tr>
<td>AUX3</td>
<td>100..5000</td>
<td>2000</td>
<td>-</td>
</tr>
</tbody>
</table>

If watchdog overflow should occur, the application is declared in error, which causes the PLC to stop immediately (HALT state).

The word %SW11 contains the watchdog value of the master task in ms. This value is not modifiable by the program.

The bit %S11 indicates a watchdog overflow. It is set to 1 by the system when the cycle time becomes greater than the watchdog.

Note:

- The reactivation of the task requires the terminal to be connected in order to analyze the cause of the error, correct it, reinitialize the PLC and switch it to RUN.
- It is not possible to exit HALT by switching to STOP. To do this you must reinitialize the application to ensure consistency of data.
Task control

When the application program is being executed, it is possible to activate or inhibit a task by using the following system bits:

<table>
<thead>
<tr>
<th>System bits</th>
<th>Task</th>
</tr>
</thead>
<tbody>
<tr>
<td>%S30</td>
<td>MAST</td>
</tr>
<tr>
<td>%S31</td>
<td>FAST</td>
</tr>
<tr>
<td>%S32</td>
<td>AUX0</td>
</tr>
<tr>
<td>%S33</td>
<td>AUX1</td>
</tr>
<tr>
<td>%S34</td>
<td>AUX2</td>
</tr>
<tr>
<td>%S35</td>
<td>AUX3</td>
</tr>
</tbody>
</table>

The task is active when the associated system bit is set to 1. These bits are tested by the system at the end of the master task.

When a task is inhibited, the inputs continue to be read and the outputs continue to be written.

On startup of the application program, for the first execution cycle only the master task is active. At the end of the first cycle the other tasks are automatically activated except if one of the tasks is inhibited (associated system bit set to 0) by the program.

Controls on input reading and output writing phases

The bits of the following system words can be used to inhibit the input reading and output writing phases.

<table>
<thead>
<tr>
<th>Inhibition of phases...</th>
<th>MAST</th>
<th>FAST</th>
<th>AUX0</th>
<th>AUX1</th>
<th>AUX2</th>
<th>AUX3</th>
</tr>
</thead>
<tbody>
<tr>
<td>reading of inputs</td>
<td>%SW8.0</td>
<td>%SW8.1</td>
<td>%SW8.2</td>
<td>%SW8.3</td>
<td>%SW8.4</td>
<td>%SW8.5</td>
</tr>
<tr>
<td>writing of outputs</td>
<td>%SW9.0</td>
<td>%SW9.1</td>
<td>%SW9.2</td>
<td>%SW9.3</td>
<td>%SW9.4</td>
<td>%SW9.5</td>
</tr>
</tbody>
</table>

Note: By default, the input reading and output writing phases are active (bits of system words %SW8 and %SW9 set to 0).
On Quantum, inputs/outputs which are distributed via DIO bus are not assigned by the words %SW8 and %SW9.
Assignment of input/output channels to master, fast and auxiliary tasks

**General**
Each task writes and reads the inputs/outputs assigned to it. The association of a channel, group of channels or an input/output module with a task is defined in the configuration screen of the corresponding module. The task that is associated by default is the MAST task.

**Reading of inputs and writing of outputs on Premium**
All the input/output channels of in-rack modules can be associated with a task (MAST, FAST or AUX 0..3).

Local and remote inputs/outputs (X bus):
For each task cycle, the inputs are read at the start of the task and the outputs are written at the end of the task.

Remote inputs/outputs on Fipio bus:
In controlled mode, the refreshing of inputs/outputs is correlated with the task period. The system guarantees that inputs/outputs are updated in a single period. Only the inputs/outputs associated with this task are refreshed.

In this mode, the period of the PLC task (MAST, FAST or AUX) must be greater than or equal to the network cycle time.

In free mode, no restriction is imposed on the task period. The PLC task period (MAST, FAST or AUX) can be less than the network cycle. If this is the case, the task can be executed without updating the inputs/outputs. Selecting this mode gives you the possibility of having the lowest possible task times for applications where speed is critical.

**Reading of inputs and writing of outputs on Quantum**
Local inputs/outputs:
Each input/output module or group of modules can be associated with a single task (MAST, FAST or AUX 0..3).

Remote inputs/outputs:
Remote input/output stations can only be associated with the master (MAST) task. The assignment is made for sections (See Execution of Quantum Sections with Remote Inputs/Outputs, p. 87), with 1 remote input station and 1 remote output station per section.
**Distributed inputs/outputs:**
Distributed input/output stations can only be associated with the master (MAST) task.
The inputs are read at the start of the master task and the outputs are written at the end of the master task.

**Example on Premium**
With its 8 successive channel modularity (channels 0 to 7, channels 8 to 15, etc.), the inputs/outputs of the Premium discrete modules can be assigned in groups of 8 channels, independently of the MAST, AUXi or FAST task.

**Example:** it is possible to assign the channels of a 28 input/output module as follows:
- inputs 0 to 7 assigned to the MAST task,
- inputs 8 to 15 assigned to the FAST task,
- outputs 0 to 7 assigned to the MAST task,
- outputs 8 to 15 assigned to the AUX0 task.
Management of event processing

General

Event processing take priority over tasks.

The following illustration describes the 3 defined levels of priority:

- EVT0 event processing is the highest priority processing. It can itself interrupt other types of event processing.
- EVTi event processing triggered by input/output modules (priority 1) take priority over TIMERi event processing triggered by timers (priority 2).
- On Premium and Atrium PLCs: types of event processing with priority level 1 are stored and processed in order.
- On Quantum PLC: the priority of priority 1 processing types is determined:
  - by the position of the input/output module in the rack,
  - by the position of the channel in the module. The module with the lowest position number has the highest level of priority.
- Event processing triggered by timer is given priority level 2. The processing priority is determined by the lowest timer number.
Control

The application program can globally validate or inhibit the various types of event processing by using the system bit %S38. If one or more events occur while they are inhibited, the associated processing is lost.

Two elementary functions of the language, \texttt{MASKEVT()} and \texttt{UNMASKEVT()}, used in the application program can also be used to mask or unmask event processing.

If one or more events occur while they are masked, they are stored by the system and the associated processing is carried out after unmasking.
Execution of TIMER-type event processing

**Description**
TIMER-type event processing is any process triggered by the ITCNTRL (See Unity Pro, Systeme Library Triggering of TIMER type event processing) function. This timer function periodically activates event processing every time the preset value is reached.

**Reference**
The following parameters are selected in the event processing properties.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Default value</th>
<th>Role</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time base</td>
<td>1 ms, 10ms, 100ms, 1 sec</td>
<td>10ms</td>
<td>Timer time base. Note: the time base of 1ms should be used with care, as there is a risk of overrun if the processing triggering frequency is too high.</td>
</tr>
<tr>
<td>Preset</td>
<td>1..1023</td>
<td>10</td>
<td>Timer preset value. The time period obtained equals: Preset x Time Base.</td>
</tr>
<tr>
<td>Phase</td>
<td>0..1023</td>
<td>0</td>
<td>The value of the temporal offset between the STOP/RUN transition of the PLC and the first restart of the timer from 0. The temporal value equals: Phase x Time Base.</td>
</tr>
</tbody>
</table>

**ITCNTRL function**
Representation in FBD:

The following table describes the input parameters:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enable</td>
<td>BOOL</td>
<td>Enable input selected</td>
</tr>
<tr>
<td>Reset_Timer</td>
<td>BOOL</td>
<td>At 1 resets the timer</td>
</tr>
<tr>
<td>Hold_Timer</td>
<td>BOOL</td>
<td>At 1, freezes timer incrementation.</td>
</tr>
<tr>
<td>Nb_Task_Event</td>
<td>BYTE</td>
<td>Input byte which determines the event processing number to be triggered.</td>
</tr>
</tbody>
</table>
The following table describes the output parameters:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status_Timer</td>
<td>WORD</td>
<td>Status word.</td>
</tr>
<tr>
<td>Current_Value</td>
<td>TIME</td>
<td>Current value of timer.</td>
</tr>
</tbody>
</table>

### Timing diagram for normal operation

The following table describes the triggering of TIMER-type event processing operations (see timing diagram above).

<table>
<thead>
<tr>
<th>Phase</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>When a rising edge is received on the <strong>RESET</strong> input, the timer is reset to 0.</td>
</tr>
<tr>
<td>2</td>
<td>The current value <strong>VALUE</strong> of the timer increases from 0 towards the preset value at a rate of one unit for each pulse of the time base.</td>
</tr>
<tr>
<td>3</td>
<td>An event is generated when the current value has reached the preset value, the timer is reset to 0, and then reactivated. The associated event processing is also triggered, if the event is not masked. It can be deferred if an event processing task with a higher or identical priority is already in progress.</td>
</tr>
<tr>
<td>4</td>
<td>When the <strong>ENABLE</strong> input is at 0, the events are no longer sent out. TIMER type event processing is no longer triggered.</td>
</tr>
<tr>
<td>5</td>
<td>When the <strong>HOLD</strong> input is at 1, the timer is frozen, and the current value stops incrementing, until this input returns to 0.</td>
</tr>
</tbody>
</table>
**Event processing synchronization**

The Phase parameter is used to trigger different TIMER-type event processing tasks at constant time intervals.

This parameter sets a temporal offset value with an absolute time origin, which is the last passage of the PLC from STOP to RUN.

**Operating condition:**

- The event processing tasks must have the same time base and preset values.
- The **RESET** and **HOLD** inputs must not be set to 1.

**Example:** 2 event processing tasks Timer1 and Timer2 to be executed at 70ms interval.

Timer1 can be defined with a phase equal to 0 and the second Timer2 with a phase of 70ms (phase of 7 and time base of 10ms).

Any event triggered by the timer associated with the Timer1 processing task shall be followed after an interval of 70ms by an event from the timer associated with the Timer2 processing task.

**Timing diagram:**

Timing diagram of the example provided above with the same preset value of 16 (160ms) for Timer1 and Timer2.
The following table describes the operation of the PLC after a transition from STOP into RUN (see timing diagram above):

<table>
<thead>
<tr>
<th>Phase</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ON a STOP RUN transition of the PLC, timing is triggered so that the preset value is reached at the end of a time period equal to Phase x time base.</td>
</tr>
<tr>
<td>2</td>
<td>The current value $VALUE$ of the timer increases from 0 towards the preset value at a rate of one unit for each pulse of the time base.</td>
</tr>
<tr>
<td>3</td>
<td>An event is generated when the current value has reached the preset value, the timer is reset to 0, and then reactivated. The associated event processing is also triggered, if the event is not masked. If can be deferred, if there is an event processing task of higher or identical priority already in progress.</td>
</tr>
</tbody>
</table>
Input/Output exchanges in event processing

General
With each type of event processing it is possible to use other input/output channels than those for the event.
As with tasks, exchanges are then performed implicitly by the system before (%I) and after (%Q) application processing.

Operation
The following table describes the exchanges and processing performed.

<table>
<thead>
<tr>
<th>Phase</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>The occurrence of an event reroutes the application program to perform the processing associated with the input/output channel which caused the event.</td>
</tr>
<tr>
<td>2</td>
<td>All inputs associated with event processing are acquired automatically.</td>
</tr>
<tr>
<td>3</td>
<td>The event processing is executed. It must be as short as possible.</td>
</tr>
<tr>
<td>4</td>
<td>All the outputs associated with the event processing are updated.</td>
</tr>
</tbody>
</table>

Premium/Atrium PLCs
The inputs acquired and the outputs updated are:
- the inputs associated with the channel which caused the event,
- the inputs and outputs used during event processing.

Note: These exchanges may relate:
- to a channel (e.g. counting module),
- or to a group of channels (discrete module). In this case, if the processing modifies, for example, outputs 2 and 3 of a discrete module, the image of outputs 0 to 7 is then transferred to the module.

Quantum PLCs
The inputs acquired and the outputs updated are selected in the configuration. Only local inputs/outputs can be selected.

Programming rule
The inputs (and the associated group of channels) exchanged during the execution of event processing are updated (loss of historical values, and thus edges). You should therefore avoid testing fronts on these inputs in the master (MAST), fast (FAST) or auxiliary (AUXi) tasks.
How to program event processing

The table below summarizes the essential steps for programming event processing.

<table>
<thead>
<tr>
<th>Step</th>
<th>Action</th>
</tr>
</thead>
</table>
| 1    | **Configuration phase** (for events triggered by input/output modules)  
In offline mode, from the configuration editor, select **Event Processing (EVT)**  
and the event processing number for the channel of the input/output module concerned. |
| 2    | **Unmasking phase**  
The task which can be interrupted must in particular:  
- Enable processing of events at system level: set bit %S38 to 1 (default value).  
- Unmask events with the instruction **UNMASKEVT** (active by default).  
- Unmask the events concerned at channel level (for events triggered by input/output modules) by setting the input/output module’s implicit language objects for unmasking of events to 1. By default, the events are masked.  
- Check that the stack of events at system level is not saturated (bit %S39 must be at 0). |
| 3    | **Event program creation phase**  
The program must:  
- Determine the origin of the event(s) on the basis of the event status word associated with the input/output module if the module is able to generate several events.  
- Carry out the reflex processing associated with the event. This process must be as short as possible.  
- Write the reflex outputs concerned.  

**Note:** the event status word is automatically reset to zero.
Illustration of event unmasking

This figure shows event unmasking in the MAST task.

- (*Unmasking events*)
  - UNMASKEVT
  - EN
  - ENO

- (*If cold start THEN authorize events*)
  - %S1

- (*If cold start THEN unmask threshold 0 crossing event: %QW1.1.0.5*)
  - %QW1.1.0.5

- (*If cold start THEN unmask threshold 1 crossing event: %QW1.1.0.6*)
  - %QW1.1.0.6

- (%S1)

- (%S38)

- (%QW1.0.1.5)

- (%QW1.0.1.6)

Illustration of the contents of event processing

This figure shows the possible contents of event processing (bit test and action).

- (*If crossing threshold event 0 THEN set physical output %Q2.0 to 1*)
  - %W1.0.3.5
  - %Q2.0

- (*If crossing threshold event 1 THEN set physical output %Q2.0 to 0*)
  - %W1.0.3.6
  - %Q2.0
Application memory structure

At a Glance

Subject of this Chapter
This chapter describes the application memory structure of Premium, Atrium and Quantum PLCs.

What's in this Chapter?
This chapter contains the following topics:

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory structure of Premium and Atrium PLCs</td>
<td>110</td>
</tr>
<tr>
<td>Detailed description of the memory zones</td>
<td>112</td>
</tr>
<tr>
<td>Memory structure of Quantum PLCs</td>
<td>113</td>
</tr>
<tr>
<td>Detailed description of the memory zones</td>
<td>115</td>
</tr>
</tbody>
</table>
Memory structure of Premium and Atrium PLCs

General
The PLC memory supports:
- **located application data**,
- **unlocated application data**,
- **the program**: task descriptors and executable code, constant words, initial values and configuration of inputs/outputs.

Structure without memory extension card
The data and program are supported by the internal RAM of the processor module. The following diagram describes the memory structure.

<table>
<thead>
<tr>
<th>Internal RAM</th>
<th>Located data</th>
<th>Unlocated data</th>
<th>Program</th>
</tr>
</thead>
</table>

Structure with memory extension card
The data is supported by the internal RAM of the processor module. The program is supported by the extension memory card. The following diagram describes the memory structure.

<table>
<thead>
<tr>
<th>Internal RAM</th>
<th>Located data</th>
<th>Unlocated data</th>
<th>Program</th>
</tr>
</thead>
<tbody>
<tr>
<td>External memory card</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Memory backup
The internal RAM is backed up by a Ni-Cad battery supported by the processor module. The RAM memory cards are backed up by a Ni-Cad battery.
Three types of memory card are offered:

- **application**: these cards contain the application program. The cards offered use either RAM or Flash EPROM technology.
- **application + file storage**: in addition to the program, these cards also contain a zone which can be used to backup/restore data using the program. The cards on offer use either RAM or Flash EPROM technology.
- **file storage**: these cards can be used to backup/restore data using the program. These cards use SRAM technology.

The following diagram describes the memory structure with an application and file storage card.

![Memory Structure Diagram](image)

**Note**: On processors with 2 memory card slots, the lower slot is reserved for the file storage function.
Detailed description of the memory zones

User data

This zone contains the located and unlocated application data.

- located data:
  - %M, %Si Boolean and %MW,%SW numerical data
  - data associated with modules (%I, %Q, %IW, %QW,%KW etc.)
- unlocated data:
  - Boolean and numerical data (instances)
  - EFB and DFB instances

User program and constants

This zone contains the executable codes and constants of the application.

- executable codes:
  - program code
  - code associated with EFs, EFBs and the management of I/O modules
  - code associated with DFBs
- constants:
  - KW constant words
  - constants associated with inputs/outputs
  - initial data values

This zone also contains the necessary information for downloading the application: graphic codes, symbols etc.

Other information

Other information relating to the configuration and structure of the application are also stored in the memory (in a data or program zone depending on the type of information).

- Configuration: other data relating to the configuration (hardware configuration, software configuration).
- System: data used by the operating system (task stack, etc.).
- Diagnostics: information relating to process or system diagnostics, diagnostics buffer.
Memory structure of Quantum PLCs

General

The PLC memory supports:
- located application data (State Ram),
- unlocated application data,
- the program: task descriptors and executable code, initial values and configuration of inputs/outputs.

Structure without memory extension card

The data and program are supported by the internal RAM of the processor module. The following diagram describes the memory structure.

Structure with memory extension card

Quantum 140 CPU 6*** processors can be fitted with a memory extension card. The data is supported by the internal RAM of the processor module. The program is supported by the extension memory card. The following diagram describes the memory structure.
Memory structure

**Memory backup**

The internal RAM is backed up by a Ni-Cad battery supported by the processor module.
The RAM memory cards are backed up by a Ni-Cad battery.

**Specificities of memory cards**

Three types of memory card are offered:

- **application**: these cards contain the application program. The cards on offer use either RAM or Flash EPROM technology
- **application + file storage**: in addition to the program, these cards also contain a zone which can be used to backup/restore data using the program. The cards on offer use either RAM or Flash EPROM technology
- **file storage**: these cards can be used to backup/restore data using the program. These cards use SRAM technology.

The following diagram describes the memory structure with an application and file storage card.

![Memory Structure Diagram]

**Note**: On processors with 2 memory card slots, the lower slot is reserved for the file storage function.
Detailed description of the memory zones

Unlocated data
- This zone contains unlocated data:
  - Boolean and numerical data
  - EFB and DFB

Located data
- This zone contains located data (State Ram):
  - User program
    - This zone contains the executable codes of the application.
      - program code
      - code associated with EFs, EFBs, and the management of I/O modules
      - code associated with DFBs
      - initial variable values
      - This zone also contains the necessary information for downloading the application: graphic codes, symbols etc.
  - Operating system
    - On 140 CPU 31••/41••/51•• processors, this contains the operating system for processing the application. This operating system is transferred from an internal EPROM memory to internal RAM on power up.
  - Application backup
    - A Flash EPROM memory zone of 1435K8, available on processors 140 CPU 31••/41••/51••, can be used to backup the program and the initial values of variables.
    - The application stored in this zone is automatically transferred to internal RAM when the PLC processor is powered up (if the PLC MEM switch is set to off on the processor front panel).

<table>
<thead>
<tr>
<th>Address</th>
<th>Object address</th>
<th>Data use</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xxxxx</td>
<td>%Qr.m.c.d,%Mi</td>
<td>output module bits and internal bits.</td>
</tr>
<tr>
<td>1xxxxx</td>
<td>%Ir.m.c.d, %i</td>
<td>input module bits.</td>
</tr>
<tr>
<td>3xxxxx</td>
<td>%IWr.m.c.d, %Wi</td>
<td>input register words of input/output modules.</td>
</tr>
<tr>
<td>4xxxxx</td>
<td>%QWr.m.c.d, %MWi</td>
<td>output words of input/output modules and internal words.</td>
</tr>
</tbody>
</table>
Other information

Other information relating to the configuration and structure of the application are also stored in the memory (in a data or program zone depending on the type of information).

- Configuration: other data relating to the configuration (hardware configuration, software configuration).
- System: data used by the operating system (task stack, etc.).
- Diagnostics: information relating to process or system diagnostics, diagnostics buffer.
Operating modes

At a Glance

Subject of this Chapter
The chapter describes the operating modes of the PLC in the event of power outage and restoral, the impacts on the application program and the updating of inputs/outputs.

What's in this Chapter?
This chapter contains the following topics:

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processing of power outage and restoral</td>
<td>118</td>
</tr>
<tr>
<td>Processing on cold start</td>
<td>121</td>
</tr>
<tr>
<td>Processing on warm restart</td>
<td>125</td>
</tr>
<tr>
<td>Automatic start in Run</td>
<td>127</td>
</tr>
<tr>
<td>PLC HALT mode</td>
<td>128</td>
</tr>
</tbody>
</table>
Operating modes

Processing of power outage and restoral

General

If the duration of the outage is less than the power supply filtering time, it has no effect on the program which continues to run normally. If this is not the case, the program is interrupted and power restoral processing is activated.

Filtering time:

<table>
<thead>
<tr>
<th>PLC</th>
<th>Alternating Current</th>
<th>Direct Current</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quantum</td>
<td>10ms</td>
<td>1ms</td>
</tr>
<tr>
<td>Premium</td>
<td>10ms</td>
<td>1ms</td>
</tr>
<tr>
<td>Atrium</td>
<td>30ms</td>
<td>-</td>
</tr>
</tbody>
</table>
### Illustration

The illustration shows the different types of power restoration detected by the system.

![Flowchart](image)

### Operation

The table below describes the power outage processing phases.

<table>
<thead>
<tr>
<th>Phase</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>On power outage the system stores the application context and the time of outage.</td>
</tr>
<tr>
<td>2</td>
<td>It sets all the outputs in the fallback state (state defined in configuration).</td>
</tr>
</tbody>
</table>
| 3     | On power restoration, the saved context is compared to the current one, which defines the type of startup to be performed:  
* if the application context has changed (i.e., loss of system context or new application), the PLC initializes the application: cold start,  
* if the application context is the same, the PLC carries out a restart without initialization of data: warm restart. |
### Operating modes

<table>
<thead>
<tr>
<th>Power outage on a rack, other than rack 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>All the channels on this rack are seen as in error by the processor, but the other racks are not affected. The values of the inputs in error are no longer updated in the application memory and are reset to zero in a discrete input module, unless they have been forced, in which case they are maintained at the forcing value.</td>
</tr>
</tbody>
</table>

If the duration of the outage is less than the filtering time, it has no effect on the program which continues to run normally.
### Processing on cold start

The following table describes the different possible causes of a cold start.

<table>
<thead>
<tr>
<th>Causes</th>
<th>Startup characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loading of an application</td>
<td>Cold start forced in STOP</td>
</tr>
<tr>
<td>RESET button pressed on processor (Premium)</td>
<td>Cold start forced in STOP or RUN mode as defined in the configuration</td>
</tr>
<tr>
<td>RESET button pressed on the processor after a blocking fault (Premium)</td>
<td>Cold start forced in STOP</td>
</tr>
<tr>
<td>Movement of handle or insertion/removal of a PCMCIA memory card</td>
<td>Cold start forced in STOP or RUN mode as defined in the configuration</td>
</tr>
<tr>
<td>Initialization from Unity Pro. Forcing the system bit %S0</td>
<td>Start in STOP or in RUN (retaining the operating mode in progress at downtime), without initialization of discrete input/output and application-specific modules</td>
</tr>
<tr>
<td>Restoral after power supply outage with loss of context</td>
<td>Cold start forced in STOP or RUN mode as defined in the configuration</td>
</tr>
</tbody>
</table>
Illustration

The diagram below describes how a cold restart operates.

1. **Acquisition of inputs**
2. **TOP program execution**
   - If %SW10.0=0, processing of warm restart
   - Power outage > micro-cut
   - Setting of bit %S0 to 0 and setting of bit %SW10.0 to 1
3. **Updating of outputs**
4. **Processor stopped**
   - Application context saved
   - Power restoral
   - Self-testing of the application
   - Initialization of the application
5. **Setting of bit %S0 to 1 and setting of bit %SW10.0 to 0**
6. **Initialization of the application**
The table below describes the program execution restart phases on cold restart.

<table>
<thead>
<tr>
<th>Phase</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>The startup is performed in RUN or in STOP depending on the status of the Automatic start in RUN parameter defined in the configuration or, if this is in use, depending on the state of the RUN/STOP input. Program execution is resumed at the start of the cycle.</td>
</tr>
</tbody>
</table>
| 2     | The system carries out the following:  
- the initialization of data (bits, I/O image, words etc.) with the initial values defined in the data editor (value set to 0, if no other initial value has been defined). For %MW words, the values can be retained on cold restart if the Reset of %MW on cold restart option is unchecked in the Configuration screen of the processor.  
- the initialization of elementary function blocks on the basis of initial data.  
- the initialization of data declared in the DFBs: either to 0, or to the initial value declared in the code, or with the value saved during execution of the SAVE function  
- the initialization of system bits and words.  
- the deactivation of tasks, other than the master task, until the end of the first master task cycle.  
- the positioning of charts to initial steps.  
- the cancellation of any forcing.  
- the initialization of message and event queues  
- the sending of configuration parameters to all discrete input/output modules and application-specific modules. |
| 3     | For this first restart cycle the system does the following:  
- relaunches the master task with the %S0 (cold restart) and %S13 (first cycle in RUN) bits set to 1, and the %SW10 word (detection of a cold restart during the first task cycle) is set to 0,  
- resets the %S0 and %S13 bits to 0, and sets each bit of the word %SW10 to 1 at the end of this first cycle of the master task,  
- activates the fast task and event processing at the end of the first cycle of the master task. |

**Processing a cold start by program**

It is advisable to test the bit %SW10.0 to detect a cold start and start processing specific to this cold start.

**Note:** It is possible to test the bit %S0, if the parameter Automatic start in RUN has been selected. If this is not the case, the PLC starts in STOP, the bit %S0 then switches to 1 on the first cycle after restart but is not visible to the program because it is not executed.
Operating modes

Output changes, for Premium and Atrium
As soon as a power outage is detected, the outputs are set in the fallback position:

- either they are assigned the fallback value,
- or the current value is maintained,

depending on the choice made in the configuration.

After power restoral, the outputs remain at zero until they are updated by the task.

Output changes, for Quantum
As soon as a power outage is detected,
- the local outputs are set to zero,
- the outputs of the remote or distributed extension racks are set in the fallback position.

After power restoral, the outputs remain at zero until they are updated by the task.

<table>
<thead>
<tr>
<th>CAUTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>the behavior of forced outputs was changed between Modsoft/NxT/Concept and Unity Pro.</td>
</tr>
<tr>
<td>With Modsoft/NxT/Concept, you cannot force outputs if the Quantum processor memory protection switch is set to &quot;On&quot;.</td>
</tr>
<tr>
<td>With Unity Pro, you can force outputs if the Quantum processor memory protection switch is set to &quot;On&quot;.</td>
</tr>
<tr>
<td>With Modsoft/NxT/Concept, forced outputs retain their status after a cold start.</td>
</tr>
<tr>
<td>With Unity Pro, forced outputs lose their status after a cold start.</td>
</tr>
<tr>
<td>Failure to follow this instruction can result in injury or equipment damage.</td>
</tr>
</tbody>
</table>

For Quantum 140 CPU 31••/41••/51••
These processors have a Flash EPROM memory of 1,435 KB which can be used to save the program and the initial values of variables.

On power restoral, you can choose the desired operating mode using the PLC MEM switch on the processor front panel. For detailed information on how this switch works, you can consult the Quantum manual (See Unity Pro, Hardware Reference Manual, Key Switches).

- **off position**: The application contained in this zone is automatically transferred to internal RAM when the PLC processor is powered up: cold restart of the application.
- **on position**: The application contained in this zone is not transferred to internal RAM: warm restart of the application.
Operating modes

Processing on warm restart

Cause of a warm restart

A warm restart may be caused:
- by a power restoral without loss of context,
- by the system bit %S1 being set to 1 by the program,
- by Unity Pro from the terminal,
- by pressing the RESET button of the power supply module of rack 0 (on Premium PLC).

Illustration

The diagram below describes how a warm restart operates.
Operating modes

The table below describes the program execution restart phases on warm restart.

<table>
<thead>
<tr>
<th>Phase</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Program execution resumes starting from the element where the power outage occurred, without updating the outputs.</td>
</tr>
</tbody>
</table>
| 2     | At the end of the restart cycle, the system carries out the following:  
  ● the initialization of message and event queues,  
  ● the sending of configuration parameters to all discrete input/output and application-specific modules,  
  ● the deactivation of the fast task and event processing (until the end of the master task cycle). |
| 3     | The system performs a restart cycle during which it:  
  ● re-acknowledges all the input modules,  
  ● relaunches the master task with the bits %S1 (warm restart) set to 1,  
  ● resets bit %S1 to 0 at the end of this first master task cycle,  
  ● reactivates the fast task, the auxiliary tasks and event processing at the end of this first cycle of the master task. |

Processing a warm restart by program

In the event of warm restart, if you want the application to be processed in a particular way, you must write the corresponding program conditional on the test that %S1 is set to 1 at the start of the master task program.

For Quantum PLCs, the switch on the front panel of the processor can be used to configure operating modes. For further details, see Quantum documentation (See Unity Pro, Harware Reference Manual, Key Switches).

Output changes, for Premium and Atrium

As soon as a power outage is detected, the outputs are set in the fallback position:  
● either they are assigned the fallback value,  
● or the current value is maintained,  
depending on the choice made in the configuration.

After power restoral, the outputs remain in the fallback position until they are updated by the task.

Output changes, for Quantum

As soon as a power outage is detected,  
● the local outputs are set to zero,  
● the outputs of the remote or distributed extension racks are set in the fallback position.

After power restoral, the outputs remain in the fallback position until they are updated by the task.
Operating modes

Automatic start in Run

Description
Automatic start in RUN is a processor configuration option. This option forces the PLC to start in RUN after a cold restart (See Processing on cold start, p. 121) (except after an application has been loaded onto PLC).

For Quantum PLCs, automatic start in RUN also depends on the position of the switch on the front panel of the processor. For more details, refer to the Quantum documentation (See Unity Pro, Harware Reference Manual, Key Switches).

<table>
<thead>
<tr>
<th>DANGER</th>
</tr>
</thead>
</table>

Using the "automatic start in RUN" option

- Inserting the PCMCIA card when the PLC is powered up will trigger automatic start in RUN
- Replacing the processor while powered up will trigger automatic start in RUN
- Unintentional or careless use of the reset button will trigger automatic start in RUN
- If the battery is found to be defective in the event of a power outage, this will trigger automatic start in RUN

It is highly recommended that you use the RUN/STOP input (on Premium PLCs) or the switch on the front panel of the processor (for Quantum PLCs) to avoid an unwanted restart when in RUN mode.

Failure to follow this instruction will result in death or serious injury.
Operating modes

PLC HALT mode

At a Glance
The following actions switches the PLC to HALT mode:
- using the HALT instruction
- watchdog overflow
- Program execution error (division by zero, overflow, etc.) if the bit %S78 (See Description of System Bits %S60 to %S78, p. 141) is set to 1.

Precautions
Caution: When the PLC is in HALT, all tasks are stopped (See Unity Pro, Operating Modes Manual, Setting a task to STOP). Check the behavior of the associated I/Os.
System objects

At a Glance

Subject of this Chapter

This chapter describes the system bits and words of Unity Pro language.

Note: The symbols, associated with each bit object or system word, mentioned in the descriptive tables of these objects, are not implemented as standard in the software, but can be entered using the data editor. They are proposed in order to ensure the homogeneity of their names in the different applications.

What's in this Chapter?

This chapter contains the following sections:

<table>
<thead>
<tr>
<th>Section</th>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1</td>
<td>System bits</td>
<td>130</td>
</tr>
<tr>
<td>6.2</td>
<td>System words</td>
<td>145</td>
</tr>
<tr>
<td>6.3</td>
<td>Atrium/Premium-specific system words</td>
<td>161</td>
</tr>
<tr>
<td>6.4</td>
<td>Quantum-specific system words</td>
<td>170</td>
</tr>
</tbody>
</table>
6.1 System bits

Description

Subject of this Section
This section describes the system bits.

What's in this Section?
This section contains the following topics:

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>System bit introduction</td>
<td>131</td>
</tr>
<tr>
<td>Description of system bits %S0 to %S7</td>
<td>132</td>
</tr>
<tr>
<td>Description of System Bits %S9 to %S13</td>
<td>133</td>
</tr>
<tr>
<td>Description of system bits %S15 to %S21</td>
<td>135</td>
</tr>
<tr>
<td>Description of System Bits %S30 to %S59</td>
<td>138</td>
</tr>
<tr>
<td>Description of System Bits %S60 to %S78</td>
<td>141</td>
</tr>
<tr>
<td>Description of system bits %S80 to %S96</td>
<td>143</td>
</tr>
<tr>
<td>Description of system bits %S100 to %S122</td>
<td>144</td>
</tr>
</tbody>
</table>
System bit introduction

General

The Premium, Atrium and Quantum PLCs use %Si system bits which indicate the state of the PLC, or they can be used to control how it operates.

These bits can be tested in the user program to detect any functional development requiring a set processing procedure.

Some of these bits must be reset to their initial or normal state by the program. However, the system bits that are reset to their initial or normal state by the system must not be reset by the program or by the terminal.
### Description of system bits %S0 to %S7

<table>
<thead>
<tr>
<th>Bit Symbol</th>
<th>Function</th>
<th>Description</th>
<th>Initial state</th>
<th>Quantum</th>
<th>Premium Atrium</th>
</tr>
</thead>
<tbody>
<tr>
<td>%S0 COLDSTART</td>
<td>Cold start</td>
<td>Normally on 0, this bit is set on 1 by:</td>
<td>1 (1 cycle)</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• power restoral with loss of data (battery fault),</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• the user program,</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• the terminal,</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• a change of cartridge.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• program loading.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>This bit is set to 1 during the first complete restored cycle of the PLC</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>either in RUN or in STOP mode. It is reset to 0 by the system before the</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>following cycle.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%S1 WARMSTART</td>
<td>Warm restart</td>
<td>Normally at 0, this bit is set to 1 by:</td>
<td>0</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• power restoral with data save,</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• the user program,</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• the terminal,</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• action on change of cartridge.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>It is reset to 0 by the system at the end of the first complete cycle and</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>before the outputs are updated.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%S4 TB10MS</td>
<td>Timebase 10 ms</td>
<td>An internal timer regulates the change in status of this bit.</td>
<td>-</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td></td>
<td></td>
<td>It is asynchronous in relation to the PLC cycle.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Graph:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%S5 TB100MS</td>
<td>Timebase 100 ms</td>
<td>Idem %S4</td>
<td>-</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>%S6 TB1SEC</td>
<td>Time base 1 s</td>
<td>Idem %S4</td>
<td>-</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>%S7 TB1MIN</td>
<td>Time base 1 min</td>
<td>Idem %S4</td>
<td>-</td>
<td>YES</td>
<td>YES</td>
</tr>
</tbody>
</table>
# Description of System Bits %S9 to %S13

## Detailed Description

Description of system bits %S9 to %S13:

<table>
<thead>
<tr>
<th>Bit Symbol</th>
<th>Function</th>
<th>Description</th>
<th>Initial state</th>
<th>Quantum</th>
<th>Premium Atrium</th>
</tr>
</thead>
<tbody>
<tr>
<td>%S9</td>
<td>OUTDIS</td>
<td>Outputs set to the fallback position on all buses</td>
<td>0</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Normally at 0, this bit is set to 1 by the program or the terminal:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>- set to 1: sets the bit to 0 or maintains the current value depending on</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>the chosen configuration (X bus, Fipio, AS-i, etc.).</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>- set to 0: outputs are updated normally.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Note:</strong> The system bit acts directly on the physical outputs and not on</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>the image bits of the outputs.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%S10</td>
<td>IOERR</td>
<td>Input/output fault</td>
<td>1</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Normally at 1, this is set to 0 when an I/O fault on an in-rack module or</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>device on Fipio is detected (e.g. non-compliant configuration, exchange</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>fault, hardware fault, etc.). The %S10 bit is reset to 1 by the system as</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>soon as the fault disappears.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%S11</td>
<td>WDG</td>
<td>Watchdog overflow</td>
<td>0</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Normally at 0, this is set to 1 by the system as soon as the task execution</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>time becomes greater than the maximum execution time (i.e. the watchdog)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>declared in the task properties.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%S12</td>
<td>PLCRUNNING</td>
<td>PLC in RUN</td>
<td>0</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td></td>
<td></td>
<td>This bit is set to 1 by the system when the PLC is in RUN. It is set to</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 by the system as soon as the PLC is no longer in RUN (STOP, INIT, etc.).</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%S13</td>
<td>1RSTSCANRUN</td>
<td>First cycle afterswitching to RUN</td>
<td>-</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Switching the PLC from STOP mode to RUN mode is indicated by setting</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>system bit %S13 to 1. This bit is reset to 0 at the end of the cycle of</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>the MAST task in RUN mode.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
CAUTION

%S10 for Quantum PLCs

On Quantum, communication errors from modules (NOM, NOE, NWM, CRA, CRP) and MMS modules are not reported on bits %S10 and %S16.

It is entirely your responsibility to ensure that these system bits are used correctly.

Failure to follow this instruction can result in injury or equipment damage.
### Description of system bits %S15 to %S21

#### Detailed description

<table>
<thead>
<tr>
<th>Bit Symbol</th>
<th>Function</th>
<th>Description</th>
<th>Initial state</th>
<th>Quantum</th>
<th>Premium</th>
<th>Atrium</th>
</tr>
</thead>
<tbody>
<tr>
<td>%S15 STRINGERROR</td>
<td>Character string fault</td>
<td>Normally set to 0, this bit is set to 1 when the destination zone for a character string transfer is not of sufficient size (including the number of characters and the end of string character) to receive this character string. The application stops in error state if the %S78 bit has been set to 1. This bit must be reset to 0 by the application.</td>
<td>0 YES YES</td>
<td>YES</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%S16 IOERRTSK</td>
<td>Task input/output fault</td>
<td>Normally set to 1, this is set to 0 by the system when a fault occurs on an in-rack I/O module or a Fipio device configured in the task. This bit must be reset to 1 by the user.</td>
<td>1 YES YES</td>
<td>YES</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%S17 CARRY</td>
<td>Rotate shift output</td>
<td>Normally at 0. During a rotate shift operation, this takes the state of the outgoing bit.</td>
<td>0 YES YES</td>
<td>YES</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### System objects

<table>
<thead>
<tr>
<th>Bit Symbol</th>
<th>Function</th>
<th>Description</th>
<th>Initial state</th>
<th>Quantum</th>
<th>Premium Atrium</th>
</tr>
</thead>
</table>
| %S18 OVERFLOW | Overflow or arithmetic error | Normally set to 0, this is set to 1 in the event of a capacity overflow if there is:  
- a result greater than +32 767 or less than -32 768, in single length,  
- result greater than +65 535, in unsigned integer,  
- a result greater than +2 147 483 647 or less than -2 147 483 648, in double length,  
- result greater than +4 294 967 296, in double length or unsigned integer,  
- real values outside limits,  
- division by 0,  
- the root of a negative number,  
- forcing to a non-existent step on a drum,  
- stacking up of an already full register,  
- emptying of an already empty register.  
It must be tested by the user program after each operation where there is a risk of overflow, then reset to 0 by the user if there is indeed an overflow.  
When the %S18 bit switches to 1, the application stops in error state if the %S78 bit has been to set to 1. | 0 | YES | YES |
| %S19 OVERRUN | Task period overrun (periodical scanning) | Normally set to 0, this bit is set to 1 by the system in the event of a time period overrun (i.e. task execution time is greater than the period defined by the user in the configuration or programmed into the %SW word associated with the task). The user must reset this bit to 0. Each task manages its own %S19 bit. | 0 | YES | YES |
### System objects

<table>
<thead>
<tr>
<th>Bit Symbol</th>
<th>Function</th>
<th>Description</th>
<th>Initial state</th>
<th>Quantum</th>
<th>Premium</th>
</tr>
</thead>
<tbody>
<tr>
<td>%S20</td>
<td>INDEXOVF</td>
<td>Normally set to 0, this is set to 1 when the address of the indexed object becomes less than 0 or exceeds the number of objects declared in the configuration. In this case, it is as if the index were equal to 0. It must be tested by the user program after each operation where there is a risk of overflow, then reset to 0 if there is indeed an overflow. When the %S20 bit switches to 1, the application stops in error state if the %S78 bit has been to set to 1.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%S21</td>
<td>1RSTTASKRUN</td>
<td>Tested in a task (Mast, Fast, Aux0, Aux1, Aux2 Aux3), the bit %S21 indicates the first cycle of this task, %S21 is set to 1 at the start of the cycle and reset to zero at the end of the cycle. Notes: the bit %S21 does not have the same meaning in PL7 as in Unity Pro.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

**CAUTION**

**%S16 for Quantum PLCs**

On Quantum, communication errors from modules (NOM, NOE, NWM, CRA, CRP) and MMS modules are not reported on bits %S10 and %S16.

It is entirely your responsibility to ensure that these system bits are used correctly

Failure to follow this instruction can result in injury or equipment damage.
**Description of System Bits %S30 to %S59**

### Detailed Description

<table>
<thead>
<tr>
<th>Bit Symbol</th>
<th>Function</th>
<th>Description</th>
<th>Initial state</th>
<th>Quantum</th>
<th>Premium Atrium</th>
</tr>
</thead>
<tbody>
<tr>
<td>%S30</td>
<td>MASTACT</td>
<td>Activation/deactivation of the master task</td>
<td>Normally set to 1, the user setting the bit to 0 deactivates the master task. This bit is taken into consideration by the system at the end of each MAST task cycle.</td>
<td>1</td>
<td>YES</td>
</tr>
<tr>
<td>%S31</td>
<td>FASTACT</td>
<td>Activation/deactivation of the fast task</td>
<td>Normally set to 1 when the user creates the task. The user setting it to 0 deactivates the task.</td>
<td>0</td>
<td>YES</td>
</tr>
<tr>
<td>%S32</td>
<td>AUX0ACT</td>
<td>Activation/deactivation of the auxiliary task 0</td>
<td>Normally set to 1 when the user creates the task. The user setting it to 0 deactivates the auxiliary task.</td>
<td>0</td>
<td>YES</td>
</tr>
<tr>
<td>%S33</td>
<td>AUX1ACT</td>
<td>Activation/deactivation of the auxiliary task 1</td>
<td>Normally set to 1 when the user creates the task. The user setting it to 0 deactivates the auxiliary task.</td>
<td>0</td>
<td>YES</td>
</tr>
<tr>
<td>%S34</td>
<td>AUX2ACT</td>
<td>Activation/deactivation of the auxiliary task 2</td>
<td>Normally set to 1 when the user creates the task. The user setting it to 0 deactivates the auxiliary task.</td>
<td>0</td>
<td>YES</td>
</tr>
<tr>
<td>%S35</td>
<td>AUX3ACT</td>
<td>Activation/deactivation of the auxiliary task 3</td>
<td>Normally set to 1 when the user creates the task. The user setting it to 0 deactivates the auxiliary task.</td>
<td>0</td>
<td>YES</td>
</tr>
<tr>
<td>%S38</td>
<td>ACTIVEVT</td>
<td>Enabling/inhibition of events</td>
<td>Normally set to 1, the user setting the bit to 0 causes events to be inhibited.</td>
<td>1</td>
<td>YES</td>
</tr>
<tr>
<td>%S39</td>
<td>EVTOVR</td>
<td>Saturation in event processing</td>
<td>This bit is set to 1 by the system to indicate that one or more events cannot be processed following saturation of the queues. The user must reset this bit to 0.</td>
<td>0</td>
<td>YES</td>
</tr>
<tr>
<td>Bit Symbol</td>
<td>Function</td>
<td>Description</td>
<td>Initial state</td>
<td>Quantum Premium Atrium</td>
<td></td>
</tr>
<tr>
<td>------------</td>
<td>----------</td>
<td>-------------</td>
<td>---------------</td>
<td>------------------------</td>
<td></td>
</tr>
<tr>
<td>%S40 RACK0ERR</td>
<td>Rack 0 input/output fault</td>
<td>The %S40 bit is assigned to rack 0. Normally set to 1, this bit is set to 0 when a fault occurs on the rack's I/Os. In this case: ● the %S10 bit is set to 0, ● the I/O processor LED is on, ● the %I.m.c.Err module bit is set to 1. This bit is reset to 1 when the fault disappears.</td>
<td>1</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td>%S41 RACK1ERR</td>
<td>Rack 1 input/output fault</td>
<td>Idem %S40 for rack 1.</td>
<td>1</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td>%S42 RACK2ERR</td>
<td>Rack 2 input/output fault</td>
<td>Idem %S40 for rack 2.</td>
<td>1</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td>%S43 RACK3ERR</td>
<td>Rack 3 input/output fault</td>
<td>Idem %S40 for rack 3.</td>
<td>1</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td>%S44 RACK4ERR</td>
<td>Rack 4 input/output fault</td>
<td>Idem %S40 for rack 4.</td>
<td>1</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td>%S45 RACK5ERR</td>
<td>Rack 5 input/output fault</td>
<td>Idem %S40 for rack 5.</td>
<td>1</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td>%S46 RACK6ERR</td>
<td>Rack 6 input/output fault</td>
<td>Idem %S40 for rack 6.</td>
<td>1</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td>%S47 RACK7ERR</td>
<td>Rack 7 input/output fault</td>
<td>Idem %S40 for rack 7.</td>
<td>1</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td>%S50 RTCWRITE</td>
<td>Updating of time and date via words %SW50 to %SW53</td>
<td>Normally set to 0, this bit is set to 1 or 0 by the program or the terminal. ● set to 0: update of system words %SW50 to %SW53 by the date and time supplied by the PLC real-time clock. ● set to 1: system words %SW50 to %SW53 are no longer updated, therefore making it possible to modify them. ● The switch from 1 to 0 updates the real-time clock with the values entered in words %SW50 to %SW53.</td>
<td>0</td>
<td>YES</td>
<td>YES</td>
</tr>
</tbody>
</table>
System objects

<table>
<thead>
<tr>
<th>Bit Symbol</th>
<th>Function</th>
<th>Description</th>
<th>Initial state</th>
<th>Quantum</th>
<th>Premium Atrium</th>
</tr>
</thead>
<tbody>
<tr>
<td>%S51 RTCERR</td>
<td>Time loss in real time clock</td>
<td>This system-managed bit set to 1 indicates that the real-time clock is missing or that its system words (%SW50 to %SW53) are meaningless. In this case the clock must be reset to the correct time.</td>
<td>-</td>
<td>YES</td>
<td>YES</td>
</tr>
</tbody>
</table>
| %S59 RTCTUNING | Incremental update of the time and date via word %SW59 | Normally set to 0, this bit can be set to 1 or 0 by the program or the terminal:  
  - set to 0: the system does not manage the system word %SW59,  
  - set to 1: the system manages edges on word %SW59 to adjust the date and current time (by increment). | 0            | YES     | YES            |
### Description of System Bits %S60 to %S78

**Detailed Description**

Description of system bits %S60 to %S78:

<table>
<thead>
<tr>
<th>Bit Symbol</th>
<th>Function</th>
<th>Description</th>
<th>Initial state</th>
<th>Quantum</th>
<th>Premium Atrium</th>
</tr>
</thead>
<tbody>
<tr>
<td>%S60</td>
<td>BACKUPCHVOV</td>
<td>Voluntary switching command</td>
<td>0</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td>%S67</td>
<td>PCMCIABAT0</td>
<td>State of the application memory card battery</td>
<td>-</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>%S68</td>
<td>PLCBAT</td>
<td>State of processor battery</td>
<td>-</td>
<td>YES</td>
<td>YES</td>
</tr>
</tbody>
</table>
### System objects

<table>
<thead>
<tr>
<th>Bit Symbol</th>
<th>Function</th>
<th>Description</th>
<th>Initial state</th>
<th>Quantum</th>
<th>Premium Atrium</th>
</tr>
</thead>
</table>
| %S75 PCMCIABAT1 | State of the data storage memory card battery | This bit is used to monitor the status of the main battery when the memory card is in the lower PCMCIA slot (Premium (TSX P57 4•• and TSX P57 5••) and Quantum (140 CPU 671 60, 140 CPU 651 60 and 140 CPU 651 50):  
- set to 1: main voltage battery is low (application is preserved but you must replace the battery following the so-called predictive maintenance (See Unity Pro, 984 to Unity Converter, User manual)) procedure,  
- set to 0: main battery voltage is sufficient (application always preserved).  
Bit %S75 is managed:  
- on the PV06 small and medium capacity RAM memory cards (product version written on the card label), i.e. offering memory size under Unity ≤ 768K: TSX MRP P 128K, TSX MRP P 224K TSX MCP C 224K, MCP C 512K, TSX MRP P 384K, TSX MRP C 448K, TSX MRP C 768K,  
- under Unity whose version is ≥ 2.02. | - | YES | YES |
| %S76 DIAGBUFFCONF | Configured diagnostics buffer | This bit is set to 1 by the system when the diagnostics option has been configured – a diagnostics buffer for storage of errors found by diagnostics DFBs is then reserved. This bit is read-only. | 0 | YES | YES |
| %S77 DIAGBUFFFULL | Full diagnostics buffer | This bit is set to 1 by the system when the buffer that receives errors from the diagnostics function blocks is full. This bit is read-only. | 0 | YES | YES |
| %S78 HALTIFERROR | Stop in the event of error | Normally at 0, this bit can be set to 1 by the user, to program a PLC stop on application fault: %S15, %S18, %20. | 0 | YES | YES |
## Description of system bits %S80 to %S96

### Detailed description

<table>
<thead>
<tr>
<th>Bit Symbol</th>
<th>Function</th>
<th>Description</th>
<th>Initial state</th>
<th>Quantum Premium Atrium</th>
</tr>
</thead>
<tbody>
<tr>
<td>%S80 RSTMSGCNT</td>
<td>Reset message counters</td>
<td>Normally set to 0, this bit can be set to 1 by the user to reset the message counters %SW80 to %SW86.</td>
<td>0</td>
<td>YES</td>
</tr>
<tr>
<td>%S90 COMRFSH</td>
<td>Refresh common words</td>
<td>Normally set to 0, this bit is set to 1 on receiving common words from another network station. This bit can be set to 0 by the program or the terminal to check the common words exchange cycle.</td>
<td>0</td>
<td>NO</td>
</tr>
<tr>
<td>%S91 LCKASYNREQ</td>
<td>Lock asynchronous request</td>
<td>When this bit is set to 1, the asynchronous communication requests processed in the monitoring task are executed in their entirety without interruption from the other MAST or FAST tasks, thus ensuring the data is read or written consistently. <strong>Reminder:</strong> the request server of the monitoring task is addressed via gate 7 (X-Way).</td>
<td>0</td>
<td>NO</td>
</tr>
<tr>
<td>%S92 EXCHGTIME</td>
<td>Measurement mode of the communication function</td>
<td>Normally set to 0, this bit can be set to 1 by the user to set communication functions to performance measurement mode. The communication functions time-out parameter then displays the round trip exchange time in tens of ms. <strong>Note:</strong> the communication functions are executed with a standard time-out of 10s.</td>
<td>0</td>
<td>NO</td>
</tr>
<tr>
<td>%S94 SAVECURRVAL</td>
<td>Saving adjustment values</td>
<td>Normally at 0, this bit can be set to 1 by the user to replace the initial values of the declared variables with a ‘Save’ attribute (e.g.: DFB variables) with the current values. The system resets the bit %S94 to 0 when the replacement has been made. <strong>Note:</strong> this bit must be used with care: do not set this bit permanently to 1 and use the master task only.</td>
<td>0</td>
<td>YES</td>
</tr>
</tbody>
</table>
### Description of system bits %S100 to %S122

#### Detailed description

<table>
<thead>
<tr>
<th>Bit Symbol</th>
<th>Function</th>
<th>Description</th>
<th>Initial state</th>
<th>Quantum</th>
<th>Premium Atrium</th>
</tr>
</thead>
<tbody>
<tr>
<td>%S100</td>
<td>PROTTERINL</td>
<td>Protocol on terminal port</td>
<td>-</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td></td>
<td></td>
<td>This bit is set to 0 or 1 by the system according to the state of the INL/DPT shunt on the console. ● if the shunt is missing (%S100=0), then the master Uni-Telway protocol is used, ● if the shunt is present (%S100=1) then the protocol used is the one indicated by the application configuration.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%S118</td>
<td>REMIOERR</td>
<td>General Fipio I/O fault</td>
<td>-</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Normally set to 1, this bit is set to 0 by the system when a fault occurs on a device connected to the RIO and Fipio remote input/output bus. This bit is reset to 1 by the system when the fault disappears.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%S119</td>
<td>LOCIOERR</td>
<td>General in-rack I/O fault</td>
<td>-</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Normally set to 1, this bit is set to 0 by the system when a fault occurs on an I/O module placed in one of the racks. This bit is reset to 1 by the system when the fault disappears.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%S120</td>
<td>DIOERRPLC</td>
<td>DIO bus fault (CPU)</td>
<td>-</td>
<td>YES</td>
<td>NO</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Normally set to 1, this bit is set to 0 by the system when a fault occurs on a device connected to the DIO bus managed by the Modbus Plus link built into the CPU.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%S121</td>
<td>DIOERRNOM1</td>
<td>DIO bus fault (NAME No. 1)</td>
<td>-</td>
<td>YES</td>
<td>NO</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Normally set to 1, this bit is set to 0 by the system when a fault occurs on a device connected to the DIO bus managed by the first 140 NAME 2•• module.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%S122</td>
<td>DIOERRNOM2</td>
<td>DIO bus fault (NAME No. 2)</td>
<td>-</td>
<td>YES</td>
<td>NO</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Normally set to 1, this bit is set to 0 by the system when a fault occurs on a device connected to the DIO bus managed by the second 140 NAME 2•• module.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
6.2 System words

Description

Subject of this Section
This chapter describes the Atrium, Premium and Quantum system words.

What's in this Section?
This section contains the following topics:

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description of system words %SW0 to %SW11</td>
<td>146</td>
</tr>
<tr>
<td>Description of System Words %SW12 to %SW19</td>
<td>149</td>
</tr>
<tr>
<td>Description of system words %SW30 to %SW47</td>
<td>151</td>
</tr>
<tr>
<td>Description of system words %SW48 to %SW59</td>
<td>153</td>
</tr>
<tr>
<td>Description of System Words %SW70 to %SW99</td>
<td>155</td>
</tr>
<tr>
<td>Description of system words %SW108 to %SW116</td>
<td>158</td>
</tr>
<tr>
<td>Description of system words %SW124 to %SW127</td>
<td>159</td>
</tr>
</tbody>
</table>
### Description of system words %SW0 to %SW11

#### Detailed description

<table>
<thead>
<tr>
<th>Word Symbol</th>
<th>Function</th>
<th>Description</th>
<th>Initial state</th>
<th>Quantum</th>
<th>Premium Atrium</th>
</tr>
</thead>
<tbody>
<tr>
<td>%SW0</td>
<td>Master task scanning period</td>
<td>Is used to modify the period of the master task via the user program or via the terminal. The period is expressed in ms (1...255 ms) %SW0=0 in cyclic operation. On a cold restart: it takes on the value defined by the configuration.</td>
<td>0</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>%SW1</td>
<td>Fast task scanning period</td>
<td>Is used to modify the period of the fast task via the user program or via the terminal. The period is expressed in ms (1...255 ms) On a cold restart: it takes on the value defined by the configuration.</td>
<td>0</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>%SW2</td>
<td>Auxiliary task scanning period</td>
<td>Is used to modify the period of the tasks defined in the configuration, via the user program or via the terminal. The period is expressed in tens of ms (10ms to 2.55s). (1) only on 140 CPU 6•• and TSX 57 5•• PLCs.</td>
<td>0</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>%SW3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%SW4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%SW5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Word Symbol</td>
<td>Function</td>
<td>Description</td>
<td>Initial state</td>
<td>Quantum</td>
<td>Premium Atrium</td>
</tr>
<tr>
<td>-------------</td>
<td>----------</td>
<td>-------------</td>
<td>---------------</td>
<td>---------</td>
<td>---------------</td>
</tr>
<tr>
<td>%SW8 TSKINHIBIN</td>
<td>Acquisition of task input monitoring</td>
<td>Normally set to 0, this bit can be set to 1 or 0 by the program or the terminal. It inhibits the input acquisition phase of each task: ● %SW8.0 = 1 inhibits the acquisition of inputs relating to the MAST task. ● %SW8.1 = 1 inhibits the acquisition of inputs relating to the FAST task. ● %SW8.2 to 5 = 1 inhibits the acquisition of inputs relating to the AUX 0...3 tasks.</td>
<td>0</td>
<td>YES (1)</td>
<td>YES</td>
</tr>
<tr>
<td>%SW9 TSKINHIBOUT</td>
<td>Monitoring of task output update</td>
<td>Normally set to 0, this bit can be set to 1 or 0 by the program or the terminal. Inhibits the output updating phase of each task. ● %SW9.0 = 1 assigned to the MAST task; outputs relating to this task are no longer managed. ● %SW9.1 = 1 assigned to the FAST task; outputs relating to this task are no longer managed. ● %SW9.2 to 5 = 1 assigned to the AUX 0...3 tasks; outputs relating to these tasks are no longer managed. (2) Note: On Quantum, inputs/outputs which are distributed via DIO bus are not assigned by the word %SW9</td>
<td>0</td>
<td>YES(2)</td>
<td>YES</td>
</tr>
<tr>
<td>%SW10 TSKINIT</td>
<td>First cycle after cold start</td>
<td>If the value of the current task bit is set to 0, this means that the task is performing its first cycle after a cold start. ● %SW10.0: is assigned to the MAST task. ● %SW10.1: is assigned to the FAST task. ● %SW10.2 to 5: is assigned to the AUX 0...3 tasks.</td>
<td>0</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>%SW11 WDGVALUE</td>
<td>Watchdog duration</td>
<td>Reads the duration of the watchdog. The duration is expressed in ms (10...1500 ms) This word cannot be modified.</td>
<td>-</td>
<td>YES</td>
<td>YES</td>
</tr>
</tbody>
</table>
CAUTION

Task outputs checked using word %SW9:

On Premium/Atrium:
Module outputs located on the X Bus automatically switch to the configured mode (fallback or maintain). On the Fipio bus, certain devices do not manage fallback mode. Here, only maintain mode is possible.

On Quantum:
All outputs, as well as the local or remote rack (RIO) are maintained in the state that preceded the switch to 1 of the %SW9 bit corresponding to the task.
The Distributed Inputs/Outputs (DIO) are not assigned by the system word %SW9.

Failure to follow this instruction can result in injury or equipment damage.
### Description of System Words %SW12 to %SW19

**Detailed Description**

Description of system words %SW12 to %SW19:

<table>
<thead>
<tr>
<th>Word Symbol</th>
<th>Function</th>
<th>Description</th>
<th>Initial state</th>
<th>Quantum</th>
<th>Premium Atrium</th>
</tr>
</thead>
<tbody>
<tr>
<td>%SW12 UTWPORTADDR</td>
<td>Uni-Telway terminal port address</td>
<td>Uni_Telway address of terminal port (in slave mode) as defined in the configuration and loaded into this word on cold start. <strong>Note:</strong> The modification of the value of this word is not taken into account by the system.</td>
<td>-</td>
<td>NO</td>
<td>YES</td>
</tr>
</tbody>
</table>
| %SW13 XWAYNETWADDR | Main address of the station | Indicates the following for the main network (Fipway or Ethway):  
- the station number (least significant byte) from 0 to 127,  
- the network number (most significant byte) from 0 to 63,  
(value of the micro-switches on the PCMCIA card). | 254 (16#00FE) | NO | YES |
| %SW14 OSCOMMVERS | Commercial version of PLC processor | This word contains the commercial version of the PLC processor.  
**Example:** 16#0135  
version: 01  
issue number: 35 | - | YES | YES |
| %SW15 OSCOMMPATCH | PLC processor patch version | This word contains the commercial version of the PLC processor patch.  
It is coded onto the least significant byte of the word.  
Coding: 0 = no patch, 1 = A, 2 = B...  
**Example:** 16#0003 corresponds to patch C. | - | YES | YES |
| %SW16 OSINTVERS | Firmware version number | This word contains the Firmware version number in hexadecimal of the PLC processor firmware.  
**Example:** 16#0017  
version: 2.1  
VN: 17 | - | YES | YES |
System objects

<table>
<thead>
<tr>
<th>Word Symbol</th>
<th>Function Description</th>
<th>Description</th>
<th>Initial state</th>
<th>Quantum</th>
<th>Premium</th>
</tr>
</thead>
</table>
| %SW17       | Error status on floating operation | On detection of an error in a floating arithmetic operation, bit %S18 is set to 1 and %SW17 error status is updated according to the following coding:  
  - %SW17.0 = Invalid operation / result is not a number  
  - %SW17.1 = Non-standardized operand / result is acceptable  
  - %SW17.2 = Division by 0 / result is infinity  
  - %SW17.3 = Overflow / result is infinity  
  - %SW17.4 = Underflow / result is 0  
  - %SW17.5 to 15 = not used  
  This word is reset to 0 by the system on cold start, and also by the program for re-usage purposes. | 0 | YES | YES |
| %SW18 %SW19 100MSCOUNTER | Absolute time counter | The words %SW18 and %SW19 are used to calculate duration. They are incremented every 1/10th of a second by the system (even when PLC is in STOP, they are no longer incremented if the PLC is powered down). They can be read and written by the user program or by the terminal. | 0 | YES | YES |
Description of system words %SW30 to %SW47

<table>
<thead>
<tr>
<th>Word Symbol</th>
<th>Function</th>
<th>Description</th>
<th>Initial state</th>
<th>Quantum</th>
<th>Premium</th>
</tr>
</thead>
<tbody>
<tr>
<td>%SW30</td>
<td>Master task execution time</td>
<td>Indicates the execution time of the last master task cycle (in ms).</td>
<td>-</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>%SW31</td>
<td>Maximum master task execution time</td>
<td>Indicates the longest master task execution time since the last cold start (in ms).</td>
<td>-</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>%SW32</td>
<td>Minimum master task execution time</td>
<td>Indicates the shortest master task execution time since the last cold start (in ms).</td>
<td>-</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>%SW33</td>
<td>Fast task execution time</td>
<td>Indicates the execution time of the last fast task cycle (in ms).</td>
<td>-</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>%SW34</td>
<td>Maximum fast task execution time</td>
<td>Indicates the longest fast task execution time since the last cold start (in ms).</td>
<td>-</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>%SW35</td>
<td>Minimum fast task execution time</td>
<td>Indicates the shortest fast task execution time since the last cold start (in ms).</td>
<td>-</td>
<td>YES</td>
<td>YES</td>
</tr>
</tbody>
</table>

Note: Execution time is the time elapsed between the start (input acquisition) and the end (output update) of a scanning period. This time includes the processing of event tasks, the fast task, and the processing of console requests.
Description of system words %SW36 to %SW47:

<table>
<thead>
<tr>
<th>Word Symbol</th>
<th>Function</th>
<th>Description</th>
<th>Initial state</th>
<th>Quantum</th>
<th>Premium</th>
</tr>
</thead>
<tbody>
<tr>
<td>%SW36 AUX0CURRTIME</td>
<td>Auxiliary task execution time</td>
<td>Indicate the execution time of the last cycle of the AUX 0...3 tasks (in ms). (1) only on 140 CPU 6•• and TSX P57 5•• PLCs.</td>
<td>-</td>
<td>YES (1)</td>
<td>YES (1)</td>
</tr>
<tr>
<td>%SW39 AUX1CURRTIME</td>
<td>Auxiliary task execution time</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%SW42 AUX2CURRTIME</td>
<td>Auxiliary task execution time</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%SW45 AUX3CURRTIME</td>
<td>Auxiliary task execution time</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%SW37 AUX0MAXTIME</td>
<td>Maximum auxiliary task execution time</td>
<td>Indicate the longest task execution time of AUX 0...3 tasks since the last cold start (in ms). (1) only on 140 CPU 6•• and TSX P57 5•• PLCs.</td>
<td>-</td>
<td>YES (1)</td>
<td>YES (1)</td>
</tr>
<tr>
<td>%SW40 AUX1MAXTIME</td>
<td>Maximum auxiliary task execution time</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%SW43 AUX2MAXTIME</td>
<td>Maximum auxiliary task execution time</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%SW46 AUX3MAXTIME</td>
<td>Maximum auxiliary task execution time</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%SW38 AUX0MINTIME</td>
<td>Minimum auxiliary task execution time</td>
<td>Indicate the shortest task execution time of AUX 0...3 tasks since the last cold start (in ms). (1) only on 140 CPU 6•• and TSX P57 5•• PLCs.</td>
<td>-</td>
<td>YES (1)</td>
<td>YES (1)</td>
</tr>
<tr>
<td>%SW41 AUX1MINTIME</td>
<td>Minimum auxiliary task execution time</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%SW44 AUX2MINTIME</td>
<td>Minimum auxiliary task execution time</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%SW47 AUX3MINTIME</td>
<td>Minimum auxiliary task execution time</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Description of system words %SW48 to %SW59

<table>
<thead>
<tr>
<th>Word Symbol</th>
<th>Function</th>
<th>Description</th>
<th>Initial state</th>
<th>Quantum</th>
<th>Premium Atrium</th>
</tr>
</thead>
<tbody>
<tr>
<td>%SW48</td>
<td>Number of Events</td>
<td>Indicates the number of events processed since the last cold start (in ms). This word can be written by the program or the terminal</td>
<td>0</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>%SW49</td>
<td>Real-time clock function</td>
<td>System words containing date and current time (in BCD):</td>
<td>-</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>%SW50</td>
<td></td>
<td>%SW49: day of the week</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%SW51</td>
<td></td>
<td>1 = Monday</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%SW52</td>
<td></td>
<td>2 = Tuesday</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%SW53</td>
<td></td>
<td>3 = Wednesday</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%SW53</td>
<td></td>
<td>4 = Thursday</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>5 = Friday</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>6 = Saturday</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>7 = Sunday</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

These words are managed by the system when bit %S50 is set to 0. These words can be written by the user program or by the terminal when the bit %S50 is set to 1.
System objects

<table>
<thead>
<tr>
<th>Word Symbol</th>
<th>Function</th>
<th>Description</th>
<th>Initial state</th>
<th>Quantum Atrium</th>
<th>Premium Atrium</th>
</tr>
</thead>
</table>
| %SW54 STOPSEC | Real-time clock function on last stop | System words containing date and time of the last power failure or PLC stop (in BCD):  
  - %SW54: Seconds (00SS).  
  - %SW55: Hours and Minutes (HHMM).  
  - %SW56: Month and Day (MMDD).  
  - %SW57: Year (YYYY).  
  - %SW58 most significant byte containing the day of the week (1 for Monday through to 7 for Sunday).  
  - %SW58 least significant byte containing code for the last stop.  
  - 1 = change from RUN to STOP by the terminal or the dedicated input  
  - 2 = stop on software fault (PLC task or SFC overrun)  
  - 4 = power outage or handling of the memory card  
  - 5 = stop on hardware fault  
  - 6 = stop on HALT instruction. | - | YES | YES |
| %SW55 STOPHIM | | | | | |
| %SW56 STOPMD | | | | | |
| %SW57 STOP YEAR | | | | | |
| %SW58 STOPDAY | | | | | |
| %SW59 ADJDATETIME | Adjustment of current date | Contains two 8 bit series to adjust the current date. The action is always performed on the rising edge of the bit. This word is enabled by bit %S59. Illustration:  

<table>
<thead>
<tr>
<th>Bits</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Day of the week (1)</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>Seconds</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Minutes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hours</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Days</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Months</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Years</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Centuries</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Initial state</td>
<td>0</td>
<td>YES</td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Description of System Words %SW70 to %SW99

<table>
<thead>
<tr>
<th>Word Symbol</th>
<th>Function</th>
<th>Description</th>
<th>Initial state</th>
<th>Quantum</th>
<th>Premium Atrium</th>
</tr>
</thead>
<tbody>
<tr>
<td>%SW70 WEEKOFYEAR</td>
<td>Real-time clock function</td>
<td>System word containing the number of the week in the year: 1 to 52.</td>
<td>-</td>
<td>YES</td>
<td>YES</td>
</tr>
</tbody>
</table>
| %SW71 KEY_SWITCH | Position of the switches on the Quantum front panel | This word provides the image of the positions of the switches on the front panel of the Quantum processor. This word is updated automatically by the system.  
• %SW71.0 = 1 switch in the "Memory protected" position.  
• %SW71.1 = 1 switch in the "STOP" position.  
• %SW71.2 = 1 switch in the "START" position.  
• %SW71.8 = 1 switch in the "MEM" position.  
• %SW71.9 = 1 switch in the "ASCII" position.  
• %SW71.10 = 1 switch in the "RTU" position.  
• %SW71.3 to 7 and 11 to 15 unused. | 0 | YES | NO |
| %SW75 TIMEREVTNB | Timer-type event counter | This word contains the number timer type events in the queue. (1) not available on the following processors: TSX 57 1*2/3/4/5*. | 0 | YES | YES (1) |
| %SW76 DLASTREG | Diagnostics function: recording | Result of the last registration  
• =0 if the recording was successful  
• = 1 if the diagnostics buffer has not been configured  
• = 2 if the diagnostics buffer is full | 0 | YES | YES |
| %SW77 DLASTDereg | Diagnostics function: non-recording | Result of the last deregistration  
• =0 if the non-recording was successful  
• = 1 if the diagnostics buffer has not been configured  
• = 21 if the error identifier is invalid  
• = 22 if the error has not been recorded | 0 | YES | YES |
<table>
<thead>
<tr>
<th>Word symbol</th>
<th>Function</th>
<th>Description</th>
<th>Initial state</th>
<th>Quantum</th>
<th>Premium Atrium</th>
</tr>
</thead>
<tbody>
<tr>
<td>%SW78 DNBERRBUF</td>
<td>Diagnostics function: number of errors</td>
<td>Number of errors currently in the diagnostics buffer</td>
<td>0</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>%SW80 MSGCNT0</td>
<td>Message management</td>
<td>★ %SW80: Number of messages sent by the system to the terminal port. ★ %SW81: Number of messages received by the system from the terminal port. ★ %SW82: Number of messages sent by the system to the PCMCIA module. ★ %SW83: Number of messages received by the system from the PCMCIA module. (1) The words %SW82 and %SW83 are not available on Quantum.</td>
<td>0</td>
<td>YES (1)</td>
<td>YES</td>
</tr>
<tr>
<td>%SW81 MSGCNT1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%SW82 MSGCNT2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%SW83 MSGCNT3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%SW84 MSGCNT4</td>
<td>Telegram management</td>
<td>★ %SW84: Number of telegrams sent by the system. ★ %SW85: Number of telegrams received by the system. ★ %SW86: Number of messages refused by the system.</td>
<td>0</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td>%SW85 MSGCNT5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%SW86 MSGCNT6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%SW87 MSTSERVCNT</td>
<td>Communication flow management</td>
<td>★ %SW87: Number of requests processed by synchronous server per master (MAST) task cycle. ★ %SW88: Number of requests processed by asynchronous server per master (MAST) task cycle. ★ %SW89: Number of requests processed by server functions (immediately) per master (MAST) task cycle. (1) The words %SW88 and %SW89 are not available on Quantum.</td>
<td>0</td>
<td>YES (1)</td>
<td>YES</td>
</tr>
<tr>
<td>%SW88 ANNSERVCNT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%SW89 APPSERVCNT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
System objects

<table>
<thead>
<tr>
<th>Word Symbol</th>
<th>Function</th>
<th>Description</th>
<th>Initial state</th>
<th>Quantum</th>
<th>Premium Atrium</th>
</tr>
</thead>
<tbody>
<tr>
<td>%SW90 MAXREQNB</td>
<td>Maximum number of requests processed per master task cycle</td>
<td>This word is used to set a maximum number of requests (all protocols included: UNI-TE, Modbus, etc.) which can be processed by the PLC per master task cycle. When the CPU is the server: This number of requests must be between 2 (minimum) and N+4 (maximum). N: number differs depending on the model. ● TSX 57 1•: N = 4 (mini=2, maxi=4+4=8) ● TSX 57 2•: N = 8 ● TSX 57 3•: N = 12 ● TSX 57 4•: N = 16 ● TSX 57 5•: N = 16 ● 140 CPU 31•/43•/53•/: N = 5 ● 140 CPU 6•: N = 20 ● For Quantum 140 CPU 31•/43•/53•/: this value is set and is equal to 5. With Fipio, the Premium range processes up to 7 requests. When the CPU is the client: N: number differs depending on the model. ● TSX 57 5•: N = 64 ● For the other CPUs mentioned above this value is equal to 80. The value 0 will not work. If a value that is outside of the range is entered, it is the value N that is taken into account.</td>
<td>0</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>%SW99 INPUTADR/SWAP</td>
<td>Communication redundancy management</td>
<td>Word used to manage the redundancy of network modules. When a problem is detected on a communication module used to access a network number x (X-WAY), it is possible to switch to another communication module (connected to the same network) by entering the network number in the %SW99 word. The %SW99 word is reset to 0 by the system.</td>
<td>0</td>
<td>NO</td>
<td>YES</td>
</tr>
</tbody>
</table>
### Description of system words %SW108 to %SW116

<table>
<thead>
<tr>
<th>Word Symbol</th>
<th>Function</th>
<th>Description</th>
<th>Initial state</th>
<th>Quantum</th>
<th>Premium Atrium</th>
</tr>
</thead>
<tbody>
<tr>
<td>%SW108 FORCEDIOIM</td>
<td>Number of forced I/O module bits</td>
<td>This system word counts the number of forced I/O module bits. This word is incremented for every forcing, and decremented for every unforcing.</td>
<td>0</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>%SW109 FORCEDANA</td>
<td>Number of forced analog channels</td>
<td>This system word counts the number of forced analog channels. This word is incremented for every forcing, and decremented for every unforcing.</td>
<td>0</td>
<td>NO</td>
<td>YES</td>
</tr>
</tbody>
</table>
| %SW116 REMIOERR | Fipio I/O error | Normally set to 0, each bit for this word signifies the Fipio exchange status of the exchange in which it is being tested. This word is to be reset to 0 by the user. More details on bits of word %SW116:  
  - %SW116.0 = 1 explicit exchange error (variable has not been exchanged on the bus),  
  - %SW116.1 = 1 time-out on an explicit exchange (no reply at the end of time-out),  
  - %SW116.2 = 1 maximum number of explicit exchanges achieved at the same time,  
  - %SW116.3 = 1 a frame is invalid,  
  - %SW116.4 = 1 the length of frame received is greater than the length that was declared,  
  - %SW116.5 = reserved on 0,  
  - %SW116.6 = 1 a frame is invalid, or an agent is initializing,  
  - %SW116.7 = 1 absence of a configured device,  
  - %SW116.8 = 1 channel fault (at least one device channel is indicating a fault),  
  - %SW116.9 to 15 = reserved on 0. | - | NO | YES |
### Description of system words %SW124 to %SW127

**Detailed description**

<table>
<thead>
<tr>
<th>Word Symbol</th>
<th>Function</th>
<th>Description</th>
<th>Initial state</th>
<th>Quantum</th>
<th>Premium</th>
</tr>
</thead>
</table>
| %SW124 CPUERR | Type of system fault  | The last type of system fault encountered is written into this word by the system (these codes are unchanged on a cold restart):  
  - 16#30: system code fault.  
  - 16#53: time-out fault during I/O exchanges.  
  - 16#60 to 64: stack overrun.  
  - 16#90: system switch fault: Unforeseen IT. | -             | YES     | YES     |
| %SW125 BLKERRTYPE | Last fault detected  | The code of the last fault detected is given in this word:  
  The following error codes cause the PLC to stop if %S78 is set to 1. %S15, %S18 and %S20 are always activated independently of %S78  
  - 16#2258: execution of HALT instruction.  
  - 16#DE87: calculation error on floating-point numbers (%S18, these errors are listed in the word %SW17).  
  - 16#DEB0: Watchdog overflow (%S11).  
  - 16#DEF0: division by 0 (%S18).  
  - 16#DEF1: character string transfer error (%S15).  
  - 16#DEF2: arithmetic error; %S18,  
  - 16#DEF3: index overflow (%S20).  
  **Note:** the following codes: 16#8xxx and 16#7xxx do not stop the application and indicate an error on function blocks. | -             | YES     | YES     |
### Blocking fault address

Address of the instruction that generated the application blocking fault. For 16 bit processors, TSX P57 1••/2••

- `%SW126` contains the offset for this address.
- `%SW127` contains the segment number for this address.

For 32 bit processors:
- `%SW126` contains the least significant word for this address.
- `%SW127` contains the most significant word for this address.

<table>
<thead>
<tr>
<th>Word Symbol</th>
<th>Function</th>
<th>Description</th>
<th>Initial state</th>
<th>Quantum</th>
<th>Premium Atrium</th>
</tr>
</thead>
<tbody>
<tr>
<td>ERRADDR0</td>
<td>Blocking fault instruction address</td>
<td></td>
<td>0</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>%SW127</td>
<td>ERRADDR1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
6.3 Atrium/Premium-specific system words

This section describes the system words %SW128 to %SW167 for Premium and Atrium PLCs.

This section contains the following topics:

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description of system words %SW128 to %SW143</td>
<td>162</td>
</tr>
<tr>
<td>Description of System Words %SW144 to %SW146</td>
<td>163</td>
</tr>
<tr>
<td>Description of System Words %SW147 to %SW152</td>
<td>165</td>
</tr>
<tr>
<td>Description of system word %SW153</td>
<td>166</td>
</tr>
<tr>
<td>Description of system word %SW154</td>
<td>168</td>
</tr>
<tr>
<td>Description of Premium/Atrium system words %SW155 to %SW167</td>
<td>169</td>
</tr>
</tbody>
</table>
Description of system words %SW128 to %SW143

### Detailed description

Each bit in this group of words indicates the state of a device connected to the Fipio bus. Normally set to 1, the presence of a 0 in one of these bits indicates the occurrence of a fault on this connection point. For a non-configured connection point, the corresponding bit is always 1.

<table>
<thead>
<tr>
<th>Word Symbol</th>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>%SW128...143</td>
<td>Faulty Fipio</td>
<td>Each bit in this group of words indicates the state of a device connected to the Fipio bus. Normally set to 1, the presence of a 0 in one of these bits indicates the occurrence of a fault on this connection point. For a non-configured connection point, the corresponding bit is always 1.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bit 0</th>
<th>Bit 1</th>
<th>Bit 2</th>
<th>Bit 3</th>
<th>Bit 4</th>
<th>Bit 5</th>
<th>Bit 6</th>
<th>Bit 7</th>
<th>Bit 8</th>
<th>Bit 9</th>
<th>Bit 10</th>
<th>Bit 11</th>
<th>Bit 12</th>
<th>Bit 13</th>
<th>Bit 14</th>
<th>Bit 15</th>
</tr>
</thead>
<tbody>
<tr>
<td>%SW128</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
</tr>
<tr>
<td>%SW129</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
</tr>
<tr>
<td>%SW130</td>
<td>32</td>
<td>33</td>
<td>34</td>
<td>35</td>
<td>36</td>
<td>37</td>
<td>38</td>
<td>39</td>
<td>40</td>
<td>41</td>
<td>42</td>
<td>43</td>
<td>44</td>
<td>45</td>
<td>46</td>
</tr>
<tr>
<td>%SW131</td>
<td>48</td>
<td>49</td>
<td>50</td>
<td>51</td>
<td>52</td>
<td>53</td>
<td>54</td>
<td>55</td>
<td>56</td>
<td>57</td>
<td>58</td>
<td>59</td>
<td>60</td>
<td>61</td>
<td>62</td>
</tr>
<tr>
<td>%SW132</td>
<td>64</td>
<td>65</td>
<td>66</td>
<td>67</td>
<td>68</td>
<td>69</td>
<td>70</td>
<td>71</td>
<td>72</td>
<td>73</td>
<td>74</td>
<td>75</td>
<td>76</td>
<td>77</td>
<td>78</td>
</tr>
<tr>
<td>%SW133</td>
<td>80</td>
<td>81</td>
<td>82</td>
<td>83</td>
<td>84</td>
<td>85</td>
<td>86</td>
<td>87</td>
<td>88</td>
<td>89</td>
<td>90</td>
<td>91</td>
<td>92</td>
<td>93</td>
<td>94</td>
</tr>
<tr>
<td>%SW134</td>
<td>96</td>
<td>97</td>
<td>98</td>
<td>99</td>
<td>100</td>
<td>101</td>
<td>102</td>
<td>103</td>
<td>104</td>
<td>105</td>
<td>106</td>
<td>107</td>
<td>108</td>
<td>109</td>
<td>110</td>
</tr>
<tr>
<td>%SW135</td>
<td>112</td>
<td>113</td>
<td>114</td>
<td>115</td>
<td>116</td>
<td>117</td>
<td>118</td>
<td>119</td>
<td>120</td>
<td>121</td>
<td>122</td>
<td>123</td>
<td>124</td>
<td>125</td>
<td>126</td>
</tr>
<tr>
<td>%SW136</td>
<td>128</td>
<td>129</td>
<td>130</td>
<td>131</td>
<td>132</td>
<td>133</td>
<td>134</td>
<td>135</td>
<td>136</td>
<td>137</td>
<td>138</td>
<td>139</td>
<td>140</td>
<td>141</td>
<td>142</td>
</tr>
<tr>
<td>%SW137</td>
<td>144</td>
<td>145</td>
<td>146</td>
<td>147</td>
<td>148</td>
<td>149</td>
<td>150</td>
<td>151</td>
<td>152</td>
<td>153</td>
<td>154</td>
<td>155</td>
<td>156</td>
<td>157</td>
<td>158</td>
</tr>
<tr>
<td>%SW138</td>
<td>160</td>
<td>161</td>
<td>162</td>
<td>163</td>
<td>164</td>
<td>165</td>
<td>166</td>
<td>167</td>
<td>168</td>
<td>169</td>
<td>170</td>
<td>171</td>
<td>172</td>
<td>173</td>
<td>174</td>
</tr>
<tr>
<td>%SW139</td>
<td>176</td>
<td>177</td>
<td>178</td>
<td>179</td>
<td>180</td>
<td>181</td>
<td>182</td>
<td>183</td>
<td>184</td>
<td>185</td>
<td>186</td>
<td>187</td>
<td>188</td>
<td>189</td>
<td>190</td>
</tr>
<tr>
<td>%SW140</td>
<td>192</td>
<td>193</td>
<td>194</td>
<td>195</td>
<td>196</td>
<td>197</td>
<td>198</td>
<td>199</td>
<td>200</td>
<td>201</td>
<td>202</td>
<td>203</td>
<td>204</td>
<td>205</td>
<td>206</td>
</tr>
<tr>
<td>%SW141</td>
<td>208</td>
<td>209</td>
<td>210</td>
<td>211</td>
<td>212</td>
<td>213</td>
<td>214</td>
<td>215</td>
<td>216</td>
<td>217</td>
<td>218</td>
<td>219</td>
<td>220</td>
<td>221</td>
<td>222</td>
</tr>
<tr>
<td>%SW142</td>
<td>224</td>
<td>225</td>
<td>226</td>
<td>227</td>
<td>228</td>
<td>229</td>
<td>230</td>
<td>231</td>
<td>232</td>
<td>233</td>
<td>234</td>
<td>235</td>
<td>236</td>
<td>237</td>
<td>238</td>
</tr>
<tr>
<td>%SW143</td>
<td>240</td>
<td>241</td>
<td>242</td>
<td>243</td>
<td>244</td>
<td>245</td>
<td>246</td>
<td>247</td>
<td>248</td>
<td>249</td>
<td>250</td>
<td>251</td>
<td>252</td>
<td>253</td>
<td>254</td>
</tr>
</tbody>
</table>
### Description of System Words %SW144 to %SW146

<table>
<thead>
<tr>
<th>Word Symbol</th>
<th>Function</th>
<th>Description</th>
<th>Initial state</th>
<th>Quantum</th>
<th>Premium Atrium</th>
</tr>
</thead>
<tbody>
<tr>
<td>%SW144</td>
<td>BAOPMOD</td>
<td>Fipio bus arbiter function operating mode</td>
<td>0</td>
<td>NO</td>
<td>YES</td>
</tr>
</tbody>
</table>

This system word is used to start and stop the bus arbiter function and the producer / consumer function. It can modify the starting, automatic and manual modes of the bus in the event of a stop.

- %SW144.0
  - = 1: producer / consumer function in RUN.
  - = 0: producer / consumer function in STOP (no variables are exchanged on the bus).
- %SW144.1
  - = 1: bus arbiter is in RUN 0.
  - = 0: bus arbiter is in STOP (no variables or message scanning is carried out on the bus).
- %SW144.2
  - = 1: automatic start in the event of an automatic bus stop.
  - = 0: manual start in the event of an automatic bus stop.
- %SW144.3 to 15 reserved, %SW144.3 = 1, %SW144.4 to 15 = 0.
Modification of Fipio Bus Arbiter Parameters

The bits are set to 1 by the user, and reset to 0 by the system when initialization has been carried out.
- %SW145.0 = 1: modification of the priority of the bus arbiter; the most significant byte for this system word contains the value of the priority of the bus arbiter which is to be applied to the bus.
- %SW145.1 and %SW145.2 are reserved.
- %SW145.3 to %SW145.7 reserved on 0.
- %SW145.8 to %SW145.15: this byte contains the value which is applied to the bus, according to the value of bit 0.

These parameters can be modified when the bus arbiter is in RUN, but for them to be taken into account by the application, the BA must be stopped then restarted.

Fipio bus arbiter function display

The least significant byte indicates the status of the producer / consumer function. The most significant byte indicates the status of the bus arbiter function.

Byte value:
- 16#00: the function does not exist (no Fipio application).
- 16#70: the function has been initialized but is not operational (in STOP).
- 16#F0: the function is currently being executed normally (in RUN).

Initial state Quantum Premium Atrium

<table>
<thead>
<tr>
<th>Word Symbol</th>
<th>Function</th>
<th>Description</th>
<th>Initial state</th>
<th>Quantum</th>
<th>Premium</th>
</tr>
</thead>
<tbody>
<tr>
<td>%SW145</td>
<td>BAPARAM</td>
<td>Modification of Fipio Bus Arbiter Parameters</td>
<td>0 NO YES</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%SW146</td>
<td>BASTATUS</td>
<td>Fipio bus arbiter function display</td>
<td>0 NO YES</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**CAUTION**

Concerns words %SW144 and %SW145

Modifying these system words can cause the PLC station to stop.

Failure to follow this instruction can result in injury or equipment damage.
### Description of System Words %SW147 to %SW152

**Detailed Description**

Description of system words %SW147 to %SW152:

<table>
<thead>
<tr>
<th>Word Symbol</th>
<th>Function</th>
<th>Description</th>
<th>Initial state</th>
<th>Quantum</th>
<th>Premium</th>
</tr>
</thead>
<tbody>
<tr>
<td>%SW147 TCRMAST</td>
<td>MAST network cycle time</td>
<td>A value which is not zero indicates (in ms) the value of the MAST task network cycle time (TCRMAST).</td>
<td>0</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td>%SW148 TCRFAST</td>
<td>FAST network cycle time</td>
<td>A value which is not zero indicates (in ms) the value of the first FAST task network cycle time (TCRFAST).</td>
<td>0</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td>%SW150 NBFRSENT</td>
<td>Number of frames sent</td>
<td>This word indicates the number of frames sent by the Fipio channel manager.</td>
<td>0</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td>%SW151 NBFRREC</td>
<td>Number of frames received</td>
<td>This word indicates the number of frames received by the Fipio channel manager.</td>
<td>0</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td>%SW152 NBRESENTMSG</td>
<td>Number of messages resent</td>
<td>This word indicates the number of messages resent by the Fipio channel manager.</td>
<td>0</td>
<td>NO</td>
<td>YES</td>
</tr>
</tbody>
</table>
Description of system word %SW153

<table>
<thead>
<tr>
<th>Word Symbol</th>
<th>Function</th>
<th>Description</th>
<th>Initial state</th>
<th>Quantum</th>
<th>Premium Atrium</th>
</tr>
</thead>
<tbody>
<tr>
<td>%SW153</td>
<td>List of Fipio manager faults</td>
<td>Each bit is set to 1 by the system, and reset to 0 by the user. See the list below.</td>
<td>0</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td>FipioERR0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Description of the bits

- bit 0 = "overrun station fault": corresponds to loss of a MAC symbol while receiving – this is linked to the receiver reacting too slowly.
- bit 1 = "message refusal fault": indicates that a message with acknowledgment was refused, or that it was not acknowledged in the first place. Receiving MAC.
- bit 3 = "underrun station fault": corresponds to the station being unable to respect transfer speed on the network.
- bit 4 = "physical layer fault": corresponds to a prolonged transmission absence in the physical layer.
- bit 5 = "non-echo fault": corresponds to a fault which occurs when the transmitter is currently sending, with a transmission current in the operating range, and when at the same time there is detection of an absence of signal on the same channel.
- bit 6 = "talking fault": corresponds to a fault whereby the transmitter is controlling the line for longer than the maximum set operating limit. This fault is caused, for example, by deterioration of the modulator, or by a faulty data link layer.
- bit 7 = "undercurrent fault": corresponds to a fault whereby the transmitter generates, when solicited, a current weaker than the minimum set operating limit. This fault is caused by increased line impedance (e.g. open line, etc.).
- bit 8 = "pierced frame fault": indicates that a pause has been received in the frame body, after identifying a delimiter at the start of the frame, and before identifying a delimiter at the end of the frame. The appearance of a pause in normal operating conditions takes place after a delimiter has been identified at the end of a frame.
- bit 9 = "Receiving frame CRC fault": indicates that the CRC calculated on a normally received frame and the CRC contained within this frame have different values.
- bit 10 = "Receiving frame code fault": indicates that certain symbols, belonging exclusively to delimitation sequences at the start and end of frames, have been received within the body of the frame.
- bit 11 = "received frame length fault": more than 256 bytes have been received for the frame body.
- bit 12 = "unknown frame type received": within the frame body, the first byte identifies the type of frame link. A set number of frame types are defined in the WorldFip standard link protocol. Any other code found within a frame is therefore an unknown frame type.
- bit 13 = "a truncated frame has been received": a frame section is recognized by a sequence of symbols delimiting the end of the frame, while the destination station awaits the arrival of a delimiter sequence for the start of the frame.
- bit 14 = "unused, non-significant value".
- bit 15 = "unused, non-significant value"
Description of system word %SW154

<table>
<thead>
<tr>
<th>Word Symbol</th>
<th>Function</th>
<th>Description</th>
<th>Initial State</th>
<th>Quantum</th>
<th>Premium Atrium</th>
</tr>
</thead>
<tbody>
<tr>
<td>%SW154</td>
<td>List of Fipio channel manager faults</td>
<td>Each bit is set to 1 by the system and reset to 0 by the user. See the list below.</td>
<td>0</td>
<td>NO</td>
<td>YES</td>
</tr>
</tbody>
</table>

Description of the bits

- bit 0 = "aperiodic sequence time-out": indicates that the messages or aperiodic variables window has overflowed its limit within an elementary cycle of the macro-cycle.
- bit 1 = "refusal of messaging request": indicates that the message queue is saturated - for the time being the bus arbiter is in no position to latch onto nor to comply with a request.
- bit 2 = "urgent update command refused": indicates that the queue for urgent aperiodic variables exchange requests is saturated - for the time being the bus arbiter is in no position to latch onto nor to comply with a request.
- bit 3 = "non-urgent update command refused": indicates that the queue for non-urgent aperiodic variable exchange requests is saturated - for the time being the bus arbiter is in no position to latch onto nor to comply with a request.
- bit 4 = "pause fault": the bus arbiter has not detected any bus activity during a time period larger than the standardized WorldFip time period.
- bit 5 = "a network collision has occurred on identifier transmission": indicates activity on the network during theoretical pause periods. Between a transmission and awaiting a reply from the bus arbiter, there should be nothing circulating on the bus. If the bus arbiter detects activity, it will generate a collision fault (for example, when several arbiters are active at the same time on the bus).
- bit 6 = "bus arbiter overrun fault": indicates a conflict on accessing the bus arbiter station memory.
- bit 7 = "unused, non-significant value".
- bit 8 to bit 15 = reserved on 0.
Detailed description

Description of system words %SW155 to %SW167:

<table>
<thead>
<tr>
<th>Word Symbol</th>
<th>Function</th>
<th>Description</th>
<th>Initial state</th>
<th>Quantum</th>
<th>Premium Atrium</th>
</tr>
</thead>
<tbody>
<tr>
<td>%SW155 NBEXPLFIP</td>
<td>Number of explicit exchanges on Fipio</td>
<td>Number of explicit exchanges currently being processed on Fipio, carried out by instructions (READ_STS, REA_PARAM, etc.). Also takes into account the explicit exchanges carried out by requests (READ_IO_OBJECT, WRITE_IO_OBJECT, etc.).</td>
<td>0</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td>%SW160 to %SW167 PREMRACK0 to PREMRACK7</td>
<td>Operating status of the PLC modules</td>
<td>The words %SW160 to %SW167 are respectively associated with racks 0 to 7. Bits 0 to 15 of each of these words are associated with the modules located in positions 0 to 15 of these racks. The bit is set to 0 if the module is faulty, and set to 1 if the module is operating correctly. <strong>Example:</strong> %SW163.5 = 0 The module located in slot 5 of rack 3 is faulty.</td>
<td>0</td>
<td>NO</td>
<td>YES</td>
</tr>
</tbody>
</table>
6.4 Quantum-specific system words

Description

This section describes the system words %SW128 to %SW547 for Quantum PLCs.

What's in this Section?

This section contains the following topics:

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description of System Words %SW60 to %SW63:</td>
<td>171</td>
</tr>
<tr>
<td>Description of Quantum System Words %SW128 to %SW179</td>
<td>173</td>
</tr>
<tr>
<td>Description of Quantum system words %SW180 to %SW640</td>
<td>176</td>
</tr>
</tbody>
</table>
### Description of System Words %SW60 to %SW63:

<table>
<thead>
<tr>
<th>Word Symbol</th>
<th>Function Description</th>
<th>Initial state</th>
<th>Quantum</th>
<th>Premium Atrium</th>
</tr>
</thead>
<tbody>
<tr>
<td>%SW60</td>
<td>Quantum Hot Standby command register</td>
<td>0</td>
<td>YES</td>
<td>NO</td>
</tr>
<tr>
<td>HSB_CMD</td>
<td>Meaning of the different bits of the word %SW60:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• %SW60.0=1 invalidates the commands entered in the display (keypad).</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• %SW60.1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• =0 sets PLC A to OFFLINE mode.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• =1 sets PLC A to RUN mode.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• %SW60.2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• =0 sets PLC B to OFFLINE mode.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• =1 sets PLC B to RUN mode.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• %SW60.3=0 forces Standby PLC to OFFLINE mode if the applications are different.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• %SW60.4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• =0 authorizes an update of the firmware only after the application has stopped.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• =1 authorizes an update of the firmware without the application stopping.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• %SW60.5=1 application transfer request from the Standby to the primary.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• %SW60.8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• =0 address switch on Modbus port 1 during a primary swap.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• =1 no address switch on Modbus port 1 during a primary swap.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• %SW60.9</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• =0 address switch on Modbus port 2 during a primary swap.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• =1 no address switch on Modbus port 2 during a primary swap.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• %SW60.10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• =0 address switch on Modbus port 3 during a primary swap.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• =1 no address switch on Modbus port 3 during a primary swap.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Meaning of the different bits of the word %SW61:
- %SW61.0 and %SW61.1 PLC operating mode bits
  - %SW61.1=0, %SW61.0=1: OFFLINE mode.
  - %SW61.1=1, %SW61.0=0: primary mode.
  - %SW61.1=1, %SW61.0=1: secondary mode (Standby).
- %SW61.2 and %SW61.3 operating mode bits from the other PLC
  - %SW61.3=0, %SW61.2=1: OFFLINE mode.
  - %SW61.3=1, %SW61.2=0: primary mode.
  - %SW61.3=1, %SW61.2=1: secondary mode (Standby).
  - %SW61.3=0, %SW61.2=0: the remote PLC is not accessible (switched off, no communication).
- %SW61.4=0 the applications are identical on both PLCs.
- %SW61.5
  - =0 the PLC is used as unit A.
  - =1 the PLC is used as unit B.
- %SW61.14=0 unlocated variables being transferred.
- %SW61.15
  - =0 Hot Standby not activated.
  - =1 Hot Standby activated.

These 2 words may be written by the user in the first section of the master task. They are then transferred automatically from the Standby processor to update the primary PLC. They may be read on the primary PLC and be used as primary application parameters.
Description of Quantum System Words %SW128 to %SW179

### Detailed Description

Description of system words %SW128 to %SW179; these words are active on Quantum 140 CPU 6 PLCs:

<table>
<thead>
<tr>
<th>Word Symbol</th>
<th>Function</th>
<th>Description</th>
<th>Initial state</th>
<th>Quantum Premium Atrium</th>
</tr>
</thead>
<tbody>
<tr>
<td>%SW128 NB_P502_CNX</td>
<td>Number of connections open</td>
<td>The Most Significant Byte of this word indicates the number of TCP connections open on the Ethernet link TCP/IP port 502.</td>
<td>0</td>
<td>YES NO</td>
</tr>
<tr>
<td>%SW129 NB_DENIED_CNX</td>
<td>Number of connections refused</td>
<td>This word indicates the number of TCP connections refused on the Ethernet link TCP/IP port 502.</td>
<td>0</td>
<td>YES NO</td>
</tr>
<tr>
<td>%SW130 NB_P502_REF</td>
<td>Number of messages refused</td>
<td>This word indicates the number of TCP messages refused on the Ethernet link TCP/IP port 502.</td>
<td>0</td>
<td>YES NO</td>
</tr>
<tr>
<td>%SW132 and %SW133 NB_SENT_MSG</td>
<td>Number of messages sent</td>
<td>This double word %SDW132 indicates the number of messages sent on the Ethernet link TCP/IP port 502.</td>
<td>0</td>
<td>YES NO</td>
</tr>
<tr>
<td>%SW134 and %SW135 NB_RCV_MSG</td>
<td>Number of messages received</td>
<td>This double word %SDW134 indicates the number of messages received on the Ethernet link TCP/IP port 502.</td>
<td>0</td>
<td>YES NO</td>
</tr>
<tr>
<td>%SW136 NB_IOS_CNX</td>
<td>Number of devices scanned</td>
<td>This word indicates the number of devices scanned on the Ethernet link TCP/IP port 502.</td>
<td>0</td>
<td>YES NO</td>
</tr>
<tr>
<td>%SW137 NB_IOS_MSG</td>
<td>Number of IO Scanning messages received</td>
<td>This word indicates the number of messages received per second from the IO Scanning service on the Ethernet link TCP/IP port 502.</td>
<td>0</td>
<td>YES NO</td>
</tr>
<tr>
<td>%SW138 GLBD_ERROR</td>
<td>Global Data coherence error</td>
<td>Global Data coherence error</td>
<td>0</td>
<td>YES NO</td>
</tr>
</tbody>
</table>
### System objects

<table>
<thead>
<tr>
<th>Word Symbol</th>
<th>Function</th>
<th>Description</th>
<th>Initial state</th>
<th>Quantum</th>
<th>Premium Atrium</th>
</tr>
</thead>
<tbody>
<tr>
<td>%SW139</td>
<td>BW_GLAD_IOS</td>
<td>Global Data and IO Scanning service load</td>
<td>0</td>
<td>YES</td>
<td>NO</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The Least Significant Byte of this word measures the percentage of load relating to IO Scanning. The Most Significant Byte of this word measures the percentage of load relating to Global Data.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%SW140</td>
<td>BW_OTHER_MSG</td>
<td>Load for messaging service and other services</td>
<td>0</td>
<td>YES</td>
<td>NO</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The Least Significant Byte of this word measures the percentage of load relating to messaging. The Most Significant Byte of this word measures the percentage of load relating to other services.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%SW141 and %SW142</td>
<td>IP_ADDR</td>
<td>IP Address</td>
<td>0</td>
<td>YES</td>
<td>NO</td>
</tr>
<tr>
<td></td>
<td></td>
<td>This double word %SDW141 receives the IP address of the Ethernet link.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%SW143 and %SW144</td>
<td>IP_NETMASK</td>
<td>IP subnetwork mask</td>
<td>0</td>
<td>YES</td>
<td>NO</td>
</tr>
<tr>
<td></td>
<td></td>
<td>This double word %SDW143 receives the subnetwork mask of the Ethernet link.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%SW145 and %SW146</td>
<td>IP_GATEWAY</td>
<td>Default Ethernet gateway address</td>
<td>0</td>
<td>YES</td>
<td>NO</td>
</tr>
<tr>
<td></td>
<td></td>
<td>This double word %SDW145 receives the address of the default Ethernet gateway.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%SW147 to %SW149</td>
<td>MAC_ADDR1 to 3</td>
<td>MAC Addresses</td>
<td>0</td>
<td>YES</td>
<td>NO</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The words %SW147, %SW148, %SW149 code the addresses MAC 1, MAC 2 and MAC 3 respectively.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%SW150</td>
<td>FW_VERSION</td>
<td>Embedded software version</td>
<td>0</td>
<td>YES</td>
<td>NO</td>
</tr>
<tr>
<td></td>
<td></td>
<td>This word codes the embedded software version for 140 CPU 651-x0 PLCs.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Coprocessor version</td>
<td>This word codes the coprocessor version for 140 CPU 671-60 PLCs.</td>
<td>0</td>
<td>YES</td>
<td>NO</td>
</tr>
</tbody>
</table>
### System objects

#### %SW151 BOARD_STS

**Status of Ethernet link**

- Bit 0 = 0 if the Ethernet link is stopped
- Bit 1 = 0
- Bit 2: 0 = half duplex mode, 1 = full duplex
- Bit 3 = 0
- Bits 4 to 11: =7 for Quantum, =6 for Hot Standby Quantum
- Bit 12: 0 = 10 Mbits link, 1 = 100 Mbits link
- Bit 13: 0 = 10/100Base-TX link (twisted pair)
- Bit 14: 0
- Bit 15: 0 = Ethernet link inactive, 1 = Ethernet link active

Initial state: 0

- YES
- NO

#### %SW160 to %SW167 REFRESH_IO

**Device operating status determined by IO scanning**

The bits of words %SW160 to %SW167 are associated with devices scanned by IO scanning.

- Bit is set to 0 if the device is faulty, and set to 1 if the device is operating correctly.
- %SW160.0: device No. 1
- %SW160.1: device No. 2
- ............
- %SW167.15: device No. 128

Initial state: -

- YES
- NO

#### %SW168 to %SW171 VALID_GD

**Operating status of Global Data**

The bits of words %SW168 to %SW171 are associated with Global Data.

- Bit is set to 0 if the device is faulty, and set to 1 if the device is operating correctly.
- %SW168.0: device No. 1
- %SW168.1: device No. 2
- ............
- %SW171.15: device No. 64

Initial state: -

- YES
- NO
### Description of Quantum system words %SW180 to %SW640

#### Detailed description

<table>
<thead>
<tr>
<th>Word Symbol</th>
<th>Function</th>
<th>Description</th>
<th>Initial state</th>
<th>Quantum</th>
<th>Premium Atrium</th>
</tr>
</thead>
<tbody>
<tr>
<td>%SW180 to %SW339</td>
<td>Operating status of the PLC modules</td>
<td>The words %SW180 to %SW339 are associated with the PLC stations: 5 words per station, corresponding to racks 1 to 5 of each station. %SW180: operating status of the station 1 rack 1 modules. %SW181: operating status of the station 1 rack 2 modules. .......... %SW185: operating status of the station 2 rack 1 modules. %SW186: operating status of the station 2 rack 2 modules. .......... Bits 0 to 15 of each of these words are associated with the modules located in positions 16 to 1 of these racks. The bit is set to 0 if the module is faulty, and set to 1 if the module is operating correctly. <strong>Example:</strong> %SW185.5 =0 The module located in station 2 rack 1 slot 11 is faulty. <strong>Note:</strong> Modules 140 XBE 100 00 (See Unity Pro, Harware Reference Manual, Operation Guidelines) require special management.</td>
<td>0</td>
<td>YES</td>
<td>NO</td>
</tr>
<tr>
<td>%SW340 MB+DIOSLOT</td>
<td>Slot number of the processor with Modbus Plus link</td>
<td>Slot number of the processor with the built-in Modbus Plus link for connection to the first DIO network. The slot number is coded from 0 to 15.</td>
<td>-</td>
<td>YES</td>
<td>NO</td>
</tr>
</tbody>
</table>
### System objects

<table>
<thead>
<tr>
<th>Word Symbol</th>
<th>Function</th>
<th>Description</th>
<th>Initial state</th>
<th>Quantum</th>
<th>Premium</th>
<th>Atrium</th>
</tr>
</thead>
<tbody>
<tr>
<td>%SW341 to %SW404 MAB+IONEALTHi i=1..64</td>
<td>Operating status of the distributed station modules of the first DIO network</td>
<td>The words %SW341 to %SW404 are associated with the distributed stations (DIO): 64 words associated with the 64 DIO stations of the first network. %SW341: operating status of the station 1 modules. %SW342: operating status of the station 2 modules. ........ %SW404: operating status of the station 64 modules. Bits 0 to 15 of each of these words are associated with the modules located in positions 16 to 1 of these stations. The bit is set to 0 if the module is faulty, and set to 1 if the module is operating correctly. <strong>Example:</strong> %SW362.5 =0 The module located in station 22 slot 11 of the first DIO network is faulty. <strong>Note:</strong> For modules 140 CRA 2•• ••• the value of this bit is not significant, and is always set to 0.</td>
<td>-</td>
<td>YES</td>
<td>NO</td>
<td></td>
</tr>
<tr>
<td>%SW405 NOM1DIOSLOT</td>
<td>Slot number of the first interface module of the DIO network</td>
<td>Slot number of module 140 NAME 2•• for connection to the second DIO network. The slot number is coded from 0 to 15.</td>
<td>-</td>
<td>YES</td>
<td>NO</td>
<td></td>
</tr>
</tbody>
</table>
### System objects

<table>
<thead>
<tr>
<th>Word Symbol</th>
<th>Function</th>
<th>Description</th>
<th>Initial state</th>
<th>Quantum</th>
<th>Premium</th>
<th>Atrium</th>
</tr>
</thead>
<tbody>
<tr>
<td>%SW406 to %SW469 NOM1DIOHEALT Hi i=1..64</td>
<td>Operating status of the distributed station modules of the second DIO network</td>
<td>The words %SW406 to %SW469 are associated with the distributed stations (DIO): 64 words associated with the 64 DIO stations of the second network. %SW406: operating status of the station 1 modules. %SW407: operating status of the station 2 modules. .......... %SW469: operating status of the station 64 modules. Bits 0 to 15 of each of these words are associated with the modules located in positions 16 to 1 of these stations. The bit is set to 0 if the module is faulty, and set to 1 if the module is operating correctly. <strong>Example:</strong> %SW412.5 =0 The module located in station 7 slot 11 of the second DIO network is faulty. <strong>Note:</strong> For modules 140 CRA 2•• ••• the value of this bit is not significant, and is always set to 0.</td>
<td>-</td>
<td>YES</td>
<td>NO</td>
<td></td>
</tr>
<tr>
<td>%SW470 NOM2DIOSLOT</td>
<td>Slot number of the second interface module of the DIO network</td>
<td>Slot number of module 140 NAME 2•• for connection to the third DIO network. The slot number is coded from 0 to 15.</td>
<td>-</td>
<td>YES</td>
<td>NO</td>
<td></td>
</tr>
</tbody>
</table>
The words %SW471 to %SW534 are associated with the distributed stations (DIO): 64 words associated with the 64 DIO stations of the third network.

%SW471: operating status of the station 1 modules.
%SW472: operating status of the station 2 modules.

........
%SW534: operating status of the station 64 modules.

Bits 0 to 15 of each of these words are associated with the modules located in positions 16 to 1 of these stations.

The bit is set to 0 if the module is faulty, and set to 1 if the module is operating correctly.

**Example:** %SW520.5 =0
The module located in station 86 slot 11 of the third DIO network is faulty.

**Note:** For modules 140 CRA 2••• •• the value of this bit is not significant, and is always set to 0.
System objects

<table>
<thead>
<tr>
<th>Word Symbol</th>
<th>Function</th>
<th>Description</th>
<th>Initial state</th>
<th>Quantum</th>
<th>Premium</th>
</tr>
</thead>
<tbody>
<tr>
<td>%SW535</td>
<td>RIO ERR STAT</td>
<td>This word stores the start-up error code. This word is always set to 0 when the system is running; in the event of error, the PLC does not start up, but generates a stop status code.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| 01: I/O assignment length |
| 02: Remote I/O link number |
| 03: Number of stations in the I/O assignment |
| 04: I/O assignment checksum |
| 10: Length of the station descriptor |
| 11: I/O station number |
| 12: Station autonomy time |
| 13: ASCII port number |
| 14: Number of station modules |
| 15: Station already configured |
| 16: Port already configured |
| 17: More than 1024 output points |
| 18: More than 1024 input points |
| 20: Module slot address |
| 21: Module rack address |
| 22: Number of output bytes |
| 23: Number of input bytes |
| 25: First reference number |
| 26: Second reference number |
| 28: Internal bits outside the 16 bit range |
| 30: Unpaired odd output module |
| 31: Unpaired odd input module |
| 32: Unpaired odd module reference |
| 33: Reference 1x after register 3x |
| 34: Reference of dummy module already used |
| 35: Module 3x is not a dummy module |
| 36: Module 4x is not a dummy module |

<table>
<thead>
<tr>
<th>YES</th>
<th>NO</th>
<th>Premium Atrium</th>
</tr>
</thead>
<tbody>
<tr>
<td>YES</td>
<td>NO</td>
<td></td>
</tr>
</tbody>
</table>
The words %SW536 to %SW538 are the communication error words on cable A.
- %SW536:
  - most significant byte: counts framing errors
  - least significant byte: counts overruns of the DMA receiver.
- %SW537:
  - most significant byte: counts receiver errors
  - least significant byte: counts incorrect station receptions.
- %SW538:
  - %SW538.15 = 1, short frame
  - %SW538.14 = 1, no end-of-frame
  - %SW538.3 = 1, CRC error
  - %SW538.2 = 1, alignment error
  - %SW538.1 = 1, overrun error
  - %SW538.13 to 4 and 0 are unused

The words %SW539 to %SW541 are the communication error words on cable B.
- %SW539:
  - most significant byte: counts framing errors
  - least significant byte: counts overruns of the DMA receiver.
- %SW540:
  - most significant byte: counts receiver errors
  - least significant byte: counts incorrect station receptions.
- %SW541:
  - %SW541.15 = 1, short frame
  - %SW541.14 = 1, no end-of-frame
  - %SW541.3 = 1, CRC error
  - %SW541.2 = 1, alignment error
  - %SW541.1 = 1, overrun error
  - %SW541.13 to 4 and 0 are unused
The words %SW542 to %SW544 are the global communication error words.  
- %SW542: displays the global communication status.  
  - %SW542.15 = 1, communication operating correctly  
  - %SW542.14 = 1, communication on cable A operating correctly  
  - %SW542.13 = 1, communication on cable B operating correctly  
  - %SW542.11 to 8 = lost communications counter  
  - %SW542.7 to 0 = retry totalizer counter.  
- %SW543: is the global error totalizer counter for cable A:  
  - most significant byte: counts the errors detected  
  - least significant byte: counts "non responses".  
- %SW544: is the global error totalizer counter for cable B:  
  - most significant byte: counts the errors detected  
  - least significant byte: counts "non responses".

<table>
<thead>
<tr>
<th>Word Symbol</th>
<th>Function</th>
<th>Description</th>
<th>Initial state</th>
<th>Quantum</th>
<th>Premium Atrium</th>
</tr>
</thead>
</table>
| %SW542 GLOBERRCNT 0 %SW543 GLOBERRCNT 1 %SW544 GLOBERRCNT2 | Global communication status | The words %SW542 to %SW544 are the global communication error words.  
  - %SW542: displays the global communication status.  
    - %SW542.15 = 1, communication operating correctly  
    - %SW542.14 = 1, communication on cable A operating correctly  
    - %SW542.13 = 1, communication on cable B operating correctly  
    - %SW542.11 to 8 = lost communications counter  
    - %SW542.7 to 0 = retry totalizer counter.  
  - %SW543: is the global error totalizer counter for cable A:  
    - most significant byte: counts the errors detected  
    - least significant byte: counts "non responses".  
  - %SW544: is the global error totalizer counter for cable B:  
    - most significant byte: counts the errors detected  
    - least significant byte: counts "non responses". | - YES | NO |
The words %SW545 to %SW640 are used to describe the status of the decentralized stations. Three status words are used for each station.

- **%SW545**: displays the global communication status for station 1.
  - %SW545.15 = 1, communication operating correctly
  - %SW545.14 = 1, communication on cable A operating correctly
  - %SW545.13 = 1, communication on cable B operating correctly
  - %SW545.11 to 8 = lost communications counter
  - %SW545.7 to 0 = retry totalizer counter.

- **%SW546**: is the global error totalizer counter for cable A station 1:
  - most significant byte: counts the errors detected
  - least significant byte: counts "non responses".

- **%SW547**: is the global error totalizer counter for cable B station 1:
  - most significant byte: counts the errors detected
  - least significant byte: counts "non responses".

The words:
- %SW548 to 550 are assigned to station 2
- %SW551 to 553 are assigned to station 3
- .......
- %SW638 to 640 are assigned to station 32

<table>
<thead>
<tr>
<th>Word Symbol</th>
<th>Function</th>
<th>Description</th>
<th>Initial State</th>
<th>Quantum</th>
<th>Premium Atrium</th>
</tr>
</thead>
<tbody>
<tr>
<td>%SW545 to %SW640</td>
<td>Status of decentralized stations</td>
<td>The words %SW545 to %SW640 are used to describe the status of the decentralized stations. Three status words are used for each station.</td>
<td>-</td>
<td>YES</td>
<td>NO</td>
</tr>
</tbody>
</table>
For the PLCs where station 1 is reserved for local input/outputs, the status words %SW545 to %SW547 are used in the following way.

- %SW545: status of the local station.
  - %SW545.15 = 1, all modules are operating correctly.
  - %SW545.14 to 8 = unused, always set to 0.
  - %SW545.7 to 0 = number of times the module has appeared defective; the counter loops back at 255.
- %SW546: this is used as a counter for 16-bit input/output bus errors.
- %SW547: this is used as a counter for 16-bit input/output bus repetitions.

<table>
<thead>
<tr>
<th>Word Symbol</th>
<th>Function</th>
<th>Description</th>
<th>Initial state</th>
<th>Quantum</th>
<th>Premium Atrium</th>
</tr>
</thead>
</table>
| %SW545 to %SW547 | Status of the local station | For the PLCs where station 1 is reserved for local input/outputs, the status words %SW545 to %SW547 are used in the following way.  
  - %SW545: status of the local station.  
    - %SW545.15 = 1, all modules are operating correctly.  
    - %SW545.14 to 8 = unused, always set to 0.  
    - %SW545.7 to 0 = number of times the module has appeared defective; the counter loops back at 255.  
  - %SW546: this is used as a counter for 16-bit input/output bus errors.  
  - %SW547: this is used as a counter for 16-bit input/output bus repetitions. | - | NO | YES |
Data Description

At a Glance

In This Chapter
This part describes the different data types that can be used in a project, and how to implement them.

What's in this Part?
This part contains the following chapters:

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Chapter Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>General overview of data</td>
<td>187</td>
</tr>
<tr>
<td>8</td>
<td>Data types</td>
<td>195</td>
</tr>
<tr>
<td>9</td>
<td>Data instances</td>
<td>255</td>
</tr>
<tr>
<td>10</td>
<td>Data references</td>
<td>271</td>
</tr>
</tbody>
</table>
General overview of data

At a Glance

Subject of this Chapter
This chapter provides a general overview of:
- the different data types
- the data instances
- the data references

What's in this Chapter?
This chapter contains the following topics:

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>General</td>
<td>188</td>
</tr>
<tr>
<td>General overview of the data type families</td>
<td>189</td>
</tr>
<tr>
<td>Overview of data instances</td>
<td>191</td>
</tr>
<tr>
<td>Overview of the data references</td>
<td>192</td>
</tr>
<tr>
<td>Syntax rules for Type\Instance names</td>
<td>193</td>
</tr>
</tbody>
</table>
General overview of data

General

Introduction

A data item designates an object which can be instantiated such as:

- a variable,
- a function block.

Data is defined in three phases. These are:

- the data types phase, which specifies the following:
  - its category,
  - its format.
- the data instances phase, which defines its storage location and property, which is:
  - located, or
  - unlocated.
- the data references phase, which defines its means of access:
  - by immediate value,
  - by name,
  - by address.

Illustration

The following are the three phases that characterize the data:

Instantiating a data item consists in allocating it a memory slot according to its type.

Referencing a data item consists in defining a reference for it (name, address, etc.) allowing it to be accessed in the memory.
General overview of the data type families

Introduction
A data type is a piece of software information which specifies for a data item:

- its structure
- its format
- a list of its attributes
- its behavior

These properties are shared by all instances of the data type.

Illustration
The data type families are filed in different categories (dark gray):

```
Data types
  Variables
    EDT
    DDT
  Function blocks
    EFB
    DFB
```
## Definitions

### Data type families and their definitions

<table>
<thead>
<tr>
<th>Family</th>
<th>Definition</th>
</tr>
</thead>
</table>
| **EDT** | Elementary data types, such as:  
- Bool,  
- Int,  
- Byte,  
- Word,  
- Dword,  
- etc. |
| **DDT** | Derived Data Types, such as:  
- tables, which contain elements of the same type:  
  - Bool tables (EDT tables),  
  - tables of tables (DDT tables),  
  - tables of structures (DDT tables),  
- structures, which contain elements of the different types:  
  - Bool structures, Word structures, etc. (EDT structures),  
  - structures of tables, structures of structures, structures of tables/structures (DDT structures),  
  - Bool structures, table structures, etc. (EDT and DDT structures),  
  - structures concerning input/output data (IODDT structures),  
  - Structures containing variables that restore the status properties of an action or transition of a Sequential Function Chart. |
| **EFB** | Elementary Function Blocks written in C language. These comprise:  
- input variables,  
- internal variables,  
- output variables,  
- a processing algorithm. |
| **DFB** | Derived Function Blocks written in automation languages (Structured Text, Instruction List, etc.). These comprise:  
- input variables,  
- internal variables,  
- output variables,  
- a processing algorithm. |
Overview of data instances

Introduction

A data instance is an individual functional entity, which has all the characteristics of the data type to which it belongs.

One or more instances can belong to a data type.

The data instance can have a memory allocation that is:

- either unlocated
- or located

Illustration

Memory allocation of instances (dark gray) belonging to the different types:

Definitions

Definition of the memory allocations of data instances:

<table>
<thead>
<tr>
<th>Data instance</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unlocated</td>
<td>The memory slot of the instance is automatically allocated by the system and can change for each generation of the application. The instance is located by a name (symbol) chosen by the user.</td>
</tr>
<tr>
<td>Located</td>
<td>The memory slot of the instance is fixed, predefined and never changes. The instance is located by a name (symbol) chosen by the user and a topological address defined by the manufacturer, or by the topological address of the manufacturer only.</td>
</tr>
</tbody>
</table>
Overview of the data references

Introduction
A data reference allows the user to access the instance of this data either by:
- immediate value, true only for data of type EDT,
- address settings, true only for data of type EDT,
- name (symbol), true for all EDT, DDT, EFB, DFB data types, as well as for SFC objects.

Illustration
Possible data references according to data type (dark gray):

Diagram with nodes and edges illustrating data references for EDT and DDT.
Syntax rules for Type\Instance names

Introduction

The syntax of names of types and variables can be written up with or without the extended character set. This option can be selected in the Language extensions tab of the Tools->Project properties menu.

- With Allow extended character set option selected, the application is compliant with the IEC standard,
- With Allow extended character set option not selected, the user has a certain degree of flexibility, but the application is not compliant with the IEC standard.

The extended character set used for names entered into the application concerns:

- DFB (Derived Function Block) user function blocks or DDT (Derived data type),
- the internal elements composing a DFB/EFB function block data type or a derived data type (DDT),
- the data instances.

If the "Allow extended ..." checkbox is selected

The names entered are strings made up of alphanumeric characters and the Underscore character.

The rules are as follows:

- the first character of the name is an alphabetic character or an Underscore,
- two Underscore characters cannot be used consecutively.

If the "Allow extended ..." checkbox is not selected

The names entered are strings made up of alphanumeric characters and the Underscore character.

Additional characters are authorized such as:

- characters corresponding to ASCII codes 192 to 223 (except for code 215),
- characters corresponding to ASCII codes 224 to 255 (except for code 247).

The rules are as follows:

- the first character of the name is an alphanumeric character or an Underscore,
- Underscore characters can be used consecutively.
Data types

At a Glance

Subject of this Chapter
This chapter describes all the data types that can be used in an application.

What’s in this Chapter?
This chapter contains the following sections:

<table>
<thead>
<tr>
<th>Section</th>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.1</td>
<td>Elementary data types (EDT) in Binary format</td>
<td>196</td>
</tr>
<tr>
<td>8.2</td>
<td>Elementary data types (EDT) in BCD format</td>
<td>207</td>
</tr>
<tr>
<td>8.3</td>
<td>Elementary Data Types (EDT) in Real format</td>
<td>214</td>
</tr>
<tr>
<td>8.4</td>
<td>Elementary Data Types (EDT) in character string format</td>
<td>217</td>
</tr>
<tr>
<td>8.5</td>
<td>Elementary data types (EDT) in bit string format</td>
<td>220</td>
</tr>
<tr>
<td>8.6</td>
<td>Derived Data Types (DDT/IODDT)</td>
<td>224</td>
</tr>
<tr>
<td>8.7</td>
<td>Function block data types (DFB/EFB)</td>
<td>237</td>
</tr>
<tr>
<td>8.8</td>
<td>Generic Data Types (GDT)</td>
<td>245</td>
</tr>
<tr>
<td>8.9</td>
<td>Data types belonging to Sequential Function Charts (SFC)</td>
<td>248</td>
</tr>
<tr>
<td>8.10</td>
<td>Compatibility between data types</td>
<td>250</td>
</tr>
</tbody>
</table>
8.1 Elementary data types (EDT) in Binary format

At a Glance

Subject of this Section
This section describes Binary format data types. These are:
- Boolean types
- Integer types
- Time types

What's in this Section?
This section contains the following topics:

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overview of data types in binary format</td>
<td>197</td>
</tr>
<tr>
<td>Boolean Types</td>
<td>199</td>
</tr>
<tr>
<td>Integer types</td>
<td>204</td>
</tr>
<tr>
<td>The Time type</td>
<td>206</td>
</tr>
</tbody>
</table>
Overview of data types in binary format

<table>
<thead>
<tr>
<th>Introduction</th>
<th>The data types in Binary format belong to the EDT (Elementary data type) family, which includes single rather than derived data types (tables, structures, function blocks).</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reminder concerning Binary format</td>
<td>A data item in binary format is made up of one or more bits, where each of these is represented by one of the base 2 figures (0 or 1). The scale of the data item depends on the number of bit(s) of which it is made. Example:</td>
</tr>
<tr>
<td></td>
<td><img src="image" alt="Binary Format Example" /></td>
</tr>
<tr>
<td></td>
<td>A data item can be:</td>
</tr>
<tr>
<td></td>
<td>• signed. Here the highest ranking bit is the sign bit:</td>
</tr>
<tr>
<td></td>
<td>• 0 indicates a positive value</td>
</tr>
<tr>
<td></td>
<td>• 1 indicates a negative value</td>
</tr>
<tr>
<td></td>
<td>The range of values is:</td>
</tr>
<tr>
<td></td>
<td>([-2^{(\text{Bits} - 1)}, 2^{(\text{Bits} - 1)} - 1])</td>
</tr>
<tr>
<td></td>
<td>• unsigned. Here all the bits represent the value</td>
</tr>
<tr>
<td></td>
<td>The range of values is:</td>
</tr>
<tr>
<td></td>
<td>([0, 2^{\text{Bits} - 1}])</td>
</tr>
<tr>
<td></td>
<td>Bits=number of bits (format).</td>
</tr>
</tbody>
</table>
### Data types in binary format

List of data types:

<table>
<thead>
<tr>
<th>Type</th>
<th>Designation</th>
<th>Format (bits)</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOOL</td>
<td>Boolean</td>
<td>8</td>
<td>0=(False)</td>
</tr>
<tr>
<td>EBOOL</td>
<td>Boolean with forcing and edge detection</td>
<td>8</td>
<td>0=(False)</td>
</tr>
<tr>
<td>INT</td>
<td>Integer</td>
<td>16</td>
<td>0</td>
</tr>
<tr>
<td>DINT</td>
<td>Double integer</td>
<td>32</td>
<td>0</td>
</tr>
<tr>
<td>UINT</td>
<td>Unsigned integer</td>
<td>16</td>
<td>0</td>
</tr>
<tr>
<td>UDINT</td>
<td>Unsigned double integer</td>
<td>32</td>
<td>0</td>
</tr>
<tr>
<td>TIME</td>
<td>Unsigned double integer</td>
<td>32</td>
<td>T=0s</td>
</tr>
</tbody>
</table>
Boolean Types

At a Glance
There are two types of Boolean. These are:
- BOOL type, which contains only the value FALSE (=0) or TRUE (=1),
- EBOOL type, which contains the value FALSE (=0) or TRUE (=1) but also information concerning the management of falling or rising edges and forcing.

Principle of the BOOL type
This type takes up one memory byte, but the value is only stored in one bit. The default value for this type is FALSE (=0). It is accessible via an address containing the offset of the corresponding byte:
Address settings:

<table>
<thead>
<tr>
<th>Memory</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>V</td>
<td></td>
</tr>
</tbody>
</table>

In the case of the word extracted bit, it is accessible via an address containing the following information:
- an offset of the corresponding byte,
- the rank defining its position in the word
Address settings:

<table>
<thead>
<tr>
<th>Memory</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>V</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Rank</td>
</tr>
</tbody>
</table>
### Principle of the EBOOL type

This type takes up one memory byte which contains:

- the bit for the value (V),
- the history bit (H) for managing rising or falling edges. Each time the object's status changes, the value is copied to this bit,
- the bit containing the forcing status (F). Equal to 0 if the object is not forced and equal to 1 if the object is forced.

The default value for the bits associated with the EBOOL type is FALSE (=0).

It is accessible via an address specifying the offset of the corresponding byte:

Address settings:

![Address settings diagram]

### Historical Trend Diagram

The trend diagram below shows the main statuses of the value and history bits associated with the EBOOL type.

The rising edges of the value bit (1, 4) are copied to the history bit in the next PLC cycle (2, 5). The falling edges of the value bit (2, 7) are copied to the history bit of the next PLC cycle (3, 8).

![Trend Diagram]

### Trend Diagram and Forcing

The trend diagram below shows the main statuses of the value, history and forcing bits associated with the EBOOL type.
The rising edges of the value bit (1, 4) are copied to the history bit in the next PLC cycle (2, 5). The falling edges of the value bit (2, 7) are copied to the history bit in the next PLC cycle (3, 8). Between (4 and 5), the forcing bit equals 1, while the value and history bits remain at 1.

<table>
<thead>
<tr>
<th>Status change action</th>
<th>(1)</th>
<th>(2)</th>
<th>(3)</th>
<th>(4)</th>
<th>(5)</th>
<th>(6)</th>
<th>(7)</th>
<th>(8)</th>
<th>(9)</th>
<th>(10)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value bit</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Forcing bit</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>History bit</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### PLC variables belonging to Boolean types

<table>
<thead>
<tr>
<th>List of variables</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Variable</strong></td>
</tr>
<tr>
<td>Internal bit</td>
</tr>
<tr>
<td>System bit</td>
</tr>
<tr>
<td>Word extracted bit</td>
</tr>
<tr>
<td><strong>%I inputs</strong></td>
</tr>
<tr>
<td>Module error bit</td>
</tr>
<tr>
<td>Channel error bit</td>
</tr>
<tr>
<td>Input bit</td>
</tr>
<tr>
<td><strong>%Q outputs</strong></td>
</tr>
<tr>
<td>Output bit</td>
</tr>
</tbody>
</table>
Compatibility between BOOL and EBOOL

The operations authorized between these two types of variables are:
- value copying,
- address copying.

### Copies between types

<table>
<thead>
<tr>
<th></th>
<th>BOOL destination</th>
<th>EBOOL destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOOL source</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>EBOOL source</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

### Compatibility between the parameters of elementary functions (EF)

<table>
<thead>
<tr>
<th>Effective parameter (external to EF)</th>
<th>Formal BOOL parameter (internal to EF)</th>
<th>Formal EBOOL parameter (internal to EF)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOOL</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>EBOOL</td>
<td>In -&gt;Yes</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>In-Out -&gt;No</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Out -&gt; No</td>
<td></td>
</tr>
</tbody>
</table>

### Compatibility between the parameters of block functions (EFB/DFB)

<table>
<thead>
<tr>
<th>Effective parameter (external to FB)</th>
<th>Formal BOOL parameter (internal to FB)</th>
<th>Formal EBOOL parameter (internal to FB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOOL</td>
<td>Yes</td>
<td>In -&gt;Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>In-Out -&gt;No</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Out -&gt; Yes</td>
</tr>
<tr>
<td>EBOOL</td>
<td>In -&gt;Yes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>In-Out -&gt;No</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Out -&gt; Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

### Compatibility between array variables

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ARRAY[I..J] OF BOOL source</td>
<td>Yes</td>
</tr>
<tr>
<td>ARRAY[I..J] OF EBOOL source</td>
<td>No</td>
</tr>
<tr>
<td>ARRAY[I..J] OF EBOOL source</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Compatibility between static variables

<table>
<thead>
<tr>
<th></th>
<th>BOOL (%MW:xi) direct addressing</th>
<th>EBOOL (%Mi) direct addressing</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOOL (Var:BOOL) declared variable</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>EBOOL (Var:EBOOL) declared variable</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Compatibility

EBBOOL data types follow the rules below:

- A EBOOL type variable cannot be passed as a BOOL type input/output parameter.
- EBOOL arrays cannot be passed as ANY type parameters of an FFB.
- BOOL and EBOOL arrays are not compatible for instructing assignment (same rule as for FFB parameters).
- On Quantum:
  - EBOOL type located variables cannot be passed as EBOOL type input/output parameters.
  - EBOOL arrays cannot be passed as parameters of a DFB.
Data types

Integer types

At a Glance

Integer types are used to represent a value in different bases. These are:

- base 10 (decimal) by default. Here the value is signed or unsigned depending on the integer type
- base 2 (binary). Here the value is unsigned and the prefix is 2#
- base 8 (octal). Here the value is unsigned and the prefix is 8#
- base 16 (octal). Here the value is unsigned and the prefix is 16#

**Note:** In decimal representation, if the chosen type is signed, the value can be preceded by the + or - signs (the sign + is optional).

Integer Type (INT)

Signed type with a 16 bit format.

This table shows the range in each base:

<table>
<thead>
<tr>
<th>Base</th>
<th>from...</th>
<th>to...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decimal (signed)</td>
<td>-32768</td>
<td>32767</td>
</tr>
<tr>
<td>Binary (unsigned)</td>
<td>2#0</td>
<td>2#1111111111111111</td>
</tr>
<tr>
<td>Octal (unsigned)</td>
<td>8#0</td>
<td>8#177777</td>
</tr>
<tr>
<td>Hexadecimal (unsigned)</td>
<td>16#0</td>
<td>16#FFFF</td>
</tr>
</tbody>
</table>

Double Integer Type (DINT)

Signed type with a 32 bit format.

This table shows the range in each base:

<table>
<thead>
<tr>
<th>Base</th>
<th>from...</th>
<th>to...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decimal (signed)</td>
<td>-2147483648</td>
<td>2147483647</td>
</tr>
<tr>
<td>Binary (unsigned)</td>
<td>2#0</td>
<td>2#11111111111111111111111111111111</td>
</tr>
<tr>
<td>Octal (signed)</td>
<td>8#0</td>
<td>8#377777777777</td>
</tr>
<tr>
<td>Hexadecimal (unsigned)</td>
<td>16#0</td>
<td>16#FFFFFFFF</td>
</tr>
</tbody>
</table>
### Unsigned Integer Type (UINT)

Unsigned type with a 16 bit format.

This table shows the range in each base:

<table>
<thead>
<tr>
<th>Base</th>
<th>from...</th>
<th>to...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decimal</td>
<td>0</td>
<td>65535</td>
</tr>
<tr>
<td>Binary</td>
<td>2#0</td>
<td>2#1111111111111111</td>
</tr>
<tr>
<td>Octal</td>
<td>8#0</td>
<td>8#177777</td>
</tr>
<tr>
<td>Hexadecimal</td>
<td>16#0</td>
<td>16#FFFF</td>
</tr>
</tbody>
</table>

### Unsigned Double Integer Type (UDINT)

Unsigned type with a 32 bit format.

This table shows the range in each base:

<table>
<thead>
<tr>
<th>Base</th>
<th>from...</th>
<th>to...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decimal</td>
<td>0</td>
<td>4294967295</td>
</tr>
<tr>
<td>Binary</td>
<td>2#0</td>
<td>2#11111111111111111111111111111111</td>
</tr>
<tr>
<td>Octal</td>
<td>8#0</td>
<td>8#377777777777</td>
</tr>
<tr>
<td>Hexadecimal</td>
<td>16#0</td>
<td>16#FFFFFFFF</td>
</tr>
</tbody>
</table>
The Time type

At a Glance

The Time type T# or TIME# is represented by an unsigned double integer (UDINT) (See Integer types, p. 204) type.

It expresses a duration in milliseconds, which approximately represents a maximum duration of 49 days.

The units of time authorized to represent the value are:

- days (D)
- hours (H)
- minutes (M)
- seconds (S)
- milliseconds (MS)

Entering a value

This table shows the possible ways of entering the maximum value of the Time type, according the authorized units of time:

<table>
<thead>
<tr>
<th>Diagram</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>T#4294967295MS</td>
<td>value in milliseconds</td>
</tr>
<tr>
<td>T#4294967S.295MS</td>
<td>value in seconds/milliseconds</td>
</tr>
<tr>
<td>T#71582M.47S.295MS</td>
<td>value in minutes/seconds/milliseconds</td>
</tr>
<tr>
<td>T#193H.2M.47S.295MS</td>
<td>value in hours/minutes/seconds/milliseconds</td>
</tr>
<tr>
<td>T#49D.17H.2M.47S.295MS</td>
<td>value in days/hours/minutes/seconds/milliseconds</td>
</tr>
</tbody>
</table>
8.2 Elementary data types (EDT) in BCD format

At a Glance

Subject of this section

This section describes BCD format (Binary Coded Decimal) data types. These are:

- Date type
- Time of Day type (TOD)
- Date and Time (DT) type

What's in this Section?

This section contains the following topics:

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overview of data types in BCD format</td>
<td>208</td>
</tr>
<tr>
<td>The Date type</td>
<td>210</td>
</tr>
<tr>
<td>The Time of Day (TOD) type</td>
<td>211</td>
</tr>
<tr>
<td>The Date and Time (DT) type</td>
<td>212</td>
</tr>
</tbody>
</table>
Overview of data types in BCD format

Introduction

The data types in BCD format belong to the EDT (Elementary data type) family, which includes single rather than derived data types (tables, structures, function blocks).

Reminder concerning BCD format

The Binary Coded Decimal (BCD) format is used to represent decimal numbers between 0 and 9 using a group of four bits (half-byte).

In this format, the four bits used to code the decimal numbers have a range of unused combinations.

Correspondence table:

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
</tr>
<tr>
<td>5</td>
<td>0101</td>
</tr>
<tr>
<td>6</td>
<td>0110</td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
</tr>
<tr>
<td>8</td>
<td>1000</td>
</tr>
<tr>
<td>9</td>
<td>1001</td>
</tr>
<tr>
<td></td>
<td>1010   (unused)</td>
</tr>
<tr>
<td></td>
<td>1011   (unused)</td>
</tr>
<tr>
<td></td>
<td>1100   (unused)</td>
</tr>
<tr>
<td></td>
<td>1101   (unused)</td>
</tr>
<tr>
<td></td>
<td>1110   (unused)</td>
</tr>
<tr>
<td></td>
<td>1111   (unused)</td>
</tr>
</tbody>
</table>

Example of coding using a 16 bit format:

<table>
<thead>
<tr>
<th>Decimal value</th>
<th>2</th>
<th>4</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>2450</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Binary value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0010 0100 0101 0000</td>
</tr>
</tbody>
</table>
### Example of coding using a 32 bit format:

<table>
<thead>
<tr>
<th>Decimal value</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>9</th>
<th>3</th>
<th>0</th>
<th>1</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>78993016</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Binary value</td>
<td>0111</td>
<td>1000</td>
<td>1001</td>
<td>1001</td>
<td>0011</td>
<td>0000</td>
<td>0001</td>
<td>0110</td>
</tr>
</tbody>
</table>

### Data types in BCD format

<table>
<thead>
<tr>
<th>Type</th>
<th>Designation</th>
<th>Scale (bits)</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATE</td>
<td>Date</td>
<td>32</td>
<td>D#1990-01-01</td>
</tr>
<tr>
<td>TIME_OF_DAY</td>
<td>Time of day</td>
<td>32</td>
<td>TOD#00:00:00</td>
</tr>
<tr>
<td>DATE_AND_TIME</td>
<td>Date and Time</td>
<td>64</td>
<td>DT#1990-01-01-00:00:00</td>
</tr>
</tbody>
</table>
The Date type

At a Glance

The Date type in 32 bit format contains the following information:

- the year coded in a 16-bit field (4 most significant half-bytes)
- the month coded in an 8-bit field (2 half bytes)
- the day coded in an 8-bit field (2 least significant half bytes)

Representation in BCD format of the date 2001-09-20:

<table>
<thead>
<tr>
<th>Year (2001)</th>
<th>Month (09)</th>
<th>Day (20)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0010 0000 0000 0001</td>
<td>0000 1001</td>
<td>0010 0000</td>
</tr>
</tbody>
</table>

Syntax rules

The Date type is entered as follows: D#<Year>-<Month>-<Day>

This table shows the lower/upper limits in each field:

<table>
<thead>
<tr>
<th>Field</th>
<th>Limits</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Year</td>
<td>[1990,2099]</td>
<td></td>
</tr>
<tr>
<td>Month</td>
<td>[01,12]</td>
<td>The left 0 is always displayed, but can be omitted at the time of entry</td>
</tr>
<tr>
<td>Day</td>
<td>[01,31]</td>
<td>For the months 01/03/05/07/08/10/12</td>
</tr>
<tr>
<td></td>
<td>[01,30]</td>
<td>For the months 04/06/09/11</td>
</tr>
<tr>
<td></td>
<td>[01,29]</td>
<td>For the month 02 (leap years)</td>
</tr>
<tr>
<td></td>
<td>[01,28]</td>
<td>For the month 02 (non leap years)</td>
</tr>
</tbody>
</table>

Example:

<table>
<thead>
<tr>
<th>Entry</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>D#2001-1-1</td>
<td>The left 0 of the month and the day can be omitted</td>
</tr>
<tr>
<td>d#1990-02-02</td>
<td>The prefix can be written in lower case</td>
</tr>
</tbody>
</table>
The Time of Day (TOD) type

At a Glance

The Time of Day type coded in 32 bit format contains the following information:

- the hour coded in an 8-bit field (2 most significant half-bytes)
- the minutes coded in an 8-bit field (2 half bytes)
- the seconds coded in an 8-bit field (2 half bytes)

Note: The 8 least significant bits are unused.

Representation in BCD format of the time of day 13:25:47:

<table>
<thead>
<tr>
<th>Hour (13)</th>
<th>Minutes (25)</th>
<th>Seconds (47)</th>
<th>Least significant byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>0001 0011</td>
<td>0010 0101</td>
<td>0100 0111</td>
<td>Unused</td>
</tr>
</tbody>
</table>

Syntax rules

The Time of Day type is entered as follows: TOD#<Hour>[:<Minutes>:<Seconds>

This table shows the lower/upper limits in each field:

<table>
<thead>
<tr>
<th>Field</th>
<th>Limits</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hour</td>
<td>[00,23]</td>
<td>The left 0 is always displayed, but can be omitted at the time of entry</td>
</tr>
<tr>
<td>Minute</td>
<td>[00,59]</td>
<td>The left 0 is always displayed, but can be omitted at the time of entry</td>
</tr>
<tr>
<td>Second</td>
<td>[00,59]</td>
<td>The left 0 is always displayed, but can be omitted at the time of entry</td>
</tr>
</tbody>
</table>

Example:

<table>
<thead>
<tr>
<th>Entry</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>TOD#1:59:0</td>
<td>The left 0 of the hours and seconds can be omitted</td>
</tr>
<tr>
<td>tod#23:10:59</td>
<td>The prefix can be written in lower case</td>
</tr>
<tr>
<td>Tod#0:0:0</td>
<td>The prefix can be mixed (lower/upper case)</td>
</tr>
</tbody>
</table>
The Date and Time (DT) type

At a Glance

The Date and Time type coded in 64 bit format contains the following information:

- The year coded in a 16-bit field (4 most significant half-bytes)
- The month coded in an 8-bit field (2 half bytes)
- The day coded in an 8-bit field (2 half bytes)
- The hour coded in an 8-bit field (2 half bytes)
- The minutes coded in an 8-bit field (2 half bytes)
- The seconds coded in an 8-bit field (2 half bytes)

Note: The 8 least significant bits are unused.

Example: Representation in BCD format of the date and time 2000-09-20:13:25:47.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0010 0000</td>
<td>0000 1001</td>
<td>0010 0000</td>
<td>0001 0011</td>
<td>0010 0101</td>
<td>0100 0111</td>
<td>Unused</td>
</tr>
</tbody>
</table>

Syntax rules

The Date and Time type is entered as follows:

DT#<Year><Month><Day><Hour><Minutes><Seconds>

This table shows the lower/upper limits in each field:

<table>
<thead>
<tr>
<th>Field</th>
<th>Limits</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Year</td>
<td>[1990,2099]</td>
<td></td>
</tr>
<tr>
<td>Month</td>
<td>[01,12]</td>
<td>The left 0 is always displayed, but can be omitted during entry</td>
</tr>
<tr>
<td>Day</td>
<td>[01,31]</td>
<td>For the months 01/03/05/07/09/10/12</td>
</tr>
<tr>
<td></td>
<td>[01,30]</td>
<td>For the months 04/06/09/11</td>
</tr>
<tr>
<td></td>
<td>[01,29]</td>
<td>For the month 02 (leap years)</td>
</tr>
<tr>
<td></td>
<td>[01,28]</td>
<td>For the month 02 (non leap years)</td>
</tr>
<tr>
<td>Hour</td>
<td>[00,23]</td>
<td>The left 0 is always displayed, but can be omitted during entry</td>
</tr>
<tr>
<td>Minute</td>
<td>[00,59]</td>
<td>The left 0 is always displayed, but can be omitted during entry</td>
</tr>
<tr>
<td>Second</td>
<td>[00,59]</td>
<td>The left 0 is always displayed, but can be omitted during entry</td>
</tr>
</tbody>
</table>
### Data types

Example:

<table>
<thead>
<tr>
<th>Entry</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>DT#2000-1-10-0:40:0</td>
<td>The left 0 of the month/hour/second can be omitted</td>
</tr>
<tr>
<td>dt#1999-12-31-23:59:59</td>
<td>The prefix can be written in lower case</td>
</tr>
<tr>
<td>Dt#1990-10-2-12:02:30</td>
<td>The prefix can be mixed (lower/upper case)</td>
</tr>
</tbody>
</table>
8.3 Elementary Data Types (EDT) in Real format

Presentation of the Real data type

Introduction

The data types in Binary format belong to the EDT (Elementary data type) family, which includes single rather than derived data types (tables, structures, function blocks).

Reminder concerning Real format

The Real format (floating point in ANSI/IEEE standard) is coded in 32 bit format which corresponds to the single decimal point floating numbers. The 32 bits representing the floating point value are organized in three distinct fields. These are:

- S, the sign bit which can have the value:
  - 0, for a positive floating point number,
  - 1, for a negative floating point number.
- e, the exponential coded in an 8 bit field,
- f, the fixed-point part coded in a 23 bit field.

Representation:

```
  30  23  22  0
  S   Exponential e   Fixed point part f
```

The value of the fixed-point part (Mantissa) is between ]1,0], and is calculated using the following formula:

\[ f = \sum_{i=0}^{22} (2^{i-23} \cdot b_i) \]

bi represents the value of the bit (0 or 1) of the corresponding rank (i=0 to 22)

Number types that can be represented

These are the numbers which are:

- standardized,
- non-standardized,
- of infinite values,
- with values +0 and -0.
This table gives the values in the different fields according to number type:

<table>
<thead>
<tr>
<th>e</th>
<th>f</th>
<th>S</th>
<th>Number type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0 or 1</td>
<td>standardized</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0 or 1</td>
<td>non-standardized DEN</td>
</tr>
<tr>
<td>255</td>
<td>0</td>
<td>0</td>
<td>+ infinity (INF)</td>
</tr>
<tr>
<td>255</td>
<td>0</td>
<td>1</td>
<td>- infinity (-INF)</td>
</tr>
<tr>
<td>255</td>
<td>0</td>
<td>0</td>
<td>0 or 1</td>
</tr>
<tr>
<td>255</td>
<td>0</td>
<td>0</td>
<td>0 or 1</td>
</tr>
</tbody>
</table>

**Note:** Standard IEC 559 defines two classes of NaN (not a number): QNAN and SNaN.
- QNAN: is a NaN whose bit 22 is set to 1,
- SNaN: is a NaN whose bit 22 is set to 0.

They behave as follows:
- QNAN do not trigger errors when they appear in operands of a function or an expression.
- SNaN trigger an error when they appear in operands of a function or an arithmetic expression (See %SW17 (See Detailed Description, p. 149) and %S18 (See Description of system bits %S15 to %S21, p. 135)).

This table gives the calculation formula of the value V of the floating-point number:

<table>
<thead>
<tr>
<th>Floating-point number type</th>
<th>Value V</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standardized</td>
<td>((-1)^S \times 2^{(e-127)} \times (1 + f))</td>
</tr>
<tr>
<td>Non-standardized (DEN)</td>
<td>((-1)^S \times 2^{-126} \times f)</td>
</tr>
</tbody>
</table>

**Note:** a real number between -1.175494e-38 and 1.175494e-38 is a DEN. When an operand is a DEN, the result is not significant (See %SW17 (See Detailed Description, p. 149) and %S18 (See Description of system bits %S15 to %S21, p. 135)).
The Real type

Presentation:

<table>
<thead>
<tr>
<th>Type</th>
<th>Scale (bits)</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>REAL</td>
<td>32</td>
<td>0.0</td>
</tr>
</tbody>
</table>

Range of values (grayed out parts):

-3.402824e+38  -1.175494e-38  0.0  1.175494e-38  3.402824e+38

When a calculation result is:
- between -1.175494e-38 and 1.175494e-38, it is a DEN,
- less than -3.402824e+38, the symbol -INF (for -infinite) is displayed,
- greater than +3.402824e+38, the symbol INF (for +infinite) is displayed,
- undefined (square root of a negative number), the symbol NAN is displayed.

Examples:

Representation of the floating point number -5.934113e-18.

<table>
<thead>
<tr>
<th>S sign</th>
<th>Exponential e</th>
<th>Fixed point part f</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>01000101</td>
<td>101101011101110000111001</td>
</tr>
</tbody>
</table>

Example of an equivalent entry:

<table>
<thead>
<tr>
<th>The entry...</th>
<th>equals to...</th>
<th>and...</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0.456</td>
<td>0.456</td>
<td>.456</td>
</tr>
<tr>
<td>-1.32e12</td>
<td>-132E10</td>
<td>-132e+13</td>
</tr>
<tr>
<td>1.0E+6</td>
<td>1000000.</td>
<td>1.e6</td>
</tr>
</tbody>
</table>
### 8.4 Elementary Data Types (EDT) in character string format

#### Overview of data types in character string format

**Introduction**

Data types in character string format belong to the EDT (Elementary data type) family, which includes single rather than derived data types (tables, structures, function blocks).

**The character string type**

The character string format is used to represent a string of ASCII characters, with each character being coded in an 8 bit format.

The characteristics of character string types are as follows:

- 16 characters by default in a string (excluding end of string characters),
- a string is composed of ASCII characters between 16#20 and 16#FF (hexadecimal representation).
- in an empty string, the end of string character (code ASCII "ZERO") is the first character of the string.
- the maximum size of a string is 65534 characters.

The size of the character string can be optimized during the definition of the type using the `STRING[<size>]` command, `<size>` being an unsigned integer UINT capable of defining a string of between 1 and 65534 ASCII characters.

**Syntax rules**

The entry is preceded by and ends with the quote character “” (ASCII code 16#27). The $ (dollar) sign is a special character, followed by certain letters which indicate:

- $L or $l, go to the next line (line feed),
- $N or $n, go to the start of the next line (new line),
- $P or $p, go to the next page,
- $R or $r, carriage return,
- $T or $t tabulation (Tab),
- $$, represents the character $ in a string,
- $', represents the quote character in a string.
### Data types

#### Examples

<table>
<thead>
<tr>
<th>Type</th>
<th>Entry</th>
<th>Contents of the string</th>
</tr>
</thead>
<tbody>
<tr>
<td>STRING</td>
<td>’ABCD’</td>
<td>ABDC•************ (16 characters)</td>
</tr>
<tr>
<td>STRING[4]</td>
<td>’john’</td>
<td>john•</td>
</tr>
<tr>
<td>STRING[10]</td>
<td>’It’s john’</td>
<td>It’s john•</td>
</tr>
<tr>
<td>STRING[5]</td>
<td>’ ’</td>
<td>••••••••••</td>
</tr>
<tr>
<td>STRING[5]</td>
<td>’$’</td>
<td>••••••••••</td>
</tr>
<tr>
<td>STRING[5]</td>
<td>’the number’</td>
<td>the no•</td>
</tr>
<tr>
<td>STRING[13]</td>
<td>’0123456789’</td>
<td>0123456789••••</td>
</tr>
<tr>
<td>STRING[5]</td>
<td>’$R$L’</td>
<td>&lt;cr&gt;&lt;lf&gt;••••</td>
</tr>
<tr>
<td>STRING[5]</td>
<td>’$1.00’</td>
<td>$1.00•</td>
</tr>
</tbody>
</table>
Strings and the ANY Pin

When you use a STRING type variable as an ANY type parameter, it is highly recommended to check that the size of the variable is less than the maximum declared size.

Example:

Use of STRING on the SEL function (Selector).

String1: STRING[8]
String2: STRING[4]
String3: STRING[4]

String1:= 'AAAAAAAA';
String3:= 'CC';
Scenario 1:
String2:= 'BBBB';
(* the size of the string is equal to the maximum declared size *)
String1:= SEL(FALSE, String2, String3);
(* the result will be: 'BBBBAAAA' *)
Scenario 2:
String2:= 'BBB';
(* the size of the string is less than the maximum declared size *)
String1:= SEL(FALSE, String2, String3);
(* the result will be: 'BBB' *)
8.5 Elementary data types (EDT) in bit string format

At a Glance

Subject of this Section
This section describes data types in bit string format. These are:
- Byte type
- Word type
- Dword type

What's in this Section?
This section contains the following topics:

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overview of data types in bit string format</td>
<td>221</td>
</tr>
<tr>
<td>Bit string types</td>
<td>222</td>
</tr>
</tbody>
</table>
Overview of data types in bit string format

Introduction
Data types in bit string format belong to the EDT (Elementary data type) family, which includes **single** rather than derived data types (tables, structure, function blocks).

Reminder concerning bit string format
The particularity of this format is that all of its component bits do not represent a numerical value, but a combination of separate bits.

The data belonging to types of this format can be represented in three bases. These are:
- hexadecimal (16#)
- octal (8#)
- binary (2#)

Data types in bit string format
Three data types:

<table>
<thead>
<tr>
<th>Type</th>
<th>Scale (bits)</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>BYTE</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>WORD</td>
<td>16</td>
<td>0</td>
</tr>
<tr>
<td>DWORD</td>
<td>32</td>
<td>0</td>
</tr>
</tbody>
</table>
Data types

Bit string types

The Byte type
The Byte type is coded in 8 bit format.
This table shows the lower/upper limits of the bases which can be used:

<table>
<thead>
<tr>
<th>Base</th>
<th>Lower limit</th>
<th>Upper limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hexadecimal</td>
<td>16#0</td>
<td>16#FF</td>
</tr>
<tr>
<td>Octal</td>
<td>8#0</td>
<td>8#777</td>
</tr>
<tr>
<td>Binary</td>
<td>2#0</td>
<td>2#11111111</td>
</tr>
</tbody>
</table>

Representation examples:

<table>
<thead>
<tr>
<th>Data content</th>
<th>Representation in one of the bases</th>
</tr>
</thead>
<tbody>
<tr>
<td>00001000</td>
<td>16#8</td>
</tr>
<tr>
<td>00110011</td>
<td>8#63</td>
</tr>
<tr>
<td>00110011</td>
<td>2#110011</td>
</tr>
</tbody>
</table>

The Word type
The Word type is coded in 16 bit format.
This table shows the lower/upper limits of the bases which can be used:

<table>
<thead>
<tr>
<th>Base</th>
<th>Lower limit</th>
<th>Upper limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hexadecimal</td>
<td>16#0</td>
<td>16#FFFF</td>
</tr>
<tr>
<td>Octal</td>
<td>8#0</td>
<td>8#177777</td>
</tr>
<tr>
<td>Binary</td>
<td>2#0</td>
<td>2#1111111111111111</td>
</tr>
</tbody>
</table>

Representation examples:

<table>
<thead>
<tr>
<th>Data content</th>
<th>Representation in one of the bases</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000000011010011</td>
<td>16#D3</td>
</tr>
<tr>
<td>1010101010101010</td>
<td>8#125252</td>
</tr>
<tr>
<td>0000000011010011</td>
<td>2#11010011</td>
</tr>
</tbody>
</table>
The Dword type

The Dword type is coded in 32 bit format.

This table shows the lower/upper limits of the bases which can be used:

<table>
<thead>
<tr>
<th>Base</th>
<th>Lower limit</th>
<th>Upper limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hexadecimal</td>
<td>16#0</td>
<td>16#FFFFFFFF</td>
</tr>
<tr>
<td>Octal</td>
<td>8#0</td>
<td>8#3777777777</td>
</tr>
<tr>
<td>Binary</td>
<td>2#0</td>
<td>2#11111111111111111111111111111111</td>
</tr>
</tbody>
</table>

Representation examples:

<table>
<thead>
<tr>
<th>Data content</th>
<th>Representation in one of the bases</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000000001010110110011011110</td>
<td>16#ADCDE</td>
</tr>
<tr>
<td>0000000000000001000000000000000000000000</td>
<td>8#200000</td>
</tr>
<tr>
<td>0000000000000010101011110011011110</td>
<td>2#10101011110011011110</td>
</tr>
</tbody>
</table>
8.6 Derived Data Types (DDT/IODDT)

At a Glance

Subject of this Section

This section presents Derived Data Types. These are:

- tables (DDT)
- structures
  - structures concerning input/output data (IODDT)
  - structures concerning other data (DDT)

What's in this Section?

This section contains the following topics:

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tables</td>
<td>225</td>
</tr>
<tr>
<td>Structures</td>
<td>228</td>
</tr>
<tr>
<td>Overview of the Derived Data Type family (DDT)</td>
<td>229</td>
</tr>
<tr>
<td>DDT: Memory occupancy</td>
<td>232</td>
</tr>
<tr>
<td>Overview of Input/Output Derived Data Types (IODDT)</td>
<td>235</td>
</tr>
</tbody>
</table>
Tables

What is a table? It is a data item that contains a set of data of the same type, such as:
- elementary data (EDT), for example:
  - a group of BOOL words,
  - a group of UINT integer words,
  - etc.
- derived data (DDT) such as:
  - a group of WORD tables,
  - a group of structures,
  - etc.

Characteristics A table is characterized by two parameters:
- a parameter which defines its organization (table dimension(s)),
- a parameter that defines the type of data it contains.

Note: The most complex organization is the table with six dimensions.

The syntax comprising these two parameters is:

```
ARRAY[<dimension(s) of table>]OF <data type>
```

```
<dimension 1>,<dimension 2>,<dimension n>
```

```
<minimum limit>,<maximum limit>
```

<minimum limit> strictly less than <maximum limit>
Data types

Defining and instancing a table

Definition of a table type:

\[ X: \text{ARRAY}[1..10] \text{ OF BOOL} \]

Instancing a table

\[
\begin{align*}
\text{Tab}_1: & \ X \\
\text{Tab}_2: & \ \text{ARRAY}[1..10] \ \text{OF BOOL}
\end{align*}
\]

The instances \( \text{Tab}_1 \) and \( \text{Tab}_2 \) are of the same type and the same dimension, the only difference being that during instancing:

- the \( \text{Tab}_1 \) type takes the name \( X \),
- the \( \text{Tab}_2 \) type must be defined (unnamed table).

Note: It is beneficial to name the type, as any modification that has to be made will only be done so once, otherwise there will be as many modifications as there are instances.

Examples

This table presents the instances of tables of different dimensions:

<table>
<thead>
<tr>
<th>Entry</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{Tab}_1: \text{ARRAY}[1..2] \text{ OF BOOL} )</td>
<td>1 dimensional table with 2 Boolean words</td>
</tr>
<tr>
<td>( \text{Tab}_2: \text{ARRAY}[10..20] \text{ OF COORD} )</td>
<td>1 dimensional table with 31 COORD type structures (structure defined by the user)</td>
</tr>
<tr>
<td>( \text{Tab}_3: \text{ARRAY}[1..10, 1..20] \text{ OF INT} )</td>
<td>2 dimensional table with 10x20 integers</td>
</tr>
<tr>
<td>( \text{Tab}_4: \text{ARRAY}[0..2, -1..1, 201..300, 0..1] \text{ OF REAL} )</td>
<td>4 dimensional table with 3x3x100x2 reals</td>
</tr>
</tbody>
</table>

Access to a data item in table \( \text{Tab}_1 \) and \( \text{Tab}_3 \):

\[
\begin{align*}
\text{Tab}_1[2] & \quad ;\text{To access second element} \\
\text{Tab}_3[4][18] & \quad ;\text{To access eighteenth element of the fourth sub-table}
\end{align*}
\]
There are the 4 following tables:

Tab_1: ARRAY[1..10] OF INT
Tab_2: ARRAY[1..10] OF INT
Tab_3: ARRAY[1..11] OF INT
Tab_4: ARRAY[101..110] OF INT

Tab_1:=Tab_2; Assignment authorized
Tab_1:=Tab_3; Assignment refused (non-IEC compliant)
Tab_1:=Tab_4; Assignment refused (non-IEC compliant)
Structures

What is a Structure?

It is a data item containing a set of data of a different type, such as:

- a group of BOOL, WORD, UINT, etc., (EDT structure),
- a group of tables (DDT structure),
- a group of REAL, DWORD, tables, etc., (EDT and DDT structures).

Note: You can create nested structures (nested DDTs) over 8 levels. Recurring structures (DDT) are not allowed.

Characteristics

A structure is composed of data which are each characterized by:

- a type,
- a name, which enables it to be identified,
- a comment (optional) describing its role.

Definition of a structure type:

```plaintext
IDENT
  Surname: STRING[12]
  First name: STRING[16]
  Age: UINT
```

;The IDENT type structure contains a UINT type data item and two STRING type data

Definition of two data instances of an IDENT type structure:

```plaintext
Person_1: IDENT
Person_2: IDENT
```

;The instances Person_1 and Person_2 are of IDENT Structure type

Access to the Data of a Structure

Access to the data of the Person_1 IDENT-type instance:

```plaintext
Person_1.Name ;To access name of Person_1
Person_1.Age ;To access age of Person_1
```
Overview of the Derived Data Type family (DDT)

Introduction

The DDT (Derived Data Type) family includes "derived" data types such as:

- tables
- structures

Illustration:
Data types

Characteristics

A data item belonging to the DDT family is made up of:

- the type name (See Syntax rules for Type/Instance names, p. 193) (32 characters maximum) defined by the user (not obligatory for tables but recommended) (See Defining and instancing a table, p. 226)
- the type (structure or table)
- an optional comment (of a maximum of 1024 characters). Authorized characters correspond to the ASCII codes 32 to 255
- the description (in the case of a structure) of these elements
  - the element name (See Syntax rules for Type/Instance names, p. 193) (32 characters maximum)
  - the element type
  - an optional comment (1024 characters maximum) describing its role. The authorized characters correspond to the ASCII codes 32 to 255
- information such as:
  - type version number
  - date of the last modification of the code or of the internal variables or of the interface variables
  - an optional descriptive file (32767 characters) describing the block function and its different modifications

Note: The total size of a table or of a structure does not exceed 64 Kbytes.

Examples

Definition of types

COORD
  X: INT
  Y: INT
  ;COORD type structure

SEGMENT
  Origin: COORD
  Destination: COORD
  ;SEGMENT type structure containing 2 COORD type structures

OUTLINE: ARRAY[0..99] OF SEGMENT
  ;OUTLINE type table containing 100 SEGMENT type structures

DRAW
  Color: INT
  Anchor: COORD
  Pattern: ARRAY[0..15,0..15] OF WORD
  Contour: OUTLINE
  ;DRAW type structure
Access to the data of a DRAW-type structure instance

Cartoon: DRAW
; Instance of DRAW type structure

Cartoon.Pattern[15,15]
; Access to last data item in the Pattern table of the Cartoon structure

Cartoon.Contour[0].Origin.X
; Access to data item X of the COORD structure belonging to the first SEGMENT structure of the Contour table.
Data types

DDT: Memory occupancy

At a Glance
The DDTs are stored in the PLC's memory in the order in which its elements are declared.
However, the following rules apply.

Principle
The storage principle is as follows:

- The elements are stored in the order in which they are declared in the structure,
- the basic element is the byte (alignment of data on the memory bytes),
- Each element has an alignment rule:
  - the BOOL and BYTE types are indiscriminately aligned on the odd or even bytes,
  - all the other elementary types are aligned on the even bytes,
  - the structures and tables are aligned according to the alignment rule for the BOOL and BYTE types if they only contain BOOL and BYTE elements, otherwise they are aligned on the memory's even bytes.

WARNING
Be careful with the compatibility of applications converted from Concept.

With the Concept programming application, the data structures do not handle any shift in offsets (each element is set one after the other in the memory, regardless of its type). Consequently, we recommend that you check everything, in particular the consistency of the data when using DDTs located on the "State RAM" (risk of shifts) or functions for communication with other devices (transfers with a different size from those programmed in Concept).

Failure to follow this instruction can result in death, serious injury, or equipment damage.
Examples 1

The table below gives some examples of data structures:

<table>
<thead>
<tr>
<th>Offset</th>
<th>Description of the structure</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Para_PWM1</td>
</tr>
<tr>
<td>offset 0</td>
<td>t_period: TIME</td>
</tr>
<tr>
<td>offset 4</td>
<td>t_min: TIME</td>
</tr>
<tr>
<td>offset 8</td>
<td>in_max: REAL</td>
</tr>
<tr>
<td></td>
<td>structure of type Para_PWM1</td>
</tr>
<tr>
<td></td>
<td>Mode_TOTALIZER</td>
</tr>
<tr>
<td>offset 0</td>
<td>hold: BOOL</td>
</tr>
<tr>
<td>offset 1</td>
<td>rst: BOOL</td>
</tr>
<tr>
<td></td>
<td>structure of type Mode_TOTALIZER</td>
</tr>
<tr>
<td></td>
<td>Info_TOTALIZER</td>
</tr>
<tr>
<td>offset 0</td>
<td>outc: REAL</td>
</tr>
<tr>
<td>offset 4</td>
<td>cter: UINT</td>
</tr>
<tr>
<td>offset 6</td>
<td>done: BOOL</td>
</tr>
<tr>
<td>offset 7</td>
<td>Reserved for the alignment</td>
</tr>
<tr>
<td></td>
<td>structure of type Info_TOTALIZER</td>
</tr>
</tbody>
</table>
#### Examples 2

The table below gives two examples of data structures with tables:

<table>
<thead>
<tr>
<th>Offset</th>
<th>Description of the structure</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>EHC105_Out</strong></td>
</tr>
<tr>
<td>offset 0</td>
<td>Quit: BYTE</td>
</tr>
<tr>
<td>offset 1</td>
<td>Control: ARRAY [1..5] OF BYTE</td>
</tr>
<tr>
<td>offset 6</td>
<td>Final: ARRAY [1..5] OF DINT</td>
</tr>
<tr>
<td></td>
<td><strong>structure of type EHC105_Out</strong></td>
</tr>
<tr>
<td></td>
<td><strong>CPCfg_ex</strong></td>
</tr>
<tr>
<td>offset 0</td>
<td>Profile_type: INT</td>
</tr>
<tr>
<td>offset 2</td>
<td>Interp_type: INT</td>
</tr>
<tr>
<td>offset 4</td>
<td>Nb_of_coords: INT</td>
</tr>
<tr>
<td>offset 6</td>
<td>Nb_of_points: INT</td>
</tr>
<tr>
<td>offset 8</td>
<td>reserved: ARRAY [0..4] OF BYTE</td>
</tr>
<tr>
<td>offset 13</td>
<td>Reserved for the alignment of Master_offset on even bytes</td>
</tr>
<tr>
<td>offset 14</td>
<td>Master_offset: INT</td>
</tr>
<tr>
<td>offset 16</td>
<td>Follower_offset: INT</td>
</tr>
<tr>
<td></td>
<td><strong>structure of type CPCfg_ex</strong></td>
</tr>
</tbody>
</table>
Overview of Input/Output Derived Data Types (IODDT)

At a Glance

The IODDTs (Input Output Derived Data Types) are predefined by the manufacturer, and contain language objects of the EDT family belonging to the channel of an application-specific module.

Illustration:

The IODDT types are structures whose size (the number of elements of which they are composed) depends on the channel or the input/output module that they represent.

A given input/output module can have more than one IODDT.

The difference with a conventional structure is that:

- the IODDT structure is predefined by the manufacturer
- The elements comprising the IODDT structure do not have a contiguous memory allocation, but rather a specific address in the module

Examples

IODDT structure for an input/output channel of an analog module

```plaintext
ANA_IN_GEN    ;ANA_IN_GEN type structure
Value:INT     ;Input value
Err: BOOL     ;Channel error
```
Access to the data of an instance of the ANA_IN_GEN type:

- Cistern_Level: ANA_IN_GEN
  ; ANA_IN_GEN type instance which corresponds for example to a tank level sensor

- Cistern_Level.Value ; Reading of the channel input value
- Cistern_Level.Err ; Reading of channel error bit

Access by direct addressing:

- For channel 0 of module 2 of rack 0 we obtain:

- Cistern_Level corresponds to %CH0.2.0
- Cistern_Level.Value corresponds to %IW0.2.0.0
- Cistern_Level_Err corresponds to %I0.2.0.ERR
8.7 Function block data types (DFB\EFB)

At a Glance

Subject of this Section

This section describes function block data types. These are:
- user function blocks (DFB)
- elementary function blocks (EFB)

What's in this Section?

This section contains the following topics:

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overview of Function Block data type families</td>
<td>238</td>
</tr>
<tr>
<td>Characteristics of Function Block data types (EFB\DFB)</td>
<td>240</td>
</tr>
<tr>
<td>Characteristics of elements belonging to function blocks</td>
<td>242</td>
</tr>
</tbody>
</table>
Overview of Function Block data type families

Introduction
Function block data type families are:
- the Elementary Function Block (EFB) (See General overview of the data type families, p. 189) type family
- the User function block (DFB) (See General overview of the data type families, p. 189) type family

Illustration:

Function blocks are entities containing:
- input and output variables acting as an interface with the application
- a processing algorithm that operates input variables and completes the output variables
- private and public internal variables operated by the processing algorithm

Illustration
Function block:
User function block (DFB)  

The user function block types (Derived Function Blocks) are developed by the user using one or more languages (according to the number of sections). These languages are:
- Ladder language
- Structured Text language
- Instruction List language
- Functional block language FBD

A DFB type can have one or more instances where each instance is referenced by a name (symbol) and possesses DFB data types.

Elementary Function Block (EFB)  

Elementary Function Blocks (EFBs) are provided by the manufacturer and are programmed in C language.

The user can create his own EFB for which he will need an optional software tool "SDKC".

An EFB type can have one or more instances where each instance is referenced by a name (symbol) and possesses EFB type data.
Data types

Characteristics of Function Block data types (EFB\DFB)

Type definition
The type of an EFB or DFB function block is defined by:
- the type name (See Syntax rules for Type\Instance names, p. 193), defined by
  the user for the DFBs,
- an optional comment. The authorized characters correspond to the ASCII codes
  32 to 255,
- the application interface data:
  - the inputs, not accessible in read\write mode from the application, but read by
    the function block code,
  - the inputs\outputs, not accessible in read\write mode from the application, but
    read and written by the function block code,
  - the outputs, accessible in read only from the application and read and written
    by the function block code.
- the internal data:
  - public internal data, accessible in read\write mode from the application, and
    read and written by the function block code,
  - private internal data, not accessible from the application, but read and written
    by the function block code.
- the code:
  - for DFBs, this is written by the user in PLC language (Structured Text,
    Instruction List, Ladder language, function block language), and is structured
    in a single section if the IEC option is active, or may be structured in several
    sections if this option is inactive
  - for EFBs, this is written in C language.
- information such as:
  - type version number,
  - date of the last modification of the code, or of the internal variables, or of the
    interface variables.
  - an optional descriptive file (32767 characters), describing the block function
    and its different modifications.
Data types

**Characteristics**

This table gives the characteristics of the elements that make up a type:

<table>
<thead>
<tr>
<th>Element</th>
<th>EFB</th>
<th>DFB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>32 characters</td>
<td>32 characters</td>
</tr>
<tr>
<td>Comment</td>
<td>1024 characters</td>
<td>1024 characters</td>
</tr>
<tr>
<td>Input Data</td>
<td>32 maximum</td>
<td>32 maximum</td>
</tr>
<tr>
<td>Input/Output data</td>
<td>32 maximum</td>
<td>32 maximum</td>
</tr>
<tr>
<td>Output data</td>
<td>32 maximum</td>
<td>32 maximum</td>
</tr>
<tr>
<td>Number of interfaces</td>
<td>32 maximum (2)</td>
<td>32 maximum (2)</td>
</tr>
<tr>
<td>(Inputs+Outputs+Inputs/Outputs)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Public data</td>
<td>No limits (1)</td>
<td>No limits (1)</td>
</tr>
<tr>
<td>Private data</td>
<td>No limits (1)</td>
<td>No limits (1)</td>
</tr>
<tr>
<td>Programming language</td>
<td>C language</td>
<td>Language:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>● Structured Text,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>● Instruction List,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>● Ladder language,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>● function block,</td>
</tr>
<tr>
<td>Section</td>
<td></td>
<td>A section is defined by:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>● a name (maximum 32 characters),</td>
</tr>
<tr>
<td></td>
<td></td>
<td>● a validation condition,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>● a comment (maximum 256 characters),</td>
</tr>
<tr>
<td></td>
<td></td>
<td>● a protection:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>● without,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>● read only,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>● read/write mode.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>A section cannot access declared variables in the application, except for:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>● system double words %SDi,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>● system words %SWi,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>● system bits %Si.</td>
</tr>
</tbody>
</table>

(1): the only limit is the size of the PLC’s memory.

(2): the EN input and ENO output are not taken into account.
Data types

Characteristics of elements belonging to function blocks

What is an element?

Each element (interface data or internal data) is defined by:
- a name (See Syntax rules for Type/Instance names, p. 193) (maximum 32 characters), defined by the user,
- a type, which can belong to the following families:
  - Elementary Data Types (EDT),
  - Derived Data Type (DDT),
  - Function Block data types (EFB/DFB).
- an optional comment (maximum 1024 characters). The authorized characters correspond to the ASCII codes 32 to 255,
- an initial value,
- an access right from the application program (sections of the application or section belonging to the DFBs see "Definition of the function block type (interface and internal variables)" (See Type definition, p. 240),
- an access right from communication requests,
- a public variables backup flag.

Authorized data types for an element belonging to a DFB

The authorized data types are:

<table>
<thead>
<tr>
<th>Element of the DFB</th>
<th>EDT types</th>
<th>DDT types</th>
<th>ANY... Function bloc types</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>IODDT</td>
<td>Unnamed</td>
<td>ANY_ARRAY</td>
</tr>
<tr>
<td>Input data</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Input/output data</td>
<td>Yes (1)</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Output data</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Public data</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Private data</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

(1): not authorized for the EBOOL type static data used on Quantum PLCs
(2): not authorized for BOOL and EBOOL type data
(3): must be completed during the execution of the DFB, and not usable outside the DFB
### Authorized data types for an element belonging to an EFB

The authorized data types are:

<table>
<thead>
<tr>
<th>Element of the EFB</th>
<th>DDT types</th>
<th>ANY...</th>
<th>Function bloc types</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>IODDT</td>
<td>Unnamed tables</td>
<td>ANY ARRAY</td>
</tr>
</tbody>
</table>

- **Input data**: Yes | No | No | Yes | Yes | Yes (1) | No
- **Input/output data**: Yes | Yes | No | Yes | Yes | Yes (1) | No
- **Output data**: Yes | No | No | No | Yes | Yes (1) | (2) No
- **Public data**: Yes | No | No | No | Yes | No | No
- **Private data**: Yes | No | No | No | Yes | No | Yes

(1): not authorized for BOOL and EBOOL type data

(2): must be completed during the execution of the EFB, and not usable outside the EFB

### Initial values for an element belonging to a DFB

This table specifies whether the initial values can be entered from the DFB type definition or the DFB instance:

<table>
<thead>
<tr>
<th>Element of the DFB</th>
<th>From the DFB type</th>
<th>From the DFB instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input data (no ANY... type)</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Input data (of ANY... type)</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Input/output data</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Output data (no ANY... type)</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Output data (of ANY... type)</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Public data</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Private data</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>
Initial values for an element belonging to an EFB

This table specifies whether the initial values can be entered from the EFB type definition or the EFB instance:

<table>
<thead>
<tr>
<th>Element of the EFB</th>
<th>From the EFB type</th>
<th>From the DFB instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input data (no ANY... type)</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>See Overview of Generic Data Types, p. 245</td>
<td></td>
</tr>
<tr>
<td>Input data (of ANY... type)</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Input/output data</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Output data (no ANY... type)</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Output data (of ANY... type)</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Public data</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Private data</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>
8.8 Generic Data Types (GDT)

Overview of Generic Data Types

Introduction Generic Data Types are conventional groups of data types (EDT, DDT) organized hierarchically, and specifically intended to determine compatibility among these conventional groups of data types.

These groups are identified by the prefix ANY_xxx, but these prefixes can under no circumstances be used to instance the data.

Their field of use concerns function block (EFB\DFB) and elementary function (EF) data type families, in order to define which data types are compatible with their interfaces for the following:

- inputs,
- input/outputs,
- outputs.
Data types

At a Glance

Hierarchical presentation of the Generic Data Types

```
ANY
  ANY_ELEMENTARY
  ANY_MAGNITUDE
    ANY_NUM
      ANY_REAL
      REAL
      ANY_INT
        DINT, INT, UDINT, UINT
    TIME
    ANY_BIT
    DWORD, WORD, BYTE, BOOL
  ANY_STRING
    STRING
  ANY_DATE
    DATE_AND_TIME, DATE, TIME_OF_DAY
  EBOOL
  ANY_DERIVED
  ANY_ARRAY
    ANY_ARRAY_ANY_EDT
    ANY_ARRAY_ANY_MAGNITUDE
      ANY_ARRAY_ANY_NUM
      ANY_ARRAY_ANY_REAL
      ANY_ARRAY_REAL
      ANY_ARRAY_ANY_INT
      ANY_ARRAY_DINT
      ANY_ARRAY_INT
      ANY_ARRAY_UDINT
      ANY_ARRAY_UINT
      ANY_ARRAY_TIME
    ANY_ARRAY_ANY_BIT
    ANY_ARRAY_DWORD
    ANY_ARRAY_WORD
    ANY_ARRAY_BYTE
    ANY_ARRAY_BOOL
    ANY_ARRAY_ANY_STRING
    ANY_ARRAY_STRING
    ANY_ARRAY_ANY_DATE
    ANY_ARRAY_DATE
    ANY_ARRAY_DATE_AND_TIME
    ANY_ARRAY_TIME_OF_DAY
    ANY_ARRAY_EBOOL
  ANY_ARRAY_ANY_DDT
  ANY_STRUCTURE
    ANY_DDT
    ANY_IODDT
    ANY_FFB
    ANY_EFB
    ANY_DFB
```
Example

This gives us the following DFB:

```

<table>
<thead>
<tr>
<th>FBI_1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parts.Counter</td>
</tr>
<tr>
<td>%I0.4.1</td>
</tr>
<tr>
<td>Pre.Val</td>
</tr>
<tr>
<td>Part_OK</td>
</tr>
<tr>
<td>Preset</td>
</tr>
</tbody>
</table>
| Nb_parts_attained | %M2
```

The **Preset** input parameter may be defined of GDT-type as ANY_ARRAY_x (with x being an EDT).

**Note:** The authorized objects for the various parameters are defined in this table (See *Authorized Objects for the Different Parameters, p. 525*).
8.9 Data types belonging to Sequential Function Charts (SFC)

Overview of the data types of the Sequential Function Chart family

Introduction

The Sequential Function Chart (SFC) data type family includes derived data types, such as the structures that restore the properties and status of the chart and its component actions.

Each step is represented by two structures. These are:

- the SFCSTEP_STATE structure,
- the SFCSTEP_TIMES structure.

Illustration:

![Diagram of SFC structure](image)

Note: The two structure types SFCSTEP_STATE and SFCSTEP_TIMES are also linked to each Macro step of the sequential function chart.
Data types

Definition of the SFCSTEP_STATE structure type
This structure includes all types of data linked to the status of the step or of the Macro step.
These data types are:
- x: BOOL elementary data type (EDT) containing the value TRUE when the step is active,
- t: TIME elementary data type (EDT) containing the activity time of the step. When deactivated, the step value is maintained until the next activation,
- tminErr: BOOL elementary data type (EDT) containing the value TRUE if the activity time of the step is less than the minimum programmed activity time,
- tmaxErr: BOOL elementary data type (EDT) containing the value TRUE if the activity time of the step is greater than the maximum programmed activity time,

These data types are accessible from the application in read only mode.

Definition of the SFCSTEP_TIMES structure type
This structure includes all types of data linked to the definition of the runtime parameters of the step or of the Macro step.
These data types are:
- delay: TIME elementary data type (EDT), defining the polling delay time of the transition situated downstream from the active step,
- tmin: TIME elementary data type (EDT) containing the minimum value during which the step must at least be executed. If this value is not respected the data tmin.Err switches to the value TRUE,
- tmax: TIME elementary data type (EDT) containing the maximum value during which the step must at least be executed. If this value is not respected the data tmax.Err switches to the value TRUE.

These data types are only accessible from the SFC editor.

Data access syntax of the structure SFCSTEP_STATE
The instance names of this structure correspond to the names of the steps or macro steps of the sequential function chart

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name_Step.x</td>
<td>Used to find out the status of the step (active/inactive)</td>
</tr>
<tr>
<td>Name_Step.t</td>
<td>Used to find out the current or total activation time for the step</td>
</tr>
<tr>
<td>Name_Step.tminErr</td>
<td>Used to find out if the minimum activation time of the step is less than the time programmed in Name_Step.tmin</td>
</tr>
<tr>
<td>Name_Step.tmaxErr</td>
<td>Used to find out if the maximum activation time of the step is greater than the time programmed in Name_Step.tmax</td>
</tr>
</tbody>
</table>
8.10 Compatibility between data types

Compatibility between data types

Introduction

The following is a presentation of the different rules of compatibility between types within each of the following families:

- the Elementary Data Type (EDT) family,
- the Derived Data Type (DDT) family,
- the Generic Data Type (GDT) family.

The Elementary Data Type (EDT) family

The Elementary Data Type (EDT) family contains the following sub-families:

- the binary format data type sub-family,
- the BCD format data type sub-family,
- the Real format data type sub-family,
- the character string format data type sub-family,
- the bit string format data type sub-family,

There is no compatibility whatsoever between two data types, even if they belong to the same sub-family.
The Derived Data Type (DDT) family contains the following sub-families:

- the table type sub-family,
- the structure type sub-family:
  - structures concerning input/output data (IODDT),
  - structures concerning other data.

Rules concerning the structures:

Two structures are compatible if their elements are:

- of the same name,
- of the same type,
- organized in the same order.

There are four types of structure:

**ELEMENT_1**
```
My_Element: INT
Other_Element: BOOL
```

**ELEMENT_2**
```
My_Element: INT
Other_Element: BOOL
```

**ELEMENT_3**
```
Element: INT
Other_Element: BOOL
```

**ELEMENT_4**
```
Other_Element: BOOL
My_Element: INT
```

Compatibility between the structure types

<table>
<thead>
<tr>
<th>Types</th>
<th>ELEMENT_1</th>
<th>ELEMENT_2</th>
<th>ELEMENT_3</th>
<th>ELEMENT_4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ELEMENT_1</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>ELEMENT_2</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>ELEMENT_3</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>ELEMENT_4</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
</tr>
</tbody>
</table>
Rules concerning the tables

Two tables are compatible if:

- their dimensions and the order of their dimensions are identical,
- each corresponding dimension is of the same type.

There are five types of table:

- TAB_1: ARRAY[10..20]OF INT ;Table one dimension of TAB_1 type
- TAB_2: ARRAY[20..30]OF INT ;Table one dimension of TAB_2 type
- TAB_3: ARRAY[20..30]OF INT ;Table one dimension of TAB_3 type
- TAB_4: ARRAY[20..30]OF TAB_1 ;Table one dimension of TAB_4 type
- TAB_5: ARRAY[20..30,10..20]OF INT ;Table two dimensions of type TAB_5

Compatibility between the table types:

<table>
<thead>
<tr>
<th>Type...</th>
<th>and type...</th>
<th>are...</th>
</tr>
</thead>
<tbody>
<tr>
<td>TAB_1</td>
<td>TAB_2</td>
<td>incompatible</td>
</tr>
<tr>
<td>TAB_2</td>
<td>TAB_3</td>
<td>compatible</td>
</tr>
<tr>
<td>TAB_4</td>
<td>TAB_5</td>
<td>compatible</td>
</tr>
<tr>
<td>TAB_4[25]</td>
<td>TAB_5[28]</td>
<td>compatible</td>
</tr>
</tbody>
</table>
The Generic Data Type (GDT) family is made up of groups organized hierarchically which contain data types belonging to the following families:

- Elementary Data Types (EDT),
- Derived Data Types (DDT).

**Rules:**

A conventional data type is compatible with the genetic data types related to it hierarchically (see the section "Generic data types") (See At a Glance, p. 246).

A generic data type is compatible with the generic data types related to it hierarchically (see the section "Generic data types") (See At a Glance, p. 246).

**Example:**

The INT type is compatible with the ANY_INT or ANY_NUM or ANY_MAGNITUDE types.

The INT type is not compatible with the ANY_BIT or ANY_REAL types.

The ANY_INT generic type is compatible with the ANY_NUM type.

The ANY_INT generic type is not compatible with the ANY_REAL type.
Data types
Data instances

At a Glance

What's in this Chapter?
This chapter describes data instances and their characteristics. These instances can be:
- unlocated data instances,
- located data instances,
- direct addressing data instances.

What's in this Chapter?
This chapter contains the following topics:

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Type Instances</td>
<td>256</td>
</tr>
<tr>
<td>Data instance attributes</td>
<td>260</td>
</tr>
<tr>
<td>Direct Addressing Data Instances</td>
<td>263</td>
</tr>
</tbody>
</table>
Data instances

Data Type Instances

Introduction

What is a data type instance? (See Overview of data instances, p. 191)

A data type instance is referenced either by:

- a name (symbol), in which case we say the data is **unlocated** because its memory allocation is not defined but is carried out automatically by the system,
- a name (symbol) and a **topological address** defined by the manufacturer, in which case we say the data is **located** since its memory allocation is known,
- a **topological address** defined by the manufacturer, in which case we say the data is **direct addressing**, and its memory allocation is known.

Unlocated Data Instances

Unlocated data instances are managed by the PLC operating system, and their physical location in the memory is **unknown to the user**.

Unlocated data instances are defined using data types belonging to one of the following families:

- Elementary Data Types (EDT),
- Derived Data Types (DDT),
- Function Block data types (EFB\DFB).

Examples:

```plaintext
Var_1: BOOL
;Instance of EDT family of Boolean type with 1 byte memory allocation

Var_2: UDINT
;Instance of EDT family of double unsigned integer type with 4 byte memory allocation

Var_3: ARRAY[1..10]OF INT
;Instance of DDT family of table type with 20 byte memory allocation

COORD
    X: INT
    Y: INT
Var_4: COORD
;Instance of DDT family of COORD type structure with 4 byte memory allocation
```
Located Data Instances

Located data instances have a predefined memory location in the PLC, **this location is known to the user**.

Located data instances are defined using data types belonging to one of the following families:

- Elementary Data Types (EDT),
- Derived Data Types (DDT),
- Sequential Function Chart data types (SFC).

**Note:** Double-type instances of located data (%MD<i>, %KD<i>) or floating (%MF<i>, %KF<i>) should be located by an integer type (%MW<i>, %KW<i>). Only I/O objects, allow to locate type instances (%MD<i>, %KD<i>, %QD, %ID, %MF<i>, %KF<i>, %QF, %IF) by using their topologic address (for example %MD0.6.0.11, %MF0.6.0.31).
Examples:

Var_1: EBOOL AT %M100
; Instance of EDT family of Boolean type (with 1 byte memory allocation) predefined in %M100

Var_2: BOOL AT %I2.1.0.ERR
; Instance of EDT family of Boolean type (with 1 byte memory allocation) predefined in %I2.1.0.ERR

Var_3: INT AT %MW10
; Instance of EDT family of integer type (with 2 byte memory allocation) predefined in %MW10

Var_4: INT AT %MW10
; Prohibited as Var_3 has the same memory allocation and is of the same type

Var_5: WORD AT %MW10
; Instance of EDT family of WORD type (with 2 byte memory allocation) predefined in %MW10

Var_6: ARRAY[1..10]OF INT AT %MW50
; Instance of EDT family of table type (with 20 byte memory allocation) predefined from %MW50

COORD
  X: INT
  Y: INT

Var_7: COORD AT %MW20
; Instance of EDT family of COORD structure type (with 4 byte memory allocation) predefined from %MW20

Var_8: DINT AT %MD0.6.0.11
; Instance of EDT family of DINT type (with 4 byte memory allocation) predefined from the topologic address of the I/O object of the application-specific module.

Var_9: REAL AT %MF0.6.0.31
; Instance of EDT family of REAL type (with 4 byte memory allocation) predefined from the topologic address of the I/O object of the application-specific module.
Direct addressing data instances have a predefined location in the PLC memory or in an application-specific module, **this location is known to the user.**

Direct addressing data instances are defined using types belonging to the Elementary Data Type (EDT) family.

Examples of direct addressing data instances:

<table>
<thead>
<tr>
<th>Internal</th>
<th>Constant</th>
<th>System</th>
<th>Input/Output</th>
<th>Network</th>
</tr>
</thead>
<tbody>
<tr>
<td>%Mi</td>
<td>%Si</td>
<td>%Q, %I</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%MWi</td>
<td>%KWi</td>
<td>%SWi</td>
<td>%QW, %IW</td>
<td>%NW</td>
</tr>
<tr>
<td>%MDi</td>
<td>%KDi</td>
<td></td>
<td>%QD, %ID</td>
<td></td>
</tr>
<tr>
<td>%MFi</td>
<td>%KFi</td>
<td></td>
<td>%QF, %IF</td>
<td></td>
</tr>
</tbody>
</table>

**Note:** Sequential Function Chart (SFC) data type instances are created the moment they are inserted in the application program, with default name the user can modify.
Data instances

Data instance attributes

At a Glance

The attributes of a data instance are its defining information.
This information is:

- its name (See Syntax rules for Type\Instance names, p. 193) (except for the direct addressing data instances (See Direct Addressing Data Instances, p. 263)),
- its topological address (except for unlocated data type instances),
- its data type, which can belong to one of the following families:
  - Elementary Data Type (EDT),
  - Derived Data Type (DDT),
  - Function Block data type (EFB\DFB),
  - Sequential Function Chart data type (SFC).
- an optional descriptive comment (1024 characters maximum). Authorized characters correspond to the ASCII codes 32 to 255

Name of a data instance

This is a symbol (32 characters maximum) chosen by the user which is used to reference the instance and must be unique.
Certain names cannot be used, for example:

- key words used in text languages,
- names of program sections,
- names of data types that are predefined or chosen by the user (structures, tables),
- names of DFB/EFB data types that are predefined or chosen by the user,
- names of Elementary Functions (EF) that are predefined or chosen by the user.
Data instances

Names of instances belonging to the SFC family

The names of instances are declared implicitly while the user drafts his sequential function chart. They are default names supplied by the manufacturer which the user can modify.

Manufacturer-supplied default names:

<table>
<thead>
<tr>
<th>SFC object</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step</td>
<td>S_&lt;section name&gt;_&lt;step No.&gt;</td>
</tr>
<tr>
<td>Step of Macro step</td>
<td>S_&lt;section name&gt;<em>&lt;macro step No.&gt;</em>&lt;step No.&gt;</td>
</tr>
<tr>
<td>Macro step</td>
<td>MS_&lt;section name&gt;_&lt;step No.&gt;</td>
</tr>
<tr>
<td>Nested macro step</td>
<td>MS_&lt;section name&gt;<em>&lt;macro step No.&gt;</em>&lt;step No.&gt;</td>
</tr>
<tr>
<td>Input step of Macro step</td>
<td>S_INSECTIONNAME_&lt;macro step No.&gt;</td>
</tr>
<tr>
<td>Output step of Macro step</td>
<td>S_OUTSECTIONNAME_&lt;macro step No.&gt;</td>
</tr>
<tr>
<td>Transition</td>
<td>T_&lt;section name&gt;_&lt;transition No.&gt;</td>
</tr>
<tr>
<td>Transition of Macro step</td>
<td>T_&lt;section name&gt;<em>&lt;macro step No.&gt;</em>&lt;transition No.&gt;</td>
</tr>
</tbody>
</table>

Note: Instance names do not include the name of the section in which the instance is used, since it can be used in different sections of the application.

Names of instances belonging to the Function Block family

Instance names are implicitly declared while the user inserts the instances into the sections of the application program. They are default names supplied by the manufacturer which the user may modify.

Syntax of manufacturer-supplied default names:

FB_<name of function block type>_<instance No.>
The access syntax is as follows:

For structure type data
<Instance name>.<Element name>

For table type data
<Instance name>[Element index]

Rule:
The maximum size of the access syntax is 1024 characters, and the possible limits of a derived data type are as follows:

- 10 nesting levels (tables/structures),
- 6 dimensions per table,
- 4 digits (figures) to define the index of a table element.
## Direct Addressing Data Instances

### At a Glance
What is a direct addressing data instance? (See *Direct Addressing Data Instances*, p. 259)

### Access syntax
The syntax of a direct addressing data instance is defined by the `%` symbol followed by a **memory location prefix** and in certain cases some additional information.

The memory location prefix can be:

- **M**, for internal variables,
- **K**, for constants (Premium),
- **S**, for system variables,
- **N**, for network variables,
- **I**, for input variables,
- **Q**, for output variables.
### %M Internal Variables

#### Access Syntax:

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Format</th>
<th>Example</th>
<th>Program Access Rights</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit</td>
<td>%M&lt;i&gt; or %MX&lt;i&gt;</td>
<td>3 bits (Ebool)</td>
<td>%M1</td>
</tr>
<tr>
<td>Word</td>
<td>%MW&lt;i&gt;</td>
<td>16 bits (Int)</td>
<td>%MW10</td>
</tr>
<tr>
<td>Word extracted bit</td>
<td>%MW&lt;i&gt;,&lt;j&gt;</td>
<td>1 bit (Bool)</td>
<td>%MW15.5</td>
</tr>
<tr>
<td>double word</td>
<td>%MD&lt;i&gt;</td>
<td>32 bits (Dint)</td>
<td>%MD8</td>
</tr>
<tr>
<td>Real (floating point)</td>
<td>%MF&lt;i&gt;</td>
<td>32 bits (Real)</td>
<td>%MF15</td>
</tr>
</tbody>
</table>

<i> represents the instance number (starts a 0 for Premium and 1 for Quantum).

#### Note:
The %M<i> or %MX<i> data detect edges and manage forcing.

#### Memory Organization:

\[
\begin{align*}
&MW0 \\
&MW1 = MF0 \\
&MW2 = MD2 = MF1 \\
&MW3 = MF2 \\
&MW4 = MD4 = MF3 \\
&MW5 = MF4
\end{align*}
\]

#### Note:
The modification of %MW<i> involves the corresponding modifications of %MD<i> and %MF<i>.}
### %K Constants

Access syntax:

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Format</th>
<th>Access Rights</th>
</tr>
</thead>
<tbody>
<tr>
<td>Word constant</td>
<td>%KW&lt;i&gt;</td>
<td>16 bits (Int)</td>
</tr>
<tr>
<td>Double word constant</td>
<td>%KD&lt;i&gt;</td>
<td>32 bits (Dint)</td>
</tr>
<tr>
<td>Real (floating point) constant</td>
<td>%KF&lt;i&gt;</td>
<td>32 bits (Real)</td>
</tr>
</tbody>
</table>

<i> represents the instance number.

**Note:** The memory organization is identical to that of internal variables. It should be noted that these variables are not available on Quantum PLCs.

### %I Constants

Access syntax:

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Format</th>
<th>Access Rights</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit</td>
<td>%I&lt;i&gt;</td>
<td>3 bits (Ebool)</td>
</tr>
<tr>
<td>Word</td>
<td>%IW&lt;i&gt;</td>
<td>16 bits (Int)</td>
</tr>
</tbody>
</table>

<i> represents the instance number.

**Note:** These data are only available on Quantum and Momentum PLCs.

### %S System Variables

Access syntax:

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Format</th>
<th>Access Rights</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit</td>
<td>%S&lt;i&gt; or %SX&lt;i&gt;</td>
<td>1 bit (Bool)</td>
</tr>
<tr>
<td>Word</td>
<td>%SW&lt;i&gt;</td>
<td>32 bits (Int)</td>
</tr>
</tbody>
</table>

<i> represents the instance number.

**Note:** The memory organization is identical to that of internal variables. The %S<i> and %SX<i> data are not used for detection of edges and do not manage forcing.
These variables contain information, which has to be exchanged between several application programs across the communication network.

Access syntax:

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Format</th>
<th>Program access rights</th>
</tr>
</thead>
<tbody>
<tr>
<td>Common word</td>
<td>%NW&lt;n&gt;.&lt;s&gt;.&lt;d&gt;</td>
<td>16 bits (Int)</td>
</tr>
<tr>
<td>Word extracted bit</td>
<td>%NW&lt;n&gt;.&lt;s&gt;.&lt;d&gt;.&lt;j&gt;</td>
<td>1 bit (Bool)</td>
</tr>
</tbody>
</table>

<n> represents the network number.

<s> represents the station number.

<d> represents the data number.

<j> represents the position of the bit in the word.
These variables are contained in the application-specific modules.

**Access syntax:**

<table>
<thead>
<tr>
<th>Access Type</th>
<th>Syntax</th>
<th>Example</th>
<th>Program access rights</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input/Output</td>
<td>%CH&lt;@mod&gt;.&lt;c&gt;</td>
<td>%CH4.3.2</td>
<td>R</td>
</tr>
<tr>
<td>structure (IODDT)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%I inputs</td>
<td>%I&lt;@mod&gt;.MOD.ERR</td>
<td>%I4.2.MOD.ERR</td>
<td>R</td>
</tr>
<tr>
<td></td>
<td>%I&lt;@mod&gt;.&lt;c&gt;.ERR</td>
<td>%I4.2.3.ERR</td>
<td>R</td>
</tr>
<tr>
<td></td>
<td>%I&lt;@mod&gt;.&lt;c&gt;</td>
<td>%I4.2.3</td>
<td>R</td>
</tr>
<tr>
<td></td>
<td>%I&lt;@mod&gt;.&lt;c&gt;.&lt;d&gt;</td>
<td>%I4.2.3.1</td>
<td>R</td>
</tr>
<tr>
<td></td>
<td>%IW&lt;@mod&gt;.&lt;c&gt;</td>
<td>%IW4.2.3</td>
<td>R</td>
</tr>
<tr>
<td></td>
<td>%IW&lt;@mod&gt;.&lt;c&gt;.&lt;d&gt;</td>
<td>%IW4.2.3.1</td>
<td>R</td>
</tr>
<tr>
<td></td>
<td>%ID&lt;@mod&gt;.&lt;c&gt;</td>
<td>%ID4.2.3</td>
<td>R</td>
</tr>
<tr>
<td></td>
<td>%ID&lt;@mod&gt;.&lt;c&gt;.&lt;d&gt;</td>
<td>%ID4.2.3.1</td>
<td>R</td>
</tr>
<tr>
<td></td>
<td>%IF&lt;@mod&gt;.&lt;c&gt;</td>
<td>%IF4.2.3</td>
<td>R</td>
</tr>
<tr>
<td></td>
<td>%IF&lt;@mod&gt;.&lt;c&gt;.&lt;d&gt;</td>
<td>%IF4.2.3.1</td>
<td>R</td>
</tr>
<tr>
<td>%Q outputs</td>
<td>%Q&lt;@mod&gt;.&lt;c&gt;</td>
<td>%Q4.20.3</td>
<td>R/W</td>
</tr>
<tr>
<td></td>
<td>%Q&lt;@mod&gt;.&lt;c&gt;.&lt;d&gt;</td>
<td>%Q4.20.3.1</td>
<td>R/W</td>
</tr>
<tr>
<td></td>
<td>%QW&lt;@mod&gt;.&lt;c&gt;</td>
<td>%QW4.2.3</td>
<td>R/W</td>
</tr>
<tr>
<td></td>
<td>%QW&lt;@mod&gt;.&lt;c&gt;.&lt;d&gt;</td>
<td>%QW4.2.3.1</td>
<td>R/W</td>
</tr>
<tr>
<td></td>
<td>%QD&lt;@mod&gt;.&lt;c&gt;</td>
<td>%QD4.2.3</td>
<td>R/W</td>
</tr>
<tr>
<td></td>
<td>%QD&lt;@mod&gt;.&lt;c&gt;.&lt;d&gt;</td>
<td>%QD4.2.3.1</td>
<td>R/W</td>
</tr>
<tr>
<td></td>
<td>%QF&lt;@mod&gt;.&lt;c&gt;</td>
<td>%QF4.2.3</td>
<td>R/W</td>
</tr>
<tr>
<td></td>
<td>%QF&lt;@mod&gt;.&lt;c&gt;.&lt;d&gt;</td>
<td>%QF4.2.3.1</td>
<td>R/W</td>
</tr>
<tr>
<td>%M variables</td>
<td>%MW&lt;@mod&gt;.&lt;c&gt;</td>
<td>%MW4.2.3</td>
<td>R/W</td>
</tr>
<tr>
<td>(Premium)</td>
<td>%MW&lt;@mod&gt;.&lt;c&gt;.&lt;d&gt;</td>
<td>%MW4.2.3.1</td>
<td>R/W</td>
</tr>
<tr>
<td></td>
<td>%MD&lt;@mod&gt;.&lt;c&gt;</td>
<td>%MD4.2.3</td>
<td>R/W</td>
</tr>
<tr>
<td></td>
<td>%MD&lt;@mod&gt;.&lt;c&gt;.&lt;d&gt;</td>
<td>%MD4.2.3.1</td>
<td>R/W</td>
</tr>
<tr>
<td></td>
<td>%MF&lt;@mod&gt;.&lt;c&gt;</td>
<td>%MF4.2.3</td>
<td>R/W</td>
</tr>
<tr>
<td></td>
<td>%MF&lt;@mod&gt;.&lt;c&gt;.&lt;d&gt;</td>
<td>%MF4.2.3.1</td>
<td>R/W</td>
</tr>
</tbody>
</table>
## Data instances

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Example</th>
<th>Program access rights</th>
</tr>
</thead>
<tbody>
<tr>
<td>%K Constants (Premium)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Int type word</td>
<td>%KW&lt;@mod&gt;.&lt;c&gt;</td>
<td>%KW4.2.3 R</td>
</tr>
<tr>
<td>%KW&lt;@mod&gt;.&lt;c&gt;.&lt;d&gt;</td>
<td>%KW4.2.3.1 R</td>
<td></td>
</tr>
<tr>
<td>Dint type double word</td>
<td>%KD&lt;@mod&gt;.&lt;c&gt;</td>
<td>%KD4.2.3 R</td>
</tr>
<tr>
<td>%KD&lt;@mod&gt;.&lt;c&gt;.&lt;d&gt;</td>
<td>%KD4.2.3.1 R</td>
<td></td>
</tr>
<tr>
<td>Read type real</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(floating point)</td>
<td>%KF&lt;@mod&gt;.&lt;c&gt;</td>
<td>%KF4.2.3 R</td>
</tr>
<tr>
<td>%KF&lt;@mod&gt;.&lt;c&gt;.&lt;d&gt;</td>
<td>%KF4.2.3.1 R</td>
<td></td>
</tr>
</tbody>
</table>

@@ mod = 'b'.'c'.'e'.'r'.'m'  
<@ mod = 'b'.'c'.'e'.'r'.'m'>  
<b> bus number (omitted if station is local).  
<e> device connection point number (omitted if station is local, the connection point is also called Drop for Quantum users).  
<r> rack number.  
<m> module slot  
<c> channel number (0 to 999) or MOD reserved word.  
<d> data number (0 to 999) or ERR reserved word (optional if 0 value).
Examples: local station and station on bus for Quantum and Premium PLCs.

Quantum example

Local rack

%IW.4.1.5
(rack 1, module 4, channel 1, data 5)

%IW.3.2.1.4.1.5
(bus 3, drop (device) 2, rack 1, module 4
channel 1 data 5)

%IW.2.1.3.4.5
(bus 2, drop (device) 1, rack 1, module 3
channel 4 data 5)

RIO (No. 3)

DIO (No. 2)

Premium example

%IW.0.4.1.5
(rack 0, module 4, channel 1, data 5)

%IW.2.7.0.5.1.5
(bus module 5 channel 1
data 5)

Available for Unity versions
higher than V1.0
Data instances
Data references

At a Glance

What's in this Chapter?
This chapter provides the references of data instances. These references can be:
- value-based references,
- name-based references,
- address-based references.

What's in this Chapter?
This chapter contains the following topics:

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>References to data instances by value</td>
<td>272</td>
</tr>
<tr>
<td>References to data instances by name</td>
<td>274</td>
</tr>
<tr>
<td>References to data instances by address</td>
<td>277</td>
</tr>
<tr>
<td>Data naming rules</td>
<td>281</td>
</tr>
</tbody>
</table>
References to data instances by value

Introduction
What is a data instance reference? (See Overview of the data references, p. 192)

At a Glance
A reference to a data instance by a value is an instance which does not have a name (symbol) or topological address.

This corresponds to an **immediate value** which can be assigned to a data type instance belonging to the EDT family.

Standard IEC 1131 authorizes immediate values on instances of the following data types:
- Booleans, such as:
  - BOOL,
  - EBOOL.
- integers,
  - INT,
  - UINT,
  - DINT,
  - UDINT,
  - TIME.
- reals
  - REAL.
- dates and times,
  - DATE,
  - DATE AND TIME,
  - TIME OF DAY.
- character strings.
  - STRING.

The programming software goes beyond the scope of the standard by adding the **bit string types**.
- BYTE,
- WORD,
- DWORD.
This table associates immediate values with types of instance:

<table>
<thead>
<tr>
<th>Immediate value</th>
<th>Type of instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>'I am a character string'</td>
<td>STRING</td>
</tr>
<tr>
<td>T#1s</td>
<td>TIME</td>
</tr>
<tr>
<td>D#2000-01-01</td>
<td>DATE</td>
</tr>
<tr>
<td>TOD#12:25:23</td>
<td>TIME_OF_DAY</td>
</tr>
<tr>
<td>DT#2000-01-01-12:25:23</td>
<td>DATE_AND_TIME</td>
</tr>
<tr>
<td>16#FFF0</td>
<td>WORD</td>
</tr>
<tr>
<td>UINT#16#9AF (typed value)</td>
<td>UINT</td>
</tr>
<tr>
<td>DWORD#16#FFFF (typed value)</td>
<td>DWORD</td>
</tr>
</tbody>
</table>
## References to data instances by name

### Introduction
What is a data instance reference? (See *Overview of the data references*, p. 192)

### References to instances of the EDT family
The user chooses a name (symbol) which can be used to access the data instance:

- **Valve_State**: BOOL
- **Upper_Threshold**: EBOOL AT %M10
- **Hopper_Content**: UINT
- **Oven_Temperature**: INT AT %MW100
- **Encoder_Value**: WORD

### References to instances of the DDT family
The user chooses a name (symbol) which can be used to access the data instance:

- **Color_Range ARRAY[1..15]OF STRING**
- **Vehicles ARRAY[1..100]OF Color_range**

Giving 2 types of table:

```
Color_Range ARRAY[1..15]OF STRING
Vehicles ARRAY[1..100]OF Color_range
```

- **Car**: Vehicles
  Instance name of the Vehicles-type table

- **Car[11, 5]**
  Access to car 11 of the color corresponding to element 5 of the Color_range table
Data references

Structures:
The user chooses a name (symbol) which can be used to access the data instance:

Giving the 2 structures:

ADDRESS
  Street: STRING[20]
  Post_Code: UDINT
  Town: STRING: [20]

IDENT
  Surname: STRING[15]
  First name: STRING[15]
  Age: UINT
  Date_of_Birth: DATE
  Location: ADDRESS

Person_1 :IDENT
;Instance name of structure of IDENT type

Person_1.Age
Access to the age of Person_1

Person_1.Location.Town
;Access to the location where Person_1 resides
References to instances of the DFB/EFB families

The user chooses a name (symbol) which can be used to access the data instance.

Giving the DFB type:

![Diagram of Parts_Counter]

- **Part_OK**
- **Preset**
- **Num_parts_attained**
- **Num_Parts_Prog**

**Screw_Counter**: Parts_Counter
Instance name of block of Parts_Counter type

**Screw_Counter.Num_Parts_Prog**
Access to public variable Num_Parts_Prog

**Screw_Counter.Num_parts_attained**
Access to the output interface Num_parts_attained
References to data instances by address

Introduction
What is a data instance reference? (See Overview of the data references, p. 192)

At a Glance
It is only possible to make a reference to a data instance by address for certain instances of data belonging to the EDT family. These instances are:
- internal variables (%M<i>, %MW<i>, %MD<i>, %MF<i>),
- constants (%KW<i>, %KD<i>, %KF<i>),
- inputs/outputs (%I<address>, %Q<address>).

Reference by direct addressing
Addressing is deemed to be direct when the address of the instance is fixed or, in other words, when it is written into the program.
Examples:

%M1
Access to first bit of the memory

%MW12
Access to twelfth word of the memory

%MD4
Access to fourth double word of the memory

%KF100
Access to hundredth floating pointing word of the memory

%Q0.4.0.5
Access to fifth bit of the output module in position 4 of rack 0
References by indexed address

Addressing is deemed to be indexed when the address of the instance is completed with an index. The index is defined either by:
- a value belonging to an Integer type,
- an arithmetical expression made up of Integer types.
An indexed variable always has a non-indexed equivalent:

\%MW<i>[<index>] \iff \%MW<j>

The rules for calculating \(<j>\) are as follows:

<table>
<thead>
<tr>
<th>Object&lt;i&gt;[index]</th>
<th>Object&lt;j&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>%M[i][index]</td>
<td>(&lt;j&gt; = \langle i \rangle + \langle index \rangle)</td>
</tr>
<tr>
<td>%MW[i][index]</td>
<td>(&lt;j&gt; = \langle i \rangle + \langle index \rangle)</td>
</tr>
<tr>
<td>%KW[i][index]</td>
<td>(&lt;j&gt; = \langle i \rangle + \langle index \rangle)</td>
</tr>
<tr>
<td>%MD[i][index]</td>
<td>(&lt;j&gt; = \langle i \rangle + \langle index \rangle \times 2)</td>
</tr>
<tr>
<td>%KD[i][index]</td>
<td>(&lt;j&gt; = \langle i \rangle + \langle index \rangle \times 2)</td>
</tr>
<tr>
<td>%MF[i][index]</td>
<td>(&lt;j&gt; = \langle i \rangle + \langle index \rangle \times 2)</td>
</tr>
<tr>
<td>%KF[i][index]</td>
<td>(&lt;j&gt; = \langle i \rangle + \langle index \rangle \times 2)</td>
</tr>
</tbody>
</table>

Examples:

\%MD6[10] \iff \%MD26

\%MW10[My_Var+8] \iff \%MW20 (with My_Var=2)

During compilation of the program a check verifies that
- the index is not negative,
- the index does not exceed the space in the memory allocated to each of these three data types.
**Word extract bits**

It is possible to extract one of the 16 bits of single words (%MW, %SW; %KW, %IW, %QW).

The address of the instance is completed with the rank of the extracted bit (<j>):

```
WORD<i> . <j>
```

**Examples:**

%MW10.4
Bit No. 4 of word %MW10

%SW8.4
Bit No. 4 of system word %SW8

%KW100.14
Bit No. 14 of constant KW100

%QW0.5.1.0.10
Bit No. 10 of word 0 of channel 1 of output module 5 of rack 0
Data references

**Bit and word tables**

These are a series of adjacent objects (bits or words) of the same type and of a defined length

OBJECT<->L

**Presentation of bit tables:**

<table>
<thead>
<tr>
<th>Type</th>
<th>Address</th>
<th>Write access</th>
</tr>
</thead>
<tbody>
<tr>
<td>Discrete I/O input bits</td>
<td>%Ix.i:L</td>
<td>No</td>
</tr>
<tr>
<td>Discrete I/O output bits</td>
<td>%Qx.i:L</td>
<td>Yes</td>
</tr>
<tr>
<td>Internal bits</td>
<td>%Mi:L</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Presentation of word tables:**

<table>
<thead>
<tr>
<th>Type</th>
<th>Address</th>
<th>Write access</th>
</tr>
</thead>
<tbody>
<tr>
<td>Internal words</td>
<td>%MWi:L</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>%MDi:L</td>
<td></td>
</tr>
<tr>
<td></td>
<td>%MFi:L</td>
<td></td>
</tr>
<tr>
<td>Constant words</td>
<td>%KWi:L</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>%KDi:L</td>
<td></td>
</tr>
<tr>
<td></td>
<td>%KFi:L</td>
<td></td>
</tr>
<tr>
<td>System words</td>
<td>%SW50:4</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Examples:**

%M2:65
Defines an EBOOL table from %M2 to %M66

%M125:30
Defines an INT table from %MW125 to %MW154
# Data naming rules

**Introduction**

In an application the user chooses a name to:
- define a type of data,
- instantiate a data item (symbol),
- identify a section.

Some rules have been defined in order to avoid conflicts occurring. This means that it is necessary to differentiate between the different domains of application of data.

**What is a domain?**

It is an area of the application from which a variable can or cannot be accessed, such as:
- the application domain which includes:
  - the various application tasks,
  - the sections of which it is composed.
- the domains for each data type such as:
  - structures/tables for the DDT family,
  - EFB/DFBs for the function block family.
**Rules**

This table defines whether or not it is possible to use a name that already exists in the application for newly-created elements:

<table>
<thead>
<tr>
<th>Application → New elements (below)</th>
<th>Section</th>
<th>Table/Structure types</th>
<th>EFB/DFB types</th>
<th>EF</th>
<th>Instances</th>
</tr>
</thead>
<tbody>
<tr>
<td>Section</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Table/Structure types</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>EFB/DFB types</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No (3)</td>
</tr>
<tr>
<td>EF</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Instances</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No (1) Yes (2)</td>
<td>No</td>
</tr>
</tbody>
</table>

(1): An instance belonging to the application domain cannot have the same name as an EF

(2): An instance belonging to the type domain (internal variable) can have the same name as an EF. The EF in question cannot be used in this type.

(3): The creation or import of EFB/DFBs with the same name as an existing instance are prohibited.

**Note:** A number of additional considerations to the rules given in the table are listed below, specifying that:

- within a type, an instance (internal variable) cannot have the same name as the type name of the object to which it belongs,
- there is no conflict between the name of an instance belonging to a section of the application and the name of the instance belonging to a section of a DFB,
- there is no conflict between the name of a section belonging to a task and the name of the section belonging to a DFB.
Programming language

Introduction

Contents of this section
This section describes the syntax of the programming languages that are available.

What’s in this Part?
This part contains the following chapters:

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Chapter Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>Function Block language FBD</td>
<td>285</td>
</tr>
<tr>
<td>12</td>
<td>Ladder Diagram LD</td>
<td>311</td>
</tr>
<tr>
<td>13</td>
<td>SFC Sequence Language</td>
<td>353</td>
</tr>
<tr>
<td>14</td>
<td>Instruction list IL</td>
<td>417</td>
</tr>
<tr>
<td>15</td>
<td>Structured Text ST</td>
<td>465</td>
</tr>
</tbody>
</table>
Function Block language FBD

Introduction

Overview
This chapter describes the function block language FBD which conforms to IEC-61131.

What's in this Chapter?
This chapter contains the following topics:

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>General information about the FBD Function Block language</td>
<td>286</td>
</tr>
<tr>
<td>Elementary Functions, Elementary Function Blocks, Derived Function Blocks and Procedures (FFBs)</td>
<td>288</td>
</tr>
<tr>
<td>Subroutine Calls</td>
<td>297</td>
</tr>
<tr>
<td>Control Elements</td>
<td>298</td>
</tr>
<tr>
<td>Link</td>
<td>299</td>
</tr>
<tr>
<td>Text Object</td>
<td>301</td>
</tr>
<tr>
<td>Execution sequence of the FFBs</td>
<td>302</td>
</tr>
<tr>
<td>Change Execution Sequence</td>
<td>304</td>
</tr>
<tr>
<td>Loop Planning</td>
<td>309</td>
</tr>
</tbody>
</table>
General information about the FBD Function Block language

Introduction
The FBD editor is used for graphical function block programming according to IEC 61131-3.

Representation of an FBD section

Representation:

Objects
The objects of the FBD programming language (Function Block Diagram) help to divide a section into a number of:
- EFs and EFBs (Elementary Functions (See Elementary Function, p. 288) and Elementary Function Blocks (See Elementary Function Block, p. 288)),
- DFBs (Derived Function Blocks) (See DFB, p. 290),
- Procedures (See Procedure, p. 290) and
- Control Elements (See Control Elements, p. 298).
These objects, combined under the name FFBs, can be linked with each other by:
- Links (See Link, p. 299) or
- Actual Parameters (See Parameters, p. 291).
Comments regarding the section logic can be provided using text objects (see Text Object, p. 301).
Section Size

One FBD section consists of a window containing a single page.
This page has a grid background. A grid unit consists of 10 coordinates. A grid unit is the smallest possible space between 2 objects in an FBD section.
The FBD programming language is not cell oriented but the objects are still aligned with the grid coordinates.
An FBD section has 360 horizontal grid coordinates (=36 grid units) and 240 vertical grid coordinates (=24 grid units).

IEC Conformity

For a description of the extent to which the FBD programming language conforms to IEC, see IEC Conformity (See IEC Compliance, p. 601).
Elementary Functions, Elementary Function Blocks, Derived Function Blocks and Procedures (FFBs)

<table>
<thead>
<tr>
<th>Introduction</th>
<th>FFB is the generic term for:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>● Elementary Function (EF) (See Elementary Function, p. 288)</td>
</tr>
<tr>
<td></td>
<td>● Elementary Function Block (EFB) (See Elementary Function Block, p. 288)</td>
</tr>
<tr>
<td></td>
<td>● DFB (Derived Function Block) (See DFB, p. 290)</td>
</tr>
<tr>
<td></td>
<td>● Procedure (See Procedure, p. 290)</td>
</tr>
</tbody>
</table>

### Elementary Function

Elementary functions (EF) have no internal states. If the input values are the same, the value on the output is the same every time the function is called. For example, the addition of two values always gives the same result.

An elementary function is represented graphically as a frame with inputs and one output. The inputs are always represented on the left and the output is always on the right of the frame.

The name of the function, i.e. the function type, is displayed in the center of the frame.

The execution number (See Execution sequence of the FFBs, p. 302) for the function is shown to the right of the function type.

The function counter is shown above the frame. The function counter is the sequential number of the function within the current section. Function counters cannot be modified.

#### Elementary Function

![Elementary Function Diagram](DIV IN1 IN2 .1 OUT)

With some elementary functions, the number of inputs can be increased.

### Elementary Function Block

Elementary function blocks (EFBs) have internal states. If the input values are the same, the value on the output can be different each time the function is called. e.g. for a counter the value on the output is incremented.

An elementary function block is represented graphically as a frame with inputs and outputs. The inputs are always represented on the left and the outputs always on the right of the frame.

Function blocks can have more than one output.
The name of the function block, i.e. the function block type, is displayed in the center of the frame.

The execution number (See Execution sequence of the FFBs, p. 302) for the function block is shown to the right of the function block type.

The instance name is displayed above the frame.

The instance name serves as a unique identification for the function block in a project.

The instance name is created automatically and has the following structure: FBI_n

FBI = Function Block Instance
n = sequential number of the function block in the project

This automatically generated name can be modified for clarification. The instance name (max. 32 characters) must be unique throughout the project and is not case-sensitive. The instance name must conform to general naming conventions.

**Note:** To conform to IEC61131-3, only letters are permitted as the first character of the name. If you want to use a numeral as your first character however, this must be enabled explicitly.

Elementary Function Block

```
FBI_1

  CTU
  ----
  CTU  Q
  R
  PV  CV
```
DFB

Derived function blocks (DFBs) have the same properties as elementary function blocks. The user can create them in the programming languages FBD, LD, IL, and/or ST.

The only difference to elementary function blocks is that the derived function block is represented as a frame with double vertical lines.

**Derived Function Block**

```
  +-----------------+
  |                  |
  |  FBI_1           |
  +-----------------+
     |                |
     |  DFB_EXAMP     |
     +-----------------+
        |                |
        | IN1           OUT1 |
        +-----------------+
        |                |
        | IN2           |
        +-----------------+
        |                |
        | IN3           OUT2 |
        +-----------------+
```

Procedure

Procedures are functions viewed technically.

The only difference to elementary functions is that procedures can occupy more than one output and they support data type `VAR_IN_OUT`.

Procedures are a supplement to IEC 61131-3 and must be enabled explicitly.

To the eye, procedures are no different than elementary functions.

**Procedure**

```
  +-----------------+
  |                  |
  |  PROC_EXAMP     |
  +-----------------+
     |                |
     | IN1           OUT1 |
     +-----------------+
        |                |
        | IN2           OUT2 |
        +-----------------+
        |                |
        | IO1           IO1 |
        +-----------------+
```
Parameters

Inputs and outputs are required to transfer values to or from an FFB. These are called formal parameters.

Objects are linked to formal parameters; these objects contain the current process states. They are called actual parameters.

Formal and actual parameters:

At program runtime, the values from the process are transferred to the FFB via the actual parameters and then output again after processing.

Only one object (actual parameter) of the following types may be linked to FFB inputs:
- Variable
- Address
- Literal
- ST Expression (See Expression, p. 467)
  ST expressions on FFB inputs are a supplement to IEC 61131-3 and must be enabled explicitly.
- Link

The following combinations of objects (actual parameters) can be linked to FFB outputs:
- one variable
- a variable and one or more connections (but not for VAR_IN_OUT (See VAR_IN_OUT Variable, p. 296) outputs)
- an address
- an address and one or more connections (but not for VAR_IN_OUT (See VAR_IN_OUT Variable, p. 296) outputs)
- one or more connections (but not for VAR_IN_OUT (See VAR_IN_OUT Variable, p. 296) outputs)

The data type of the object to be linked must be the same as that of the FFB input/output. If all actual parameters consist of literals, a suitable data type is selected for the function block.

Exception: For generic FFB inputs/outputs with data type ANY_BIT, it is possible to link objects of data type INT or DINT (not UINT and UDINT).
This is a supplement to IEC 61131-3 and must be enabled explicitly.

Example:

**Allowed:**

```
  AND
  IntVar1 IN1 OUT IntVar3
  IntVar2 IN2
```

**Not allowed:**

```
  AND WORD
  IntVar1 IN1 OUT IntVar3
  IntVar2 IN2
```

(In this case, `AND_INT` must be used.)

Not all formal parameters have to be assigned an actual parameter. However, this does not apply in the case of negated pins. These must always be assigned an actual parameter. This is also the case with some formal parameter types. These types are shown in the following table.

Table of formal parameter types:

<table>
<thead>
<tr>
<th>Parameter type</th>
<th>EDT</th>
<th>STRING</th>
<th>ARRAY</th>
<th>ANY ARRAY</th>
<th>IODDT</th>
<th>STRUCT</th>
<th>FB</th>
<th>ANY</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFB: Input</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>/</td>
<td>+</td>
<td>/</td>
<td>+</td>
</tr>
<tr>
<td>EFB: VAR_IN_OUT</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>/</td>
<td>+</td>
</tr>
<tr>
<td>EFB: Output</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>/</td>
<td>+</td>
</tr>
<tr>
<td>DFB: Input</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>/</td>
<td>+</td>
<td>/</td>
<td>+</td>
</tr>
<tr>
<td>DFB: VAR_IN_OUT</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>/</td>
<td>+</td>
</tr>
<tr>
<td>DFB: Output</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>/</td>
<td>-</td>
<td>/</td>
<td>+</td>
</tr>
<tr>
<td>EF: Input</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>EF: VAR_IN_OUT</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>/</td>
<td>+</td>
</tr>
<tr>
<td>EF: Output</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>/</td>
<td>-</td>
</tr>
<tr>
<td>Procedure: Input</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Procedure: VAR_IN_OUT</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>/</td>
<td>+</td>
</tr>
<tr>
<td>Procedure: Output</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>/</td>
<td>+</td>
</tr>
</tbody>
</table>

+  Actual parameter required
-  Actual parameter not required
/  not applicable
FFBs that use actual parameters on the inputs that have not yet received any value assignment, work with the initial values of these actual parameters.

If no value is allocated to a formal parameter, then the initial value will be used for executing the function block. If no initial value has been defined then the default value (0) is used.

If a formal parameter is not assigned a value and the function block/DFB is instanced more than once, then the subsequent instances are run with the old value.

Public Variables

In addition to inputs/outputs, some function blocks also provide public variables. These variables transfer statistical values (values that are not influenced by the process) to the function block. They are used for setting parameters for the function block.

Public variables are a supplement to IEC 61131-3.

The assignment of values to public variables is made using their initial values.

Public variables are read via the instance name of the function block and the names of the public variables.

Example:

Programming notes

Attention should be paid to the following programming notes:

- FFBs are only executed if the input EN=1 or if the input EN is grayed out (see also EN and ENO, p. 294).
- Boolean inputs and outputs can be inverted.
- Special conditions apply when using VAR_IN_OUT variables (See VAR_IN_OUT Variable, p. 296).
Function block/DFB instances can be called multiple times (also see Multiple Function Block Instance Call, p. 294).

**Multiple Function Block Instance Call**

Function block/DFB instances can be called more than once; other than instances from communication EFBs and function blocks/DFBs with an ANY output but no ANY input: these can only be called once.

Calling the same function block/DFB instance more than once makes sense, for example, in the following cases:

- If the function block/DFB has no internal value or it is not required for further processing.
  In this case, memory is saved by calling the same function block/DFB instance more than once since the code for the function block/DFB is only loaded once. The function block/DFB is then handled like a “Function”.
- If the function block/DFB has an internal value and this is supposed to influence various program segments, for example, the value of a counter should be increased in different parts of the program.
  In this case, calling the same function block/DFB means that temporary results do not have to be saved for further processing in another part of the program.

**EN and ENO**

One EN input and one ENO output can be used in all FFBs.

If the value of EN is equal to "0" when the FFB is invoked, the algorithms defined by the FFB are not executed and ENO is set to "0".

If the value of EN is equal to "1" when the FFB is invoked, the algorithms defined by the FFB will be executed. After the algorithms have been executed successfully, the value of ENO is set to "1": If an error occurs when executing these algorithms, ENO is set to "0".

If EN is set to "0" (caused by EN=0 or an error during execution):

- Function blocks
  - EN/ENO handling with function blocks that (only) have one link as an output parameter:

    | FUNCBLOCK_1 | FUNCBLOCK_2 |
    |--------------|--------------|
    | EN           | EN           |
    | IN1          | ENO          |
    | IN2          | IN1          |
    | OUT          | OUT          |

  If EN of FUNCBLOCK_1 is set to "0", the link on output OUT of FUNCBLOCK_1 maintains the old status it had during the last correctly executed cycle.

- EN/ENO handling with function blocks that have one variable and one link as output parameters:
If EN of FUNCBLOCK_1 is set to “0”, the link on output OUT of FUNCBLOCK_1 maintains the old status it had during the last correctly executed cycle. The OUT1 variable on the same pin either retains its previous status or can be changed externally without influencing the link. The variable and the link are saved independently of each other.

- Functions/Procedures
  - As defined in IEC61131-3, the outputs from deactivated functions (EN input set to “0”) are undefined. (The same applies to procedures.)
  - Here nevertheless an explanation of the output statuses in this case:
    - EN/ENO handling with function/procedure blocks that (only) have one link as an output parameter:
      - If EN of FUNC_PROC_1 is set to “0”, the link on output OUT of FUNC_PROC_1 maintains the old status it had during the last correctly executed cycle.
    - EN/ENO handling with function/procedure blocks that have one variable and one link as output parameters:
      - If EN of FUNC_PROC_1 is set to “0”, the link on output OUT of FUNC_PROC_1 maintains the old status it had during the last correctly executed cycle. The OUT1 variable on the same pin either retains its previous status or can be changed externally without influencing the link. The variable and the link are saved independently of each other.

The output behavior of the FFBs does not depend on whether the FFBs are invoked without EN/ENO or with EN=1.
VAR_IN_OUT Variable

FFBs are often used to read a variable at an input (input variables), to process it and to output the altered values of the same variable (output variables). This special type of input/output variable is also called a VAR_IN_OUT variable. The link between input and output variables is represented by a line in the FFB.

The following special features are to be noted when using FFBs with VAR_IN_OUT variables:

- All VAR_IN_OUT inputs must be assigned a variable.
- Via graphical links only VAR_IN_OUT outputs with VAR_IN_OUT inputs can be connected.
- Only one graphical link can be connected to a VAR_IN_OUT input/output.
- A combination of variable/address and graphical connections is not possible for VAR_IN_OUT outputs.
- No literals or constants can be connected to VAR_IN_OUT inputs/outputs.
- No negations can be used on VAR_IN_OUT inputs/outputs.
- Different variables/variable components can be connected to the VAR_IN_OUT input and the VAR_IN_OUT output. In this case the value of the variables/variable component on the input is copied to the at the output variables/variable component.
Subroutine Calls

Calling a Subroutine

In FBD, subroutines are called using the following blocks:

<table>
<thead>
<tr>
<th>Condition</th>
<th>EN</th>
<th>ENO</th>
</tr>
</thead>
<tbody>
<tr>
<td>MySR</td>
<td></td>
<td>SR_NAME</td>
</tr>
</tbody>
</table>

If the status of EN is 1, the respective subroutine (variable name in SR_Name) is called.

The output ENO is not used to display the error status for this type of block. The output ENO is always 1 for this type of block and is used to call multiple subroutines simultaneously.

The following construction makes it possible to call multiple subroutines simultaneously:

<table>
<thead>
<tr>
<th>Condition</th>
<th>EN</th>
<th>ENO</th>
</tr>
</thead>
<tbody>
<tr>
<td>MySR1</td>
<td></td>
<td>SR_NAME</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Condition</th>
<th>EN</th>
<th>ENO</th>
</tr>
</thead>
<tbody>
<tr>
<td>MySR2</td>
<td></td>
<td>SR_NAME</td>
</tr>
</tbody>
</table>

The subroutine to be called must be located in the same task as the FBD section called.

Subroutines can also be called from within subroutines.

Subroutine calls are a supplement to IEC 61131-3 and must be enabled explicitly.

In SFC action sections, subroutine calls are only allowed when Multitoken Operation is enabled.
Control Elements

Introduction

Control elements are used for executing jumps within an FBD section and for returning from a subroutine (SRx) or derived function block (DFB) to the main program.

Control Elements

The following control elements are available:

<table>
<thead>
<tr>
<th>Designation</th>
<th>Representation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jump</td>
<td>NEXT</td>
<td>When the status of the left link is 1, a jump is made to a label (in the current section). To generate a conditional jump, a jump object is linked to a Boolean FFB output. To generate an unconditional jump, the jump object is assigned the value 1 for example, using the AND function.</td>
</tr>
<tr>
<td>Label</td>
<td>LABEL:</td>
<td>Labels (jump targets) are indicated as text with a colon at the end. This text is limited to 32 characters and must be unique within the entire section. The text must conform to general naming conventions. Jump labels can only be placed between the first two grid points on the left edge of the section. Note: Jump labels may not &quot;cut through&quot; networks, i.e. an assumed line from the jump label to the right edge of the section may not be crossed by any object. This is also valid for links.</td>
</tr>
<tr>
<td>Return</td>
<td>RETURN</td>
<td>Every subroutine and every DFB (derived function block) is exited after being processed, i.e. it returns to the main program called. If the subroutine/DFB is left prematurely, the return to the main program can be forced by the return object. If the status of the left link is 1, a return from the subroutine or DFB (derived function block) to the main program is carried out. Return objects can only be used in DFBs or SR subroutines. They cannot be used in the main program. To generate a conditional return, a return object is linked to a Boolean FFB output.</td>
</tr>
</tbody>
</table>
**Link**

**Description**
Links are vertical and horizontal connections between FFBs.

**Representation**
The link coordinates are identified by a filled circle.

Crossed links are indicated by a “broken” link.
**Programming Notes**

Attention should be paid to the following programming notes:

- Links can be used for any data type.
- The data types of the inputs/outputs to be linked must be the same.
- Several links can be connected with one FFB output. Only one may be linked with an FFB input however.
- Inputs and outputs may be linked to one-another. Linking more than one output together is not possible. That means that no OR connection is possible using links in FBD. An OR function is to be used in this case.
- Overlapping links with other objects is permitted.
- Links may not be used to create loops since the sequence of execution in this case cannot be clearly determined in the section. Loops must be created using actual parameters (see *Loop Planning, p. 309*).
- To avoid links crossing each other, links can also be represented in the form of connectors.

The source and target for the connection are labeled with a name that is unique within the section.

The connector name has the following structure depending on the type of source object for the connection:

- For functions: "Function counter/formal parameter" for the source of the connection

  ![Function Block Diagram](image)

  - For function blocks: "Instance name/formal parameter" for the source of the connection

  ![Function Block Diagram](image)
## Text Object

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Text can be positioned as text objects using FBD Function Block language. The size of these text objects depends on the length of the text. The size of the object, depending on the size of the text, can be extended vertically and horizontally to fill further grid units. Text objects may not overlap with FFBs; however they may overlap with links.</td>
</tr>
</tbody>
</table>
## Execution sequence of the FFBs

### Introduction

The execution sequence is determined by the position of the FFBs within the section (executed from left to right and from top to bottom). If the FFBs are then linked graphically, the execution sequence is determined by the signal flow.

The execution sequence is indicated by the execution number (number in the top right corner of the FFB frame).

### Execution Sequence on Networks

For network execution sequences, the following rules apply:

- Executing a section is completed network by network based on the FFB links from above and below.
- Links may not be used to create loops since the sequence of execution in this case cannot be clearly determined. Loops must be created using actual parameters (see *Loop Planning*, p. 309).
- The execution sequence for networks that are not linked is determined by the graphic sequence (from top-right to bottom-left). This execution sequence can be influenced (see *Change Execution Sequence*, p. 304).
- Processing on a network is ended completely before the processing begins on another network for which outputs are used on the previous network.
- No element of a network is deemed to be processed as long as the status of all inputs of this element are not calculated.
- Processing on a network is only ended if all outputs on this network have been processed.

### Signal flow within a network

For execution sequences within a network, the following rules apply:

- An FFB is only processed if all elements (FFB outputs etc.) with which its inputs are linked are processed.
- The execution sequence of FFBs that are linked with various outputs of the same FFB runs from top to bottom.
- The execution sequence of FFBs is not influenced by the location within the network.

This does not apply if more than one FFB is linked to the same output of the "calling" FFB. In this case, the execution sequence is determined by the graphic sequence (from top to bottom).
Priorities in Defining the Signal Flow Within a Section:

<table>
<thead>
<tr>
<th>Priority</th>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Link</td>
<td>Links have the highest priorities in defining the signal flow within a FBD section.</td>
</tr>
<tr>
<td>2</td>
<td>User Definition</td>
<td>User Access to Execution Sequence.</td>
</tr>
<tr>
<td>3</td>
<td>Network by Network</td>
<td>Processing on a network is ended completely before the processing begins on another network.</td>
</tr>
<tr>
<td>4</td>
<td>Output Sequence</td>
<td>FFBs that are linked to the outputs of the same &quot;calling&quot; FFB are processed from top to bottom.</td>
</tr>
<tr>
<td>5</td>
<td>Rung by Rung</td>
<td>Lowest priority. (Only applies if none of the other rules apply).</td>
</tr>
</tbody>
</table>

Example

Example of the Execution Sequence of Objects in an FBD Section:
Change Execution Sequence

Introduction

The execution order of networks and the execution order of objects within a network are defined by a number of rules (See Priorities, p. 303).

In some cases the execution order suggested by the system should be changed. The procedure for defining/changing the execution sequence of networks is as follows:

- Using links instead of actual parameters
- Network positions.
- Explicit execution sequence definition.

The procedure for defining/changing the execution sequence of networks is as follows:

- FFB positions.

Original situation

The following diagram shows two networks for which the execution sequences are simply defined by their positions within the section, without taking into account the fact that blocks .4/.5 and .7/.8 require a different execution sequence.
Link instead of actual parameters

By using a link instead of a variable the two networks are executed in the proper sequence (see also *Original situation, p. 304*).
The correct execution sequence can be achieved by changing the position of the networks in the section (see also *Original situation, p. 304*).
Explicit definition

The correct execution sequence can be achieved by explicitly changing the execution sequence of an FFB. To indicate that which FFB’s had their execution order changed, the execution number is shown in a black field (see also Original situation, p. 304).

Note: Only one reference of an instance is allowed, e.g. the instance ".7" may only be referenced once.
**FFB positions**

The position of FFBs only influences the execution sequence if more than one FFB is linked to the same output of the "calling" FFB (see also *Original situation, p. 304*).

In the first network, block positions 4 and 5 are switched. In this case (common origins for both block inputs) the execution sequence of both blocks is switched as well (processed from top to bottom).

In the second network, block positions 7 and 8 are switched. In this case (different origins for the block inputs) the execution sequence of the blocks is not switched (processed in the order the block outputs are called).
Loop Planning

Non-permitted Loops

Configuring loops exclusively via links is not permitted since it is not possible to clearly specify the signal flow (the output of one FFB is the input of the next FFB, and the output of this one is the input of the first).

Non-permitted Loops via Links

Generating via an actual parameter

This type of logic must be resolved using feedback variables so that the signal flow can be determined.

Feedback variables must be initialized. The initial value is used during the first execution of the logic. Once they have been executed the initial value is replaced by the actual value.

Pay attention to the two different types of execution sequences (number in brackets after the instance name) for the two blocks.

Loop generated with an actual parameter: Type 1

Loop generated with an actual parameter: Type 2
Introduction

Overview
This chapter describes the ladder diagram language LD which conforms to IEC-611311.

What's in this Chapter?
This chapter contains the following topics:

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>General information about the LD Ladder Diagram language</td>
<td>312</td>
</tr>
<tr>
<td>Contacts</td>
<td>315</td>
</tr>
<tr>
<td>Coils</td>
<td>317</td>
</tr>
<tr>
<td>Elementary Functions, Elementary Function Blocks, Derived Function Blocks and Procedures (FFBs)</td>
<td>319</td>
</tr>
<tr>
<td>Control elements</td>
<td>329</td>
</tr>
<tr>
<td>Operation and compare blocks</td>
<td>330</td>
</tr>
<tr>
<td>Links</td>
<td>332</td>
</tr>
<tr>
<td>Text Object</td>
<td>335</td>
</tr>
<tr>
<td>Edge recognition</td>
<td>336</td>
</tr>
<tr>
<td>Execution Sequence and Signal Flow</td>
<td>343</td>
</tr>
<tr>
<td>Loop Planning</td>
<td>345</td>
</tr>
<tr>
<td>Change Execution Sequence</td>
<td>347</td>
</tr>
</tbody>
</table>
General information about the LD Ladder Diagram language

Introduction

This section describes the Ladder Diagram (LD) according to IEC 61131-3. The structure of an LD section corresponds to a rung for relay switching.

The left power rail is located on the left-hand side of the LD editor. This left power rail corresponds to the phase (L ladder) of a rung. With LD programming, in the same way as in a rung, only the LD objects which are linked to a power supply, that is to say connected to the left power rail, are "processed". The right power rail corresponds to the neutral wire. However, all coils and FFB outputs are linked with it directly or indirectly, and this creates a power flow.

A group of objects which are linked together one below the other, and have no links to other objects (excluding the power rail), is called a network or a rung.
Representation of an LD section

Representation:

- **AND**
  - IN1
  - TRUE
  - 1

- **EN Input**
  - Variable
  - Literals

- **ENO Output**
  - Topological Addresses

- **Horizontal Boolean Link**
  - %IX1.5.4
  - %IX1.5.5

- **Vertical Boolean Link**
  - %M50

- **Normally Closed**
  - %M50

- **Jump**
  - %QX100

- **Label (Jump target)**
  - NEXT:

- **Left Power Rail**

- **Right Power Rail**

- **XOR**
  - IN4

- **ST Expression**
  - Error1
  - Result1

- **Error2**

- **Result2**
The objects of the LD programming language help to divide a section into a number of:

- Contacts (See Contacts, p. 315),
- Coils (See Coils, p. 317),
- EFs and EFBs (Elementary Functions (See Elementary Function, p. 319) and Elementary Function Blocks (See Elementary Function Block, p. 320)),
- DFBs (Derived Function Blocks (See DFB, p. 321))
- Procedures (See Procedure, p. 321)
- Control Elements (See Control elements, p. 329) and
- Operation and Comparison blocks (See Operation and compare blocks, p. 330)

that represent an extension to IEC 61131-3

These objects can be connected with each other by means of:

- Links (See Links, p. 332)
- Actual Parameters (See Parameters, p. 322) (FFBs only).

Comments regarding the section logic can be provided using text objects (see Text Object, p. 335).

One LD section consists of a window containing a single page.

This page has a grid that divides the section into rows and columns.

A width of 11-64 columns and 17-2000 lines can be defined for LD sections.

The LD programming language is cell oriented, i.e. only one object can be placed in each cell.

The processing sequence of the individual objects in an LD section is determined by the data flow within the section. Networks connected to the left power rail are processed from top to bottom (link to the left power rail). Networks that are independent of each other within the section are processed according to their position (from top to bottom) (see also Execution Sequence and Signal Flow, p. 343).

For a description of IEC conformity for the LD programming language, see IEC Conformity (See IEC Compliance, p. 601).
Contacts

Introduction

A contact is an LD element that transfers a status on the horizontal link to its right side. This status is the result of a Boolean AND operation on the status of the horizontal link on the left side with the status of the relevant Boolean actual parameter.

A contact does not change the value of the relevant actual parameter.

Contacts take up one cell.

The following are permitted as actual parameters:

- Boolean variables
- Boolean constants
- Boolean addresses (topological addresses or symbolic addresses)
- ST expression (See Expression, p. 467) delivering a Boolean result (e.g. VarA OR VarB)

ST expressions as actual parameters for contacts are a supplement to IEC 61131-3 and must be enabled explicitly.
Contact Types

The following contacts are available:

<table>
<thead>
<tr>
<th>Designation</th>
<th>Representation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normally open</td>
<td>xxx</td>
<td>In the case of normally open contacts, the status of the left link is transferred to the right link if the status of the relevant Boolean actual parameter (indicated with xxx) is ON. Otherwise, the status of the right link is OFF.</td>
</tr>
<tr>
<td>Normally closed</td>
<td>xxx</td>
<td>In the case of normally closed contacts, the status of the left link is transferred to the right link if the status of the relevant Boolean actual parameter (indicated with xxx) is OFF. Otherwise, the status of the right link is OFF.</td>
</tr>
<tr>
<td>Contact for detecting positive</td>
<td>xxx</td>
<td>With contacts for detection of positive transitions, the right link for a program cycle is ON if a transfer of the relevant actual parameter (labeled by xxx) goes from OFF to ON and the status of the left link is ON at the same time. Otherwise, the status of the right link is 0. Also see Edge recognition, p. 336.</td>
</tr>
<tr>
<td>transitions</td>
<td>p</td>
<td></td>
</tr>
<tr>
<td>Contact for detecting negative</td>
<td>xxx</td>
<td>With contacts for detection of negative transitions, the right link for a program cycle is ON if a transfer of the relevant actual parameter (labeled by xxx) goes from ON to OFF and the status of the left link is ON at the same time. Otherwise, the status of the right link is 0. Also see Edge recognition, p. 336.</td>
</tr>
<tr>
<td>transitions</td>
<td>n</td>
<td></td>
</tr>
</tbody>
</table>
Coils

Introduction
A coil is an LD element which transfers the status of the horizontal link on the left side, unchanged, to the horizontal link on the right side. The status is stored in the respective Boolean actual parameter.

Normally, coils follow contacts or FFBs, but they can also be followed by contacts. Coils take up one cell.

The following are permitted as actual parameters:
- Boolean variables
- Boolean addresses (topological addresses or symbolic addresses)

Coil types
The following coils are available:

<table>
<thead>
<tr>
<th>Designation</th>
<th>Representation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coil</td>
<td>xxx</td>
<td>With coils, the status of the left link is transferred to the relevant Boolean actual parameter (indicated by xxx) and the right link.</td>
</tr>
<tr>
<td>negated coil</td>
<td>xxx</td>
<td>With negated coils, the status of the left link is copied onto the right link. The inverted status of the left link is copied to the relevant Boolean actual parameter (indicated by xxx). If the left link is OFF, then the right link will also be OFF and the relevant Boolean actual parameter will be ON.</td>
</tr>
<tr>
<td>Coils for detecting positive transitions</td>
<td>xxx</td>
<td>With coils that detect positive transitions, the status of the left link is copied onto the right link. The relevant actual parameter of data type EBOOL (indicated by xxx) is 1 for a program cycle, if a transition of the left link from 0 to 1 is made. Also see Edge recognition, p. 336.</td>
</tr>
<tr>
<td>Coils for detecting negative transitions</td>
<td>xxx</td>
<td>With coils that detect negative transitions, the status of the left link is copied onto the right link. The relevant actual Boolean parameter (indicated by xxx) is 1 for a program cycle, if a transition of the left link from 1 to 0 is made. Also see Edge recognition, p. 336.</td>
</tr>
</tbody>
</table>
Set coil

With set coils, the status of the left link is copied onto the right link. The relevant Boolean actual parameter (indicated by xxx) is set to ON if the left link has a status of ON, otherwise it remains unchanged. The relevant Boolean actual parameter can be reset through the reset coil. Also see Edge recognition, p. 336.

Reset coil

With reset coils, the status of the left link is copied onto the right link. The relevant Boolean actual parameter (indicated by xxx) is set to OFF if the left link has a status of ON, otherwise it remains unchanged. The relevant Boolean actual parameter can be set through the set coil. Also see Edge recognition, p. 336.

Stop coil

With halt coils, if the status of the left link is 1, the program execution is stopped immediately. (With stop coils the status of the left link is not copied to the right link.)

Call coil

With call coils, the status of the left link is copied to the right link. If the status of the left link is ON then the respective sub-program (indicated by xxx) is called.

The subroutine to be called must be located in the same task as the calling LD section. Subroutines can also be called from within subroutines. Subroutines are a supplement to IEC 61131-3 and must be enabled explicitly.

In SFC action sections, call coils (subroutine calls) are only allowed when Multitoken Operation is enabled.

<table>
<thead>
<tr>
<th>Designation</th>
<th>Representation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set coil</td>
<td>xxx</td>
<td>With set coils, the status of the left link is copied onto the right link. The relevant Boolean actual parameter (indicated by xxx) is set to ON if the left link has a status of ON, otherwise it remains unchanged. The relevant Boolean actual parameter can be reset through the reset coil. Also see Edge recognition, p. 336.</td>
</tr>
<tr>
<td>Reset coil</td>
<td>xxx</td>
<td>With reset coils, the status of the left link is copied onto the right link. The relevant Boolean actual parameter (indicated by xxx) is set to OFF if the left link has a status of ON, otherwise it remains unchanged. The relevant Boolean actual parameter can be set through the set coil. Also see Edge recognition, p. 336.</td>
</tr>
<tr>
<td>Stop coil</td>
<td>xxx</td>
<td>With halt coils, if the status of the left link is 1, the program execution is stopped immediately. (With stop coils the status of the left link is not copied to the right link.)</td>
</tr>
<tr>
<td>Call coil</td>
<td>xxx</td>
<td>With call coils, the status of the left link is copied to the right link. If the status of the left link is ON then the respective sub-program (indicated by xxx) is called. The subroutine to be called must be located in the same task as the calling LD section. Subroutines can also be called from within subroutines. Subroutines are a supplement to IEC 61131-3 and must be enabled explicitly. In SFC action sections, call coils (subroutine calls) are only allowed when Multitoken Operation is enabled.</td>
</tr>
</tbody>
</table>
Elementary Functions, Elementary Function Blocks, Derived Function Blocks and Procedures (FFBs)

Introduction

FFB is the generic term for:
- Elementary Function (EF) (See Elementary Function, p. 319)
- Elementary Function Block (EFB) (See Elementary Function Block, p. 320)
- Derived Function Block (DFB) (See DFB, p. 321)
- Procedure (See Procedure, p. 321)

FFBs occupy 1 to 3 columns (depending on the length of the formal parameter names) and 2 to 33 lines (depending on the number of formal parameter rows).

Elementary Function

Functions have no internal states. If the input values are the same, the value on the output is the same every time the function is called. For example, the addition of two values always gives the same result.

An elementary function is represented graphically as a frame with inputs and one output. The inputs are always represented on the left and the output is always on the right of the frame.

The name of the function, i.e. the function type, is displayed in the center of the frame.

The execution number (See Execution Sequence and Signal Flow, p. 343) for the function is shown to the right of the function type.

The function counter is shown above the frame. The function counter is the sequential number of the function within the current section. Function counters cannot be modified.

With some elementary functions, the number of inputs can be increased.
Elementary Function Block

Elementary function blocks have internal states. If the input values are the same, the value on the output can be different each time the function is called. e.g. for a counter the value on the output is incremented.

An elementary function block is represented graphically as a frame with inputs and outputs. The inputs are always represented on the left and the outputs always on the right of the frame. The name of the function block, i.e. the function block type, is displayed in the center of the frame. The instance name is displayed above the frame.

Function blocks can have more than one output.

The name of the function block, i.e. the function block type, is displayed in the center of the frame.

The execution number (See Execution Sequence and Signal Flow, p. 343) for the function block is shown to the right of the function block type.

The instance name is displayed above the frame.

The instance name serves as a unique identification for the function block in a project.

The instance name is created automatically and has the following structure: FBI_n

FBI = Function Block Instance

n = sequential number of the function block in the project

This automatically generated name can be modified for clarification. The instance name (max. 32 characters) must be unique throughout the project and is not case-sensitive. The instance name must conform to general naming conventions.

Note: To conform to IEC61131-3, only letters are permitted as the first character of the name. If you want to use a numeral as your first character however, this must be enabled explicitly.
Derived function blocks (DFBs) have the same properties as elementary function blocks. The user can create them in the programming languages FBD, LD, IL, and/or ST.

The only difference to elementary function blocks is that the derived function block is represented as a frame with double vertical lines.

### Derived Function Block

![DFB Diagram]

### Procedure

Procedures are functions viewed technically.

The only difference to elementary functions is that procedures can occupy more than one output and they support data type `VAR_IN_OUT`.

To the eye, procedures are no different than elementary functions. Procedures are a supplement to IEC 61131-3 and must be enabled explicitly.

![Procedure Diagram]
Parameters

Inputs and outputs are required to transfer values to or from an FFB. These are called formal parameters.

Objects are linked to formal parameters; these objects contain the current process states. They are called actual parameters.

Formal and actual parameters:

At program runtime, the values from the process are transferred to the FFB via the actual parameters and then output again after processing.

Only one object (actual parameter) of the following types may be linked to FFB inputs:

- Contact
- Variable
- Address
- Literal
- ST Expression
  - ST expressions on FFB inputs are a supplement to IEC 61131-3 and must be enabled explicitly.
- Link

The following combinations of objects (actual parameters) can be linked to FFB outputs:

- one or more coils
- one or more contacts
- one variable
- a variable and one or more connections (but not for VAR_IN_OUT (See VAR_IN_OUT-Variable, p. 328) outputs)
- an address
- an address and one or more connections (but not for VAR_IN_OUT (See VAR_IN_OUT-Variable, p. 328) outputs)
- one or more connections (but not for VAR_IN_OUT (See VAR_IN_OUT-Variable, p. 328) outputs)
The data type of the object to be linked must be the same as that of the FFB input/output. If all actual parameters consist of literals, a suitable data type is selected for the function block.

Exception: For generic FFB inputs/outputs with data type ANY_BIT, it is possible to link objects of data type INT or DINT (not UINT and UDINT).

This is a supplement to IEC 61131-3 and must be enabled explicitly.

Example:

Allowed:

```
    AND
IntVar1 IN1 OUT IntVar3
IntVar2 IN2
```

Not allowed:

```
    AND_WORD
IntVar1 IN1 OUT IntVar3
IntVar2 IN2
```

(In this case, AND_INT must be used.)

Not all formal parameters have to be assigned an actual parameter. However, this does not apply in the case of negated pins. These must always be assigned an actual parameter. This is also the case with some formal parameter types. These types are shown in the following table.

Table of formal parameter types:

<table>
<thead>
<tr>
<th>Parameter type</th>
<th>EDT</th>
<th>STRING</th>
<th>ARRAY</th>
<th>ANY_ARRAY</th>
<th>IODDT</th>
<th>STRUCT</th>
<th>FB</th>
<th>ANY</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFB: Input</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>/</td>
<td>+</td>
<td>/</td>
<td>+</td>
</tr>
<tr>
<td>DFB: Output</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>/</td>
<td>/</td>
<td>-</td>
<td>/</td>
<td>+</td>
</tr>
<tr>
<td>EFB: VAR_IN_OUT</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>/</td>
<td>+</td>
</tr>
<tr>
<td>DFB: Input</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>/</td>
<td>+</td>
<td>/</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>DFB: VAR_IN_OUT</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>/</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>EFB: Output</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>/</td>
<td>+</td>
</tr>
<tr>
<td>EF: Input</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>/</td>
<td>+</td>
</tr>
<tr>
<td>EF: VAR_IN_OUT</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>/</td>
<td>/</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>EF: Output</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>/</td>
<td>-</td>
</tr>
</tbody>
</table>

35006144 04 June 2006 323
FFBs that use actual parameters on the inputs that have not yet received any value assignment, work with the initial values of these actual parameters.

If no value is allocated to a formal parameter, then the initial value will be used for executing the function block. If no initial value has been defined then the default value (0) is used.

If a formal parameter is not assigned a value and the function block/DFB is instanced more than once, then the subsequent instances are run with the old value.

<table>
<thead>
<tr>
<th>Parameter type</th>
<th>EDT</th>
<th>STRING</th>
<th>ARRAY</th>
<th>ANY_ARRAY</th>
<th>IODDT</th>
<th>STRUCT</th>
<th>FB</th>
<th>ANY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Procedure: Input</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Procedure: VAR_IN_OUT</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>/</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Procedure: Output</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>/</td>
<td>+</td>
<td>+</td>
</tr>
</tbody>
</table>

+  Actual parameter required
-  Actual parameter not required
/  not applicable
Public Variables

In addition to inputs/outputs, some function blocks also provide public variables. These variables transfer statistical values (values that are not influenced by the process) to the function block. They are used for setting parameters for the function block.

Public variables are a supplement to IEC 61131-3. The assignment of values to public variables is made using their initial values. Public variables are read via the instance name of the function block and the names of the public variables.

Example:

(FB1 has the public variables AREA_NR and OP_CTRL.)

Programming notes

Attention should be paid to the following programming notes:

- FFBs will only be processed when they are directly or indirectly connected to the left bus bar.
- If the FFB will be conditionally executed, the EN input may be pre-linked through contacts or other FFBs (also see EN and ENO, p. 326).
- Boolean inputs and outputs can be inverted.
- Special conditions apply when using VAR_IN_OUT variables (See VAR_IN_OUT-Variable, p. 328).
- Function block/DFB instances can be called multiple times (also see Multiple Function Block Instance Call, p. 326).
Multiple Function Block Instance Call

Function block/DFB instances can be called more than once; other than instances from communication EFBs and function blocks/DFBs with an \textit{ANY} output but no \textit{ANY} input: these can only be called once.

Calling the same function block/DFB instance more than once makes sense, for example, in the following cases:
- If the function block/DFB has no internal value or it is not required for further processing.
  In this case, memory is saved by calling the same function block/DFB instance more than once since the code for the function block/DFB is only loaded once. The function block/DFB is then handled like a "Function".
- If the function block/DFB has an internal value and this is supposed to influence various program segments, for example, the value of a counter should be increased in different parts of the program.
  In this case, calling the same function block/DFB means that temporary results do not have to be saved for further processing in another part of the program.

EN and ENO

One \textit{EN} input and one \textit{ENO} output can be used in all FFBs.

If the value of \textit{EN} is equal to "0" when the FFB is invoked, the algorithms defined by the FFB are not executed and \textit{ENO} is set to "0".

If the value of \textit{EN} is equal to "1" when the FFB is invoked, the algorithms defined by the FFB will be executed. After the algorithms have been executed successfully, the value of \textit{ENO} is set to "1". If an error occurs when executing these algorithms, \textit{ENO} is set to "0".

If \textit{ENO} is set to "0" (caused by \textit{EN}=0 or an error during execution):
- Function blocks
  - \textit{EN/ENO} handling with function blocks that (only) have one link as an output parameter:

\begin{center}
\begin{tabular}{|c|c|}
\hline
\textit{EN} & \textit{ENO} \\
\hline
\textit{IN1} & \textit{OUT} \\
\hline
\textit{IN2} & \\
\hline
\end{tabular}
\begin{tabular}{|c|c|}
\hline
\textit{EN} & \textit{ENO} \\
\hline
\textit{IN1} & \textit{OUT} \\
\hline
\textit{IN2} & \\
\hline
\end{tabular}
\end{center}

If \textit{ENO} of F\textsc{uncblock}_1 is set to "0", the link on output \textsc{out} of F\textsc{uncblock}_1 maintains the old status it had during the last correctly executed cycle.
- \textit{EN/ENO} handling with function blocks that have one variable and one link as output parameters:

\begin{center}
\begin{tabular}{|c|c|}
\hline
\textit{EN} & \textit{ENO} \\
\hline
\textit{IN1} & \textit{OUT} \\
\hline
\textit{IN2} & \\
\hline
\end{tabular}
\begin{tabular}{|c|c|}
\hline
\textit{EN} & \textit{ENO} \\
\hline
\textit{IN1} & \textit{OUT} \\
\hline
\textit{IN2} & \\
\hline
\end{tabular}
\end{center}

\begin{center}
\textsc{out1}
\end{center}
If EN of FUNCBLOCK_1 is set to "0", the link on output OUT of FUNCBLOCK_1 maintains the old status it had during the last correctly executed cycle. The OUT1 variable on the same pin either retains its previous status or can be changed externally without influencing the link. The variable and the link are saved independently of each other.

- Functions/Procedures
  As defined in IEC61131-3, the outputs from deactivated functions (EN input set to "0") are undefined. (The same applies to procedures.) Here nevertheless an explanation of the output statuses in this case:
  - EN/ENO handling with function/procedure blocks that (only) have one link as an output parameter:
    
    ![Ladder Diagram LD](image)

    If EN of FUNC_PROC_1 is set to "0", the link on output OUT of FUNC_PROC_1 maintains the old status it had during the last correctly executed cycle.

  - EN/ENO handling with function/procedure blocks that have one variable and one link as output parameters:
    
    ![Ladder Diagram LD](image)

    If EN of FUNC_PROC_1 is set to "0", the link on output OUT of FUNC_PROC_1 maintains the old status it had during the last correctly executed cycle. The OUT1 variable on the same pin either retains its previous status or can be changed externally without influencing the link. The variable and the link are saved independently of each other.

The output behavior of the FFBs does not depend on whether the FFBs are invoked without EN/ENO or with EN=1.
**VAR_IN_OUT-Variable**

FFBs are often used to read a variable at an input (input variables), to process it and to output the altered values of the *same* variable (output variables).

This special type of input/output variable is also called a **VAR_IN_OUT** variable.

The link between input and output variables is represented by a line in the FFB.

**VAR_IN_OUT variable**

```
<table>
<thead>
<tr>
<th>EXAMP1</th>
<th>.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>EN</td>
<td>ENO</td>
</tr>
<tr>
<td>IN1</td>
<td>OUT1</td>
</tr>
<tr>
<td>IN2</td>
<td>OUT2</td>
</tr>
<tr>
<td>IO1</td>
<td>IO1</td>
</tr>
<tr>
<td>Comb_IN_OUT</td>
<td>Comb_IN_OUT</td>
</tr>
</tbody>
</table>
```

The following special features are to be noted when using FFBs with **VAR_IN_OUT** variables.

- All **VAR_IN_OUT** inputs must be assigned a variable.
- Via graphical links only **VAR_IN_OUT** outputs with **VAR_IN_OUT** inputs can be connected.
- Only one graphical link can be connected to a **VAR_IN_OUT** input/output.
- A combination of variable/address and graphical connections is not possible for **VAR_IN_OUT** outputs.
- No literals or constants can be connected to **VAR_IN_OUT** inputs/outputs.
- No negations can be used on **VAR_IN_OUT** inputs/outputs.
- Different variables/variable components can be connected to the **VAR_IN_OUT** input and the **VAR_IN_OUT** output. In this case the value of the variables/variable component on the input is copied to the at the output variables/variable component.
Control elements

Introduction
Control elements are used for executing jumps within an LD section and for returning from a sub-routine (SRx) or derived function block (DFB) to the main program.

Control elements take up one cell.

Control elements
The following control elements are available:

<table>
<thead>
<tr>
<th>Designation</th>
<th>Representation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jump</td>
<td>![NEXT]</td>
<td>When the status of the left link is 1, a jump is made to a label (in the current section). To generate an unconditional jump, the jump object must be placed directly on the left power rail. To generate a conditional jump, a jump object is placed at the end of a series of contacts.</td>
</tr>
<tr>
<td>Tag</td>
<td>![LABEL:]</td>
<td>Tags (jump targets) are indicated as text with a colon at the end. This text is limited to 32 characters and must be unique within the entire section. The text must conform to general naming conventions. Jump labels can only be placed in the first cell directly on the power rail. <strong>Note:</strong> Jump labels may not “cut through” networks, i.e. an assumed line from the jump label to the right edge of the section may not be crossed by any object. This also applies to Boolean links and FFB links.</td>
</tr>
<tr>
<td>Return</td>
<td>![RETURN]</td>
<td>Every subroutine and every DFB (derived function block) is exited after being processed, i.e. it returns to the main program called. If the subroutine/DFB is left prematurely, the return to the main program can be forced by the return object. If the status of the left link is 1, a return from the subroutine or DFB (derived function block) to the main program is carried out. Return objects can only be used in DFBs or SR subroutines. They cannot be used in the main program. To generate a conditional return, a return object is placed at the end of a series of contacts.</td>
</tr>
</tbody>
</table>
Operation and compare blocks

Introduction

In addition to the objects defined in IEC 61131-3, there are several other blocks for executing ST instructions (See Instructions, p. 467) and ST expressions (See Expression, p. 467) and for simple compare operations. These blocks are only available in the LD programming language.

Objects

The following objects are available:

<table>
<thead>
<tr>
<th>Designation</th>
<th>Representation</th>
<th>Description</th>
</tr>
</thead>
</table>
| Operation block | ![OPERATE RES <FCT_NAME>(...)](image) | When the status of the left link is 1, the ST instruction in the block is executed. All ST instructions (See Instructions, p. 467) are allowed except the control instructions:  
- RETURN,  
- JUMP,  
- IF,  
- CASE,  
- FOR  
- etc.)  
For operate blocks, the state of the left link is passed to the right link (regardless of the result of the ST instruction). A block can contain up to 4096 characters. If not all characters can be displayed then the beginning of the character sequence will be followed by suspension points (...). An operation block takes up 1 line and 4 columns. Example:  

In the example, Instruction1 is executed if In1=1. Instruction2 is executed if In1=1 and In2=1 (the result of Instruction1 has no meaning for the execution of Instruction2). Out1 becomes 1 if In1=1 and In2=1 (the results of Instruction1 and Instruction2 have no meaning for the status of Out1).
Horizontal compare blocks used to execute a compare expression (\(<\), \(\leq\), \(\geq\), \(=\), \(!=\)) in the ST programming language. (Note: The same functionality is also possible using ST expressions (See Expression, p. 467).)

If the state of the left link is 1 and the result of the comparison is 1, the state of the right link is 1.

A horizontal matching block can contain up to 4096 characters. If not all characters can be displayed then the beginning of the character sequence will be followed by suspension points (...).

A horizontal matching block takes up 1 line and 2 columns.

**Example:**

In the example, `Compare1` is executed if `In1 = 1`, `Compare2` is executed if `In1 = 1, In2 = 1` and the result of `Compare1` is 1. Out1 becomes 1 if `In1 = 1, In2 = 1, the result of Compare1 = 1` and the result of `Compare2 = 1`.

<table>
<thead>
<tr>
<th>Designation</th>
<th>Representation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Horizontal Matching Block</td>
<td><code>COMPARE &lt;obj1 OP obj2&gt;</code></td>
<td>Horizontal compare blocks used to execute a compare expression ((&lt;), (\leq), (\geq), (=), (!=)) in the ST programming language. (Note: The same functionality is also possible using ST expressions (See Expression, p. 467)). If the state of the left link is 1 and the result of the comparison is 1, the state of the right link is 1. A horizontal matching block can contain up to 4096 characters. If not all characters can be displayed then the beginning of the character sequence will be followed by suspension points (...). A horizontal matching block takes up 1 line and 2 columns. Example:</td>
</tr>
</tbody>
</table>

```
```

In the example, `Compare1` is executed if `In1 = 1`, `Compare2` is executed if `In1 = 1, In2 = 1` and the result of `Compare1 = 1`. Out1 becomes 1 if `In1 = 1, In2 = 1, the result of Compare1 = 1` and the result of `Compare2 = 1`. |
Links

There are 2 different types of links:

- **Boolean Links**
  Boolean links consist of one or more segments linking Boolean objects (contacts, coils) with one another.
  There are different types of Boolean links as well:
  - **Horizontal Boolean Links**
    Horizontal Boolean links enable sequential contacts and coil switching.
  - **Vertical Boolean Links**
    Vertical Boolean links enable parallel contacts and coil switching.

- **FFB Links**
  FFB connections are a combination of horizontal and vertical segments that connect FFB inputs/outputs with other objects.

Connections:
Attention should be paid to the following general programming notes:

- The data types of the inputs/outputs to be linked must be the same.
- Links between parameters with variable lengths (e.g. ANY_ARRAY_INT) are not allowed.
- Several links can be connected with one output (right-hand side of one contact, one coil or one FFB output). However, only one link can be connected with an input (left-hand side of one contact, one coil or one FFB output).
- Unconnected contacts, coils and FFB inputs are specified as "0" by default.
- Links may not be used to create loops since the sequence of execution in this case cannot be clearly determined in the section. Loops must be created using actual parameters (see Loop Planning, p. 309).

Notes on Programming Boolean Links:

- Overlapping Boolean links with other objects is not permitted.
- The signal flow (power flow) is from left to right for Boolean links. Therefore, backwards links are not allowed.
- If two Boolean links are crossed, the links are connected automatically. Since crossing Boolean links is not possible, links are not indicated in any special way.

Notes on Programming FFB Links:

- At least one side of an FFB link must be connected with an FFB input or output.
- To differentiate them from Boolean links, FFB links are shown with a doubly thick line.
- The signal flow (power flow) in FFB links is from the FFB output to the FFB input, no matter which direction they are made in. Therefore, backwards links are allowed.
- Only FFB inputs and FFB outputs may be linked to one-another. Linking more than one FFB outputs together is not possible. That means that no OR connection is possible in LD using FFB links.
- Overlapping FFB links with other objects is permitted.
- Crossing FFB links is also permitted. Crossed links are indicated by a "broken" link.

- Connection points between more FFB links are shown with a filled circle.
To avoid links crossing each other, FFB links can also be represented in the form of connectors. The source and target for the FFB connection are labeled with a name that is unique within the section. The connector name has the following structure depending on the type of source object for the connection:

- For functions: "Function counter/formal parameter" for the source of the connection

- For function blocks: "Instance name/formal parameter" for the source of the connection

- For contacts: "OUT1_sequential number"

**Vertical Links**

The "Vertical Link" is special. The vertical link serves as a logical OR. With this form of the OR link, 32 inputs (contacts) and 64 outputs (coils, links) are possible.
Text Object

Introduction  Text can be positioned as text objects in the Ladder Diagram (LD). The size of these text objects depends on the length of the text. The size of the object, depending on the size of the text, can be extended vertically and horizontally to fill further grid units. Text objects may overlap with other objects.
Edge recognition

Introduction

During the edge recognition, a bit is monitored during a transition from 0 -> 1 (positive edge) or from 1 -> 0 (negative edge).

For this, the value of the bit in the previous cycle is compared to the value of the bit in the current cycle. In this case, not only the current value, but also the old value, are needed.

Instead of a bit, 2 bits are therefore needed for edge recognition (current value and old value).

Because the data type BOOL only offers one single bit (current value), there is another data type for edge recognition, EBOOL (expanded BOOL). In addition to edge recognition, the data type EBOOL provides an option for forcing. It must also be saved whether forcing the bit is enabled or not.

The data type EBOOL saves the following data:
- the current value of the bit in Value bit
- the old value of the bit in History bit
  (the content of the value bit is copied to the History bit at the beginning of each cycle)
- Information whether forcing of the bit is enabled in Force-Bit
  (0 = Forcing disabled, 1 = Forcing enabled)
Recognizing positive edges

A contact to recognize positive edges is used to recognize positive edges. With this contact, the right connection for a program cycle is 1 when the transition of the associated actual parameter (A) is from 0 to 1 and, at the same time, the status of the left connection is 1. Otherwise, the status of the right link is 0.

In the example, a positive edge of the variable A is supposed to be recognize and B should therefore be set for a cycle.

Anytime the value bit of A equals 1 and the history bit equals 0, B is set to 1 for a cycle (cycle 1, 4, and 9).
Recognizing negative edges

A contact to recognize negative edges is used to recognize negative edges. With this contact, the right connection for a program cycle is 1 when the transition of the associated actual parameter ($A$) is from 1 to 0 and, at the same time, the status of the left connection is 1. Otherwise, the status of the right link is 0.

In the example, a negative edge of the variable $A$ is supposed to be recognize and $B$ should therefore be set for a cycle.

Anytime the value bit of $A$ equals 0 and the history bit equals 1, $B$ is set to 1 for a cycle (cycle 2 and 8).
Forcing bits

When forcing bits, the value of the variable determined by the logic will be overwritten by the force value.

In the example, a negative edge of the variable $A$ is supposed to be recognized and $B$ should therefore be set for a cycle.

Anytime the value bit or force bit of $A$ equals 0 and the history bit equals 1, $B$ is set to 1 for a cycle (cycle 1 and 8).
Forcing of coils can cause the loss of edge recognition

Forcing of coils can cause the loss of edge recognition

In the example, when \( A \) equals 1, \( B \) should equal 1, and with a rising edge from \( A \), the coil \( B \) will be set for a cycle.

In this example, the variable \( B \) is first assigned to the coil, and then to the link to recognize positive edges.

At the beginning of the second cycle, the value bit of \( B \) equals 0. When forcing \( B \) within this cycle, the force bit and value bit are set to 1. While processing the first line of the logic in the third cycle, the history bit of the coil (\( B \)) will also be set to 1.

Problem:

During edge recognition (comparison of the value bit and the history bit) in the second line of the logic, no edge is recognized, because due to the updating, the value bit and history bit on line 1 of \( B \) are always identical.
Solution:

In this example, the variable B is first assigned to the link to recognize positive edges and then the coil.

At the beginning of the second cycle, the value bit of B equals 0. When forcing B within this cycle, the force bit and value bit are set to 1. While processing the first line of the logic in the third cycle, the history bit of the link (B) will remain set to 0.

Edge recognition recognizes the difference between value bits and history bit and sets the coil (C) to 1 for one cycle.

Using set coil or reset coil can cause the loss of edge recognition with EBOOL variables.

The variable above the set/reset coil (variable C in the example) is always affected by the value of the left link.

If the left link is 1, the value bit (variable C in the example) is copied to the history bit and the value bit is set to 1.

If the left link is 0, the value bit (variable C in the example) is copied to the history bit, but the value bit is not changed.

This means that whatever value the left link has before the set or reset coil, the history bit is always updated.
In the example, a positive edge of the variable C should be recognized and set D for a cycle.

<table>
<thead>
<tr>
<th>Code line</th>
<th>Behavior in LD</th>
<th>Corresponds to in ST</th>
</tr>
</thead>
</table>
| 1         | Original situation: C = 0, History bit = 0 | IF A AND B 
THEN C := 1; 
ELSE C := C; 
END_IF; |
|           | A = 1, B = 1, C = 1, History bit = 0 | IF NOT(A) AND NOT(B) 
THEN C := 0; 
ELSE C := C; 
END_IF; |
| 2         | C = 1, History = 1 | - |
|           | D = 0, as the value bit and history bit of C are identical. The rising edge of C, shown in code line 1, is not recognized by the code in line 2, as this forces the history bit to be updated. (If the condition is FALSE, the present value of C is again assigned to C, see ELSE statement in code line 2 in ST example.) | - |
Execution Sequence and Signal Flow

**Execution Sequence of Networks**

The following rules apply to network execution sequences:

- Executing a section is completed network by network based on the object links from above and below.
- Links may not be used to create loops since the sequence of execution in this case cannot be clearly determined. Loops must be created using actual parameters (see *Loop Planning, p. 345*).
- The execution sequence of networks which are only linked by the left power rail, is determined by the graphical sequence (from top to bottom) in which these are connected to the left power rail. This does not apply if the sequence is influenced by control elements.
- Processing on a network is ended completely before the processing begins on another network.
- No element of a network is deemed to be processed until the status of all inputs of this element have been processed.
- Processing on a network is only ended if all outputs on this network have been processed. This also applies if the network contains one or more control elements.

**Signal flow within a network**

For signal flow within a network (rungs), the following rules apply:

- The signal flow for Boolean links is:
  - left to right with horizontal Boolean links and
  - from top to bottom with vertical Boolean links.
- The signal flow with FFB links is from the FFB output to the FFB input, regardless of which direction they are made in.
- An FFB is only processed if all elements (FFB outputs etc.) to which its inputs are linked are processed.
- The execution sequence of FFBs that are linked with various outputs of the same FFB runs from top to bottom.
- The execution sequence of objects is not influenced by their positions within the network.
- The execution sequence for FFBs is represented as execution number by the FFB.
Priorities

Priorities when defining the signal flow within a section:

<table>
<thead>
<tr>
<th>Priority</th>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Link</td>
<td>Links have the highest priorities in defining the signal flow within an LD section.</td>
</tr>
<tr>
<td>2</td>
<td>Network by Network</td>
<td>Processing on a network is ended completely before the processing begins on another network.</td>
</tr>
<tr>
<td>3</td>
<td>Output sequence</td>
<td>Outputs of the same function block or outputs to vertical links are processed from top to bottom.</td>
</tr>
<tr>
<td>4</td>
<td>Rung by Rung</td>
<td>Lowest priority. The execution sequence of networks which are only linked by the left power rail, is determined by the graphical sequence (from top to bottom) in which these are connected to the left power rail. (Only applies if none of the other rules apply).</td>
</tr>
</tbody>
</table>

Example

Example of the execution sequence of objects in an LD section:

Note: The execution numbers for contacts and coils is not shown. They are only shown in the graphic to provide a better overview.
Loops Planning

Non-permitted Loops

Creating loops using links alone is not permitted because it is not possible to clearly define the signal flow (the output of one FFB is the input of the next FFB, and the output of this one is the input of the first again).

Non-permitted loops via links:
Generating via an actual parameter

This type of logic must be generated using feedback variables so that the signal flow can be determined.

Feedback variables must be initialized. The initial value is used during the first execution of the logic. Once they have been executed, the initial value is replaced by the actual value.

Pay attention to the two different types of execution sequences (number in brackets after the instance name) for the two blocks.

Loop generated with an actual parameter: Type 1

Loop generated with an actual parameter: Type 2
Change Execution Sequence

Introduction

The order of execution in networks and the execution order of objects within a network are defined by a number of rules (See Execution Sequence and Signal Flow, p. 343).

In some cases the execution order suggested by the system should be changed. The procedure for defining/changing the execution sequence of networks is as follows:

- Using Links Instead of Actual Parameters
- Network Positions

The procedure for defining/changing the execution sequence of networks is as follows:

- Positioning of Objects
The following representation shows two networks for which the execution sequences are only defined by their position within the section, without taking into account that block 0.4/0.5 and 0.7/0.8 require another execution sequence.
By using a link instead of a variable the two networks are run in the proper sequence (see also Original Situation, p. 348).
The correct execution sequence can be achieved by changing the position of the networks in the section (see also *Original Situation*, p. 348).
Positioning of Objects

The position of objects can only have an influence on the execution order if several inputs (left link of Contacts/Coils, FFB inputs) are linked with the same output of the object "to be called" (right link of Contacts/Coils, FFB outputs) (see also Original Situation, p. 348).

Original situation:

In the first network, block positions 0.1 and 0.2 are switched. In this case (common origins for both block inputs) the execution sequence of both blocks is switched as well (processed from top to bottom). The same applies when switching coils C and D in the second network.
In the third network, block positions 0.4 and 0.5 are switched. In this case (different origins for the block inputs) the execution sequence of the blocks is not switched (processed in the sequence that the block outputs are called in). The same applies when switching coils G and H in the last network.
SFC Sequence Language

Introduction

Overview

This chapter describes the SFC sequence language which conforms to IEC 611311.

What's in this Chapter?

This chapter contains the following sections:

<table>
<thead>
<tr>
<th>Section</th>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>13.1</td>
<td>General information about SFC sequence language</td>
<td>354</td>
</tr>
<tr>
<td>13.2</td>
<td>Steps and Macro Steps</td>
<td>360</td>
</tr>
<tr>
<td>13.3</td>
<td>Actions and Action Sections</td>
<td>369</td>
</tr>
<tr>
<td>13.4</td>
<td>Transitions and Transition Sections</td>
<td>377</td>
</tr>
<tr>
<td>13.5</td>
<td>Jump</td>
<td>382</td>
</tr>
<tr>
<td>13.6</td>
<td>Link</td>
<td>383</td>
</tr>
<tr>
<td>13.7</td>
<td>Branches and Merges</td>
<td>384</td>
</tr>
<tr>
<td>13.8</td>
<td>Text Objects</td>
<td>388</td>
</tr>
<tr>
<td>13.9</td>
<td>Single-Token</td>
<td>389</td>
</tr>
<tr>
<td>13.10</td>
<td>Multi-Token</td>
<td>400</td>
</tr>
</tbody>
</table>
13.1 General information about SFC sequence language

Introduction

Overview
This section contains a general overview of the SFC sequence language.

What's in this Section?
This section contains the following topics:

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>General information about SFC sequence language</td>
<td>355</td>
</tr>
<tr>
<td>Link rules</td>
<td>359</td>
</tr>
</tbody>
</table>
General information about SFC sequence language

Introduction
The sequence language SFC (Sequential Function Chart), which conforms to IEC 61131-3, is described in this section.

Structure of a Sequence Controller
IEC conforming sequential control is created in Unity Pro from SFC sections (top level), transition sections and action sections.

These SFC sections are only allowed in the Master Task of the project. SFC sections cannot be used in other tasks or DFBs.

In Single Token, each SFC section contains exactly one SFC network (sequence).

In Multi-Token, an SFC section can contain one or more independent SFC networks.

Objects
An SFC section provides the following objects for creating a program:
- Step (See Step, p. 361)
- Macro Step (embedded sub-step) (See Macro Steps and Macro Sections, p. 365)
- Transition (transition condition) (See Transition, p. 378)
- Jump (See Jump, p. 382)
- Link (See Link, p. 383)
- Alternative branch (See Alternative Branches and Alternative Joints, p. 385)
- Alternative junction (See Alternative Branches and Alternative Joints, p. 385)
- Parallel branch (See Parallel Branch and Parallel Joint, p. 387)
- Parallel junction (See Parallel Branch and Parallel Joint, p. 387)

Comments regarding the section logic can be provided using text objects (related topics Text Object, p. 388).
Representation of an SFC section

Appearance:

SFC Sequence Language
An SFC section is a "Status Machine", i.e. the status is created by the active step and the transitions pass on the switch/change behavior. Steps and transitions are linked to one another through directional links. Two steps can never be directly linked and must always be separated by a transition. The active signal status processes take place along the directional links and are triggered by switching a transition. The direction of the chain process follows the directional links and runs from the end of the preceding step to the top of the next step. Branches are processed from left to right.

Every step has zero or more actions. A transition condition is necessary for every transition.

The last transition in the chain is always connected to another step in the chain (via a graphic link or jump symbol) to create a closed loop. Step chains are therefore processed cyclically.

When an SFC section is created, it is automatically assigned a variable of data type SFCCHART_STATE. The variable that is created always has the name of the respective SFC section.

This variable is used to assign the SFC control blocks to the SFC section to be controlled.

The behavior of an SFC network is greatly affected by the number of tokens selected, i.e. the number of active steps. Explicit behavior is possible by using one token (single token). (Parallel branches each with an active token [step] per branch as a single token). This corresponds to a step chain as defined in IEC 61131-3).

A step chain with a number of maximum active steps (Multi Token) defined by the user increases the degree of freedom. This reduces/eliminates the restrictions for enforcing unambiguousness and non-blocking and must be guaranteed by the user.

Step chains with Multi Token do not conform to IEC 61131-3.

An SFC section consists of a single-page window.
- The window has a logical grid of 200 lines and 32 columns.
- Steps, transitions and jumps each require a cell.
- Branches and links do not require their own cells, they are inserted in the respective step or transition cell.
- A maximum of 1024 steps can be placed per SFC section (including all their macro sections).
- A maximum of 100 steps can be active (Multi Token) per SFC section (including all their macro sections).
- A maximum of 64 steps can be set manually at the same time per SFC section (Multi Token).
- A maximum of 100 steps can be active at the same time per SFC section (Multi Token).
SFC Sequence Language

- A maximum of 20 actions can be assigned to each SFC step.
- The nesting depth of macros, i.e., macro steps within macro steps, is to 8 levels.

IEC Conformity

For a description of the extent to which the SFC programming language conforms to IEC, see IEC Conformity (See IEC Compliance, p. 601).
Link rules

The table indicates which object outputs can be linked with which object inputs.

<table>
<thead>
<tr>
<th>From object output of</th>
<th>To object input of</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step</td>
<td>Transition</td>
</tr>
<tr>
<td></td>
<td>Alternative Branch</td>
</tr>
<tr>
<td></td>
<td>Parallel joint</td>
</tr>
<tr>
<td>Transition</td>
<td>Step</td>
</tr>
<tr>
<td></td>
<td>Jump</td>
</tr>
<tr>
<td></td>
<td>Parallel Branch</td>
</tr>
<tr>
<td></td>
<td>Alternative joint</td>
</tr>
<tr>
<td>Alternative Branch</td>
<td>Transition</td>
</tr>
<tr>
<td>Alternative joint</td>
<td>Step</td>
</tr>
<tr>
<td></td>
<td>Jump</td>
</tr>
<tr>
<td></td>
<td>Parallel Branch</td>
</tr>
<tr>
<td></td>
<td>Alternative joint</td>
</tr>
<tr>
<td>Parallel Branch</td>
<td>Step</td>
</tr>
<tr>
<td></td>
<td>Jump</td>
</tr>
<tr>
<td></td>
<td>Alternative joint</td>
</tr>
<tr>
<td></td>
<td>Alternative joint</td>
</tr>
<tr>
<td></td>
<td>Alternative branch (only with Multi-Token (See Multi-Token, p. 400))</td>
</tr>
<tr>
<td>Parallel joint</td>
<td>Transition</td>
</tr>
<tr>
<td></td>
<td>Alternative branch (only with Multitoken (See Multi-Token, p. 400))</td>
</tr>
<tr>
<td></td>
<td>Alternative joint</td>
</tr>
</tbody>
</table>
13.2 Steps and Macro Steps

Introduction

Overview
This section describes the step and macro step objects of the SFC sequence language.

What's in this Section?
This section contains the following topics:

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step</td>
<td>361</td>
</tr>
<tr>
<td>Macro Steps and Macro Sections</td>
<td>365</td>
</tr>
</tbody>
</table>
### Step Types

The following types of steps exist:

<table>
<thead>
<tr>
<th>Type</th>
<th>Representation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Normal&quot; Step</td>
<td>S_1_2</td>
<td>A step becomes active when the previous step becomes inactive (a delay that may be defined must pass) and the upstream transition is satisfied. A step normally becomes inactive when a delay that may be defined passes and the downstream transition is satisfied. For a parallel joint, all previous steps must satisfy these conditions. Zero or more actions belong to every step. Steps without action are known as waiting steps.</td>
</tr>
<tr>
<td>Initial step</td>
<td>S_1_1</td>
<td>The initial status of a sequence string is characterized by the initial step. After initializing the project or initializing the sequence string, the initial step is active. Initial steps are not normally assigned with any actions. With Single-Token (Conforming with IEC 61131-3) only one initial step is allowed per sequence. With Multi-Token, a definable number (0 to 100) of initial steps are possible.</td>
</tr>
<tr>
<td>Macro Step</td>
<td>MS_1_1</td>
<td>See Macro Step, p. 365</td>
</tr>
<tr>
<td>Input step</td>
<td>MS_1_1_IN</td>
<td>see Input Step, p. 365</td>
</tr>
<tr>
<td>Output step</td>
<td>MS_1_1_OUT</td>
<td>see Output Step, p. 366</td>
</tr>
</tbody>
</table>
Step Names

When creating a step, it is assigned with a suggested number. The suggested number is structured as follows $S_{i,j}$, whereas $i$ is the (internal) current number of the section and $j$ is the (internal) current step number in the current section.

You can change the suggested numbers to give you a better overview. Step names (maximum 32 characters) must be unique over the entire project, i.e. no other step, variable or section etc. may exist with the same name. There are no case distinctions. The step name must correspond with the standardized name conventions.

Step Times

Each step can be assigned a minimum supervision time, a maximum supervision time and a delay time:

- **Minimum Supervision Time**
  The minimum supervision time sets the minimum time for which the step should normally be active. If the step becomes inactive before this time has elapsed, an error message is generated. In animation mode, the error is additionally identified by a colored outline (yellow) around the step object.
  If no minimum supervision time or a minimum supervision time of 0 is entered, step supervision is not carried out.
  The error status remains the same until the step becomes active again.

- **Maximum Supervision Time**
  The maximum supervision time specifies the maximum time in which the step should normally be active. If the step is still active after this time has elapsed, an error message is generated. In animation mode, the error is additionally identified by a colored outline (pink) around the step object.
  If no maximum supervision time or a maximum supervision time of 0 is entered, step supervision is not carried out.
  The error status remains the same until the step becomes inactive.

- **Delay Time**
  The delay time (step dwell time) sets the minimum time for which the step must be active.

Note: The defined times apply for the step only, not for the allocated actions. Individual times can be defined for these.

Setting the Step Times

The following formula is to be used for defining/determining these times:

```
Delay time < minimum supervision time < maximum supervision time
```

There are 2 ways to assign the defined values to a step:

- As a duration literal
- Use of the data structure SFCSTEP_TIMES
SFCSTEP_TIME
S variable

Every step can be implicitly allocated a variable of data type SFCSTEP_TIMES. The elements for this data structure can be read from and written to (read/write).

The data structure is handled the same as any other data structure, i.e. they can be used in variable declarations and therefore accessing the entire data structure (e.g. as FFB parameter) is possible.

Structure of the Data Structure:

<table>
<thead>
<tr>
<th>Element Name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;VarName&quot;.delay</td>
<td>TIME</td>
<td>Delay Time</td>
</tr>
<tr>
<td>&quot;VarName&quot;.min</td>
<td>TIME</td>
<td>Minimum Supervision Time</td>
</tr>
<tr>
<td>&quot;VarName&quot;.max</td>
<td>TIME</td>
<td>Maximum Supervision Time</td>
</tr>
</tbody>
</table>
SFCSTEP_STAT

Every step is implicitly allocated a variable of data type SFCSTEP_STATE. This step variable has the name of the allocated step. The elements for this data structure can only be read (read only).

The data structure cannot be used in variable declarations. Therefore, accessing the entire data structure (e.g. as FFB parameter) is not possible.

Structure of the Data Structure:

<table>
<thead>
<tr>
<th>Element Name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;StepName&quot;.t</td>
<td>TIME</td>
<td>Current dwell time in the step. If the step is deactivated, the value of this element is retained until the step is activated again.</td>
</tr>
</tbody>
</table>
| "StepName".x         | BOOL      | 1: Step active  
                        0: Step inactive                                                           |
| "StepName".tminErr   | BOOL      | This element is a supplement to IEC 61131-3.  
                        1: Underflow of minimum supervision time  
                        0: No underflow of minimum supervision time  
                        The element is automatically reset in the following cases:  
                        • If the step is activated again  
                        • If the sequence control is reset  
                        • If the command button Reset Time Error is activated |
| "StepName".tmaxErr   | BOOL      | This element is a supplement to IEC 61131-3.  
                        1: Overflow of maximum supervision time  
                        0: No overflow of maximum supervision time  
                        The element is automatically reset in the following cases:  
                        • If the step is exited  
                        • If the sequence control is reset  
                        • If the command button Reset Time Error is activated |
Macro Steps and Macro Sections

Macro Step
Macro steps are used for calling macro sections and thus for hierarchical structuring of sequential controls.

Representation of a Macro Step:

Macro steps have the following properties:
- Macro steps can be positioned in "Sequence Control" sections and in macro sections.
- The number of macro steps is unlimited.
- The nesting depth, i.e. macro steps within macro steps is to 8 levels.
- Each macro step is implicitly allocated a variable of data type SFCSTEP_STATE, see SFCSTEP_STATE variable, p. 364.
- Macro steps can be allocated a variable of data type SFCSTEP_TIMES, see SFCSTEP_TIMES variable, p. 363.
- Macro steps can NOT be allocated with actions.
- Each macro step can be replaced with the sequence string in the allocated macro section.

Macro steps are a supplement to IEC 61131-3 and must be enabled explicitly.

Input Step
Every macro section begins with an input step.

Representation of an input step:

Input steps have the following properties:
- Input steps can only be placed in macro sections.
- Only one individual input step can be placed for each macro section.
- Each input step is implicitly allocated a variable of data type SFCSTEP_STATE, see SFCSTEP_STATE variable, p. 364.
- Input steps can be allocated a variable of data type SFCSTEP_TIMES, see SFCSTEP_TIMES variable, p. 363.
- Input steps can be allocated actions.
Output Step

Every macro section ends with an output step.

Representation of an output step:

```
MS_1_1_OUT
```

Output steps have the following properties:
- Output steps can only be placed in macro sections.
- Only one individual output step can be placed for each macro section.
- Output steps can NOT be allocated with actions.
- Output steps can only be assigned a delay time. Assigning supervision times is not possible, see Step Times, p. 362.

Macro Section

A macro section consists of a single sequence string having principally the same elements as a "sequence control" section (e.g. steps, initial step[s], macro steps, transitions, branches, joints, etc.).

Additionally, each macro section contains an input step at the beginning and an output step at the end.

Each macro step can be replaced with the sequence string in the allocated macro section.

Therefore, macro sections can contain 0, 1 or more initial steps, see Step Types, p. 361 as well.
- Single-Token
  - 0 Initial steps are used in macro sections, if there is already an initial step in the higher or lower section.
  - 1 Initial step is used in macro sections, if there are no initial steps in the higher or lower section.
- Multi-Token
  A maximum of 100 initial steps can be placed per section (including all their macro sections).
Using macro sections:

The name of the macro section is identical to the name of the macro step that it is called from. If the name of the macro step is changed then the name of the respective macro section is changed automatically.

A macro section can only be used once.

Macro Step Processing:

<table>
<thead>
<tr>
<th>Phase</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A macro step is activated if the previous transition condition is TRUE. At the same time, the input step in the macro section is activated.</td>
</tr>
<tr>
<td>2</td>
<td>The sequence string of the macro section is processed. The macro step remains active as long as at least one step in the macro section is active.</td>
</tr>
<tr>
<td>3</td>
<td>If the output step of the macro section is active then the transitions following the macro step are enabled.</td>
</tr>
<tr>
<td>4</td>
<td>The macro step becomes inactive when the output step is activated which causes the following transition conditions to be enabled and the transition condition to be TRUE. At the same time, the output step in the macro section is activated.</td>
</tr>
</tbody>
</table>
Step Names

When creating a step, it is assigned with a suggested number.

Meanings of the Suggested Numbers:

<table>
<thead>
<tr>
<th>Step Type</th>
<th>Suggested Number</th>
<th>Description</th>
</tr>
</thead>
</table>
| Macro Step             | MS_{i}_j         | MS = Macro Step  
  \(i\) = (internal) current (sequential) number of the current section  
  \(j\) = (internal) current (sequential) macro step number of the current section |
| Input step             | MS_{k_l}IN       | MS = Macro Step  
  \(k\) = (internal) current (sequential) number of the calling section  
  \(l\) = (internal) current (sequential) macro step number of the calling section  
  \(IN\) = Input Step |
| Output step            | MS_{k_l}OUT      | MS = Macro Step  
  \(k\) = (internal) current (sequential) number of the calling section  
  \(l\) = (internal) current (sequential) macro step number of the calling section  
  \(OUT\) = Output Step |
| "Normal" Step (within a macro section) | S_{k_m}     | S = Step  
  \(k\) = (internal) current (sequential) number of the calling section  
  \(m\) = (internal) current (sequential) step number of the calling section |

You can change the suggested numbers to give you a better overview. Step names (maximum 28 characters for macro step names, maximum 32 characters for step names) must be unique within the entire project, i.e. no other step, variable or section (with the exception of the name of the macro section assigned to the macro step) etc. may exist with the same name. There are no case distinctions. The step name must correspond with the standardized name conventions.

If the name of the macro step is changed then the name of the respective macro section and the steps within it are changed automatically.

For example if MS_{1_1} is renamed to MyStep then the step names in the macro section are renamed to MyStep_{IN}, MyStep_{1}, ..., MyStep_{n}, MyStep_{OUT}. 
13.3 Actions and Action Sections

Introduction

Overview

This section describes the actions and action sections of the SFC sequence language.

What's in this Section?

This section contains the following topics:

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Action</td>
<td>370</td>
</tr>
<tr>
<td>Action Section</td>
<td>372</td>
</tr>
<tr>
<td>Identifier</td>
<td>373</td>
</tr>
</tbody>
</table>
Action

Introduction

Actions have the following properties:

- An action can be a Boolean variable (action variable (See Action Variable, p. 370)) or a section (action section (See Action Section, p. 372)) of programming language FDB, LD, IL or ST.
- A step can be assigned none or several actions. A step which is assigned no action has a waiting function, i.e. it waits until the assigned transition is completed.
- If more than one action is assigned to a step they are processed in the sequence in which they are positioned in the action list field.
  Exception: Independent of their position in the action list field, actions with the identifier (See Available Identifiers, p. 373) P1 are always processed first and actions with the identifier P0 are processed last.
- The control of actions is expressed through the use of identifiers (See Identifier, p. 373).
- A maximum of 20 actions can be assigned to each step.
- The action variable that is assigned to an action can also be used in actions from other steps.
- The action variable can also be used for reading or writing in any other section of the project (multiple assignment).
- Actions that are assigned an identifier with duration can only be activated one time.
- Only Boolean variables/addresses or Boolean elements of multi-element variables are allowed as action variables.
- Actions have unique names.
  The name of the action is either the name of the action variable or the name of the action section.

Action Variable

The following are authorized as action variables:

- Address of data type BOOL
  An action can be assigned to a hardware output using an address. In this case, the action can be used as enable signal for a transition, as input signal in another section and as output signal for the hardware.
- Simple variable or element of a multi-element variable of data type BOOL
  The action can be used as an input signal with assistance from a variable in another section.
  - Unlocated Variable
    With unlocated variables, the action can be used as enable signal for a transition and as input signal in another section.
  - Located Variable
With located variables the action can be used as an enabling signal for a transition, as an input signal in another section and as an output signal for the hardware.

**Action Names**

If an address or a variable is used as an action then that name (e.g. %Q10.4, Variable1) is used as the action name.

If an action section is used as an action then the section name is used as the action name.

Action names (maximum 32 characters) must be unique over the entire project, i.e. no other transition, variable or section etc. may exist with the same name. There are no case distinctions. The action name must correspond with the standardized name conventions.
### Action Section

**Introduction**

An action section can be created for every action. This is a section which contains the logic of the action and it is automatically linked with the action.

**Name of the Action Section**

The name of the action section is always identical to the assigned action, see *Action Names*, p. 371.

**Programming languages**

FBD, LD, IL and ST are possible as programming languages for action sections.

**Properties of Action sections**

Action sections have the following properties:

- Action sections can have any amount of outputs.
- Subroutine calls are only possible in action sections when Multitoken operation is enabled.
  
  **Note:** The called subroutines are not affected by the controller of the sequence string, i.e.
  - the identifier assigned to the called action section does not affect the subroutine,
  - the subroutine also remains active when the called step is deactivated.
- No diagnosis functions, diagnosis function blocks or diagnosis procedures may be used in action sections.
- Action sections can have any amount of networks.
- Action sections belong to the SFC section in which they were defined and can be assigned any number of actions within this SFC section (including all of their macro sections).
- Action sections which are assigned an identifier with duration, can only be activated one time.
- Action sections belong to the SFC section that they were defined in. If the respective SFC section is deleted then all action sections of this SFC section are also deleted automatically.
- Action sections can be called exclusively from actions.
Identifier

Introduction
Each action that is linked to a step must have an identifier which defines the control for that action.

Available Identifiers
The following identifiers are available:

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Meaning</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>N / None</td>
<td>Not Stored</td>
<td>If the step is active then action is 1 and if the step is inactive the action is 0.</td>
</tr>
<tr>
<td>R</td>
<td>Overriding reset</td>
<td>The action, which is set in another step with the identifier S, is reset. The activation of any action can also be prevented. <strong>Note:</strong> Identifiers are automatically declared as unbuffered. This means that the value is reset to 0 after stop and cold restart, e.g. when voltage is on/off. Should a buffered output be required, please use the RS or SR function block from the standard block library.</td>
</tr>
</tbody>
</table>
Set (saved) The set action also remains active, when the associated step is inactive. The action only becomes inactive when it is reset in another step in the current sequence string using the identifier \( S \), i.e. resetting the action from another section is not possible for example.

Note: Identifiers are automatically declared as unbuffered. This means that the value is reset to 0 after stop and cold restart, e.g. when voltage is on/off. Should a buffered output be required, please use the RS or SR function block from the standard block library.

Note: A maximum of 100 actions are permitted using the \( S \) identifier per SFC Section.

Time limited If the step is active, the action is also active. After the process of the time duration, defined manually for the action, the action returns to 0, even if the step is still active. The action also becomes 0 if the step is inactive.

Note: For this identifier, an additional duration of data type TIME must be defined.
### Identifier Meaning Description

#### S
- **Set (saved)** The set action also remains active, when the associated step is inactive. The action only becomes inactive when it is reset in another step in the current sequence string using the identifier `R`, i.e. resetting the action from another section is not possible for example.

**Note:** Identifiers are automatically declared as unbuffered. This means that the value is reset to 0 after stop and cold restart, e.g. when voltage is on/off. Should a buffered output be required, please use the `RS` or `SR` function block from the standard block library.

**Note:** A maximum of 100 actions are permitted using the `S` identifier per SFC Section.

#### L
- **Time limited** If the step is active, the action is also active. After the process of the time duration, defined manually for the action, the action returns to 0, even if the step is still active. The action also becomes 0 if the step is inactive.

**Note:** For this identifier, an additional duration of data type `TIME` must be defined.
If the step is active, the internal timer is started and the action becomes 1 after the process of the time duration, which was defined manually for the action. If the step becomes inactive after that, the action becomes inactive as well. If the step becomes inactive before the internal time has elapsed then the action does not become active.

Note: For this identifier, an additional duration of data type TIME must be defined.

If the step becomes active, the action becomes 1 and this remains for one program cycle, independent of whether or not the step remains active.

Note: For this identifier, an additional duration of data type TIME must be defined.

If the step becomes active (0->1-edge), the action becomes 1 and this remains for one program cycle, independent of whether or not the step remains active.

Note: Independent of their position in the action list field, actions with the identifier P1 are always processed first, see Action, p. 370 as well.

If the step becomes active (1->0-edge), the action becomes 1 and this remains for one program cycle.

Note: Independent of their position in the action list field, actions with the identifier P0 are always processed last, see Action, p. 370 as well.

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Meaning</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>Delayed</td>
<td>If the step is active, the internal timer is started and the action becomes 1 after the process of the time duration, which was defined manually for the action. If the step becomes inactive after that, the action becomes inactive as well. If the step becomes inactive before the internal time has elapsed then the action does not become active. Note: For this identifier, an additional duration of data type TIME must be defined.</td>
</tr>
<tr>
<td>P</td>
<td>Pulse</td>
<td>If the step becomes active, the action becomes 1 and this remains for one program cycle, independent of whether or not the step remains active.</td>
</tr>
<tr>
<td>DS</td>
<td>Delayed and saved</td>
<td>If the step becomes active, the internal timer is started and the action becomes active after the process of the manually defined time duration. The action first becomes inactive again when identifier R is used for a reset in another step. If the step becomes inactive before the internal time has elapsed then the action does not become active. Note: For this identifier, an additional duration of data type TIME must be defined.</td>
</tr>
<tr>
<td>P1</td>
<td>Pulse (rising edge)</td>
<td>If the step becomes active (0-&gt;1-edge), the action becomes 1 and this remains for one program cycle, independent of whether or not the step remains active. Note: Independent of their position in the action list field, actions with the identifier P1 are always processed first, see Action, p. 370 as well.</td>
</tr>
<tr>
<td>P0</td>
<td>Pulse (falling edge)</td>
<td>If the step becomes active (1-&gt;0-edge), the action becomes 1 and this remains for one program cycle. Note: Independent of their position in the action list field, actions with the identifier P0 are always processed last, see Action, p. 370 as well.</td>
</tr>
</tbody>
</table>
13.4 Transitions and Transition Sections

Introduction

Overview
This section describes the transition objects and transition sections of the SFC sequence language.

What's in this Section?
This section contains the following topics:

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transition</td>
<td>378</td>
</tr>
<tr>
<td>Transition Section</td>
<td>380</td>
</tr>
</tbody>
</table>
**Transition**

**Introduction**

A transition provides the condition through which the checks of one or more pre-transition steps pass on one or more consecutive steps along the corresponding link.

**Transition Condition**

Every transition is allocated with a transition condition of data type **BOOL**.

The following are authorized as transaction conditions:

- an address (input or output)
- a variable (input or output)
- a Literal or
- a Transition Section (See Transition Section, p. 380)

The type of transition condition determines the position of the name:

<table>
<thead>
<tr>
<th>Transition Condition</th>
<th>Position of the Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address</td>
<td>Trans.Variable</td>
</tr>
<tr>
<td>Variable</td>
<td></td>
</tr>
<tr>
<td>Literal</td>
<td>Trans.Section</td>
</tr>
<tr>
<td>Transition Section</td>
<td></td>
</tr>
</tbody>
</table>

**Transition Name**

If an address or a variable is used as a transition condition then the transition name is defined with that name (e.g., `%T10.4, Variable1`).

If a transition section is used as a transition condition then the section name is used as the transition name.

Transition names (maximum 32 characters) must be unique over the entire project, i.e. no other transition, variable or section (with the exception of the assigned transition section) etc., may exist with the same name. There are no case distinctions. The transition name must correspond with the standardized name conventions.
<table>
<thead>
<tr>
<th><strong>Enabling a Transition</strong></th>
<th>A transition is enabled if the steps immediately preceding it are active. Transitions whose immediately preceding steps are not active are not normally analyzed.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Note:</strong></td>
<td>If no transition condition is defined, the transition will never be active.</td>
</tr>
<tr>
<td><strong>Triggering a Transition</strong></td>
<td>A transition is triggered when the transition is enabled and the associated transition conditions are satisfied.</td>
</tr>
<tr>
<td></td>
<td>Triggering a transition leads to the disabling (resetting) of all immediately preceding steps that are linked to the transition, followed by the activation of all immediately following steps.</td>
</tr>
<tr>
<td><strong>Trigger Time for a Transition</strong></td>
<td>The transition trigger time (switching time) can theoretically be as short as possible, but can never be zero. The transition trigger time lasts at least the duration of a program cycle.</td>
</tr>
</tbody>
</table>
### Transition Section

**Introduction**

For every transition, a transition section can be created. This is a section containing the logic of the transition condition and it is automatically linked with the transition.

**Name of transition section**

The name of the transition section is always identical to the assigned transition, see *Transition Name, p. 378.*

**Programming Languages**

FBD, LD, IL and ST are possible as programming languages for transition sections.

Suggested Networks for Transition Section:

<table>
<thead>
<tr>
<th>Language</th>
<th>Suggested Network</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FBD</td>
<td><img src="image" alt="FBD Network" /></td>
<td>The suggested network contains an AND block with 2 inputs for which the output is linked with a variable having the name of the transition section. The suggested block can either be linked or it can be deleted if desired.</td>
</tr>
<tr>
<td>LD</td>
<td><img src="image" alt="LD Network" /></td>
<td>The suggested network contains a coil which is linked with a variable having the name of the transition section. The suggested coil can either be linked or it can be deleted if desired.</td>
</tr>
<tr>
<td>IL</td>
<td>-</td>
<td>The suggested network is empty. The content may only be created of Boolean logic. The assignment of the logic result on the output (the transition variable) is done automatically, i.e. the memory assignment ST is not allowed. Example: LD A AND B</td>
</tr>
</tbody>
</table>
### Properties of Transition Sections

Transition sections have the following properties:
- Transition sections only have one single output (transition variable), whose data type is `BOOL`. The name of these variables are identical to the names of the transition sections.
- The transition variable can only be used once in written form.
- The transition variable can be read in any position within the project.
- Only functions can be used, function blocks or procedures cannot.
- Only one coil may be used in LD.
- There is only one network, i.e. all functions used are linked with each other either directly or indirectly.
- Transition sections can only be used once.
- Transition sections belong to the SFC section in which they were defined. If the respective SFC section is deleted then all transition sections of this SFC section are also deleted automatically.
- Transition sections can be called exclusively from transitions.

<table>
<thead>
<tr>
<th>Language</th>
<th>Suggested Network</th>
<th>Description</th>
</tr>
</thead>
</table>
| ST       | -                 | The suggested network is empty. The content may only be created of Boolean logic in the form of a (nested) expression. The assignment of the logic result on the output (the transition variable) is done automatically, i.e. the instruction assignment `:=` is not allowed. The expression is not terminated by a semicolon (`;`).
Example:
A AND B
or
A AND (WORD_TO_BOOL (B)) |
13.5 Jump

Jump

General

Jumps are used to indicate directional links that are not represented in their full length.

Representation of a jump:

Properties of jumps

Jumps have the following properties:

- More than one jump may have the same target step.
- In accordance with IEC 61131-3, jumps into a parallel sequence (See Parallel Branch and Parallel Joint, p. 387) or out of a parallel sequence are not possible.
  If it should also be used again then it must be enabled explicitly.
- With jumps, there is a difference between a Sequence Jump (See Sequence Jump, p. 392) and a Sequence Loop (See Sequence Loop, p. 393).
- The jump target is indicated by the jump target symbol (>).

Jump name

Jumps do not actually have their own names. Instead, the name of the target step (jump target) is shown inside of the jump symbol.
13.6 Link

Introduction

Links connect steps and transitions, transitions and steps etc.

Properties of Links

Links have the following properties:

- Links between objects of the same type (step with step, transition with transition, etc.) are not possible
- Links are possible between:
  - unlinked object outputs and
  - unlinked or linked step inputs
    (i.e. multiple step inputs can be linked)
- Overlapping links and other SFC objects (step, transition, jump, etc.) is not possible
- Overlapping links and links is possible
- Crossing links with links is possible and is indicated by a "broken" link:

  ![Diagram](image)

- Links consist of vertical and horizontal segments
- Standard signal flow in a sequence string is from top to bottom. To create a loop however, links can be made from below to a step above. This applies to links from transitions, parallel branches or alternative joints to a step. In these cases, the direction of the link is indicated with an arrow symbol:

  ![Diagram](image)

- With links, there is a difference between a String Jump (See Sequence Jump, p. 392) and a String Loop (See Sequence Loop, p. 393)
13.7 Branches and Merges

Introduction

Overview
This section describes the branch and merge objects of the SFC sequence language.

What's in this Section?
This section contains the following topics:

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alternative Branches and Alternate Joints</td>
<td>385</td>
</tr>
<tr>
<td>Parallel Branch and Parallel Joint</td>
<td>387</td>
</tr>
</tbody>
</table>
Alternative Branches and Alternative Joints

Introduction
The alternative branch offers the possibility to program branches conditionally in the control flow of the SFC structure.

With alternative branches, as many transitions follow a step under the horizontal line as there are different processes.

All alternative branches are run together into a single branch again with alternative joints or Jumps (See Jump, p. 382) where they are processed further.

Example of an Alternative Sequence

```plaintext
Example of an Alternative Sequence
```

```
<table>
<thead>
<tr>
<th>S_5_10</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
</tr>
<tr>
<td>S_5_11</td>
</tr>
<tr>
<td>b</td>
</tr>
<tr>
<td>S_5_12</td>
</tr>
<tr>
<td>c</td>
</tr>
<tr>
<td>S_5_13</td>
</tr>
<tr>
<td>d</td>
</tr>
<tr>
<td>S_5_14</td>
</tr>
<tr>
<td>e</td>
</tr>
<tr>
<td>S_5_16</td>
</tr>
<tr>
<td>f</td>
</tr>
<tr>
<td>S_5_15</td>
</tr>
<tr>
<td>g</td>
</tr>
<tr>
<td>S_5_16</td>
</tr>
<tr>
<td>h</td>
</tr>
<tr>
<td>S_5_16</td>
</tr>
</tbody>
</table>

```
Properties of an Alternative Sequence

The properties of an alternative sequence mainly depend on whether the sequence control is operating in single token or multi-token mode.

See
- Properties of an Alternative Sequence in Single Token (See Alternative String, p. 391)
- Properties of an Alternative Sequence in Multi Token (See Alternative String, p. 402)
Parallel Branch and Parallel Joint

**Introduction**

With parallel branches, switching a single transition leads to a parallel activation of more than one (maximum 32) step (branches). Execution is from left to right. After this common activation, the individual branches are processed independently from one another.

All parallel branches are grouped using a parallel joint according to IEC 61131-1. The transition following a parallel joint is evaluated when all the immediately preceding steps of the parallel joint have been set.

Combining a parallel branch with an alternative joint is only possible in Multi-Token (See Parallel Strings, p. 405) operation.

**Example of a Parallel Sequence**

Properties of a Parallel Sequence

- Properties of a Parallel Sequence in Single Token (See Alternative String, p. 391)
- Properties of a Parallel Sequence in Multi-Token (See Alternative String, p. 402)
13.8 Text Objects

Text Object

Introduction

Text can be positioned in the form of text objects using SFC sequence language. The size of these text objects depends on the length of the text. This text object is at least the size of a cell and can be vertically and horizontally enlarged to other cells according to the size of the text. Text objects can overlap with other SFC objects.
13.9 Single-Token

Introduction

Overview

This section describes the "Single-Token" operating mode for sequence controls.

What's in this Section?

This section contains the following topics:

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution Sequence Single-Token</td>
<td>390</td>
</tr>
<tr>
<td>Alternative String</td>
<td>391</td>
</tr>
<tr>
<td>Sequence Jumps and Sequence Loops</td>
<td>392</td>
</tr>
<tr>
<td>Parallel Strings</td>
<td>395</td>
</tr>
<tr>
<td>Asymmetric Parallel String Selection</td>
<td>397</td>
</tr>
</tbody>
</table>
**Execution Sequence Single-Token**

<table>
<thead>
<tr>
<th>Description</th>
<th>The following rules apply for single token:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>● The output situation is defined by the initial step. The sequence string contains on initial step only.</td>
</tr>
<tr>
<td></td>
<td>● Only one step is ever active in the sequence string. The only exceptions are parallel branches in which one step is active per branch.</td>
</tr>
<tr>
<td></td>
<td>● The active signal status processes take place along the directional links, triggered by switching one or more transitions. The direction of the string process follows the directional links and runs from the under side of the predecessor step to the top side of the successive step.</td>
</tr>
<tr>
<td></td>
<td>● A transition is enabled if the steps immediately preceding it are active. Transitions whose immediately preceding steps are not active are not normally analyzed.</td>
</tr>
<tr>
<td></td>
<td>● A transition is triggered when the transition is enabled and the associated transition conditions are satisfied.</td>
</tr>
<tr>
<td></td>
<td>● Triggering a transition leads to the disabling (resetting) of all immediately preceding steps that are linked to the transition, followed by the activation of all immediately following steps.</td>
</tr>
<tr>
<td></td>
<td>● If more than one transition condition in a row of sequential steps has been satisfied then one step is processed per cycle.</td>
</tr>
<tr>
<td></td>
<td>● Steps cannot be activated or deactivated by other non-SFC sections.</td>
</tr>
<tr>
<td></td>
<td>● The use of macro steps is possible.</td>
</tr>
<tr>
<td></td>
<td>● Only one branch is ever active in alternative branches. The branch to be run is determined by the result of the transition conditions of the transitions that follow the alternative branch. If a transition condition is satisfied, the remaining transitions are no longer processed. The branch with the satisfied transition is activated. This gives rise to a left to right priority for branches. All alternative branches are combined at the end by an alternative joint or jumps.</td>
</tr>
<tr>
<td></td>
<td>● With parallel branches, switching a single transition leads to the activation of more than one step (branch). After this common activation, the individual branches are processed independent of one another. All parallel branches are combined at the end by a parallel joint. Jumps into a parallel branch or out of a parallel branch are not possible.</td>
</tr>
</tbody>
</table>
Alternative String

According to IEC 61131-3, only one switch (1-off-n-select) can be made from the transitions. The branch to be run is determined by the result of the transition conditions of the transitions that follow the alternative branch. Branch transitions are processed from left to right. If a transition condition is satisfied, the remaining transitions are no longer processed. The branch with the satisfied transition is activated. This results in a left to right priority for branches.

If none of the transitions are switched, the step that is currently set remains set.

Alternative Strings:

<table>
<thead>
<tr>
<th>S_5_10</th>
<th>S_5_11</th>
<th>S_5_12</th>
<th>S_5_13</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>If...</th>
<th>Then</th>
</tr>
</thead>
<tbody>
<tr>
<td>If S_5_10 is active and transition condition a is true (independent of b).</td>
<td>then a sequence is run from S_5_10 to S_5_11.</td>
</tr>
<tr>
<td>If S_5_10 is active and transition condition b is true and a is false.</td>
<td>then a sequence is run from S_5_10 to S_5_12.</td>
</tr>
</tbody>
</table>
Sequence Jumps and Sequence Loops

Sequence Jump  A sequence jump is a special type of alternative branch that can be used to skip several steps of a sequence.

A sequence jump can be made with jumps or with links.

Sequence jump:

<table>
<thead>
<tr>
<th>If...</th>
<th>Then</th>
</tr>
</thead>
<tbody>
<tr>
<td>If transition condition a is true,</td>
<td>then a sequence is run from S_5_10 to S_5_11, S_5_12 and S_5_13.</td>
</tr>
<tr>
<td>If transition condition b is true,</td>
<td>then a jump is made from S_5_10 directly to S_5_13.</td>
</tr>
<tr>
<td>If transition condition e is true,</td>
<td>then a sequence is run from S_5_10 to S_5_14 and S_5_13.</td>
</tr>
</tbody>
</table>
**Sequence Loop**

A sequence loop is a special type of alternative branch with which one or more branches lead back to a previous step.

A sequence loop can be made with jumps or with links.

**Sequence loop:**

If... Then

<table>
<thead>
<tr>
<th>Condition</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>a is true</td>
<td>A sequence runs from S_1_11 to S_1_12.</td>
</tr>
<tr>
<td>b is true</td>
<td>A sequence runs from S_1_12 to S_1_13.</td>
</tr>
<tr>
<td>b is false and c is true</td>
<td>A sequence runs from S_1_12 to S_1_14.</td>
</tr>
<tr>
<td>f is true</td>
<td>A jump is made from S_1_14 back to S_1_12.</td>
</tr>
</tbody>
</table>

The loop from S_1_12 by means of transition conditions c and f back to S_1_12 is repeated until transition condition b is true or c is false and d is true.

If transition conditions b and c are false and d is true, then a jump is made from S_1_12 directly back to S_1_11.

The loop from S_1_11 to S_1_12 and back to S_1_11 via transition conditions a and d is repeated until transition condition b or c is true.

Infinite sequence loops are not permitted within an alternative sequence.
Infinite sequence loops:

If transition condition $b$ is true, then a sequence runs from $S_{1\_1}$ to $S_{1\_3}$.
If transition condition $e$ is true, then a jump is made to $S_{1\_4}$.
If transition condition $f$ is true, then a jump is made to $S_{1\_3}$.
The loop from $S_{1\_3}$ via transition condition $e$, to $S_{1\_4}$ via transition condition $f$ and a jump back to $S_{1\_3}$ again, is now repeated infinitely.
Parallel Strings

With parallel branches, switching a single transition leads to a parallel activation of more than one (maximum 32) steps (branches). This applies with Single-Token as well as with Multi-Token.

Processing Parallel Strings:

If... Then

| If S_5_10 is active and transition condition a, which belongs to the common transition, is also true, | then a sequence runs from S_5_10 to S_5_11, S_5_12 and S_5_13. |
| If steps S_5_11, S_5_12 and S_5_13 are activated, | then the strings run independently of one another. |
| If S_5_14, S_5_15 and S_5_16 are active at the same time and transition condition e, which belongs to the common transition, is true, | then a sequence is run from S_5_14, S_5_15 and S_5_16 to S_5_17. |
If a single alternative branch is used in a parallel string, it leads to blocking the string with Single-Token.

Using an Alternative Branch in a Parallel String:

If transition condition \( a \) is true, then a sequence is run to \( S_7.1 \) and \( S_7.2 \).

If steps \( S_7.1 \) and \( S_7.2 \) are activated, then the strings run independently of one another.

If transition condition \( d \) is true, then a sequence runs to \( S_7.5 \).

If transition condition \( b \) is true and \( c \) is false, then a sequence runs to \( S_7.3 \).

Since \( S_7.3 \), \( S_7.4 \) and \( S_7.5 \) are linked with a parallel merge, no sequence can follow to \( S_7.6 \) because \( S_7.3 \) and \( S_7.4 \) can never be active at the same time.

(Either \( S_7.3 \) is activated with transition condition \( b \) or \( S_7.4 \) with transition condition \( c \), never both at the same time.)

Therefore \( S_7.3 \), \( S_7.4 \) and \( S_7.5 \) can never be active at the same time either. The string is blocked.

The same problem occurs if transition condition \( b \) is false and \( c \) is true when entering the alternative branch.
Asymmetric Parallel String Selection

Introduction

According to IEC 61131-3, a parallel branch must always be terminated with a parallel merge. The number of parallel branches must not coincide with the number of parallel merges however.

Greater Amount of Merges

String with 1 Parallel Branch and 2 Parallel Merges:

If... Then
If transition condition $a$ is true, then a sequence runs to $S_{19\_2}$, $S_{19\_3}$ and $S_{19\_4}$.
If steps $S_{19\_2}$, $S_{19\_3}$ and $S_{19\_4}$ are activated, then the strings run independently of one another.
If transition condition $b$ is true, then a sequence runs to $S_{19\_5}$.
If steps $S_{19\_2}$ and $S_{19\_5}$ are active and transition condition $c$ is true, then the parallel string is departed.
Greater Amount of Branches

String with 2 Parallel Branches and 1 Parallel Merge:

If... Then

If transition condition a is true, then a sequence runs to S_19_2 and S_19_3.
If steps S_19_2 and S_19_3 are activated, then the strings run independently of one another.
If transition condition b is true, then a sequence runs to S_19_4 and S_19_5.
If steps S_19_4 and S_19_5 are activated, then the strings run independently of one another.
If steps S_19_2, S_19_4 and S_19_5 are active and transition condition c is true, then the parallel string is departed.
Nested Parallel Strings:

If... Then
If transition condition \( a \) is true, then a sequence runs to \( S_{8.10} \) and \( S_{8.11} \).
If transition condition \( b \) is true, then a sequence runs to \( S_{8.12} \) and \( S_{8.13} \).
If transition condition \( c \) is true, then a sequence runs to \( S_{8.14} \), \( S_{8.15} \) and \( S_{8.16} \).
If steps \( S_{8.13} \) and \( S_{8.14} \) are active and transition condition \( d \) is true, then a sequence runs to \( S_{8.17} \).
If steps \( S_{8.12} \) and \( S_{8.17} \) are active and transition condition \( e \) is true, then a sequence runs to \( S_{8.18} \).
...
13.10 Multi-Token

Introduction

Overview

This section describes the "Multi-Token" operating mode for sequence controls.

What's in this Section?

This section contains the following topics:

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multi-Token Execution Sequence</td>
<td>401</td>
</tr>
<tr>
<td>Alternative String</td>
<td>402</td>
</tr>
<tr>
<td>Parallel Strings</td>
<td>405</td>
</tr>
<tr>
<td>Jump into a Parallel String</td>
<td>409</td>
</tr>
<tr>
<td>Jump out of a Parallel String</td>
<td>411</td>
</tr>
</tbody>
</table>
Multi-Token Execution Sequence

Description

The following rules apply for Multi-Token:

- The original situation is defined in a number of initial steps (0 to 100) which can be defined.
- A number of steps which can be freely defined can be active at the same time in a sequence string.
- The active signal status processes take place along the directional links, triggered by switching one or more transitions. The direction of the string process follows the directional links and runs from the under side of the predecessor step to the top side of the successive step.
- A transition is enabled if the steps immediately preceding it are active. Transitions whose immediately preceding steps are not active are not analyzed.
- A transition is triggered when the transition is enabled and the associated transition conditions are satisfied.
- Triggering a transition leads to the disabling (resetting) of all immediately preceding steps that are linked to the transition, followed by the activation of all immediately following steps.
- If more than one transition condition in a row of sequential steps has been satisfied then one step is processed per cycle.
- Steps and macro steps can be activated or deactivated by other non-SFC sections or by user operations.
- If an active step is activated and deactivated at the same time then the step remains active.
- The use of macro steps is possible. Whereas the macro step section can also contain initial steps.
- More than one branch can be active with alternative branches. The branches to be run are determined by the result of the transition conditions of the transitions that follow the alternative branch. Branch transitions are processed in parallel. The branches with satisfied transitions are activated. All alternative branches do not have to be combined at the end by an alternative joint or jumps.
- If jumps are to be made into a parallel branch or out of a parallel branch then this option can be enabled. All parallel branches do not have to be combined at the end by a parallel joint in this case.
- Subroutine calls be used in an action section.
- Multiple tokens can be created with:
  - Multiple initial steps
  - Alternative or parallel branches that are not terminated
  - Jumps in combination with alternative and parallel strings
  - Activation of steps using the SFC control block `SETSTEP` from a non-SFC section or with SFC control instructions
- Tokens can be ended with:
  - Simultaneous meeting of two or more tokens in a step
Deactivation of steps using the SFC control block `RESETSTEP` from a non-SFC section or with SFC control instructions

**Alternative String**

The user can define the behavior for the evaluation of transition conditions in alternative branches with Multi-Token.

The following are possible:

- Processing is from left to right with a stop after the first active transition (1-off-n-select). This corresponds with the behavior of alternative strings with Single-Token (See Alternative String, p. 391).
- Parallel processing of all transitions of the alternative branch (x-off-n-select)

**x-off-n-select**

With Multi-Token, more than one parallel switch can be made from the transitions (1-off-n-select). The branches to be run are determined by the result of the transition conditions of the transitions that follow the alternative branch. The transitions of the branches are all processed. All branches with satisfied transitions are activated.

If none of the transitions are switched, the step that is currently set remains set.

```
x-off-n-select: 

S_5_10 ——— a ——— b
          |         |
S_5_11 ——— c ——— d
          |         |
S_5_13
```
If alternative branches should only be switched exclusively in this mode of operation, then this must be defined explicitly with the transition logic.

Example:

```
S_5_10
```

```
S_5_11
```

```
S_5_12
```

If $S_5\_10$ is active and transition condition $a$ is true and $b$ is false, then a sequence is run from $S_5\_10$ to $S_5\_11$.

If $S_5\_10$ is active and transition condition $a$ is false and $b$ is true, then a sequence is run from $S_5\_10$ to $S_5\_12$.

If $S_5\_10$ is active and transition conditions $a$ and $b$ are true, then a sequence is run from $S_5\_10$ to $S_5\_11$ and $S_5\_12$.

A second token is created by the parallel activation of the two alternative branches. These two tokens are now running parallel to one another, i.e. $S_5\_11$ and $S_5\_12$ are active at the same time.

<table>
<thead>
<tr>
<th>Token 1 ($S_5_11$)</th>
<th>Token 2 ($S_5_12$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>If...</td>
<td>Then</td>
</tr>
<tr>
<td>If the transition condition $c$ is true, then a sequence is run from $S_5_11$ to $S_5_13$.</td>
<td>If transition condition $d$ is true, then a sequence is run from $S_5_12$ to $S_5_13$.</td>
</tr>
</tbody>
</table>

If $S_5\_13$ is still active (token 1) because of the activation of transition condition $c$, then token 2 is ended and the string will be further processed as Single-Token. If $S_5\_13$ is no longer active (token 1), then it is reactivated by token 2 and both tokens continue running parallel (Multi-Token).
SFC Sequence Language

Terminating an Alternative Branch with a Parallel Merge

If a parallel merge is used to terminate an alternative branch, it may block the string.

Terminating an Alternative Branch with a Parallel Merge:

If transition condition \( a \) is true and \( b \) is false, then a sequence runs to \( S_{6.1} \).

Since \( S_{6.1} \) and \( S_{6.2} \) are linked by a parallel merge, the branch cannot be departed because \( S_{6.1} \) and \( S_{6.2} \) can never be active at the same time. (Either \( S_{6.1} \) is activated with transition condition \( a \) or \( S_{6.2} \) with transition condition \( b \).)

Therefore \( S_{6.1} \) and \( S_{6.2} \) can never be active at the same time either. The string is blocked.

This block can be removed, for example, by a second timed token that runs via transition \( b \).
Parallel Strings

With parallel branches, switching a single transition leads to a parallel activation of more than one (maximum 32) steps (branches). This applies with Single-Token as well as with Multi-Token Processing Parallel Strings:

<table>
<thead>
<tr>
<th>If...</th>
<th>Then</th>
</tr>
</thead>
<tbody>
<tr>
<td>If $S_{5.10}$ is active and transition condition $a$, which belongs to the common transition, is also true,</td>
<td>then a sequence runs from $S_{5.10}$ to $S_{5.11}$, $S_{5.12}$ and $S_{5.13}$.</td>
</tr>
<tr>
<td>If steps $S_{5.11}$, $S_{5.12}$ and $S_{5.13}$ are activated,</td>
<td>then the strings run independently of one another.</td>
</tr>
<tr>
<td>If $S_{5.14}$, $S_{5.15}$ and $S_{5.16}$ are active at the same time and transition condition $e$, which belongs to the common transition, is true,</td>
<td>then a sequence is run from $S_{5.14}$, $S_{5.15}$ and $S_{5.16}$ to $S_{5.17}$.</td>
</tr>
</tbody>
</table>
Terminating a parallel branch can also be done with an alternative merge instead of a parallel merge with Multi-Token.

Terminating a Parallel String with an Alternative Branch (variation 1):

If... Then

If the transition condition \( a \) is true, then a sequence runs to \( S_{5.1} \) and \( S_{5.2} \).

If steps \( S_{5.1} \) and \( S_{5.2} \) are activated, then the strings run independently of one another.

If transition condition \( b \) is true and \( c \) is false, then a sequence runs to \( S_{5.3} \).

A second token is created by the sequence running on the alternative merge out of the parallel string. The two tokens are running parallel to one another, i.e. \( S_{5.2} \) and \( S_{5.3} \) are active at the same time.

**Token 1 (S.5.3) | Token 2 (S.5.2)**

<table>
<thead>
<tr>
<th>If...</th>
<th>Then</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step ( S_{5.3} ) is active.</td>
<td>Step ( S_{5.2} ) is active.</td>
</tr>
</tbody>
</table>

If the transition condition \( c \) is true, then a sequence runs to \( S_{5.3} \).

If \( S_{5.3} \) is still active (token 1) then token 2 is ended and the string is further processed as Single-Token. If \( S_{5.3} \) is no longer active (token 1), then it is reactivated by token 2 and both tokens continue running parallel (Multi-Token).
Terminating a Parallel String with an Alternative Branch (variation 2):

If... Then
If the transition condition \( a \) is true, then a sequence runs to \( S_{5.1} \) and \( S_{5.2} \).

A second token is created by the sequence running on the alternative merge out of the parallel string. These two tokens are now running parallel to one another, i.e. \( S_{5.1} \) and \( S_{5.2} \) are active at the same time.

<table>
<thead>
<tr>
<th>Token 1 (S_{5.2})</th>
<th>Token 2 (S_{5.1})</th>
</tr>
</thead>
<tbody>
<tr>
<td>If...</td>
<td>Then</td>
</tr>
<tr>
<td>Step ( S_{5.2} ) is active.</td>
<td>Step ( S_{5.1} ) is active.</td>
</tr>
<tr>
<td></td>
<td>If transition condition ( b ) is true, then a sequence runs to ( S_{5.2} ).</td>
</tr>
</tbody>
</table>

If \( S_{5.2} \) is still active (token 1) then token 2 is ended and the string is further processed as Single-Token.
If \( S_{5.2} \) is no longer active (token 1), then it is reactivated by token 2 and both tokens continue running parallel (Multi-Token).
Using an Alternative Branch in a Parallel String

If one single alternative branch is used in a parallel string, it may block the string.

Using an Alternative Branch in a Parallel String:

If transition condition a is true, then a sequence is run to S_7_1 and S_7_2.
If steps S_7_1 and S_7_2 are activated, then the strings run independently of one another.
If transition condition d is true, then a sequence runs to S_7_5.
If transition condition b is true, then a sequence runs to S_7_3.

Since S_7_3, S_7_4 and S_7_5 are linked by a parallel merge, the parallel string cannot be departed because S_7_3 and S_7_4 can never be active at the same time.
(Either S_7_3 is activated with transition condition b or S_7_4 with transition condition c.)
Therefore S_7_3, S_7_4 and S_7_5 cannot be active at the same time either. The string is blocked.
This block can be removed for example, by a second timed token that runs via transition c.
Jump into a Parallel String

Description
The ability to jump into a parallel string or out of a parallel string can be enabled optionally with multi-token.

A jump into a parallel string does not activate all branches. Since the transition after the parallel joint is only evaluated if all steps which directly precede the transition are set, the parallel string can no longer be departed, the string is blocking.
If the transition condition \( a \) is true, then a sequence runs to \( S_{1\_1} \) and \( S_{1\_2} \).

If steps \( S_{1\_1} \) and \( S_{1\_2} \) are activated, then the strings run independently of one another.

If \( S_{1\_2} \) is active and transition condition \( b \), is true, then a sequence runs from \( S_{1\_2} \) to \( S_{1\_3} \).

If \( S_{1\_1} \) and \( S_{1\_3} \) are active and transition condition \( c \), which belongs to the common transition, is true, then a sequence runs from \( S_{1\_1} \) and \( S_{1\_3} \) to a jump to \( S_{1\_1} \).

If \( S_{1\_1} \) is activated by the jump, then only the branch from \( S_{1\_1} \) is active. The branch from \( S_{1\_2} \) is not active.

Since \( S_{1\_1} \) and \( S_{1\_3} \) are not active at the same time, the string cannot continue. The string is blocked. This block can removed by e.g. a second timed token that is set to reactivate step \( S_{1\_2} \).

<table>
<thead>
<tr>
<th>If...</th>
<th>Then</th>
</tr>
</thead>
<tbody>
<tr>
<td>If the transition condition ( a ) is true,</td>
<td>then a sequence runs to ( S_{1_1} ) and ( S_{1_2} ).</td>
</tr>
<tr>
<td>If steps ( S_{1_1} ) and ( S_{1_2} ) are activated,</td>
<td>then the strings run independently of one another.</td>
</tr>
<tr>
<td>If ( S_{1_2} ) is active and transition condition ( b ), is true,</td>
<td>then a sequence runs from ( S_{1_2} ) to ( S_{1_3} ).</td>
</tr>
<tr>
<td>If ( S_{1_1} ) and ( S_{1_3} ) are active and transition condition ( c ), which belongs to the common transition, is true,</td>
<td>then a sequence runs from ( S_{1_1} ) and ( S_{1_3} ) to a jump to ( S_{1_1} ).</td>
</tr>
<tr>
<td>If ( S_{1_1} ) is activated by the jump,</td>
<td>then only the branch from ( S_{1_1} ) is active. The branch from ( S_{1_2} ) is not active.</td>
</tr>
</tbody>
</table>

Since \( S_{1\_1} \) and \( S_{1\_3} \) are not active at the same time, the string cannot continue. The string is blocked. This block can removed by e.g. a second timed token that is set to reactivate step \( S_{1\_2} \).
Jump out of a Parallel String

Introduction

The ability to jump into a parallel string or out of a parallel string can be enabled optionally with multi-token.

Extra tokens are generated in all cases.

Jump out of a Parallel String:

```
  a
  S_2_1 ---- S_2_2 ---- S_2_3
     |            |
     c          d
     S_2_3 ---- S_2_4
  e
  S_2_5

```
If... Then
If the transition condition $a$ is true and $b$ is false, then a sequence runs to $S_{2 \_1}$ and $S_{2 \_2}$.
If steps $S_{2 \_1}$ and $S_{2 \_2}$ are activated, then the strings run independently of one another.
If the transition condition $c$ is true, then a jump is made to $S_{2 \_3}$.

A second token is created by the jump out of the parallel string. Both tokens are running parallel to one another, i.e. $S_{2 \_1}$ and $S_{2 \_3}$ are active at the same time.

Token 1 ($S_{2 \_1}$) Then
If the transition condition $e$ is true, then a sequence runs to $S_{2 \_5}$.

Token 2 ($S_{2 \_3}$) Then
If transition condition $d$ is true, then a sequence runs to $S_{2 \_4}$.
If transition condition $f$ is true, then a sequence runs to $S_{2 \_5}$.

If $S_{2 \_5}$ is still active (token 1) because of the activation of transition condition $e$, then token 2 is ended and the string will be further processed as Single-Token.
If $S_{2 \_5}$ is no longer active (token 1), then it is reactivated by token 2 and both tokens continue running parallel (Multi-Token).

Jump Between Two Branches of a Parallel String:

---

Jump Between Two Branches of a Parallel String:
If the transition condition $a$ is true, then a sequence runs to $S_{4 \_1}$ and $S_{4 \_2}$.

If steps $S_{4 \_1}$ and $S_{4 \_2}$ are activated, then the strings run independently of one another.

If transition condition $b$ is true, then a sequence runs to $S_{4 \_3}$.

If the transition condition $c$ is true, then a jump is made to $S_{4 \_1}$.

A second token is created by the jump out of a branch string. Both tokens are running parallel to one another, i.e. $S_{4 \_3}$ and $S_{4 \_1}$ are active at the same time.

<table>
<thead>
<tr>
<th>Token 1 ($S_{4 _3}$)</th>
<th>Token 2 ($S_{4 _1}$)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>If...</strong></td>
<td><strong>Then</strong></td>
</tr>
</tbody>
</table>
| Step $S_{4 \_3}$ is processed | Step $S_{4 \_1}$ is processed | If transition condition $b$ is true, | then a sequence runs to $S_{4 \_3}$.

If step $S_{4 \_3}$ is still active (token 1) during the activation by token 2 then token 2 is ended and the string will continue to be processed as Single-Token.

If step $S_{4 \_3}$ is no longer active (token 1) because of the activation by token 2, then it is reactivated by token 2 and both tokens continue running parallel (Multi-Token).

In both cases, true transition condition $d$ causes the parallel string to be left.
Leaving a Parallel String with an Alternative Branch:

```
<table>
<thead>
<tr>
<th>a</th>
<th>S_3.1</th>
<th>S_3.2</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>S_3.4</td>
<td>S_3.5</td>
</tr>
<tr>
<td>c</td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td></td>
<td></td>
</tr>
<tr>
<td>e</td>
<td>S_3.6</td>
<td></td>
</tr>
</tbody>
</table>
```
If... Then
If the transition condition \( a \) is true, then a sequence runs to \( S_{3\_1} \) and \( S_{3\_2} \).
If steps \( S_{3\_1} \) and \( S_{3\_2} \) are activated, then the strings run independently of one another.
If transition condition \( b \) is false and \( c \) is true, then a sequence runs to \( S_{3\_5} \).

A second token is created by the sequence running on the alternative branch out of the parallel string. Both tokens are running parallel to one another, i.e. \( S_{3\_1} \) and \( S_{3\_5} \) are active at the same time.

<table>
<thead>
<tr>
<th>Token 1 (( S_{3_1} ))</th>
<th>Token 2 (( S_{3_5} ))</th>
</tr>
</thead>
<tbody>
<tr>
<td>If...</td>
<td>Then</td>
</tr>
<tr>
<td>Since ( S_{3_4} ) cannot become active, ( S_{3_1} ) remains (token 1) active.</td>
<td>If transition condition ( d ) is true, then a sequence runs to ( S_{3_6} ).</td>
</tr>
<tr>
<td>If transition condition ( a ) is true then a sequence runs to ( S_{3_1} ) and ( S_{3_2} ). This ends token 2 and the string is again processed as Single-Token.</td>
<td></td>
</tr>
<tr>
<td>If the transition condition ( a ) is true, then a sequence runs to ( S_{3_1} ) and ( S_{3_2} ).</td>
<td></td>
</tr>
<tr>
<td>If transition condition ( b ) is true and ( c ) is false, then a sequence runs to ( S_{3_4} ).</td>
<td></td>
</tr>
<tr>
<td>Since ( S_{3_4} ) cannot become active, ( S_{3_1} ) remains (token 1) active until a sequence appears on ( S_{3_2} ) (token 2) and the transition is ( b ).</td>
<td></td>
</tr>
<tr>
<td>If ( S_{4_4} ) is no longer active (token 1), then it is reactivated by token 2 and both tokens continue running parallel (Multi-Token).</td>
<td></td>
</tr>
<tr>
<td>(Merging the two tokens can also be done in ( S_{4_3} ).)</td>
<td></td>
</tr>
</tbody>
</table>
Introduction

Overview
This chapter describes the programming language instruction list IL which conforms to IEC 61131.

What's in this Chapter?
This chapter contains the following sections:

<table>
<thead>
<tr>
<th>Section</th>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>14.1</td>
<td>General information about the IL instruction list</td>
<td>418</td>
</tr>
<tr>
<td>14.2</td>
<td>Calling Elementary Functions, Elementary Function Blocks, Derived Function Blocks and Procedures</td>
<td>441</td>
</tr>
</tbody>
</table>
14.1 General information about the IL instruction list

Introduction

Overview

This section contains a general overview of the IL instruction list.

What's in this Section?

This section contains the following topics:

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>General information about the IL instruction list</td>
<td>419</td>
</tr>
<tr>
<td>Operands</td>
<td>422</td>
</tr>
<tr>
<td>Modifier</td>
<td>424</td>
</tr>
<tr>
<td>Operators</td>
<td>426</td>
</tr>
<tr>
<td>Subroutine Call</td>
<td>437</td>
</tr>
<tr>
<td>Tags and Jumps</td>
<td>438</td>
</tr>
<tr>
<td>Comment</td>
<td>440</td>
</tr>
</tbody>
</table>
General information about the IL instruction list

Introduction
Using the Instruction list programming language (IL), you can call function blocks and functions conditionally or unconditionally, perform assignments and make jumps conditionally or unconditionally within a section.

Instructions
An instruction list is composed of a series of instructions.

Each instruction begins on a new line and consists of:
- an Operator (See Operands, p. 422),
- if necessary with a Modifier (See Modifier, p. 424) and
- if necessary one or more Operands (See Operands, p. 422)

Should several operands be used, they are separated by commas. It is possible for a Label (See Tags and Jumps, p. 438) to be in front of the instruction. This label is followed by a colon. A Comment (See Comment, p. 440) can follow the instruction.

Example:

<table>
<thead>
<tr>
<th>Label</th>
<th>Operators</th>
<th>Operands</th>
<th>Modifier</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>START</td>
<td>LD A</td>
<td>(* Key 1 *)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ANDN B</td>
<td>(* and not key 2 *)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ST C</td>
<td>(* Ventilator on *)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Structure of the programming language

IL is a so-called accumulator orientated language, i.e. each instruction uses or alters the current content of the accumulator (a form of internal cache). IEC 61131 refers to this accumulator as the "result".

For this reason, an instruction list should always begin with the LD operand ("Load in accumulator command").

Example of an addition:

<table>
<thead>
<tr>
<th>Command</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD 10</td>
<td>Load the value 10 into the accumulator.</td>
</tr>
<tr>
<td>ADD 25</td>
<td>&quot;25&quot; is added to the contents of the accumulator.</td>
</tr>
<tr>
<td>ST A</td>
<td>The result is stored in the variable A. The content of the variable A and the accumulator is now 35. Any further instruction will work with accumulator contents &quot;35&quot; if it does not begin with LD.</td>
</tr>
</tbody>
</table>

Compare operations likewise always refer to the accumulator. The Boolean result of the comparison is stored in the accumulator and therefore becomes the current accumulator content.

Example of a comparison:

<table>
<thead>
<tr>
<th>Command</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD B</td>
<td>The value B is loaded into the accumulator.</td>
</tr>
<tr>
<td>GT 10</td>
<td>10 is compared with the contents of the accumulator.</td>
</tr>
<tr>
<td>ST A</td>
<td>The result of the comparison is stored in the variable A. If B is less than or equal to 10, the value of both variable A and the accumulator content is 0 (FALSE). If B is greater than 10, the value of both variable A and the accumulator content is 1 (TRUE).</td>
</tr>
</tbody>
</table>

Section Size

The length of an instruction line is limited to 300 characters.

The length of an IL section is not limited within the programming environment. The length of an IL section is only limited by the size of the PLC memory.

Syntax

Identifiers and Keywords are not case sensitive.

Spaces and tabs have no influence on the syntax and can be used as and when required.

**Exception:** Not allowed - spaces and tabs
- keywords,
- literals,
- values,
- identifiers,
• variables and
• limiter combinations [e.g. (* for comments)].

**Execution sequence**

Instructions are executed line by line, from top to bottom. This sequence can be altered with the use of parentheses.

If, for example, A, B, C and D have the values 1, 2, 3 and 4, and are calculated as follows:

```
LD A
ADD B
SUB C
MUL C
ST E
```

the result in E will be 0.

In the case of the following calculation:

```
LD A
ADD B
SUB (LD C MUL D)
ST E
```

the result in E will be -9.

**Error behavior**

The following conditions are handled as an error when executing an expression:

• Attempting to divide by 0.
• Operands do not contain the correct data type for the operation.
• The result of a numerical operation exceeds the value range of its data type.

**IEC Conformity**

For a description of IEC conformity for the IL programming language, see IEC Conformity (See *IEC Compliance, p. 601*).
Operands

Introduction
Operators are used for operands.

An operand can be:
- an address
- a literal
- a variable
- a multi-element variable
- an element of a multi-element variable
- an EFB/DFB output or
- an EFB/DFB call.

Data types

The operand and the current accumulator content must be of the same type. Should operands of various types be processed, a type conversion must be performed beforehand.

In the example the integer variable \( i_1 \) is converted into a real variable before being added to the real variable \( r_4 \).

LD  \( i_1 \)
INT_TO_REAL
ADD r4
ST  r3

As an exception to this rule, variables with data type \( \text{TIME} \) can be multiplied or divided by variables with data type \( \text{INT}, \text{DINT}, \text{UINT} \) or \( \text{UDINT} \).

Permitted operations:
- LD  timeVar1
  DIV  dintVar1
  ST  timeVar2
- LD  timeVar1
  MUL  intVar1
  ST  timeVar2
- LD  timeVar1
  MUL  10
  ST  timeVar2

This function is listed by IEC 61131-3 as "undesired" service.
### Direct use of addresses

Addresses can be used directly (without a previous declaration). In this case the data type is assigned to the address directly. The assignment is made using the "Large prefix".

The different large prefixes are given in the following table:

<table>
<thead>
<tr>
<th>Large prefix / Symbol</th>
<th>Example</th>
<th>Data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>no prefix</td>
<td>%I10,%CH203.MOD,%CH203.MOD.ERR</td>
<td>BOOL</td>
</tr>
<tr>
<td>X</td>
<td>%M20</td>
<td>BOOL</td>
</tr>
<tr>
<td>B</td>
<td>%QB102.3</td>
<td>BYTE</td>
</tr>
<tr>
<td>W</td>
<td>%MW10</td>
<td>INT</td>
</tr>
<tr>
<td>D</td>
<td>%QD100</td>
<td>DINT</td>
</tr>
<tr>
<td>F</td>
<td>%MF100</td>
<td>REAL</td>
</tr>
</tbody>
</table>

### Using other data types

Should other data types be assigned as the default data types of an address, this must be done through an explicit declaration. This variable declaration takes place comfortably using the variable editor. The data type of an address can not be declared directly in an ST section (e.g. declaration AT %MW1: UINT; not permitted).

The following variables are declared in the variable editor:

```
UnlocV1: ARRAY [1..10] OF INT;
LocV1:  ARRAY [1..10] OF INT AT %MW100;
LocV2:  TIME AT %MW100;
```

The following calls then have the correct syntax:

```
%MW200 := 5;
LD LocV1[%MW200]
ST UnlocV1[2]
```

### Accessing field variables

When accessing field variables (ARRAY), only literals and variables of INT, DINT, UINT and UDINT types are permitted in the index entry.

The index of an ARRAY element can be negative if the lower threshold of the range is negative.

Example: Saving a field variable

```
LD var1[1]
ST var2.otto[4]
```
Modifiers influence the execution of the operators (see Operators, p. 426).

<table>
<thead>
<tr>
<th>Modifier</th>
<th>Use of Operators of data type</th>
<th>Description</th>
</tr>
</thead>
</table>
| N        | BOOL, BYTE, WORD, DWORD       | The modifier \(N\) is used to invert the value of the operands bit by bit. **Example:** In the example \(C\) is 1, if \(A\) is 1 and \(B\) is 0. 
\[LD\ A\]
\[ANDN\ B\]
\[ST\ C\] |
| C        | BOOL                         | The modifier \(C\) is used to carry out the associated instruction, should the value of the accumulator be 1 (TRUE). **Example:** In the example, the jump after \(START\) is only performed when \(A\) is 1 (TRUE) and \(B\) is 1 (TRUE). 
\[LD\ A\]
\[AND\ B\]
\[JMP\ C\ \(START\)\] |
| CN       | BOOL                         | If the modifiers \(C\) and \(N\) are combined, the associated instruction is only performed if the value of the accumulator be a Boolean 0 (FALSE). **Example:** In the example, the jump after \(START\) is only performed when \(A\) is 0 (FALSE) and \(B\) is 0 (FALSE). 
\[LD\ A\]
\[AND\ B\]
\[JMP\ CN\ \(START\)\] |
### Modifier | Use of Operators of data type | Description
--- | --- | ---
( | all | The left bracket modifier is used to move back the evaluation of the operand, until the right bracket operator appears. The number of right parenthesis operations must be equal to the number of left bracket modifiers. Brackets can be nested. **Example:** In the example E is 1, if C and/or D is 1 and A and B are 1.
LD A
AND B
AND ( C OR D )
ST E
The example can also be programmed in the following manner:
LD A
AND B
AND ( LD C OR D )
ST E
Operators

Introduction

An operator is a symbol for:

- an arithmetic operation to be executed,
- a logical operation to be executed or
- calling an elementary function block - DFBs or subroutines.

Operators are generic, i.e. they adapt automatically to the data type of the operands.
### Load and save operators

IL programming language load and save operators:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Modifier</th>
<th>Meaning</th>
<th>Operands</th>
<th>Description</th>
</tr>
</thead>
</table>
| **LD**   | N        | Loads the operands value into the accumulator | Literal, variable, direct address of any data type | The value of an operand is loaded into the accumulator using **LD**. The size of the accumulator adapts automatically to the data type of the operand. This also applies to the derived data types. **Example:** In this example the value of A is loaded into the accumulator, the value of B then added and the result saved in E.  
**LD** A  
**ADD** B  
**ST** E |
| **ST**   | N        | Saves the accumulator value in the operand | Variable, direct address of any data type | The current value of the accumulator is stored in the operand using **ST**. The data type of the operand must be the same as the "data type" of the accumulator. **Example:** In this example the value of A is loaded into the accumulator, the value of B then added and the result saved in E.  
**LD** A  
**ADD** B  
**ST** E  
The "old" result is used in subsequent calculations, depending on whether or not an **LD** follows an **ST**. **Example:** In this example the value of A is loaded into the accumulator, the value of B then added and the result saved in E. The value of B is then subtracted from the value of E (current accumulator content) and the result saved in C.  
**LD** A  
**ADD** B  
**ST** E  
**SUB** 3  
**ST** C |
Set and reset operators of the IL programming language:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Modifier</th>
<th>Meaning</th>
<th>Operands</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>-</td>
<td>Sets the operand to 1, when the accumulator content is 1</td>
<td>Variable, direct address of BOOL data type</td>
<td>S sets the operand to &quot;1&quot; when the current content of the accumulator is a Boolean 1. Example: In this example the value of A is loaded to the accumulator. If the content of the accumulator (value of A) is 1, then OUT is set to 1. LD A S OUT Usually this operator is used together with the reset operator R as a pair. Example: This example shows a RS flip-flop (reset dominant) that is controlled through the two Boolean variables A and C. LD A S OUT LD C R OUT</td>
</tr>
<tr>
<td>R</td>
<td>-</td>
<td>Sets the operand to 0 when the accumulator content is 1</td>
<td>Variable, direct address of BOOL data type</td>
<td>R sets the operand to &quot;0&quot; when the current content of the accumulator is a Boolean 1. Example: In this example the value of A is loaded to the accumulator. If the content of the accumulator (value of A) is 1, then OUT is set to 0. LD A R OUT Usually this operator is used together with the set operator S as a pair. Example: This example shows a SR flip-flop (set dominant) that is controlled through the two Boolean variables A and C. LD A R OUT LD C S OUT</td>
</tr>
</tbody>
</table>
### Logical Operators

IL programming language logic operators:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Modifier</th>
<th>Meaning</th>
<th>Operands</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND</td>
<td>N, N (, )</td>
<td>Logical AND</td>
<td>Literal, variable,</td>
<td>The AND operator makes a logical AND link between the accumulator content and the operand.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>direct address of</td>
<td>BOOL, BYTE, WORD or DWORD data</td>
<td>In the case of BYTE, WORD and DWORD data types, the link is made bit by bit.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>types</td>
<td></td>
<td>Example: In the example D is 1 if A, B and C are 1.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>LD A</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>AND B</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>AND C</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ST D</td>
</tr>
<tr>
<td>OR</td>
<td>N, N (, )</td>
<td>Logical OR</td>
<td>Literal, variable,</td>
<td>The OR operator makes a logical OR link between the accumulator content and the operand.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>direct address of</td>
<td>BOOL, BYTE, WORD or DWORD data</td>
<td>In the case of BYTE, WORD and DWORD data types, the link is made bit by bit.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>types</td>
<td></td>
<td>Example: In the example D is 1 if A or B are 1 and C is 1.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>LD A</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OR B</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OR C</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ST D</td>
</tr>
<tr>
<td>XOR</td>
<td>N, N (, )</td>
<td>Logical exclusive OR</td>
<td>Literal, variable,</td>
<td>The XOR operator makes a logical exclusive OR link between the accumulator content and the operand.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>direct address of</td>
<td>BOOL, BYTE, WORD or DWORD data</td>
<td>If more than two operands are linked, the result with an uneven number of 1-states is 1, and is 0 with an even number of 1-states.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>types</td>
<td></td>
<td>In the case of BYTE, WORD and DWORD data types, the link is made bit by bit.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Example: In the example D is 1 if A or B is 1. If A and B have the same status (both 0 or 1), D is 0.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>LD A</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>XOR B</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ST D</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Example: In the example F is 1 if 1 or 3 operands are 1. F is 0 if 0, 2 or 4 operands are 1.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>LD A</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>XOR B</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>XOR C</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>XOR D</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>XOR E</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ST F</td>
</tr>
</tbody>
</table>
**NOT**  
Logical negation (complement)  
Accumulator contents of data types **BOOL**, **BYTE**, **WORD** or **DWORD**  
The accumulator content is inverted bit by bit with **NOT**.  
**Example:** In the example \( B \) is 1 if \( A \) is 0 and \( B \) is 0 if \( A \) is 1.  
LD A  
**NOT**  
ST B
### IL programming language Arithmetic operators:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Modifier</th>
<th>Meaning</th>
<th>Operands</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td></td>
<td>Addition</td>
<td>Literal, variable,</td>
<td>With <code>ADD</code> the value of the operand is added to the value of the accumulator contents.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Direct address of</td>
<td>direct address of</td>
<td>Example: The example corresponds to the formula $D = A + B + C$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>data types INT,</td>
<td>DINT, UINT, UDINT,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>REAL or TIME</td>
<td>LD A</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td><code>ADD</code> B</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td><code>ADD</code> C</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td><code>ST</code> D</td>
</tr>
<tr>
<td>SUB</td>
<td></td>
<td>Subtraction</td>
<td>Literal, variable,</td>
<td>With <code>SUB</code> the value of the operand is subtracted from the accumulator content.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>direct address of</td>
<td>Example: The example corresponds to the formula $D = A - B - C$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>direct address of</td>
<td>LD A</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>data types INT,</td>
<td><code>SUB</code> B</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>DINT, UINT, UDINT,</td>
<td><code>SUB</code> C</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>REAL or TIME</td>
<td><code>ST</code> D</td>
</tr>
<tr>
<td>MUL</td>
<td></td>
<td>Multiplication</td>
<td>Literal, variable,</td>
<td>The <code>MUL</code> operator multiplies the content of the accumulator by the value of the operand.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>direct address of</td>
<td>Example: The example corresponds to the formula $D = A \times B \times C$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>data type INT,</td>
<td>LD A</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>DINT, UINT, UDINT,</td>
<td><code>MUL</code> B</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>or REAL</td>
<td><code>MUL</code> C</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td><code>ST</code> D</td>
</tr>
<tr>
<td>DIV</td>
<td></td>
<td>Division</td>
<td>Literal, variable,</td>
<td>The <code>DIV</code> operator divides the contents of the accumulator by the value of the operand.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>direct address of</td>
<td>Example: The example corresponds to the formula $D = A / B / C$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>data types INT,</td>
<td>LD A</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>DINT, UINT, UDINT,</td>
<td><code>DIV</code> B</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>or REAL</td>
<td><code>DIV</code> C</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td><code>ST</code> D</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Note: The <code>MULTIME</code> function in the obsolete library is available for multiplications involving the data type <code>Time</code>.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Note: The <code>DIVTIME</code> function in the obsolete library is available for divisions involving the data type <code>Time</code>.</td>
</tr>
</tbody>
</table>
### MOD (Modulo Division)

The MOD operator divides the value of the first operand by the value of the second and returns the remainder (Modulo) as the result.

**Example:** In this example
- C is 1 if A is 7 and B is 2
- C is 1 if A is 7 and B is -2
- C is -1 if A is -7 and B is 2
- C is -1 if A is -7 and B is -2

<table>
<thead>
<tr>
<th>Operator</th>
<th>Modifier</th>
<th>Meaning</th>
<th>Operands</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOD</td>
<td>(</td>
<td>Literal, variable, direct address of INT, DINT, UINT or UDINT data types</td>
<td></td>
<td>The MOD operator divides the value of the first operand by the value of the second and returns the remainder (Modulo) as the result.</td>
</tr>
</tbody>
</table>

LD A
MOD B
ST C
### IL programming language comparison operators:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Modifier</th>
<th>Meaning</th>
<th>Operands</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>GT</strong></td>
<td>(</td>
<td>Comparison: &gt;</td>
<td>Literal, variable, direct address of data type BOOL, BYTE, WORD, DWORD, STRING, INT, DINT, UINT, UDINT, REAL, TIME, DATE, DT or TOD</td>
<td>The <code>GT</code> operator compares the contents of the accumulator with the contents of the operand. If the contents of the accumulator are greater than the contents of the operands, the result is a Boolean 1. If the contents of the accumulator are less than/equal to contents of the operands, the result is a Boolean 0. <strong>Example:</strong> In the example the value of D is 1 if A is greater than 10, otherwise the value of D is 0.</td>
</tr>
<tr>
<td><strong>GE</strong></td>
<td>(</td>
<td>Comparison: &gt;=</td>
<td>Literal, variable, direct address of data type BOOL, BYTE, WORD, DWORD, STRING, INT, DINT, UINT, UDINT, REAL, TIME, DATE, DT or TOD</td>
<td>The <code>GE</code> operator compares the contents of the accumulator with the contents of the operand. If the contents of the accumulator are greater than/equal to the contents of the operands, the result is a Boolean 1. If the contents of the accumulator are less than the contents of the operands, the result is a Boolean 0. <strong>Example:</strong> In the example the value of D is 1 if A is greater than or equal to 10, otherwise the value of D is 0.</td>
</tr>
<tr>
<td><strong>EQ</strong></td>
<td>(</td>
<td>Comparison: =</td>
<td>Literal, variable, direct address of data type BOOL, BYTE, WORD, DWORD, STRING, INT, DINT, UINT, UDINT, REAL, TIME, DATE, DT or TOD</td>
<td>The <code>EQ</code> operator compares the contents of the accumulator with the contents of the operand. If the contents of the accumulator is equal to the contents of the operands, the result is a Boolean 1. If the contents of the accumulator are not equal to the contents of the operands, the result is a Boolean 0. <strong>Example:</strong> In the example the value of D is 1 if A is equal to 10, otherwise the value of D is 0.</td>
</tr>
</tbody>
</table>
### Instruction list IL

<table>
<thead>
<tr>
<th>Operator</th>
<th>Modifier</th>
<th>Meaning</th>
<th>Operands</th>
<th>Description</th>
</tr>
</thead>
</table>
| NE       | (        | Comparison: <>           | Literal, variable, direct address of data type BOOL, BYTE, WORD, DWORD, STRING, INT, DINT, UINT, UDINT, REAL, TIME, DATE, DT or TOD | The **NE** operator compares the contents of the accumulator with the contents of the operand. If the contents of the accumulator are not equal to the contents of the operands, the result is a Boolean 1. If the contents of the accumulator are equal to the contents of the operands, the result is a Boolean 0. **Example:** In the example the value of $D$ is 1 if $A$ is not equal to 10, otherwise the value of $D$ is 0.  
LD A  
NE 10  
ST D |
| LE       | (        | Comparison: <=          | Literal, variable, direct address of data type BOOL, BYTE, WORD, DWORD, STRING, INT, DINT, UINT, UDINT, REAL, TIME, DATE, DT or TOD | The **LE** operator compares the contents of the accumulator with the contents of the operand. If the contents of the accumulator are less than/equal to the contents of the operands, the result is a Boolean 1. If the contents of the accumulator are greater than the contents of the operands, the result is a Boolean 0. **Example:** In the example the value of $D$ is 1 if $A$ is smaller than or equal to 10, otherwise the value of $D$ is 0.  
LD A  
LE 10  
ST D |
| LT       | (        | Comparison: <           | Literal, variable, direct address of data type BOOL, BYTE, WORD, DWORD, STRING, INT, DINT, UINT, UDINT, REAL, TIME, DATE, DT or TOD | The **LT** operator compares the contents of the accumulator with the contents of the operand. If the contents of the accumulator is less than the contents of the operands, the result is a Boolean 1. If the contents of the accumulator is greater than/equal to the contents of the operands, the result is a Boolean 0. **Example:** In the example the value of $D$ is 1 if $A$ is smaller than 10, otherwise the value of $D$ is 0.  
LD A  
LT 10  
ST D |
## Call Operators

IL programming language call operators:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Modifier</th>
<th>Meaning</th>
<th>Operands</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAL</td>
<td>C, CN</td>
<td>Call of a function block, DFB or subprogram</td>
<td>Instance name of the function block, DFB or subprogram</td>
<td>A function block, DFB or subprogram is called up conditionally or unconditionally with <strong>CAL</strong>. see also <em>Calling Elementary Function Blocks and Derived Function Blocks</em>, p. 447 and <em>Subroutine Call</em>, p. 437</td>
</tr>
<tr>
<td>FUNCTIO NNAME</td>
<td>-</td>
<td>Executing a function</td>
<td>Literal, variable, direct address (data type is dependent on function)</td>
<td>A function is performed by specifying the name of the function. see also <em>Calling Elementary Functions</em>, p. 442</td>
</tr>
<tr>
<td>PROCEDU RENAME</td>
<td>-</td>
<td>Executing a procedure</td>
<td>Literal, variable, direct address (data type is dependent on procedure)</td>
<td>A procedure is performed by specifying the name of the procedure. see also <em>Calling Procedures</em>, p. 457</td>
</tr>
</tbody>
</table>
### Structuring Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Modifier</th>
<th>Meaning</th>
<th>Operands</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>JMP</strong></td>
<td>C, CN</td>
<td>Jump to label <em>LABEL</em></td>
<td>LABEL</td>
<td>With JMP a jump to the label can be conditional or unconditional.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>see also <em>Tags and Jumps, p. 438</em></td>
</tr>
<tr>
<td><strong>RET</strong></td>
<td>C, CN</td>
<td>Return to the next highest program organization unit</td>
<td>-</td>
<td>Every subroutine and every DFB (derived function block) is exited after being processed, i.e. it returns to the main program from which it was called.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If the subroutine/DFB is left prematurely, the return to the main program can be forced using <strong>RET</strong> (Return).</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td><strong>RET</strong> can only be used in subroutines or DFBs. It cannot be used in the main program.</td>
</tr>
<tr>
<td><strong>)</strong></td>
<td></td>
<td>Processing deferred operations</td>
<td>-</td>
<td>A right bracket <code>)</code> starts the processing of the deferred operator. The number of right bracket operations must be equal to the number of left bracket modifiers. Brackets can be nested.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td><strong>Example:</strong> In the example <code>E</code> is 1 if <code>C</code> and/or <code>D</code> is 1 and <code>A</code> and <code>B</code> are 1.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td><code>LD A</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td><code>AND B</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td><code>AND ( C OR D )</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td><code>ST E</code></td>
</tr>
</tbody>
</table>
Subroutine Call

Call Subroutine

A subroutine call consists of the **CAL** operator, followed by the name of the subroutine section, followed by an empty parameter list (optional).

Subroutine calls do not return a value.

The subroutine to be called must be located in the same task as the IL section called.

Subroutines can also be called from within subroutines.

**e.g.**

```
ST A
CAL SubroutineName ()
LD B
```

or

```
ST A
CAL SubroutineName
LD B
```

Subroutines are a supplement to IEC 61131-3 and must be enabled explicitly.

In SFC action sections, subroutine calls are only allowed when Multitoken Operation is enabled.
Tags and Jumps

Introduction  Tags serve as destinations for Jumps.

Tag properties:  Tag properties:
  ● Tags must always be the first element in a line.
  ● The name must be clear throughout the directory, and it is not upper/lower case sensitive.
  ● Tags can be 32 characters long (max.).
  ● Tags must conform to the IEC name conventions.
  ● Tags are separated by a colon : from the following instruction.
  ● Tags are only permitted at the beginning of "Expressions", otherwise an undefined value can be found in the battery.

Example:
  start: LD A
  AND B
  OR C
  ST D
  JMP start

Jump properties:  Jump properties:
  ● With JMP operation a jump to the tag can be restricted or unrestricted.
  ● JMP can be used with the modifiers C and CN (only if the battery content is data type BOOL).
  ● Jumps can be made within program and DFB sections.
  ● Jumps are only possible in the current section.

Possible destinations are:
  ● the first LD instruction of an EFB/DFB call with assignment of input parameters (see start2),
  ● a normal LD instruction (see start1),
  ● a CAL instruction, which does not work with assignment of input parameters (see start3),
  ● a JMP instruction (see start4),
  ● the end of an instruction list (see start5).
Example

start2: LD A
        ST counter.CU
        LD B
        ST counter.R
        LD C
        ST counter.PV
        CAL counter
        JMPCN start4

start1: LD A
        AND B
        OR C
        ST D
        JMPC start3
        LD A
        ADD E
        JMP start5

start3: CAL counter {
        CU:=A
        R:=B
        PV:=C
    }
        JMP start1

start4: JMPC start1

start5:
Comment

Description
In the IL editor, comments always start with the string (* and end in the string *). Any comments can be entered between these character strings.

Nesting comments is not permitted according to IEC 61131-3. If comments are nested nevertheless, then they must be enabled explicitly.
14.2 Calling Elementary Functions, Elementary Function Blocks, Derived Function Blocks and Procedures

Introduction

Overview

Calling Elementary Functions, Elementary Function Blocks, Derived Function Blocks and Procedures in the IL programming language.

What's in this Section?

This section contains the following topics:

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Calling Elementary Functions</td>
<td>442</td>
</tr>
<tr>
<td>Calling Elementary Function Blocks and Derived Function Blocks</td>
<td>447</td>
</tr>
<tr>
<td>Calling Procedures</td>
<td>457</td>
</tr>
</tbody>
</table>
Calling Elementary Functions

Using functions

Elementary functions are provided in the form of libraries. The logic of the functions is created in the programming language C and may not be modified in the IL editor. Functions have no internal states. If the input values are the same, the value on the output is the same every time the function is called. For example, the addition of two values always gives the same result. With some elementary functions, the number of inputs can be increased.

Elementary functions only have one return value (output).

Parameters

"Inputs" and one "output" are required to transfer values to or from a function. These are called formal parameters.

The current process states are transferred to the formal parameters. These are called actual parameters.

The following can be used as actual parameters for function inputs:
- Variable
- Address
- Literal

The following can be used as actual parameters for function outputs:
- Variable
- Address

The data type of the actual parameters must match the data type of the formal parameters. The only exceptions are generic formal parameters whose data type is determined by the actual parameter.

When dealing with generic ANY_BIT formal parameters, actual parameters of the INT or DINT (not UINT and UDINT) data types can be used.

This is a supplement to IEC 61131-3 and must be enabled explicitly.

Example:

Allowed:

\[
\text{AND (AnyBitParam := IntVar1, AnyBitParam2 := IntVar2)}
\]

Not allowed:

\[
\text{AND_WORD (WordParam1 := IntVar1, WordParam2 := IntVar2)}
\]

(In this case, AND_INT must be used.)

\[
\text{AND_ARRAY_WORD (ArrayInt, ...)}
\]

(In this case an explicit type conversion must be carried out using INT_ARR_TO_WORD_ARR (...).)
Not all formal parameters must be assigned a value for formal calls. Which formal parameter types must be assigned a value can be seen in the following table.

<table>
<thead>
<tr>
<th>Parameter type</th>
<th>EDT</th>
<th>STRING</th>
<th>ARRAY</th>
<th>ANY.ARRAY</th>
<th>IODDT</th>
<th>STRUCT</th>
<th>FB</th>
<th>ANY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>VAR_IN_OUT</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>/</td>
<td>/</td>
<td>+</td>
</tr>
<tr>
<td>Output</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>/</td>
<td>-</td>
</tr>
<tr>
<td>+</td>
<td>Actual parameter required</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>Actual parameter not required</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>/</td>
<td>not applicable</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

If no value is assigned to a formal parameter, the initial value will be used when the function is executed. If no initial value has been defined, the default value (0) is used.

**Programming notes**

Attention should be paid to the following programming notes:

- Functions are only executed if the input \( \text{EN} = 1 \) or the \( \text{EN} \) input is not used (see also \( \text{EN and ENO, p. 446} \)).
- All generic functions are overloaded. This means the functions can be called with or without entering the data type.

E.g.

LD i1
ADD i2
ST i3
is identical to

LD i1
ADD_INT i2
ST i3

- In contrast to ST, functions in IL cannot be nested.
- There are two ways of invoking a function:
  - Formal call (calling a function with formal parameter names)
  - Informal call (calling a function without formal parameter names)

**Formal call**

With this type of call (call with formal parameter names), the function is called using an instruction sequence consisting of the function name, followed by the bracketed list of value assignments (actual parameters) to the formal parameters. The order in which the formal parameters are listed is **not significant**. The list of actual parameters may be wrapped immediately following a comma. After executing the function the result is loaded into the accumulator and can be stored using \( \text{ST} \). \( \text{EN} \) and \( \text{ENO} \) can be used for this type of call.
Calling a function with formal parameter names:

Function Name: LIMIT
Formal Parameters:
- MN:=0
- IN:=var1
- MX:=var2

ST out

or

LIMIT (MN:=0, IN:=var1, MX:=var2)
ST out

Calling the same function in FBD:

<table>
<thead>
<tr>
<th>LIMIT</th>
<th>MN</th>
<th>IN</th>
<th>OUT</th>
<th>out</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>var1</td>
<td>var2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

With formal calls, values do not have to be assigned to all formal parameters (see also Parameters, p. 442).

LIMIT (MN:=0, IN:=var1)
ST out

Calling the same function in FBD:

<table>
<thead>
<tr>
<th>LIMIT</th>
<th>MN</th>
<th>IN</th>
<th>OUT</th>
<th>out</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>var1</td>
<td>var2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Informal call

With this type of call (call without formal parameter names), the function is called using an instruction sequence made up by loading the first actual parameter into the accumulator, followed by the function name and an optional list of actual parameters. The order in which the actual parameters are listed is significant. The list of actual parameters cannot be wrapped. After executing the function the result is loaded into the accumulator and can be stored using \textit{ST}. \textit{EN} and \textit{ENO} cannot be used for this type of call.

Calling a function with formal parameter names:

\begin{itemize}
  \item Actual Parameters \quad Function Name
  \item Limit \quad \text{B,C}
  \item ST \quad \text{result}
\end{itemize}

Calling the same function in FBD:

\begin{itemize}
  \item Limit
  \item A \quad \text{MN}
  \item B \quad \text{IN}
  \item C \quad \text{MX}
\end{itemize}

\begin{itemize}
  \item Limit \quad result
\end{itemize}

\textbf{Note:} Note that when making an informal call, the list of actual parameters cannot be put in brackets. IEC 61133-3 requires that the brackets be left out in this case to illustrate that the first actual parameter is not a part of the list.

\textbf{Invalid} informal call for a function:

\begin{itemize}
  \item LD A
  \item Limit (B,C)
\end{itemize}

If the value to be processed (first actual parameter) is already in the accumulator, the load instruction can be omitted.

\begin{itemize}
  \item Limit B,C
  \item ST \quad \text{result}
\end{itemize}

If the result is to be used immediately, the store instruction can be omitted.

\begin{itemize}
  \item LD A
  \item Limit \_REAL B,C
  \item MUL E
\end{itemize}

If the function to be executed only has one input, the name of the function is not followed by a list of actual parameters.
Calling a function with one actual parameter:

**Actual Parameters**
- LD A
- SIN
- ST result

**Function Name**
- Result of the function

Calling the same function in FBD:
- SIN
- A
- IN
- OUT
- result

**EN and ENO**

With all functions an EN input and an ENO output can be configured.

If the value of EN is equal to "0" when the function is called, the algorithms defined by the function are not executed and ENO is set to "0".

If the value of EN is equal to 1 when the function is called, the algorithms defined by the function are executed. After the algorithms have been executed successfully, the value of ENO is set to "1". If an error occurred while executing the algorithms, ENO is set to "0".

If ENO is set to "0" (caused when EN=0 or an error occurred during execution), the output of the function is set to "0".

The output behavior of the function does not depend on whether the function was called up without EN/ENO or with EN=1.

If EN/ENO are used, the function call must be formal.

**LIMIT** (EN:=1, MN:=0, IN:=var1, MX:=5, ENO=>var2)

ST out

Calling the same function in FBD:

- LIMIT
- 1
- 0
- var1
- 5
- EN
- MN
- IN
- MX
- ENO
- result
- var2
- out
Calling Elementary Function Blocks and Derived Function Blocks

**Elementary Function Block**

Elementary function blocks have internal states. If the inputs have the same values, the value on the output can have another value during the individual operations. For example, with a counter, the value on the output is incremented.

Function blocks can have several output values (outputs).

**Derived Function Block**

Derived function blocks (DFBs) have the same properties as elementary function blocks. The user can create them in the programming languages FBD, LD, IL, and/or ST.

**Parameter**

"Inputs and outputs" are required to transfer values to or from function blocks. These are called formal parameters.

The current process states are transferred to the formal parameters. They are called actual parameters.

The following can be used as actual parameters for function block inputs:
- Variable
- Address
- Literal

The following can be used as actual parameters for function block outputs:
- Variable
- Address

The data type of the actual parameters must match the data type of the formal parameters. The only exceptions are generic formal parameters whose data type is determined by the actual parameter.

Exception:

When dealing with generic ANY_BIT formal parameters, actual INT or DINT (not UINT and UDINT) parameters can be used.

This is a supplement to IEC 61131-3 and must be enabled explicitly.

Example:

**Allowed:**

\[
\text{AND (AnyBitParam := IntVar1, AnyBitParam2 := IntVar2)}
\]

**Not allowed:**

\[
\text{AND_WORD (WordParam1 := IntVar1, WordParam2 := IntVar2)}
\]

(In this case, \text{AND\_INT} must be used.)

\[
\text{AND\_ARRAY\_WORD (ArrayInt, ...)}
\]
(In this case an explicit type conversion must be carried out using
\texttt{INT\_ARR\_TO\_WORD\_ARR (...}).

Not all formal parameters need be assigned a value. You can see which formal
parameter types must be assigned a value in the following table.

<table>
<thead>
<tr>
<th>Parameter type</th>
<th>EDT</th>
<th>STRING</th>
<th>ARRAY</th>
<th>ANY_ARRAY</th>
<th>IODDT</th>
<th>STRUCT</th>
<th>FB</th>
<th>ANY</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFB: Input</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>/</td>
<td>+</td>
<td>/</td>
<td>/</td>
<td>+</td>
</tr>
<tr>
<td>EFB: VAR_IN_OUT</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>/</td>
<td>+</td>
<td>/</td>
<td>+</td>
</tr>
<tr>
<td>EFB: Output</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>/</td>
<td>-</td>
<td>/</td>
<td>+</td>
</tr>
<tr>
<td>DFB: Input</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>/</td>
<td>+</td>
<td>/</td>
<td>/</td>
<td>+</td>
</tr>
<tr>
<td>DFB: VAR_IN_OUT</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>/</td>
<td>+</td>
<td>/</td>
<td>+</td>
</tr>
<tr>
<td>DFB: Output</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>/</td>
<td>/</td>
<td>-</td>
<td>/</td>
<td>+</td>
</tr>
</tbody>
</table>

\begin{itemize}
  \item + Actual parameter required
  \item - Actual parameter not required
  \item / not applicable
\end{itemize}

If no value is allocated to a formal parameter, then the initial value will be used for
executing the function block. If no initial value has been defined then the default
value (0) is used.

If a formal parameter is not assigned a value and the function block/DFB is
instanced more than once, then the following instances are run with the old value.

\section*{Public Variables}

In addition to "inputs/outputs", some function blocks also provide public variables.

These variables transfer statistical values (values that are not influenced by the
process) to the function block. They are used for setting parameters for the function
block.

Public variables are a supplement to IEC 61131-3.

The assignment of values to public variables is made via their initial values or via the
load and save instructions.

Example:

\begin{verbatim}
LD 1
ST D\_ACT1\_OP\_CTRL
\end{verbatim}

(D\_ACT1 is an instance of the
function block D\_ACT and has
the public variables AREA\_NR
and OP\_CTRL.)

Public variables are read via the instance name of the function block and the names
of the public variables.
Programming notes

Attention should be paid to the following programming notes:

- Functions are only executed if the input $\text{EN}=1$ or the $\text{EN}$ input is not used (see also $\text{EN}$ and $\text{ENO}$, p. 454).
- The assignment of variables to $\text{ANY}$ or $\text{ARRAY}$ output types must be made using the $\Rightarrow$ operator (see also $\text{Formal form of CAL with a list of input parameters}$, p. 450).
  Assignments cannot be made outside the function block call.
  The instruction
  \[
  \text{My Var} := \text{My SAH.OUT}
  \]
  is invalid, if the output $\text{OUT}$ of the $\text{SAH}$ function block is of type $\text{ANY}$.
  The instruction
  \[
  \text{Cal My SAH (OUT}=>\text{My Var)}
  \]
  is valid.
- Special conditions apply when using $\text{VAR\_IN\_OUT}$ variables (See $\text{VAR\_IN\_OUT}$ variable, p. 455).
- The use of function blocks consists of two parts:
  - the Declaration (See Declaration, p. 449)
  - calling the function block,
- There are four ways of calling a function block:
  - $\text{Formal Form of CAL}$ with a list of input parameters (See $\text{Formal form of CAL with a list of input parameters}$, p. 450) (call with formal parameter names)
    In this case variables can be assigned to outputs using the $\Rightarrow$ operator.
  - $\text{Informal form of CAL}$ with a list of input parameters (See $\text{Informal form of CAL with a list of input parameters}$, p. 451) (call without formal parameter names)
  - $\text{CAL}$ and $\text{Load/Save}$ (See $\text{CAL}$ and $\text{Load/Save the input parameters}$, p. 452) the input parameter
  - Use of the input operators (See $\text{Use of the input operators}$, p. 453)
- Function block/DFB instances can be called multiple times; other than instances of communication EFBs, these can only be called once (see $\text{Multiple Function Block Instance Call}$, p. 454).

Declaration

Before calling a function block it must be declared in the variables editor.
Formal form of CAL with a list of input parameters

With this type of call (call with formal parameter names), the function block is called using a CAL instruction which follows the instance name of the function block and a bracketed list of actual parameter assignments to the formal parameters. The assignment of the input formal parameter is made using the := assignment and the output formal parameter is made using the => assignment. The sequence in which the input formal parameters and output formal parameters are enumerated is not significant. The list of actual parameters may be continued immediately following a comma.

EN and ENO can be used for this type of call.

Function block call in the formal form of CAL with a list of input parameters:

```
CAL MY_COUNT (CU:=var1, R:=reset, PV:=100, Q=>out, CV=>current)
```

Calling the same function block in FBD:

```
MY_COUNT

<table>
<thead>
<tr>
<th>CU</th>
<th>O</th>
<th>out</th>
</tr>
</thead>
<tbody>
<tr>
<td>var1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R</td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>CV</td>
<td>current</td>
</tr>
</tbody>
</table>
```

It is not necessary to assign a value to all formal parameters (see also Parameter, p. 447).

```
CAL MY_COUNT (CU:=var1, R:=reset, Q=>out, CV=>current)
```

Calling the same function block in FBD:

```
MY_COUNT

<table>
<thead>
<tr>
<th>CU</th>
<th>O</th>
<th>out</th>
</tr>
</thead>
<tbody>
<tr>
<td>var1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PV</td>
<td>CV</td>
<td>current</td>
</tr>
</tbody>
</table>
```
The value of a function block output can be stored and then saved by loading the function block output (function block instance name and separated by a full stop or entering the formal parameter).

Loading and saving function block outputs:

**Instance Name Formal Parameter (Output)**

```
LD  MY_COUNT.Q
ST  out
LD  MY_COUNT.CV
ST  current
```

**Informal form of CAL with a list of input parameters**

With this type of call (call without formal parameter names), the function block is called using a CAL instruction, that follows the instance name of the function block and a bracketed list of actual parameter for the inputs and outputs. The order in which the actual parameters are listed in a function block call is significant. The list of actual parameters cannot be wrapped.

EN and ENO cannot be used for this type of call.

Function block call in the informal form of CAL with a list of input parameters:

```
CAL MY_COUNT (var1, reset, 100, out, current)
```

**Calling the same function block in FBD:**

![FBD diagram]

With informal calls it is not necessary to assign a value to all formal parameters (see also Parameter, p. 447).

This is a supplement to IEC 61131-3 and must be enabled explicitly.

An empty parameter field is used to omit a parameter.

Call with empty parameter field:

```
CAL MY_COUNT (var1, , 100, out, current)
```
Calling the same function block in FBD:

```
MY_COUNT
```

An empty parameter field does not have to be used if formal parameters are omitted at the end.

MY_COUNT (var1, reset)

Calling the same function block in FBD:

```
MY_COUNT
```

**CAL and Load/Save the input parameters**

Function blocks may be called with an instruction list consisting of loading the actual parameters, followed by saving into the formal parameter, followed by the *CAL* instruction. The sequence of loading and saving the parameters is **not significant**.

Only load and save instructions for the function block currently being configured are allowed between the first load instruction for the actual parameters and the call of the function block. All other instructions are not allowed in this position.

It is not necessary to assign a value to all formal parameters (see also *Parameter*, p. 447).

**CAL with Load/Save the input parameters:**

```
LD var1
ST MY_COUNT.CU
LD reset
ST MY_COUNT.R
LD value
ST MY_COUNT.CV
CAL MY_COUNT.
LD MY_COUNT.Q
ST out
LD MY_COUNT.CV
ST current
```

Actual Parameter (Input)

Formal Parameter (Input)

Calling the function block

Formal Parameter (Output)

Actual Parameter (Output)
Use of the input operators

Function blocks can be called using an instruction list that consists of loading the actual parameters followed by saving them in the formal parameters followed by an input operator. The sequence of loading and saving the parameters is not significant.

Only load and save instructions for the function block currently being configured are allowed between the first load instruction for the actual parameters and the input operator of the function block. All other instructions are not allowed in this position.

EN and ENO cannot be used for this type of call.

It is not necessary to assign a value to all formal parameters (see also Parameter, p. 447).

The possible input operators for the various function blocks can be found in the table. Additional input operators are not available.

<table>
<thead>
<tr>
<th>Input Operator</th>
<th>FB type</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1, R</td>
<td>SR</td>
</tr>
<tr>
<td>S, R1</td>
<td>RS</td>
</tr>
<tr>
<td>CLK</td>
<td>R_TRIG</td>
</tr>
<tr>
<td>CLK</td>
<td>F_TRIG</td>
</tr>
<tr>
<td>CU, R, PV</td>
<td>CTU_INT, CTU_DINT, CTU_UINT, CTU_UDINT</td>
</tr>
<tr>
<td>CD, LD, PV</td>
<td>CTD_INT, CTD_DINT, CTD_UINT, CTD_UDINT</td>
</tr>
<tr>
<td>CU, CD, R, LD, PV</td>
<td>CTUD_INT, CTUD_DINT, CTUD_UINT, CTUD_UDINT</td>
</tr>
<tr>
<td>IN, PT</td>
<td>TP</td>
</tr>
<tr>
<td>IN, PT</td>
<td>TON</td>
</tr>
<tr>
<td>IN, PT</td>
<td>TOP</td>
</tr>
</tbody>
</table>

Use of the input operators:

- **LD reset**
- **ST MY_COUNT.R**
- **LD value**
- **ST MY_COUNT.PV**
- **LD vari**
- **CU MY_COUNT**
- **LD MY_COUNT.Q**
- **ST out**
- **LD MY_COUNT.CV**
- **ST current**
Calling a function block without inputs

Even if the function block has no inputs or the inputs are not to be parameterized, the function block should be called before its outputs can be used. Otherwise the initial values of the outputs will be transferred, i.e. “0”.

E.g.

Calling the function block in the IL programming language:

```
CAL MY_CLOCK ()
CAL MY_COUNT (CU:=MY_CLOCK.CLK1, R:=reset, PV:=100)
LD MY_COUNT.Q
ST out
LD MY_COUNT.CV
ST current
```

Calling the same function block in FBD:

```
MY_CLOCK
MY_COUNT

SYSCLK

CLK1
CLK2
CLK3
CLK4
CLK5

MY_CLOCK

CLK1
CLK2
CLK3
CLK4
CLK5

CTU

CU

R

reset

100

PV

SYSCLOCK

CLK1
CLK2
CLK3
CLK4
CLK5

MY_COUNT

CLK1
CLK2
CLK3
CLK4
CLK5

CTU

CU

R

reset

100

PV

MY_COUNT

CLK1
CLK2
CLK3
CLK4
CLK5

```

Multiple Function Block Instance Call

Function block/DFB instances can be called multiple times; other than instances of communication EFBs, these can only be called once.

Calling the same function block/DFB instance more than once makes sense, for example, in the following cases:

- If the function block/DFB has no internal value or it is not required for further processing.
  In this case, memory is saved by calling the same function block/DFB instance more than once since the code for the function block/DFB is only loaded one time. The function block/DFB is then handled like a “Function”.
- If the function block/DFB has an internal value and this is supposed to influence various program segments, for example, the value of a counter should be increased in different parts of the program.
  In this case, calling the same function block/DFB means that temporary results do not have to be saved for further processing in another part of the program.

EN and ENO

With all function blocks/DFBs, an EN input and an ENO output can be configured.

If the value of EN is equal to “0”, when the function block/DFB is called, the algorithms defined by the function block/DFB are not executed and ENO is set to “0”.

EN and ENO

With all function blocks/DFBs, an EN input and an ENO output can be configured.

If the value of EN is equal to “0”, when the function block/DFB is called, the algorithms defined by the function block/DFB are not executed and ENO is set to “0”.
If the value of $EN$ is equal to "1", when the function block/DFB is invoked, the algorithms which are defined by the function block/DFB will be executed. After the algorithms have been executed successfully, the value of $ENO$ is set to "1". If an error occurs when executing these algorithms, $ENO$ is set to "0".

If $ENO$ is set to "0" (results from $EN=0$ or an error during execution), the outputs of the function block/DFB retain the status from the last cycle in which they were correctly executed.

The output behavior of the function blocks/DFBs does not depend on whether the function blocks/DFBs are called without $EN/ENO$ or with $EN=1$.

If $EN/ENO$ are used, the function block call must be formal. The assignment of variables to $ENO$ must be made using the => operator.

Calling the same function block in FBD:

```
VAR_IN_OUT
variable
Function blocks are often used to read a variable at an input (input variables), to process it and to output the updated values of the same variable (output variables). This special type of input/output variable is also called a VAR_IN_OUT variable.

The following special features are to be noted when using function blocks/DFBs with VAR_IN_OUT variables.

- All VAR_IN_OUT inputs must be assigned a variable.
- VAR_IN_OUT inputs may not have literals or constants assigned to them.
- VAR_IN_OUT outputs may not have values assigned to them.
- VAR_IN_OUT variables cannot be used outside the block call.

Calling a function block with a VAR_IN_OUT variable in IL:

```
CAL MY_FBLOCK(IN1:=V1, IN2:=V2, IO1:=V3, OUT1=>V4, OUT2=>V5)
```

Calling the same function block in FBD:

```
VAR_IN_OUT variables cannot be used outside the function block call.
```
The following function block calls are therefore invalid:

**Invalid call, example 1:**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD V1</td>
<td>Loading a V1 variable in the accumulator</td>
</tr>
<tr>
<td>CAL InOutFB</td>
<td>Calling a function block with the VAR_IN_OUT parameter. The accumulator now contains a reference to a VAR_IN_OUT parameter.</td>
</tr>
<tr>
<td>AND V2</td>
<td>AND operation on accumulator contents and V2 variable. <strong>Error:</strong> The operation cannot be performed since the VAR_IN_OUT parameter (accumulator contents) cannot be accessed from outside the function block call.</td>
</tr>
</tbody>
</table>

**Invalid call, example 2:**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD V1</td>
<td>Loading a V1 variable in the accumulator</td>
</tr>
<tr>
<td>AND InOutFB.inout</td>
<td>AND operation on accumulator contents and a reference to a VAR_IN_OUT parameter. <strong>Error:</strong> The operation cannot be performed since the VAR_IN_OUT parameter cannot be accessed from outside the function block call.</td>
</tr>
</tbody>
</table>

The following function block calls are always valid:

**Valid call, example 1:**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAL InOutFB (IN1:=V1,inout:=V2)</td>
<td>Calling a function block with the VAR_IN_OUT parameter and assigning the actual parameter within the function block call.</td>
</tr>
</tbody>
</table>

**Valid call, example 2:**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD V1</td>
<td>Loading a V1 variable in the accumulator</td>
</tr>
<tr>
<td>ST InOutFB.IN1</td>
<td>Assigning the accumulator contents to the IN1 parameter of the IN1 function block.</td>
</tr>
<tr>
<td>CAL InOutFB(inout:=V2)</td>
<td>Calling the function block with assignment of the actual parameter (V2) to the VAR_IN_OUT parameter.</td>
</tr>
</tbody>
</table>
Calling Procedures

Procedure

Procedures are provided in the form of libraries. The logic of the procedure is created in the programming language C and may not be modified in the IL editor.

Procedures - like functions - have no internal states. If the input values are the same, the value on the output is the same every time the procedure is executed. For example, the addition of two values gives the same result every time.

In contrast to functions, procedures do not return a value and support VAR_IN_OUT variables.

Procedures are a supplement to IEC 61131-3 and must be enabled explicitly.

Parameter

"Inputs and outputs" are required to transfer values to or from procedures. These are called formal parameters.

The current process states are transferred to the formal parameters. These are called actual parameters.

The following can be used as actual parameters for procedure inputs:
- Variable
- Address
- Literal

The following can be used as actual parameters for procedure outputs:
- Variable
- Address

The data type of the actual parameter must match the data type of the formal parameter. The only exceptions are generic formal parameters whose data type is determined by the actual parameter.

When dealing with generic ANY_BIT formal parameters, actual parameters of the INT or DINT (not UINT and UDINT) data types can be used.

This is a supplement to IEC 61131-3 and must be enabled explicitly.

Example:

Allowed:

\[ \text{AND (AnyBitParam := IntVar1, AnyBitParam2 := IntVar2)} \]

Not allowed:

\[ \text{AND\_WORD (WordParam1 := IntVar1, WordParam2 := IntVar2)} \]

(In this case, AND\_INT must be used.)

\[ \text{AND\_ARRAY\_WORD (ArrayInt, ...)} \]
(In this case an explicit type conversion must be carried out using 
\texttt{INT\_ARR\_TO\_WORD\_ARR} (\ldots).

Not all formal parameters must be assigned a value for formal calls. Which formal parameter types must be assigned a value can be seen in the following table.

<table>
<thead>
<tr>
<th>Parameter type</th>
<th>EDT</th>
<th>STRING</th>
<th>ARRAY</th>
<th>ANY_ARRAY</th>
<th>IODDT</th>
<th>STRUCT</th>
<th>FB</th>
<th>ANY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>VAR_IN_OUT</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>/</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Output</td>
<td>-</td>
<td></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>/</td>
<td>+</td>
</tr>
</tbody>
</table>

+ Actual parameter required
- Actual parameter not required
/ not applicable

If no value is allocated to a formal parameter, then the initial value will be used for executing the function block. If no initial value has been defined, the default value (0) is used.

**Programming notes**

Attention should be paid to the following programming notes:

- Procedures are only executed if the input \texttt{EN}\texttt{=}1 or the \texttt{EN} input is not used (see also \textit{EN and ENO, p. 461}).
- Special conditions apply when using \texttt{VAR\_IN\_OUT} variables (See \texttt{VAR\_IN\_OUT variable, p. 462}).
- There are two ways of calling a procedure:
  - Formal call (calling a function with formal parameter names)
    - In this case variables can be assigned to outputs using the \texttt{=} \rightarrow operator (calling a function block in shortened form).
  - Informal call (calling a function without formal parameter names)

**Formal call**

With this type of call (call with formal parameter names), the procedure is called using an optional \texttt{CAL} instruction sequence followed by the name of the procedure and a bracketed list of actual parameter to formal parameter assignments. The assignment of the input formal parameter is made using the \texttt{=} assignment and the output formal parameter is made using the \texttt{=} \rightarrow assignment. The order in which the input formal parameters and output formal parameters are listed is not significant.

The list of actual parameters may be wrapped immediately following a comma. \texttt{EN} and \texttt{ENO} can be used for this type of call.
Calling a procedure with formal parameter names:

```
PROC (IN1:=var1, IN2:=var1, OUT1=>result1, OUT2=>result2)
```

or

```
CAL PROC (IN1:=var1, IN2:=var1, OUT1=>result1, OUT2=>result2)
```

or

```
PROC (IN1:=var1, IN2:=var1, OUT1=>result1, OUT2=>result2)
```

or

```
CAL PROC (IN1:=var1, IN2:=var1, OUT1=>result1, OUT2=>result2)
```

Calling the same procedure in FBD:

```
PROC

<table>
<thead>
<tr>
<th>var1</th>
<th>IN1</th>
<th>OUT1</th>
<th>result1</th>
</tr>
</thead>
<tbody>
<tr>
<td>var2</td>
<td>IN2</td>
<td>OUT2</td>
<td>result2</td>
</tr>
</tbody>
</table>
```

With formal calls, values do not have to be assigned to all formal parameters (see also Parameter, p. 457).

```
PROC (IN1:=var1, OUT1=>result1, OUT2=>result2)
```

or

```
CAL PROC (IN1:=var1, OUT1=>result1, OUT2=>result2)
```

Calling the same procedure in FBD:

```
PROC

<table>
<thead>
<tr>
<th>var1</th>
<th>IN1</th>
<th>OUT1</th>
<th>result1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>IN2</td>
<td>OUT2</td>
<td>result2</td>
</tr>
</tbody>
</table>
```
Informal Call Without CAL Instruction

With this type of call (call without formal parameter names), procedures are called using an instruction sequence consisting of the first actual parameter loaded into the accumulator, followed by the procedure name, followed by a list of the input and output actual parameters. The order in which the actual parameters are listed is significant. The list of actual parameters cannot be wrapped.

**EN** and **ENO** cannot be used for this type of call.

Calling a procedure with formal parameter names:

```
LD var1
PROC var2,result1,result2
```

Calling the same procedure in FBD:

```
PROC
  var1 IN1 OUT1 result1
  var2 IN2 OUT2 result2
```

**Note:** Note that when making an informal call, the list of actual parameters cannot be put in brackets. IEC 61133-3 requires that the brackets be left out in this case to illustrate that the first actual parameter is not a part of the list.

**Invalid** informal call for a procedure:

```
LD A
LIMIT (B,C)
```

If the value to be processed (first actual parameter) is already in the accumulator, the load instruction can be omitted.

```
EXAMP1 var2,result1,result2
```

Informal Call With CAL Instruction

With this type of call, procedures are called using an instruction sequence consisting of the **CAL** instruction, followed by the procedure name followed by a list of the input and output actual parameters. The order in which the actual parameters are listed is significant. The list of actual parameters cannot be wrapped.

**EN** and **ENO** cannot be used for this type of call.
Calling a procedure with formal parameter names using CAL instruction:

Actual Parameters (Inputs)

CAL PROC (var1, var2, result1, result2)

Procedure Name

Actual Parameters (Outputs)

or

CAL PROC (var1, var2, result1, result2)

Calling the same procedure in FBD:

<table>
<thead>
<tr>
<th>var1</th>
<th>IN1</th>
<th>PROC</th>
<th>result1</th>
</tr>
</thead>
<tbody>
<tr>
<td>var2</td>
<td>IN2</td>
<td>OUT1</td>
<td>result2</td>
</tr>
</tbody>
</table>

Note: Unlike informal calls without a CAL instruction, when making informal calls with a CAL instruction, the value to be processed (first actual parameter) is not explicitly loaded in the battery. Instead it is part of the list of actual parameters. For this reason, when making informal calls with a CAL instruction, the list of actual parameters must be put in brackets.

EN and ENO

With all procedures, an EN input and an ENO output can be configured.

If the value of EN is equal to "0" when the procedure is called, the algorithms defined by the procedure are not executed and ENO is set to "0".

If the value of EN is "1" when the procedure is called, the algorithms defined by the function are executed. After the algorithms have been executed successfully, the value of ENO is set to "1". If an error occurs when executing these algorithms, ENO is set to "0".

If ENO is set to "0" (caused when EN=0 or an error occurred during executing), the outputs of the procedure are set to "0".

If EN/ENO are used, the procedure call must be formal. The assignment of variables to ENO must be made using the => operator.

PROC (EN:=1, IN1:=var1, IN2:=var2, ENO=>error, OUT1=>result1, OUT2=>result2) ;
Calling the same procedure in FBD:

VAR_IN_OUT variables cannot be used outside the procedure call.

The following procedure calls are therefore invalid:

Invalid call, example 1:

<table>
<thead>
<tr>
<th>LD V1</th>
<th>Loading a V1 variable in the accumulator</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAL InOutProc</td>
<td>Calling a procedure with the VAR_IN_OUT parameter. The accumulator now contains a reference to a VAR_IN_OUT parameter.</td>
</tr>
<tr>
<td>AND V2</td>
<td>AND operation on contents of accumulator with variable V2. Error: The operation cannot be carried out since the VAR_IN_OUT parameter (contents of accumulator) cannot be accessed outside the procedure call.</td>
</tr>
</tbody>
</table>
**Invalid call, example 2:**

<table>
<thead>
<tr>
<th>LD V1</th>
<th>Loading a V1 variable in the accumulator</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND InOutProc.inout</td>
<td>AND operation on the contents of the accumulator and a reference to a VAR_IN_OUT parameter.</td>
</tr>
<tr>
<td><strong>Fehler:</strong></td>
<td>The operation cannot be carried out since the VAR_IN_OUT parameter cannot be accessed outside the procedure call.</td>
</tr>
</tbody>
</table>

**Invalid call, example 3:**

<table>
<thead>
<tr>
<th>LD V1</th>
<th>Loading a V1 variable in the accumulator</th>
</tr>
</thead>
<tbody>
<tr>
<td>InOutFB V2</td>
<td>Calling the procedure with assignment of the actual parameter (V2) to the VAR_IN_OUT parameter.</td>
</tr>
<tr>
<td><strong>Error:</strong></td>
<td>The operation cannot be carried out as with this type of procedure call only the VAR_IN_OUT parameter would be stored in the accumulator for later use.</td>
</tr>
</tbody>
</table>

The following procedure calls are always **valid**:

**Valid call, example 1:**

| CAL InOutProc (IN1:=V1,inout:=V2) | Calling a procedure with the VAR_IN_OUT parameter and formal assignment of the actual parameter within the procedure call. |

**Valid call, example 2:**

| InOutProc (IN1:=V1,inout:=V2) | Calling a procedure with the VAR_IN_OUT parameter and formal assignment of the actual parameter within the procedure call. |

**Valid call, example 3:**

| CAL InOutProc (V1,V2) | Calling a procedure with the VAR_IN_OUT parameter and informal assignment of the actual parameter within the procedure call. |
Structured Text ST

Introduction

Overview
This chapter describes the programming language structured text ST which conforms to IEC 61131.

What's in this Chapter?
This chapter contains the following sections:

<table>
<thead>
<tr>
<th>Section</th>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>15.1</td>
<td>General information about the structured text ST</td>
<td>466</td>
</tr>
<tr>
<td>15.2</td>
<td>Instructions</td>
<td>476</td>
</tr>
<tr>
<td>15.3</td>
<td>Calling Elementary Functions, Elementary Function Blocks,</td>
<td>494</td>
</tr>
<tr>
<td></td>
<td>Derived Function Blocks and Procedures</td>
<td></td>
</tr>
</tbody>
</table>
### 15.1 General information about the structured text ST

**Introduction**

This section contains a general overview of the structured text ST.

**What's in this Section?**

This section contains the following topics:

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>General information about structured text (ST)</td>
<td>467</td>
</tr>
<tr>
<td>Operands</td>
<td>469</td>
</tr>
<tr>
<td>Operators</td>
<td>471</td>
</tr>
</tbody>
</table>
General information about structured text (ST)

Introduction
With the programming language of structured text (ST), it is possible, for example, to call up function blocks, perform functions and assignments, conditionally perform instructions and repeat tasks.

Expression
The ST programming language works with "Expressions". Expressions are constructions consisting of operators and operands that return a value when executed.

Operator
Operators are symbols representing the operations to be executed.

Operand
Operators are used for operands. Operands are variables, literals, FFB inputs/outputs etc.

Instructions
Instructions are used to assign the values returned from the expressions to actual parameters and to structure and control the expressions.

Representation of an ST section
Representation of an ST section:

Expression

Operator

Operand

Instruction

D := B*B - 4*A*C ;

IF D < 0.0 THEN NROOTS := 0 ;
ELSIF D = 0.0 THEN
NROOTS := 1 ;
X1 := - B / (2.0*A) ;
ELSE
NROOTS := 2 ;
X1 := (- B + SQRT(D)) / (2.0*A) ;
X2 := (- B - SQRT(D)) / (2.0*A) ;
END_IF ;

Section Size
The length of an instruction line is limited to 300 characters.
The length of an ST section is not limited within the programming environment. The length of an ST section is only limited by the size of the PLC memory.

Syntax
Identifiers and Keywords are not case sensitive.

Exception: Not allowed - spaces and tabs
- keywords,
- literals,
values, identifiers, variables and limiter combinations [e.g. (* for comments)].

Execution sequence

The evaluation of an expression consists of applying the operators to the operands in the sequence as defined by the rank of the operators (see Table of Operators (See Table of Operators, p. 471)). The operator with the highest rank in an expression is performed first, followed by the operator with the next highest rank, etc., until the evaluation is complete. Operators with the same rank are performed from left to right, as they are written in the expression. This sequence can be altered with the use of parentheses.

If, for example, A, B, C and D have the values 1, 2, 3 and 4, and are calculated as follows:

\[ A + B - C \times D \]

the result is -9.

In the case of the following calculation:

\[(A + B - C) \times D\]

the result is 0.

If an operator contains two operands, the left operand is executed first, e.g. in the expression

\[ \sin(A) \times \cos(B) \]

the expression \( \sin(A) \) is calculated first, then \( \cos(B) \) and only then is the product calculated.

Error behavior

The following conditions are handled as an error when executing an expression:

- Attempting to divide by 0.
- Operands do not contain the correct data type for the operation.
- The result of a numerical operation exceeds the value range of its data type

If an error occurs when executing the operation, the corresponding Systembit (%S) is set (if supported by the PLC being used).

IEC Conformity

For a description of IEC conformity for the ST programming language, see IEC Conformity (See IEC Compliance, p. 601).
Operands

Introduction
An operand can be:
- an address
- a literal
- a variable
- a multi-element variable
- an element of a multi-element variable
- a function call
- an FFB output

Data types
Data types, which are in an instruction of processing operands, must be identical. Should operands of various types be processed, a type conversion must be performed beforehand.

In the example the integer variable i1 is converted into a real variable before being added to the real variable r4.

\[ r3 := r4 + \sin(\text{INT\_TO\_REAL}(i1)) \]

As an exception to this rule, variables with data type TIME can be multiplied or divided by variables with data type INT, DINT, UINT or UDINT.

Permitted operations:
- \( \text{timeVar1} := \text{timeVar2} / \text{dintVar1} \);
- \( \text{timeVar1} := \text{timeVar2} \times \text{intVar1} \);
- \( \text{timeVar} := 10 \times \text{time#10s} \);

This function is listed by IEC 61131-3 as "undesired" service.
Direct use of addresses

Addresses can be used directly (without a previous declaration). In this case the addresses data type is assigned directly. The assignment is made using the "Large prefix".

The different large prefixes are given in the following table:

<table>
<thead>
<tr>
<th>Large prefix / Symbol</th>
<th>Example</th>
<th>Data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>no prefix</td>
<td>%I10,%CH203.MOD,%CH203.MOD.ERR</td>
<td>BOOL</td>
</tr>
<tr>
<td>X</td>
<td>%MW20</td>
<td>BOOL</td>
</tr>
<tr>
<td>B</td>
<td>%QB102.3</td>
<td>BYTE</td>
</tr>
<tr>
<td>W</td>
<td>%MW43</td>
<td>INT</td>
</tr>
<tr>
<td>D</td>
<td>%QD100</td>
<td>DINT</td>
</tr>
<tr>
<td>F</td>
<td>%MF100</td>
<td>REAL</td>
</tr>
</tbody>
</table>

Using other data types

Should other data types be assigned as the default data types of an address, this must be done through an explicit declaration. This variable declaration takes place comfortably using the variable editor. The data type of an address can not be declared directly in an ST section (e.g. declaration AT %MW1: UINT; not permitted).

For example, the following variables are declared in the variable editor:

```plaintext
UnlocV1: ARRAY [1..10] OF INT;
LocV1: ARRAY [1..10] OF INT AT %MW100;
LocV2: TIME AT %MW100;
```

The following calls then have the correct syntax:

```plaintext
%MW200 := 5;
UnlocV1[2] := LocV1[%MW200];
LocV2 := t#3s;
```

Accessing field variables

When accessing field variables (ARRAY), only literals and variables of the INT, UINT, DINT and UDINT data types are permitted in the index entry.

The index of an ARRAY element can be negative if the lower threshold of the range is negative.

Example: Using field variables

```plaintext
var1[i] := 8;
var2.otto[4] := var3;
var4[1+i+j*5] := 4;
```
Operators

Introduction
An operator is a symbol for:
- an arithmetic operation to be executed or
- a logical operation to be executed or
- a function edit (call).

Operators are generic, i.e. they adapt automatically to the data type of the operands.

Table of Operators
Operators are executed in sequence according to priority, see also Execution sequence, p. 468.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
<th>Order of rank</th>
<th>possible operands</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>()</td>
<td>Use of Brackets:</td>
<td>1 (highest)</td>
<td>Expression</td>
<td>Brackets are used to alter the execution sequence of the operators. Example: If the operands A, B, C and D have the values 1, 2, 3, and 4, A+B-C*D has the result -9 and (A+B-C)*D has the result 0.</td>
</tr>
<tr>
<td>FUNCNAME (Actual parameter-list)</td>
<td>Function processing (call)</td>
<td>2</td>
<td>Expression, Literal, Variable, Address (all data types)</td>
<td>Function processing is used to execute functions (see Calling Elementary Functions, p. 495).</td>
</tr>
<tr>
<td>-</td>
<td>Negation</td>
<td>3</td>
<td>Expression, Literal, Variable, Address of Data Type INT, DINT, UINT, UDINT or REAL</td>
<td>During negation – a sign reversal for the value of the operand takes place. Example: In the example OUT is -4 if IN1 is 4. OUT := - IN1 ;</td>
</tr>
<tr>
<td>NOT</td>
<td>Complement</td>
<td>3</td>
<td>Expression, Literal, Variable, Address of Data Type BOOL, BYTE, WORD or DWORD</td>
<td>In NOT a bit by bit inversion of the operands takes place. Example: In the example OUT is 0011001100 if IN1 is 1100110011. OUT := NOT IN1 ;</td>
</tr>
<tr>
<td>**</td>
<td>Exponentiation</td>
<td>4</td>
<td>Expression, Literal, Variable, Address of Data Type REAL (Basis) and INT, DINT, UINT, UDINT or REAL (Exponent)</td>
<td>In exponentiation, ** the value of the first operand (basis) is raised to the power of the second operand (exponent). Example: In the example OUT is 625.0 if IN1 is 5.0 and IN2 is 4.0. OUT := IN1 ** IN2 ;</td>
</tr>
</tbody>
</table>
In multiplication, \( \ast \) the value of the first operand is multiplied by the value of the second operand (exponent).  
**Example:** In the example \( \text{OUT} \) is 20.0 if \( \text{IN1} \) is 5.0 and \( \text{IN2} \) is 4.0.  
\[
\text{OUT} := \text{IN1} \ast \text{IN2} ;
\]
**Note:** The MULTIME function in the obsolete library is available for multiplications involving the data type `Time`.

In division, \(/\) the value of the first operand is divided by the value of the second operand.  
**Example:** In the example \( \text{OUT} \) is 4.0 if \( \text{IN1} \) is 20.0 and \( \text{IN2} \) is 5.0.  
\[
\text{OUT} := \text{IN1} \div \text{IN2} ;
\]
**Note:** The DIVTIME function in the obsolete library is available for divisions involving the data type `Time`.

For MOD the value of the first operand is divided by that of the second operand and the remainder of the division (Modulo) is displayed as the result.  
**Example:** In this example  
- \( \text{OUT} \) is 1 if \( \text{IN1} \) is 7 and \( \text{IN2} \) is 2  
- \( \text{OUT} \) is 1 if \( \text{IN1} \) is 7 and \( \text{IN2} \) is -2  
- \( \text{OUT} \) is -1 if \( \text{IN1} \) is -7 and \( \text{IN2} \) is 2  
- \( \text{OUT} \) is -1 if \( \text{IN1} \) is -7 and \( \text{IN2} \) is -2  
\[
\text{OUT} := \text{IN1} \mod \text{IN2} ;
\]

In addition, \(+\) the value of the first operand is added to the value of the second operand.  
**Example:** In this example  
\( \text{OUT} \) is 9, if \( \text{IN1} \) is 7 and \( \text{IN2} \) is 2  
\[
\text{OUT} := \text{IN1} + \text{IN2} ;
\]

In subtraction, \(-\) the value of the second operand is subtracted from the value of the first operand.  
**Example:** In the example \( \text{OUT} \) is 6 if \( \text{IN1} \) is 10 and \( \text{IN2} \) is 4.  
\[
\text{OUT} := \text{IN1} - \text{IN2} ;
\]
<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
<th>Order of rank</th>
<th>possible operands</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;</td>
<td>Less than comparison</td>
<td>7</td>
<td>Expression, Literal, Variable, Address of Data Type BOOL, BYTE, INT, DINT, UINT, UDINT, REAL, TIME, WORD, DWORD, STRING, DT, DATE or TOD</td>
<td>The value of the first operand is compared with the value of the second using &lt;. If the value of the first operand is less than the value of the second, the result is a Boolean 1. If the value of the first operand is greater than or equal to the value of the second, the result is a Boolean 0. <strong>Example:</strong> In the example OUT is 1 if IN1 is less than 10 and is otherwise 0. OUT := IN1 &lt; 10 ;</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than comparison</td>
<td>7</td>
<td>Expression, Literal, Variable, Address of Data Type BOOL, BYTE, INT, DINT, UINT, UDINT, REAL, TIME, WORD, DWORD, STRING, DT, DATE or TOD</td>
<td>The value of the first operand is compared with the value of the second using &gt;. If the value of the first operand is greater than the value of the second, the result is a Boolean 1. If the value of the first operand is less than or equal to the value of the second, the result is a Boolean 0. <strong>Example:</strong> In the example OUT is 1 if IN1 is greater than 10, and is 0 if IN1 is less than 0. OUT := IN1 &gt; 10 ;</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less than or equal to comparison</td>
<td>7</td>
<td>Expression, Literal, Variable, Address of Data Type BOOL, BYTE, INT, DINT, UINT, UDINT, REAL, TIME, WORD, DWORD, STRING, DT, DATE or TOD</td>
<td>The value of the first operand is compared with the value of the second operand using &lt;=. If the value of the first operand is less than or equal to the value of the second, the result is a Boolean 1. If the value of the first operand is greater than the value of the second, the result is a Boolean 0. <strong>Example:</strong> In the example OUT is 1 if IN1 is less than or equal to 10, and otherwise is 0. OUT := IN1 &lt;= 10 ;</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater than or equal to comparison</td>
<td>7</td>
<td>Expression, Literal, Variable, Address of Data Type BOOL, BYTE, INT, DINT, UINT, UDINT, REAL, TIME, WORD, DWORD, STRING, DT, DATE or TOD</td>
<td>The value of the first operand is compared with the value of the second operand using &gt;=. If the value of the first operand is greater than or equal to the value of the second, the result is a Boolean 1. If the value of the first operand is less than the value of the second, the result is a Boolean 0. <strong>Example:</strong> In the example OUT is 1 if IN1 is greater than or equal to 10, and otherwise is 0. OUT := IN1 &gt;= 10 ;</td>
</tr>
</tbody>
</table>
The value of the first operand is compared with the value of the second operand using `=`. If the value of the first operand is equal to the value of the second, the result is a Boolean 1. If the value of the first operand is not equal to the value of the second, the result is a Boolean 0.

**Example:** In the example `OUT` is 1 if `IN1` is equal to 10 and is otherwise 0.

```plaintext
OUT := IN1 = 10 ;
```

The value of the first operand is compared with the value of the second using `< >`. If the value of the first operand is not equal to the value of the second, the result is a Boolean 1. If the value of the first operand is equal to the value of the second, the result is a Boolean 0.

**Example:** In the example `OUT` is 1 if `IN1` is not equal to 10 and is otherwise 0.

```plaintext
OUT := IN1 <> 10 ;
```

With `&`, there is a logical AND link between the operands. In the case of `BYTE`, `WORD` and `DWORD` data types, the link is made bit by bit.

**Example:** In the examples `OUT` is 1 if `IN1`, `IN2` and `IN3` are 1.

```plaintext
OUT := IN1 & IN2 & IN3 ;
```

With `AND`, there is a logical OR link between the operands. In the case of `BYTE`, `WORD` and `DWORD` data types, the link is made bit by bit.

**Example:** In the examples `OUT` is 1 if `IN1`, `IN2` and `IN3` are 1.

```plaintext
OUT := IN1 AND IN2 AND IN3 ;
```
### XOR

**Operator**: XOR  
**Meaning**: Logical Exclusive OR  
**Order of rank**: 10  
**possible operands**: Expression, Literal, Variable, Address of Data Type BOOL, BYTE, WORD or DWORD  
**Description**:  
With XOR, there is a logical Exclusive OR link between the operations. In the case of BYTE, WORD and DWORD data types, the link is made bit by bit.  
**Example**: In the example OUT is 1 if IN1 and IN2 are not equal. If A and B have the same status (both 0 or 1), D is 0.  
OUT := IN1 XOR IN2 ;  
If more than two operands are linked, the result with an uneven number of 1-states is 1, and is 0 with an even number of 1-states.  
**Example**: In the example OUT is 1 if 1 or 3 operands are 1. OUT is 0 if 0, 2 or 4 operands are 1.  
OUT := IN1 XOR IN2 XOR IN3 XOR IN4 ;

### OR

**Operator**: OR  
**Meaning**: Logical OR  
**Order of rank**: 11 (lowest)  
**possible operands**: Expression, Literal, Variable, Address of Data Type BOOL, BYTE, WORD or DWORD  
**Description**:  
With OR, there is a logical OR link between the operands. With the BYTE and WORD, DWORD data types, the link is made bit by bit.  
**Example**: In the example OUT is 1 if IN1, IN2 or IN3 is 1.  
OUT := IN1 OR IN2 OR IN3 ;
15.2 Instructions

Introduction

Overview

This section describes the instructions for the programming language of structured text ST.

What's in this Section?

This section contains the following topics:

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instructions</td>
<td>477</td>
</tr>
<tr>
<td>Assignment</td>
<td>478</td>
</tr>
<tr>
<td>Select Instruction IF...THEN...END_IF</td>
<td>480</td>
</tr>
<tr>
<td>Select Instruction ELSE</td>
<td>481</td>
</tr>
<tr>
<td>Select Instruction ELSIF...THEN</td>
<td>482</td>
</tr>
<tr>
<td>Select Instruction CASE...OF...END_CASE</td>
<td>483</td>
</tr>
<tr>
<td>Repeat Instruction FOR...TO...BY...DO...END_FOR</td>
<td>484</td>
</tr>
<tr>
<td>Repeat Instruction WHILE...DO...END_WHILE</td>
<td>486</td>
</tr>
<tr>
<td>Repeat Instruction REPEAT...UNTIL...END_REPEAT</td>
<td>487</td>
</tr>
<tr>
<td>Repeat Instruction EXIT</td>
<td>488</td>
</tr>
<tr>
<td>Subroutine Call</td>
<td>489</td>
</tr>
<tr>
<td>RETURN</td>
<td>490</td>
</tr>
<tr>
<td>Empty Instruction</td>
<td>491</td>
</tr>
<tr>
<td>Labels and Jumps</td>
<td>492</td>
</tr>
<tr>
<td>Comment</td>
<td>493</td>
</tr>
</tbody>
</table>
Instructions

Description

Instructions are the “Commands” of the ST programming language.
Instructions must be terminated with semicolons.
Several instructions (separated by semicolons) can be present in one line.
A single semicolon represents an Empty instruction (See Empty Instruction, p. 491).
Assignment

Introduction
When an assignment is performed, the current value of a single or multi-element variable is replaced by the result of the evaluation of the expression.

An assignment consists of a variable specification on the left side, followed by the assignment operator :=, followed by the expression to be evaluated.

Both variables (left and right sides of the assignment operator) must have the same data type.

Arrays are a special case. After being explicitly enabled, assignment of two arrays with different lengths can be made.

Assigning the value of a variable to another variable
Assignments are used to assign the value of a variable to another variable.

The instruction
A := B ;

is used, for example, to replace the value of the variable A with the current value of variable B. If A and B are elementary data types, the individual value of B is passed to A. If A and B are derived data types, the values of all B elements are passed to A.

Assigning the value of a literal to a variable
Assignments are used to assign a literal to variables.

The instruction
C := 25 ;

is used, for example, to assign the value 25 to the variable C.

Assigning the value of an operation to a variable
Assignments are used to assign a value which is the result of an operation.

The instruction
X := (A+B-C)*D ;

is used, for example, to assign the result of the operation (A+B-C)*D to the variable X.

Assigning the value of an FFB to a variable
Assignments are used to assign a value returned by a function or a function block to a variable.

The instruction
B := MOD(C,A) ;

is used, for example, to call the MOD (Modulo) function and assign the result of the calculation to the variable B.
The instruction
A := MY_TON.Q ;

is used, for example, to assign the value of the Q output of the MY_TON function block (instance of the TON function block) to the variable A. (This is not a function block call)

Multiple assignments are a supplement to IEC 61131-3 and must be enabled explicitly.

Even after being enabled, multiple assignments are NOT allowed in the following cases:
- in the parameter list for a function block call
- in the element list to initialize structured variables

The instruction
X := Y := Z

is allowed.

The instructions
FB(in1 := 1, In2 := In3 := 2) ;
and
strucVar := (comp1 := 1, comp2 := comp3 := 2) ;

are not allowed.

Assignments between arrays and WORD/DWORD variables are only possible if a type conversion has previously been carried out, e.g.:

%Q3.0:16 := INT_TO_AR_BOOL(%MW20) ;

The following conversion functions are available (General Library), family Array):
- MOVE_BOOL_ARBOOL
- MOVE_WORD_ARWORD
- MOVE_DWORD_ARDWORD
- MOVE_INT_ARINT
- MOVE_DINT_ARDINT
- MOVE_REAL_ARREAL
Select Instruction IF...THEN...END_IF

Description

The IF instruction determines that an instruction or a group of instructions will only be executed if its related Boolean expression has the value 1 (true). If the condition is 0 (false), the instruction or the instruction group will not be executed.

The THEN instruction identifies the end of the condition and the beginning of the instruction(s).

The END_IF instruction marks the end of the instruction(s).

Example

IF...THEN...END_IF

The condition can be expressed using a Boolean variable.
If FLAG is 1, the instructions will be executed; if FLAG is 0, they will not be executed.

IF FLAG THEN
C:=SIN(A) * COS(B) ;
B:=C - A ;
END_IF ;

The condition can be expressed using an operation that returns a Boolean result.
If A is greater than B, the instructions will be executed; if A is less than or equal to B, they will not be executed.

IF A>B THEN
C:=SIN(A) * COS(B) ;
B:=C - A ;
END_IF ;

The condition can be inverted using NOT (execution of both instructions at 0).

IF NOT FLAG THEN
C:=SIN_REAL(A) * COS_REAL(B) ;
B:=C - A ;
END_IF ;

See also

ELSE (See Select Instruction ELSE, p. 481)
ELSIF (See Select Instruction ELSIF...THEN, p. 482)
Select Instruction ELSE

Description
The \texttt{ELSE} instruction always comes after an \texttt{IF...THEN, ELSIF...THEN} or \texttt{CASE} instruction.

If the \texttt{ELSE} instruction comes after an \texttt{IF} or \texttt{ELSIF} instruction, the instruction or group of instructions will only be executed if the associated Boolean expressions of the \texttt{IF} and \texttt{ELSIF} instruction are 0 (false). If the condition of the \texttt{IF} or \texttt{ELSIF} instruction is 1 (true), the instruction or group of instructions will not be executed.

If the \texttt{ELSE} instruction comes after \texttt{CASE}, the instruction or group of instructions will only be executed if no tag contains the value of the selector. If an identification contains the value of the selector, the instruction or group of instructions will not be executed.

\textbf{Note:} Any number of \texttt{IF...THEN...ELSE...END_IF} instructions may be nested to generate complex selection instructions.

Example ELSE
\begin{verbatim}
IF A>B THEN
  C:=SIN(A) * COS(B) ;
  B:=C - A ;
ELSE
  C:=A + B ;
  B:=C * A ;
END_IF ;
\end{verbatim}

See also
IF (See Select Instruction IF...THEN...END_IF, p. 480)
ELSIF (See Select Instruction ELSIF...THEN, p. 482)
CASE (See Select Instruction CASE...OF...END_CASE, p. 483)
Select Instruction ELSIF...THEN

Description

The ELSE instruction always comes after an IF...THEN instruction. The ELSIF instruction determines that an instruction or group of instructions is only executed if the associated Boolean expression for the IF instruction has the value 0 (false) and the associated Boolean expression of the ELSIF instruction has the value 1 (true). If the condition of the IF instruction is 1 (true) or the condition of the ELSIF instruction is 0 (false), the command or group of commands will not be executed.

The THEN instruction identifies the end of the ELSIF condition(s) and the beginning of the instruction(s).

Note: Any number of IF...THEN...ELSIF...THEN...END_IF instructions may be nested to generate complex selection instructions.

Example

ELSIF...THEN

IF A>B THEN
    C:=SIN(A) * COS(B) ;
    B:=SUB(C,A) ;
ELSIF A=B THEN
    C:=ADD(A,B) ;
    B:=MUL(C,A) ;
END_IF ;

For example nested instructions

IF A>B THEN
    IF B=C THEN
        C:=SIN(A) * COS(B) ;
    ELSE
        B:=SUB(C,A) ;
    END_IF ;
ELSIF A=B THEN
    C:=ADD(A,B) ;
    B:=MUL(C,A) ;
ELSE
    C:=DIV(A,B) ;
END_IF ;

See also

IF (See Select Instruction IF...THEN...END_IF, p. 480)
ELSE (See Select Instruction ELSE, p. 481)
Select Instruction CASE...OF...END_CASE

Description

The CASE instruction consists of an INT data type expression (the “selector”) and a list of instruction groups. Each group is provided with a tag which consists of one or several whole numbers (INT, DINT, UINT, UDINT) or ranges of whole number values. The first group is executed by instructions, whose tag contains the calculated value of the selector. Otherwise none of the instructions will be executed.

The OF instruction indicates the start of the tag.

An ELSE instruction may be carried out within the CASE instruction, whose instructions are executed if no tag contains the selector value.

The END_CASE instruction marks the end of the instruction(s).

Example

CASE...OF...END_CASE

Selector

\ CASE SELECT OF
   1,5: C:=SIN(A) * COS(B) ;
   2:  B:=C - A ;
   6..10: C:=C * A ;
ELSE
   B:=C * A ;
   C:=A / B ;
END_CASE ;

See also

ELSE (See Select Instruction ELSE, p. 481)
Repeat Instruction FOR...TO...BY...DO...END_FOR

**Description**
The **FOR** instruction is used when the number of occurrences can be determined in advance. Otherwise **WHILE** (See Repeat Instruction WHILE...DO...END_WHILE, p. 486) or **REPEAT** (See Repeat Instruction REPEAT...UNTIL...END_REPEAT, p. 487) are used.

The **FOR** instruction repeats an instruction sequence until the **END_FOR** instruction. The number of occurrences is determined by start value, end value and control variable.

The control variable, initial value and end value must be of the same data type (**DINT** or **INT**).

The control variable, initial value and end value can be changed by a repeated instruction. This is a supplement to IEC 61131-3.

The **FOR** instruction increments the control variable value of one start value to an end value. The increment value has the default value 1. If a different value is to be used, it is possible to specify an explicit increment value (variable or constant). The control variable value is checked before each new loop. If it is outside the start value and end value range, the loop will be left.

Before running the loop for the first time a check is made to determine whether incrementation of the control variables, starting from the initial value, is moving toward the end value. If this is not the case (e.g. initial value \(\leq\) end value and negative increment), the loop will not be processed. The control variable value is not defined outside of the loop.

The **DO** instruction identifies the end of the repeat definition and the beginning of the instruction(s).

The occurrence may be terminated early using the **EXIT**. The **END_FOR** instruction marks the end of the instruction(s).

**Example:** FOR with increment 1

FOR with increment 1

<table>
<thead>
<tr>
<th>Control variable</th>
<th>Start value</th>
<th>End value</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>FOR</strong> i:= 1 TO 50 <strong>DO</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C:= C * COS(B)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>END_FOR</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
FOR with increment not equal to 1

If an increment other than 1 is to be used, it can be defined by BY. The increment, the initial value, the end value and the control variable must be of the same data type (DINT or INT). The criterion for the processing direction (forwards, backwards) is the sign of the BY expression. If this expression is positive, the loop will run forward; if it is negative, the loop will run backward.

Example: Counting forward in two steps

The increment, the initial value, the end value and the control variable must be of the same data type (DINT or INT). The criterion for the processing direction (forwards, backwards) is the sign of the BY expression. If this expression is positive, the loop will run forward; if it is negative, the loop will run backward.

Example: Counting backwards

The loops in the example are run exactly once, as the initial value = end value. In this context it does not matter whether the increment is positive or negative.

Example: Critical loops

If the increment is \( j \) > 0 in the example, the instruction is executed.

If \( j \) < 0, the instructions are not executed because the situation initial value < only allows the end value to be incremented by \( \geq 0 \).

If \( j = 0 \), the instructions are executed and an endless loop is created as the end value will never be reached with an increment of 0.
Repeat Instruction WHILE...DO...END_WHILE

Description
The WHILE instruction has the effect that a sequence of instructions will be executed repeatedly until its related Boolean expression is 0 (false). If the expression is false right from the start, the group of instructions will not be executed at all.

The DO instruction identifies the end of the repeat definition and the beginning of the instruction(s).

The occurrence may be terminated early using the EXIT.

The END_WHILE instruction marks the end of the instruction(s).

In the following cases WHILE may not be used as it can created an endless loop which causes the program to crash:
- WHILE may not be used for synchronization between processes, e.g. as a "Waiting Loop" with an externally defined end condition.
- WHILE may not be used in an algorithm, as the completion of the loop end condition or execution of an EXIT instruction can not be guaranteed.

Example

```plaintext
x := 1
WHILE x <= 100 DO
  x := x + 4;
END_WHILE;
```

See also
EXIT (See Repeat Instruction EXIT, p. 488)
Repeat Instruction REPEAT...UNTIL...END_REPEAT

Description

The REPEAT instruction has the effect that a sequence of instructions is executed repeatedly (at least once), until its related Boolean condition is 1 (true).

The UNTIL instruction marks the end condition.

The occurrence may be terminated early using the EXIT.

The END_REPEAT instruction marks the end of the instruction(s).

In the following cases REPEAT may not be used as it can create an endless loop which causes the program to crash:

- REPEAT may not be used for synchronization between processes, e.g. as a "Waiting Loop" with an externally defined end condition.
- REPEAT may not be used in an algorithm, as the completion of the loop end condition or execution of an EXIT instruction cannot be guaranteed.

Example

```
x := -1
REPEAT
  x := x + 2
UNTIL x >= 101
END_REPEAT;
```

See also

EXIT (See Repeat Instruction EXIT, p. 488)
Repeat Instruction EXIT

Description

The **EXIT** instruction is used to terminate repeat instructions (**FOR**, **WHILE**, **REPEAT**) before the end condition has been met.

If the **EXIT** instruction is within a nested repetition, the innermost loop (in which **EXIT** is situated) is left. Next, the first instruction following the loop end (**END_FOR**, **END_WHILE** or **END_REPEAT**) is executed.

Example **EXIT**

If **FLAG** has the value 0, **SUM** will be 15 following the execution of the instructions.

If **FLAG** has the value 1, **SUM** will be 6 following the execution of the instructions.

```plaintext
SUM := 0 ;
FOR I := 1 TO 3 DO
    FOR J := 1 TO 2 DO
        IF FLAG=1 THEN EXIT;
        END_IF ;
        SUM := SUM + J ;
    END_FOR ;
    SUM := SUM + I ;
END_FOR
```

See also

**CASE** (See Select Instruction **CASE**...**OF**...**END_CASE**, p. 483)

**WHILE** (See Repeat Instruction **WHILE**...**DO**...**END_WHILE**, p. 486)

**REPEAT** (See Repeat Instruction **REPEAT**...**UNTIL**...**END_REPEAT**, p. 487)
Subroutine Call

**Subroutine call**

A subroutine call consists of the name of the subroutine section followed by an empty parameter list.

Subroutine calls do not return a value.

The subroutine to be called must be located in the same task as the ST section called.

Subroutines can also be called from within subroutines.

For example:

```plaintext
SubroutineName () ;
```

Subroutine calls are a supplement to IEC 61131-3 and must be enabled explicitly.

In SFC action sections, subroutine calls are only allowed when Multitoken Operation is enabled.
RETURN

Description

Every subroutine and every DFB (derived function block) is exited after being processed, i.e. it returns to the main program called.

If the subroutine/DFB is left prematurely, the return to the main program can be forced using **RETURN**.

**RETURN** can only be used in subroutines or DFBs. They cannot be used in the main program.
Empty Instruction

Description
A single semicolon ; represents an empty instruction.

For example,

IF x THEN ; ELSE ..

In this example, an empty instruction follows the THEN instruction. This means that the program exits the IF instruction as soon as the IF condition is 1.
Structured Text ST

Labels and Jumps

Introduction
Labels serve as destinations for jumps.

Jumps and labels in ST are a supplement to the IEC 61131-3 and must be enabled explicitly.

Label properties
Label properties:
- Labels must always be the first element in a line.
- Labels may only come before instructions of the first order (not in loops).
- The name must be clear throughout the directory, and it is not upper/lower case sensitive.
- Labels can be 32 characters long (max.).
- Labels must conform to the general naming conventions.
- Labels are separated by a colon : from the following instruction.

Properties of jumps
- Jumps can be made within program and DFB sections.
- Jumps are only possible in the current section.

Example
IF var1 THEN
    JMP START; 
    :
    :
START: ...
In the ST editor, comments always start with the string \( * \) and end in the string \( * \). Any comments can be entered between these character strings. Comments can be entered in any position in the ST editor, except in keywords, literals, identifiers and variables.

Nesting comments is not permitted according to IEC 61131-3. If comments are nested nevertheless, then they must be enabled explicitly.
15.3 Calling Elementary Functions, Elementary Function Blocks, Derived Function Blocks and Procedures

Introduction

Overview
Calling Elementary Functions, Elementary Function Blocks, Derived Function Blocks and Procedures in the ST programming language.

What's in this Section?
This section contains the following topics:

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Calling Elementary Functions</td>
<td>495</td>
</tr>
<tr>
<td>Call Elementary Function Block and Derived Function Block</td>
<td>501</td>
</tr>
<tr>
<td>Procedures</td>
<td>509</td>
</tr>
</tbody>
</table>
Calling Elementary Functions

Elementary Functions

Elementary functions are provided in the form of libraries. The logic of the functions is created in the programming language C and may not be modified in the ST editor. Functions have no internal states. If the input values are the same, the value at the output is the same for all executions of the function. For example, the addition of two values gives the same result at every execution.

Some elementary functions can be extended to more than 2 inputs.

Elementary functions only have one return value (Output).

Parameters

"Inputs" and one "output" are required to transfer values to or from a function. These are called formal parameters.

The current process states are transferred to the formal parameters. These are called actual parameters.

The following can be used as actual parameters for function inputs:

- Variable
- Address
- Literal
- ST Expression

The following can be used as actual parameters for function outputs:

- Variable
- Address

The data type of the actual parameters must match the data type of the formal parameters. The only exceptions are generic formal parameters whose data type is determined by the actual parameter.

When dealing with generic ANY_BIT formal parameters, actual parameters of the INT or DINT (not UINT and UDINT) data types can be used.

This is a supplement to IEC 61131-3 and must be enabled explicitly.

Example:

Allowed:

AND (AnyBitParam := IntVar1, AnyBitParam2 := IntVar2);

Not allowed:

AND_WORD (WordParam1 := IntVar1, WordParam2 := IntVar2);

(In this case, AND_INT must be used.)

AND_ARRAY_WORD (ArrayInt, ...);
(In this case an explicit type conversion must be carried out using
INT_ARR_TO_WORD_ARR (...);

Not all formal parameters must be assigned with a value. You can see which formal
parameter types must be assigned with a value in the following table.

<table>
<thead>
<tr>
<th>Parameter type</th>
<th>EDT</th>
<th>STRING</th>
<th>ARRAY</th>
<th>ANY_ARRAY</th>
<th>IODDT</th>
<th>STRUCT</th>
<th>FB</th>
<th>ANY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>VAR_IN_OUT</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>/</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>Output</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>/</td>
<td>-</td>
</tr>
</tbody>
</table>

+ Actual parameter required
- Actual parameter not required
/ not applicable

If no value is allocated to a formal parameter, then the initial value will be used for
executing the function block. If no initial value has been defined then the default
value (0) is used.

Programming notes

Attention should be paid to the following programming notes:

- All generic functions are overloaded. This means the functions can be called with
  or without entering the data type.
  E.g.
  \[ i1 := \text{ADD (i2, 3)}; \]
  is identical to
  \[ i1 := \text{ADD_INT (i2, 3)}; \]
- Functions can be nested (see also Nesting functions, p. 499).
- Functions are only executed if the input \( \text{EN}=1 \) or the \( \text{EN} \) input is not used (see also
  \( \text{EN} \) and \( \text{ENO} \), p. 499).
- There are two ways of calling a function:
  - Formal call (calling a function with formal parameter names)
  - Informal call (calling a function without formal parameter names)

Formal call

With formal calls (calls with formal parameter names), the call consists of the actual
parameter of the output, followed by the assignment instruction :=, then the function
name and then by a bracketed list of value assignments (actual parameters) to the
formal parameter. The order in which the formal parameters are enumerated in a
function call is not significant.

\( \text{EN} \) and \( \text{ENO} \) can be used for this type of call.
Calling a function with formal parameter names:

```
Actual Parameter (Output)    Formal Parameters
out:=LIMIT (MN:=0, IN:=var1, MX:=5 + var) ;
```

Function Name

Actual Parameters (Inputs)

Calling the same function in FBD:

With formal calls it is not necessary to assign a value to all formal parameters (see also Parameters, p. 495).

```
out:=LIMIT (MN:=0, IN:=var1) ;
```

Calling the same function in FBD:
Informal call

With informal calls (calls without formal parameter names), the call consists of the actual parameter of the output, followed by the symbol of the assignment instruction :=, then the function name and then by a bracketed list of the inputs actual parameters. The order that the actual parameters are enumerated in a function call is significant.

EN and ENO cannot be used for this type of call.

Calling a function without formal parameter names:

Actual Parameter (Output)

\[ \text{out} := \text{LIMIT} (0, \text{var1}, 5 + \text{var}) ; \]

Calling the same function in FBD:

An empty parameter field is used to skip a parameter.

Call with empty parameter field:

\[ \text{out} := \text{LIMIT} (, \text{var1}, 5 + \text{var}) ; \]

Calling the same function in FBD:

An empty parameter field does not have to be used if formal parameters are omitted at the end.

\[ \text{out} := \text{LIMIT} (0, \text{var1}) ; \]

Calling the same function in FBD:
Nesting functions

A function call can include the call of further functions. The nesting depth is not limited.

Nested call of array function:

\[
\text{out} := \text{LIMIT} \left( \text{MN}:=4, \text{IN}:=\text{MUL} \left( \text{IN1}:=\text{var1}, \text{IN2}:=\text{var2} \right), \text{MX}:=5 \right) ;
\]

Calling the same function in FBD:

\[
\begin{array}{c}
\text{var1} \\
\text{var2}
\end{array} 
\begin{array}{c}
\text{MUL} \\
\text{IN1} \\
\text{IN2}
\end{array} 
\begin{array}{c}
\text{OUT} \\
\text{MN} \\
\text{IN} \\
\text{MX} \\
\text{out}
\end{array}
\]

Functions that return a value of the ANY_ARRAY data type can not be used within a function call.

Unauthorized nesting with ANY_ARRAY:

\[
\text{out} := \text{LIMIT} \left( \text{MN}:=4, \text{IN}:=\text{EXAMP} \left( \text{IN1}:=\text{var1}, \text{IN2}:=\text{var2} \right), \text{MX}:=5 \right) ;
\]

ANY_ARRAY is permitted as the return value of the function called or as a parameter of the nested functions.

Authorized nesting with ANY_ARRAY:

\[
\begin{array}{c}
\text{ANY_ARRAY} \\
\text{MN} \\
\text{IN} \\
\text{MX} \\
\text{ENO}
\end{array} 
\begin{array}{c}
\text{OUT} \\
\text{MN} \\
\text{IN} \\
\text{MX}
\end{array} 
\begin{array}{c}
\text{ANY_ARRAY} \\
\text{OUT}
\end{array}
\]

EN and ENO

With all functions an EN input and an ENO output can be configured.

If the value of EN is equal to "0", when the function is called, the algorithms defined by the function are not executed and ENO is set to "0".

If the value of EN is equal to "1", when the function is called, the algorithms which are defined by the function are executed. After successful execution of these algorithms, the value of ENO is set to "1". If an error occurs during execution of these algorithms, ENO will be set to "0".

If ENO is set to "0" (caused when EN=0 or an error occurred during executing), the output of the function is set to "0".

The output behavior of the function does not depend on whether the function was called up without EN/ENO or with EN=1.

If EN/ENO are used, the function call must be formal.

\[
\text{out} := \text{LIMIT} \left( \text{EN}:=1, \text{MN}:=0, \text{IN}:=\text{var1}, \text{MX}:=5, \text{ENO}=>\text{var2} \right) ;
\]
Calling the same function in FBD:

```
LIMIT

1    EN
0    MN
var1 IN
5    MX

ENO    var2
out
```
Call Elementary Function Block and Derived Function Block

**Elementary Function Block**
Elementary function blocks have internal states. If the inputs have the same values, the value on the output can have another value during the individual operations. For example, with a counter, the value on the output is incremented.
Function blocks can have several output values (outputs).

**Derived Function Block**
Derived function blocks (DFBs) have the same characteristics as elementary function blocks. The user can create them in the programming languages FBD, LD, IL, and/or ST.

**Parameter**
"Inputs and outputs" are required to transfer values to or from function blocks. These are called formal parameters.
The current process states are transferred to the formal parameters. They are called actual parameters.
The following can be used as actual parameters for function block inputs:
- Variable
- Address
- Literal

The following can be used as actual parameters for function block outputs:
- Variable
- Address

The data type of the actual parameters must match the data type of the formal parameters. The only exceptions are generic formal parameters whose data type is determined by the actual parameter.

When dealing with generic \texttt{ANY_BIT} formal parameters, actual parameters of the \texttt{INT} or \texttt{DINT} (not \texttt{UINT} and \texttt{UDINT}) data types can be used.
This is a supplement to IEC 61131-3 and must be enabled explicitly.

Example:

**Allowed:**
\begin{verbatim}
AND (AnyBitParam := IntVar1, AnyBitParam2 := IntVar2);
\end{verbatim}

**Not allowed:**
\begin{verbatim}
AND_WORD (WordParam1 := IntVar1, WordParam2 := IntVar2);
\end{verbatim}
(In this case, \texttt{AND_INT} must be used.)
\begin{verbatim}
AND_ARRAY_WORD (ArrayInt, ...);
\end{verbatim}
(In this case an explicit type conversion must be carried out using \texttt{INT_ARR_TO_WORD_ARR (...);}.)
Not all formal parameters must be assigned with a value. Which formal parameter types must be assigned a value can be seen in the following table.

<table>
<thead>
<tr>
<th>Parameter type</th>
<th>EDT</th>
<th>STRING</th>
<th>ARRAY</th>
<th>ANY_ARRAY</th>
<th>IODDT</th>
<th>STRUCT</th>
<th>FB</th>
<th>ANY</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFB: Input</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>/</td>
<td>+</td>
<td>/</td>
<td>+</td>
</tr>
<tr>
<td>EFB: VAR_IN_OUT</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>/</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>EFB: Output</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>/</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>DFB: Input</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>/</td>
<td>+</td>
<td>/</td>
<td>+</td>
</tr>
<tr>
<td>DFB: VAR_IN_OUT</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>/</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>DFB: Output</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>/</td>
<td>/</td>
<td>-</td>
<td>/</td>
<td>+</td>
</tr>
</tbody>
</table>

+  Actual parameter required
-  Actual parameter not required
/  not applicable

If no value is allocated to a formal parameter, then the initial value will be used for executing the function block. If no initial value has been defined then the default value (0) is used.

If a formal parameter is not assigned with a value and the function block/DFB is instanced more than once, then the following instances are run with the old value.

**Public Variables**

In addition to "inputs/outputs", some function blocks also provide public variables.

These variables transfer statistical values (values that are not influenced by the process) to the function block. They are used for setting parameters for the function block.

Public variables are a supplement to IEC 61131-3.

The assignment of values to public variables is made via their initial values or assignments.

**Example:**

```plaintext
D_ACT1.OP_CTRL := 1 ; (D_ACT1 is an instance of the function block D_ACT and has the public variables AREA_NR and OP_CTRL.
```

Public variables are read via the instance name of the function block and the names of the public variables.
Example:

```
Var1 := D_ACT1.OP_CTRL;
```

**Programming notes**

Attention should be paid to the following programming notes:

- Functions blocks are only executed if the input \( EN=1 \) or the \( EN \) input is not used (see also **EN and ENO, p. 506**).
- The assignment of variables to \( \text{ANY} \) or \( \text{ARRAY} \) output types must be made using the \( \Rightarrow \) operator (see also **Formal call, p. 503**). Assignments cannot be made outside of the function block call.

  The instruction

  ```
  My_Var := My_SAH.OUT;
  ```

  is **invalid**, if the output \( \text{OUT} \) of the \( \text{SAH} \) function block is of type \( \text{ANY} \).

  The instruction

  ```
  Cal My_SAH (OUT=>My_Var);
  ```

  is **valid**.

- Special conditions apply when using \( \text{VAR_IN_OUT} \) variables (See **VAR_IN_OUT-Variable, p. 507**).
- The use of function blocks consists of two parts in ST:
  - the Declaration (See **Declaration, p. 503**)
  - calling the function block,

- There are two ways of calling a function block:
  - Formal call (See **Formal call, p. 503**) (calling a function with formal parameter names)
    
    This way variables can be assigned to outputs using the \( \Rightarrow \) operator.
  - Informal call (See **Informal call, p. 504**) (call without formal parameter names)
  - Function block/DFB instances can be called multiple times; other than instances of communication EFBs, these can only be called once (see **Multiple Function Block Instance Call, p. 506**).

**Declaration**

Before calling a function block it must be declared in the variables editor.

**Formal call**

With formal calls (call with formal parameter names), the function block is called using an instruction sequence made from the function blocks instance names that follows a bracketed list of actual parameter assignments to the formal parameters.

Assign input formal parameters via \( := \) Assignment and the assignment of the input formal parameter using the \( := \) assignment. The sequence in which the input formal parameters and output formal parameters are enumerated is **not significant**.

**EN** and **ENO** can be used for this type of call.
Calling a function block with formal parameter names:

Instance Names

Formal Parameters (Inputs)

MY_COUNT (CU:=var1, R:=reset, PV:=100 + value, Q=>out, CV=>current);

Actual Parameters (Inputs)

Calling the same function block in FBD:

Assigning the value of a function block output is made by entering the actual parameter name, followed by the assignment instruction := followed by the instance name of the function block and loading the formal parameter of the function block output (separated by a full-stop).

E.g.

MY_COUNT (CU:=var1, R:=reset, PV:=100 + value);
Q := MY_COUNT.out;
CV := MY_COUNT.current;

It is not necessary to assign a value to all formal parameters (see also Parameter, p. 501).

MY_COUNT (CU:=var1, R:=reset, Q=>out, CV=>current);

Calling the same function block in FBD:

Informal call

With informal calls (call without Formal parameter names), the function block is called using an instruction made from the function block instance names, followed by a bracketed list of the actual parameters for the inputs and outputs. The order in which the actual parameters are listed in a function block call is significant.

EN and ENO cannot be used for this type of call.
Calling a function block without formal parameter names:

Instance Name Actual Parameter (Input)

MY_COUNT (var1, reset, 100+value, out, current) ;

Actual Parameter (Output)

Calling the same function block in FBD:

An empty parameter field is used to skip a parameter.

Call with empty parameter field:
MY_COUNT (var1, , 100 + value, out, current) ;

Calling the same function block in FBD:

An empty parameter field does not have to be used if formal parameters are omitted at the end.
MY_COUNT (var1, reset) ;

Calling the same function block in FBD:
Calling a function block without inputs

Even if the function block has no inputs or the inputs are not to be parameterized, the function block should be called before its outputs can be used. Otherwise the initial values of the outputs will be transferred, i.e. "0".

E.g.

Calling the function block in ST:

```
MY_CLOCK () ;
MY_COUNT (CU:=MY_CLOCK.CLK1, R:=reset, PV:=100,
          Q=>out, CV=>current) ;
```

Calling the same function block in FBD:

<table>
<thead>
<tr>
<th>MY_CLOCK</th>
<th>MY_COUNT</th>
</tr>
</thead>
<tbody>
<tr>
<td>MY_CLOCK</td>
<td>MY_CLOCK</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SYSCLOCK</th>
<th>CTU</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLK1</td>
<td>CU</td>
</tr>
<tr>
<td>CLK2</td>
<td>reset</td>
</tr>
<tr>
<td>CLK3</td>
<td>R</td>
</tr>
<tr>
<td>CLK4</td>
<td>100</td>
</tr>
<tr>
<td>CLK5</td>
<td>PV</td>
</tr>
<tr>
<td>TIMER</td>
<td>CV</td>
</tr>
</tbody>
</table>

Multiple Function Block Instance Call

Function block/DFB instances can be called multiple times; other than instances of communication EFBs, these can only be called once.

Calling the same function block/DFB instance more than once makes sense, for example, in the following cases:

- If the function block/DFB has no internal value or it is not required for further processing.
  
  In this case, memory is saved by calling the same function block/DFB instance more than once since the code for the function block/DFB is only loaded once. The function block/DFB is then handled like a "Function".

- If the function block/DFB has an internal value and this is supposed to influence various program segments, for example, the value of a counter should be increased in different parts of the program.

  In this case, calling the same function block/DFB means that temporary results do not have to be saved for further processing in another part of the program.

EN and ENO

With all function blocks/DFBs, an EN input and an ENO output can be configured.

If the value of EN is equal to "0", when the function block/DFB is called, the algorithms defined by the function block/DFB are not executed and ENO is set to "0".

If the value of EN is equal to "1", when the function block/DFB is invoked, the algorithms which are defined by the function block/DFB will be executed. After the algorithms have been executed successfully, the value of ENO is set to "1". If an error occurred while executing the algorithms, ENO is set to "0".
If ENO is set to “0” (results from EN=0 or an error during execution), the outputs of the function block/DFB retain the status from the last cycle in which they were correctly executed.

The output behavior of the function blocks/DFBs does not depend on whether the function blocks/DFBs are called without EN/ENO or with EN=1.

If EN/ENO are used, the function block call must be formal. The assignment of variables to ENO must be made using the => operator.

MY_COUNT (EN:=1, CU:=var1, R:=reset, PV:=100 + value, ENO=>error, Q=>out, CV=>current) ;

Calling the same function block in FBD:

### VAR_IN_OUT-Variable

Function blocks are often used to read a variable at an input (input variables), to process it and to restate the altered values of the same variable (output variables). This special type of input/output variable is also called a VAR_IN_OUT variable.

The following special features are to be noted when using function blocks/DFBs with VAR_IN_OUT variables.

- All VAR_IN_OUT inputs must be assigned a variable.
- VAR_IN_OUT inputs may not have literals or constants assigned to them.
- VAR_IN_OUT outputs may not have values assigned to them.
- VAR_IN_OUT variables cannot be used outside of the function block call.

Calling a function block with VAR_IN_OUT variable in ST:

MY_FBLOCK(IN1:=V1, IN2:=V2, IO1:=V3, OUT1=>V4, OUT2=>V5);

Calling the same function block in FBD:

VAR_IN_OUT variables cannot be used outside the function block call.

The following function block calls are therefore invalid:
### Invalid call, example 1:

| InOutFB.inout := V1; | Assigning the variables V1 to a VAR_IN_OUT parameter. Error: The operation cannot be executed since the VAR_IN_OUT parameter cannot be accessed outside of the function block call. |

### Invalid call, example 2:

| V1 := InOutFB.inout; | Assigning a VAR_IN_OUT parameter to the V1 variable. Error: The operation cannot be executed since the VAR_IN_OUT parameter cannot be accessed outside of the function block call. |

### The following function block calls are always valid:

### Valid call, example 1:

| InOutFB (inout:=V1); | Calling a function block with the VAR_IN_OUT parameter and formal assignment of the actual parameter within the function block call. |

### Valid call, example 2:

| InOutFB (V1); | Calling a function block with the VAR_IN_OUT parameter and informal assignment of the actual parameter within the function block call. |
Procedures

Procedure

Procedures are provided in the form of libraries. The logic of the procedure is created in the programming language C and may not be modified in the ST editor. Procedures - like functions - have no internal states. If the input values are the same, the value on the output is the same for all executions of the procedure. For example, the addition of two values gives the same result at every execution.

In contrast to functions, procedures do not return a value and support VAR_IN_OUT variables.

Procedures are a supplement to IEC 61131-3 and must be enabled explicitly.

Parameter

"Inputs and outputs" are required to transfer values to or from procedures. These are called formal parameters.

The current process states are transferred to the formal parameters. These are called actual parameters.

The following can be used as actual parameters for procedure inputs:
- Variable
- Address
- Literal
- ST Expression

The following can be used as actual parameters for procedure outputs:
- Variable
- Address

The data type of the actual parameters must match the data type of the formal parameters. The only exceptions are generic formal parameters whose data type is determined by the actual parameter.

When dealing with generic ANY_BIT formal parameters, actual parameters of the INT or DINT (not UINT and UDINT) data types can be used.

This is a supplement to IEC 61131-3 and must be enabled explicitly.

Example:

Allowed:

AND (AnyBitParam := IntVar1, AnyBitParam2 := IntVar2);

Not allowed:

AND_WORD (WordParam1 := IntVar1, WordParam2 := IntVar2);

(In this case, AND_INT must be used.)

AND_ARRAY_WORD (ArrayInt, ...);
In this case an explicit type conversion must be carried out using
\[ \text{INT\_ARR\_TO\_WORD\_ARR} \ldots \].

Not all formal parameters must be assigned with a value. You can see which formal parameter types must be assigned with a value in the following table.

<table>
<thead>
<tr>
<th>Parameter type</th>
<th>EDT</th>
<th>STRING</th>
<th>ARRAY</th>
<th>ANY_ARRAY</th>
<th>IODDT</th>
<th>STRUCT</th>
<th>FB</th>
<th>ANY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>VAR_IN_OUT</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>/</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Output</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>/</td>
<td>+</td>
<td>-</td>
</tr>
</tbody>
</table>

+ Actual parameter required
- Actual parameter not required
/ not applicable

If no value is allocated to a formal parameter, then the initial value will be used for executing the function block. If no initial value has been defined then the default value (0) is used.

**Programming notes**

Attention should be paid to the following programming notes:

- Procedures are only executed if the input \( \text{EN}=1 \) or the \( \text{EN} \) input is not used (see also \( \text{EN} \) and \( \text{ENO} \), p. 512).
- Special conditions apply when using VAR\_IN\_OUT variables (See \( \text{VAR\_IN\_OUT variable} \), p. 513).
- There are two ways of calling a procedure:
  - Formal call (See Formal call, p. 510) (calling a function with formal parameter names)
    This way variables can be assigned to outputs using the \( \text{=} \rightarrow \) operator.
  - Informal call (See Informal call, p. 511) (call without formal parameter names)

**Formal call**

With formal calls (call with formal parameter names), the procedures are called using an instruction sequence made from the procedure name, followed by a bracketed list of actual parameter assignments to the formal parameters. The assignment of the input formal parameter is made using the \( \text{=} \) assignment and the output formal parameter is made using the \( \text{=} \rightarrow \) assignment. The sequence in which the input formal parameters and output formal parameters are enumerated is not significant.

\( \text{EN} \) and \( \text{ENO} \) can be used for this type of call.
Calling a procedure with formal parameter names:

**Procedure Name**

Formal Parameters (Inputs)  Formal Parameters (Outputs)

PROC (IN1:=var1, IN2:=var2, OUT1=>result1, OUT2=>result2);

Calling the same procedure in FBD:

Actual Parameters (Inputs)  Actual Parameters (Outputs)

With formal calls it is not necessary to assign a value to all formal parameters (see also Parameter, p. 509).

PROC (IN1:=var1, OUT1=>result1, OUT2=>result2);

Calling the same procedure in FBD:

Informal call

With informal calls (call without formal parameter names), procedures are called using an instruction made from the procedure name, followed by a bracketed list of the inputs and outputs actual parameters. The order that the actual parameters are enumerated in a procedure call is significant. EN and ENO cannot be used for this type of call.

**Calling a procedure without formal parameter names:**

**Procedure Name**  **Actual Parameters (Inputs)**

PROC (var1, var2, result1, result2);

**Calling the same procedure in FBD:**

Actual Parameter (Output)

With informal calls it is not necessary to assign a value to all formal parameters (see also Parameter, p. 509).
This is a supplement to IEC 61131-3 and must be enabled explicitly. An empty parameter field is used to skip a parameter.

Call with empty parameter field:

```
PROC (var1, , result1, result2);
```

**Calling the same procedure in FBD:**

![FBD diagram]

An empty parameter field does not have to be used if formal parameters are omitted at the end.

```
PROC (var1, var2, result1);
```

**Calling the same procedure in FBD:**

![FBD diagram]

---

**EN and ENO**

With all procedures, an `EN` input and an `ENO` output can be configured.

If the value of `EN` is equal to "0", when the procedure is called, the algorithms defined by the procedure are not executed and `ENO` is set to "0".

If the value of `EN` is "1" when the procedure is called, the algorithms defined by the function are executed. After successful execution of these algorithms, the value of `ENO` is set to "1". If an error occurs during execution of these algorithms, `ENO` will be set to "0".

If `ENO` is set to "0" (caused when `EN=0` or an error occurred during executing), the outputs of the procedure are set to "0".

The output behavior of the procedure does not depend on whether the function is called without `EN` or with `EN=1`.

If `EN/ENO` are used, the procedure call must be formal. The assignment of variables to `ENO` must be made using the `=>` operator.

```
PROC (EN:=1, IN1:=var1, IN2:=var2, ENO=>error, OUT1=>result1, OUT2=>result2);
```

**Calling the same procedure in FBD:**

![FBD diagram]
**VAR_IN_OUT variable**

Procedures are often used to read a variable at an input (input variables), to process it and to restate the altered values of the same variable (output variables). This special type of input/output variable is also called a VAR_IN_OUT variable.

The following special features are to be noted when using procedures with VAR_IN_OUT variables.

- All VAR_IN_OUT inputs must be assigned a variable.
- VAR_IN_OUT inputs may not have literals or constants assigned to them.
- VAR_IN_OUT outputs may not have values assigned to them.
- VAR_IN_OUT variables cannot be used outside of the procedure call.

**Calling a procedure with VAR_IN_OUT variable in ST:**

```st
PROC2 (IN1:=V1, IN2:=V2, IO1:=V3,
       OUT1=>V4, OUT2=>V5) ;
```

**Calling the same procedure in FBD:**

<table>
<thead>
<tr>
<th>PROC2</th>
</tr>
</thead>
<tbody>
<tr>
<td>V1</td>
</tr>
<tr>
<td>IN1</td>
</tr>
<tr>
<td>V2</td>
</tr>
<tr>
<td>IN2</td>
</tr>
<tr>
<td>V3</td>
</tr>
<tr>
<td>IO1</td>
</tr>
<tr>
<td>V4</td>
</tr>
<tr>
<td>OUT1</td>
</tr>
<tr>
<td>V5</td>
</tr>
<tr>
<td>OUT2</td>
</tr>
</tbody>
</table>

VAR_IN_OUT variables cannot be used outside of the procedure call.

The following procedure calls are therefore **invalid**:

**Invalid call, example 1:**

```st
InOutProc.inout := V1;  
```

Assigning the variables V1 to a VAR_IN_OUT parameter.  
**Error:** The operation cannot be executed since the VAR_IN_OUT parameter cannot be accessed outside of the procedure call.

**Invalid call, example 2:**

```st
V1 :=InOutProc.inout;
```

Assigning a VAR_IN_OUT parameter to the V1 variable.  
**Error:** The operation cannot be executed since the VAR_IN_OUT parameter cannot be accessed outside of the procedure call.

The following procedure calls are always **valid**:

**Valid call, example 1:**

```st
InOutProc (inout:=V1);
```

Calling a procedure with the VAR_IN_OUT parameter and formal assignment of the actual parameter within the procedure call.
**Valid call, example 2:**

| InOutProc (V1); | Calling a procedure with the VAR_IN_OUT parameter and informal assignment of the actual parameter within the procedure call. |
User function blocs (DFB)

At a Glance

In This Chapter

This part presents:
- the user function blocks (DFB)
- the internal structure of DFBs
- Diagnostics DFBs
- the types and instances of DFBs
- instance calls using different languages

What's in this Part?

This part contains the following chapters:

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Chapter Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>Overview of User Function Blocks (DFB)</td>
<td>517</td>
</tr>
<tr>
<td>17</td>
<td>Description of User Function Blocks (DFB)</td>
<td>523</td>
</tr>
<tr>
<td>18</td>
<td>User Function Blocks (DFB) instance</td>
<td>531</td>
</tr>
<tr>
<td>19</td>
<td>Use of the DF Bs from the different programming languages</td>
<td>539</td>
</tr>
<tr>
<td>20</td>
<td>User diagnostics DFB</td>
<td>557</td>
</tr>
</tbody>
</table>
Overview of User Function Blocks (DFB)

At a Glance

Subject of this Chapter
This chapter provides an overview of the user function blocks (DFB), and the different steps in their implementation.

What's in this Chapter?
This chapter contains the following topics:

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction to user function blocks</td>
<td>518</td>
</tr>
<tr>
<td>How to implement a DFB function block</td>
<td>520</td>
</tr>
</tbody>
</table>
**Introduction to user function blocks**

### Introduction

Unity Pro software enables you to create DFB user function blocks, using automation languages. A DFB is a program block that you write to meet the specific requirements of your application. It includes:

- one or more sections written in Ladder (LD), Instruction List (IL), Structured Text (ST) or Functional Block Diagram (FBD) language
- input/output parameters
- public or private internal variables

Function blocks can be used to structure and optimize your application. They can be used whenever a program sequence is repeated several times in your application, or to set a standard programming operation (for example, an algorithm that controls a motor, incorporating local safety requirements).

By exporting then importing these blocks, they can be used by a group of programmers working on a single application or in different applications.

### Benefits of using a DFB

Using a DFB function block in an application enables you to:

- simplify the design and entry of the program
- increase the legibility of the program
- facilitate the debugging of the application (all of the variables handled by the function block are identified on its interface)
- reduce the volume of code generated (the code that corresponds to the DFB is only loaded once - however many calls are made to the DFB in the program, only the data corresponding to the instances are generated)

### Comparison with a subroutine

Compared to a subroutine, using a DFB makes it possible to:

- set processing parameters more easily
- use internal variables that are specific to the DFB and therefore independent from the application
- test its operation independently from the application

Furthermore, LD and FBD languages provide a graphic view of the DFBs, facilitating the design and debugging of your program.

### DFB created with previous software versions

DFBs created using PL7 and Concept must first be converted using the converters that come with the product, before being used in the application.
The following table shows the domain of use for the DFBs.

<table>
<thead>
<tr>
<th>Function</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>PLCs for which DFBs can be used.</td>
<td>Premium/Atrium and Quantum</td>
</tr>
<tr>
<td>DFB creation software</td>
<td>Unity Pro</td>
</tr>
<tr>
<td>Software with which DFBs can be used.</td>
<td>Unity Pro or Unity Pro Medium</td>
</tr>
<tr>
<td>Programming language for creating the DFB code.</td>
<td>IL, ST, LD or FBD (1)</td>
</tr>
<tr>
<td>Programming language with which DFBs can be used.</td>
<td>IL, ST, LD or FBD (1)</td>
</tr>
</tbody>
</table>

(1) IL: Instruction List, ST: Structured Text, LD: LaDder, FBD: Functional Block Diagram language.
How to implement a DFB function block

Implementation procedure

There are 3 steps in the DFB function block implementation procedure:

<table>
<thead>
<tr>
<th>Step</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Create your DFB model (called: DFB type).</td>
</tr>
<tr>
<td>2</td>
<td>Create a copy of this function block, called an instance, every time the DFB is used in the application.</td>
</tr>
<tr>
<td>3</td>
<td>Use the DFB instances in your application program.</td>
</tr>
</tbody>
</table>

Creation of the DFB type

This operation consists in designing a model of the DFB you want to use in your application. To do this, use the DFB editor to define and code all the elements that make up the DFB:

- Description of the function block: name, type (DFB), activation of diagnostics, comment.
- Structure of the function block: parameters, variables, code sections.

Description of a DFB type

The following diagram shows a graphic representation of a DFB model.

The function block comprises the following elements:

- Name: name of the DFB type (max. 32 characters). This name must be unique in the libraries, the authorized characters used depend on the choice made in the Identifiers area of the Language extensions tab in the Project Settings (See Unity Pro, Operating Modes Manual, Language extensions):
- Inputs: input parameters (excluding input/output parameters).
- Outputs: output parameters (excluding input/output parameters).
- Inputs/Outputs: input/output parameters.
- Public variables: internal variables accessible by the application program.
- Private variables: nested internal variables or DFBs, not accessible by the application program.
- Sections: DFB code sections in LD, IL, ST or FBD.
### Creation of a DFB Instance

Once the DFB type is created, you can define an instance of this DFB via the variable editor or when the function is called in the program editor.

### Use of DFB Instances

A DFB instance is used as follows:
- as a standard function block in Ladder (LD) or Functional Block Diagram (FBD) language,
- as an elementary function in Structured Text (ST) or Instruction List (IL) language.

A DFB instance can be used in all application program tasks, except event tasks and Sequential Function Chart (SFC) transitions.

### Storage

The DFB types the user creates can be stored (See Unity Pro, Operating Modes Manual, Archiving a DFB Type) in the function and function block library.
Overview of DFBs
Description of User Function Blocks (DFB)

At a Glance

Subject of this Chapter
This chapter provides an overview of the different elements that make up the user function blocks.

What's in this Chapter?
This chapter contains the following topics:

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Definition of DBF Function Block Internal Data</td>
<td>524</td>
</tr>
<tr>
<td>DFB Parameters</td>
<td>526</td>
</tr>
<tr>
<td>DFB variables</td>
<td>528</td>
</tr>
<tr>
<td>DFB code section</td>
<td>529</td>
</tr>
</tbody>
</table>
Definition of DBF Function Block Internal Data

At a Glance

There are two types of DFB internal data:

- The parameters: Input, Output or Input/Output,
- Public or Private variables.

The internal data of the DFB must be defined using symbols (this data cannot be addressed as an address).

Elements to Define for Each Parameter

When the function block is created, the following must be defined for each parameter:

- Name: Name of DFB type (max. 32 characters). This name must be unique in the libraries; the authorized characters used depend on the choice made in the Identifiers area of the Language extensions tab in Project Settings (See Unity Pro, Operating Modes Manual, Language extensions):
  - A type of object (BOOL, INT, REAL, etc.),
  - A comment of a maximum of 1024 characters (optional). Formatting characters (carriage return, tab, etc.) are not allowed,
  - An initial value,
  - The read/write attribute that defines whether the variable may or may not be written in runtime: R (read only) or R/W (read/write). This attribute must only be defined for public variables,
  - The backup attribute that defines whether the variable may or may not be saved.

Types of Objects

The types of objects that may be defined for the DFB parameters belong to the following families:

- Elementary data family: EDT. This family includes the following object types: Boolean (BOOL, EBOOL), Integer (INT, DINT, etc.), Real (REAL), Character string (STRING), Bit string (BYTE, WORD, etc.), etc,
- Derived data family: DDT. This family includes table (ARRAY) and structure (user or IODDT) object types,
- Generic data families: ANY_INT, ANY_REAL, ANY_ARRAY, etc. For example:
  - The ANY_INT family includes integer data (INT, DINT, UINT, UDINT),
  - The ANY_DATE family includes temporal data (DATE, DATE_AND_TIME, TIME_OF_DAY),
  - The ANY_STRUCT family includes any structures, etc.
- The function block family: FB. This family includes EFB and DFB object types.
For each of the DFB parameters, the following object families may be used:

<table>
<thead>
<tr>
<th>Object families</th>
<th>EDT</th>
<th>DDT (1)</th>
<th>IODDT</th>
<th>ANY,...(^{(2)})</th>
<th>ANY_ARRAY(_x)(^{(4)})</th>
<th>FB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Inputs/outputs</td>
<td>Yes (3)</td>
<td>Yes</td>
<td>Yes (See Use of IODDTs in a DFB, p. 542)</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Outputs</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Public variables</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Private variables</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Key:

(1) Derived data family, except input/output derived data types (IODDT).
(2) All generic data families, except the ANY_ARRAY family.
(3) Except for EBOOL-type static variables, with Quantum PLCs.
(4) \(x\) being a data type from an elementary data family: EDT.
DFB Parameters

Illustration

This illustration shows some examples of DFB parameters

![Diagram showing DFB parameters]

Description of the Parameters

This table shows the role of each parameter

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Maximum number</th>
<th>Role</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs</td>
<td>32 (1)</td>
<td>These parameters can be used to transfer the values of the application program to the internal program of the DFB. They are accessible in read-only by the DFB, but are not accessible by the application program.</td>
</tr>
<tr>
<td>Outputs</td>
<td>32 (2)</td>
<td>These parameters can be used to transfer the values of the DFB to the application program. They are accessible for reading by the application program except for ARRAY-type parameters.</td>
</tr>
<tr>
<td>Inputs/Outputs</td>
<td>32</td>
<td>These parameters may be used to transfer data from the application program to the DFB, which can then modify it and return it to the application program. These parameters are not accessible by the application program.</td>
</tr>
</tbody>
</table>

Legend:

(1) Number of inputs + Number of inputs/outputs less than or equal to 32
(2) Number of outputs + Number of inputs/outputs less than or equal to 32

Parameters that can be accessed by the application program

The only parameters that can be accessed by the application program outside the call are output parameters. To make this possible, the following syntax must be used in the program: `DFB_Name.Parameter_name`

`DFB_Name` represents the name of the instance of the DFB used (maximum of 32 characters).
Parameter_Name represents the name of the output parameter (maximum 32 characters).

Example: Control.Accel indicates the output Accel of the DFB instance called Control.

EN and ENO parameters

EN is an input parameter, and ENO is an output parameter. They are both of BOOL type, and may or may not be used (optional) in the definition of a DFB type.

Where the user wishes to use these parameters, the editor sets them automatically:
- EN is the first input parameter and ENO the first output parameter.

Example of implementation of EN/ENO parameters.

If the EN input parameter of an instance is assigned the value 0 (FALSE), then:
- the section(s) that make up the code of the DFB is/are not executed (this is managed by the system),
- the ENO output parameter is set to 0 (FALSE) by the system.

If the EN input parameter of an instance is assigned the value 1 (TRUE), then:
- the section(s) that make up the code of the DFB is/are executed (this is managed by the system),
- the ENO output parameter is set to 1 (TRUE) by the system.

If an error is detected (for example a processing error) by the DFB instance, the user has the option of setting the ENO output parameter to 0 (FALSE). In this case:
- either the output parameters are frozen in the state they were in during the previous process until the fault disappears,
- or the user provides a function in the DFB code whereby the outputs are forced to the required state until the fault disappears.
Description of DFBs

DFB variables

This table shows the role of each type of variable.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Maximum number</th>
<th>Role</th>
</tr>
</thead>
<tbody>
<tr>
<td>Public</td>
<td>unlimited</td>
<td>These internal variables of the DFB may be used by the DFB, by the application program and by the user in adjust mode.</td>
</tr>
<tr>
<td>Private</td>
<td>unlimited</td>
<td>These internal variables of the DFB can only be used by this function block, and are therefore not accessible by the application program. These variables are generally necessary to the programming of the block, but are of no interest to the user (for example, the result of an intermediate calculation, etc.).</td>
</tr>
</tbody>
</table>

Variables that can be accessed by the application program

The only variables that can be accessed by the application program are public variables. To make this possible, the following syntax must be used in the program:

DFB_Name.Variable_Name

DFB_Name represents the name of the instance of the DFB used (maximum of 32 characters).

Variable_Name represents the name of the public variable (maximum of 8 characters).

Example: Control.Gain indicates the public variable Gain of the DFB instance called Control

Saving public variables

Setting the %S94 system bit to 1 causes the public variables you have modified to be saved by program or by adjustment, in place of the initial values of these variables (defined in the DFB instances).

Replacement is only possible if the backup attribute is correctly set for the variable.
**DFB code section**

**General**

The code section(s) define(s) the process the DFB is to carry out, as a function of the declared parameters.

If the IEC option is set, a single section may be attached to the DFB. Otherwise, a DFB may contain several code sections; the number of sections being unlimited.

**Programming languages**

To program DFB sections, you can use the following languages:

- Instruction List (IL)
- Structured Text (ST)
- Ladder language (LD)
- Functional Block Diagram (FBD)

**Defining a section**

A section is defined by:

- a symbolic name that identifies the section (maximum of 32 characters)
- a validation condition that defines the execution of the section
- a comment (maximum of 256 characters)
- a protection attribute (no protection, write-protected section, read/write-protected section)

**Programming rules**

When executed, a DFB section can only use the parameters you have defined for the function block (input, output and input/output parameters and internal variables). Consequently, a DFB function block cannot use either the global variables of the application, or the input/output objects, except the system words and bits (%Si, %SWi and %SDi).

A DFB section has maximum access rights (read and write) for its parameters.
Example of code

The following program provides an example of Structured Text code

```plaintext
CHR_200:=CHR_100;
CHR_114:=CHR_104;
CHR_116:=CHR_106;
RESET DEMARRE;
("We increment 80 times CHR_100")
FOR CHR_102:=1 TO 80 DO
  INC CHR_100;
  WHILE((CHR_104-CHR_114)<100)DO
    IF(CHR_104>400) THEN
      EXIT;
    END_IF;
    INC CHR_104;
    REPEAT
    IF(CHR_106>300) THEN
      EXIT;
    END_IF;
    INC CHR_106;
    UNTIL (CHR_100-CHR_116)>100
    END_REPEAT;
  END_WHILE;
("Loop as long as CHR_106")
  IF (CHR_106=CHR_116) THEN
    EXIT;
  ELSE
    CHR_114:=CHR_104;
    CHR_116:=CHR_106;
    END_IF;
    INC CHR_200;
END_FOR;
```
User Function Blocks (DFB) instance

At a Glance

Subject of this Chapter
This chapter provides an overview of the creation of a DFB instance, and its execution.

What's in this Chapter?
This chapter contains the following topics:

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Creation of a DFB instance</td>
<td>532</td>
</tr>
<tr>
<td>Execution of a DFB instance</td>
<td>534</td>
</tr>
<tr>
<td>Programming example for a Derived Function Block (DFB)</td>
<td>535</td>
</tr>
</tbody>
</table>
DFB instance

**Creation of a DFB instance**

A DFB instance is a copy of the DFB model (DFB type):
- it uses the DFB type code (the code is not duplicated),
- it creates a data zone specific to this instance, which is a copy of the parameters and variables of the DFB type. This zone is situated in the application's data area.

You must identify each DFB instance you create with a name of a maximum 32 characters, the authorized characters used depend on the choice made in the **Identifiers** area of the **Language extensions** tab in the **Project Settings** (See Unity Pro, Operating Modes Manual, Language extensions).

The first character must be a letter! Keywords and symbols are prohibited.

**Creation of an instance**

From a DFB type, you can create as many instances as necessary; the only limitation is the size of the PLC memory.

**Initial values**

The initial values of the parameters and public variables that you defined when creating the DFB type can be modified for each DFB instance.

Not all DFB parameters have an initial value.

**Modification of the initial values of the elements in the DFB instances**

<table>
<thead>
<tr>
<th></th>
<th>EDT (except String type)</th>
<th>String Type</th>
<th>EDT Structure</th>
<th>FB</th>
<th>ANY_ARRAY</th>
<th>IODDT</th>
<th>ANY...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>-</td>
<td>No</td>
<td>-</td>
<td>No</td>
</tr>
<tr>
<td>Input/Output</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>-</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Outputs</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>-</td>
<td>-</td>
<td>No</td>
</tr>
<tr>
<td>Public variables</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Private Variables</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
Modification of the initial values of the elements in the DFB type

<table>
<thead>
<tr>
<th></th>
<th>EDT (except String type)</th>
<th>String Type</th>
<th>EDT</th>
<th>DDT structure</th>
<th>FB</th>
<th>ANY_ARRAY</th>
<th>IODDT</th>
<th>ANY_...</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Inputs</strong></td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td><strong>Input/Output</strong></td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>-</td>
<td>-</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td><strong>Outputs</strong></td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>No</td>
</tr>
<tr>
<td><strong>Public variables</strong></td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td><strong>Private Variables</strong></td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
Execution of a DFB instance

A DFB instance is executed as follows:

<table>
<thead>
<tr>
<th>Step</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Loading the values in the input and input/output parameters. On initialization (or on cold restart), all non-assigned inputs take the initial value defined in the DFB type. They then keep the last value assigned to them.</td>
</tr>
<tr>
<td>2</td>
<td>Execution of the internal program of the DFB.</td>
</tr>
<tr>
<td>3</td>
<td>Writing the output parameters.</td>
</tr>
</tbody>
</table>

Debugging of DFBs

The Unity Pro software offers several DFB debugging tools:

- animation table: all parameters, and public and private variables are displayed and animated in real-time. Objects may be modified and forced
- breakpoint, step by step and program diagnostics
- runtime screens: for unitary debugging
Programming example for a Derived Function Block (DFB)

**General**
This example of programming a counter using a DFB is provided for instruction purposes.

**Characteristics of the DFB type**
The DFB type used to create the counter is as follows:

<table>
<thead>
<tr>
<th>Elements</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cpt_parts</td>
<td></td>
</tr>
<tr>
<td>Reset</td>
<td>EBOOL</td>
</tr>
<tr>
<td>Presel</td>
<td>DINT</td>
</tr>
<tr>
<td>Count</td>
<td>EBOOL</td>
</tr>
<tr>
<td>V_cour</td>
<td>DINT</td>
</tr>
</tbody>
</table>

The elements of the `Cpt_parts` DFB type are as follows:

<table>
<thead>
<tr>
<th>Elements</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name of the DFB type</td>
<td><code>Cpt_parts</code></td>
</tr>
<tr>
<td>Input parameters</td>
<td>- Reset: counter reset (EBOOL type)</td>
</tr>
<tr>
<td></td>
<td>- Presel: Preset value of the counter (DINT type)</td>
</tr>
<tr>
<td></td>
<td>- Count: upcounter input (EBOOL type)</td>
</tr>
<tr>
<td>Output parameters</td>
<td>Done: preset value reached output (BOOL type)</td>
</tr>
<tr>
<td>Public internal variable</td>
<td>V_cour: current value of the counter (DINT type)</td>
</tr>
</tbody>
</table>

**Operation of the counter**
The operation of the counter must be as follows:

<table>
<thead>
<tr>
<th>Phase</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>The DFB counts the rising edges on the <code>Count</code> input.</td>
</tr>
<tr>
<td>2</td>
<td>The number of edges it counts is then stored by the variable <code>V_cour</code>. This</td>
</tr>
<tr>
<td></td>
<td>variable is reset by a rising edge on the <code>Reset</code> input.</td>
</tr>
<tr>
<td>3</td>
<td>When the number of edges counted is equal to the preset value, the <code>Done</code></td>
</tr>
<tr>
<td></td>
<td>output is set to 1. This variable is reset by a rising edge on the <code>Reset</code></td>
</tr>
<tr>
<td></td>
<td>input.</td>
</tr>
</tbody>
</table>
The internal program of the DFB type `Cpt_parts` is defined in Structured Text as follows:

```plaintext
IF RE (Reset) THEN
  V_cour:=0;
END_IF;
IF RE (Count) THEN
  V_cour:=V_cour+1;
END_IF;
IF(V_cour>=Presel) THEN
  SET (Done);
ELSE
  RESET (Done);
END_IF;
```

Let us suppose your application needs to count 3 part types (for example, bolts, nuts and screws). The DFB type `Cpt_parts` can be used three times (3 instances) to perform these different counts.

The number of parts to be procured for each type is defined in the words `%MD10`, `%MD12` and `%MD14` respectively. When the number of parts is reached, the counter sends a command to an output (%Q1.2.1, %Q1.2.2 or %Q1.2.3) which then stops the procurement system for the corresponding parts.
The application program is entered in Ladder language as follows. The 3 DFBs (instances) Cpt_bolts, Cpt_nuts and Cpt_screws are used to count the different parts.
Use of the DFBs from the different programming languages

At a Glance

Subject of this Chapter

This chapter provides an overview of DFB instance calls made using the different programming languages.

What’s in this Chapter?

This chapter contains the following topics:

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rules for using DFBs in a program</td>
<td>540</td>
</tr>
<tr>
<td>Use of IODDTs in a DFB</td>
<td>542</td>
</tr>
<tr>
<td>Use of a DFB in a Ladder language program</td>
<td>546</td>
</tr>
<tr>
<td>Use of a DFB in a Structured Text language program</td>
<td>548</td>
</tr>
<tr>
<td>Use of a DFB in an Instruction List program</td>
<td>551</td>
</tr>
<tr>
<td>Use of a DFB in a program in Function Block Diagram language</td>
<td>555</td>
</tr>
</tbody>
</table>
Use of DFBs

Rules for using DFBs in a program

DFB instances can be used in all languages [Instruction List (IL), Structured Text (ST), Ladder (LD) and Function Block Diagram (FBD)] and in all the tasks of the application program (sections, subroutine, etc.), except for event tasks and SFC program transitions.

General rules of use

When using a DFB, you must comply with the following rules for whatever language is being used:

- It is not necessary to connect all the input, input/output or output parameters, except the following parameters, which it is compulsory for you to assign:
  - generic data-type input parameters (ANY_INT, ANY_ARRAY, etc.),
  - input/output parameters,
  - generic data-type output parameters (other than tables) (ANY_INT, ANY_REAL, etc.).
- unconnected input parameters keep the value of the previous call or the initialization value defined for these parameters, if the block has never been called.
- all of the objects assigned to the input, input/output and output parameters must be of the same type as those defined when the DFB type was created (for example: if the type INT is defined for the input parameter "speed", then you cannot assign it the type DINT or REAL). The only exceptions are BOOL and EBOOL types for input and output parameters (not for input/output parameters), which can be mixed.

**Example:** The input parameter "Validation" may be defined as BOOL and associated with a %Mi internal bit of type EBOOL. However, in the internal code of the DFB type, the input parameter actually has BOOL-type properties (it cannot manage edges).

Assignment of parameters

The following table summarizes the different possibilities for assigning parameters in the different programming languages:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Assignment of the parameter (1)</th>
<th>Assignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs</td>
<td>EDT (2)</td>
<td>Connected, value, object or expression</td>
<td>Optional</td>
</tr>
<tr>
<td></td>
<td>BOOL</td>
<td>Connected, value, object or expression</td>
<td>Optional</td>
</tr>
<tr>
<td></td>
<td>DDT</td>
<td>Connected, value or object</td>
<td>Compulsory</td>
</tr>
<tr>
<td></td>
<td>ANY,...</td>
<td>Connected or object</td>
<td>Compulsory</td>
</tr>
<tr>
<td></td>
<td>ANY_ARRAY</td>
<td>Connected or object</td>
<td>Compulsory</td>
</tr>
</tbody>
</table>
Use of DFBs

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Assignment of the parameter (1)</th>
<th>Assignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs/outputs</td>
<td>EDT</td>
<td>Connected or object</td>
<td>Compulsory</td>
</tr>
<tr>
<td></td>
<td>DDT</td>
<td>Connected or object</td>
<td>Compulsory</td>
</tr>
<tr>
<td></td>
<td>IODDT</td>
<td>Connected or object</td>
<td>Compulsory</td>
</tr>
<tr>
<td></td>
<td>ANY_...</td>
<td>Connected or object</td>
<td>Compulsory</td>
</tr>
<tr>
<td></td>
<td>ANY_ARRAY</td>
<td>Connected or object</td>
<td>Compulsory</td>
</tr>
<tr>
<td>Outputs</td>
<td>EDT</td>
<td>Connected or object</td>
<td>Optional</td>
</tr>
<tr>
<td></td>
<td>DDT</td>
<td>Connected or object</td>
<td>Optional</td>
</tr>
<tr>
<td></td>
<td>ANY_...</td>
<td>Connected or object</td>
<td>Compulsory</td>
</tr>
<tr>
<td></td>
<td>ANY_ARRAY</td>
<td>Connected or object</td>
<td>Optional</td>
</tr>
</tbody>
</table>

(1) Connected in Ladder (LD) or Function Block Diagram (FBD) language. Value or object in Instruction List (IL) or Structured Text (ST) language.

(2) Except BOOL-type parameters
# Use of IODDTs in a DFB

## At a Glance

The following tables present the different IODDTs for the Premium and Quantum (See Analog application, p. 544) PLCs that can be used in a DFB (exclusively as input/output (See Authorized Objects for the Different Parameters, p. 525)) parameters.

## Discrete application

The following table lists the IODDTs of the discrete application for Premium PLCs.

<table>
<thead>
<tr>
<th>IODDT families</th>
<th>Can be used in a DFB</th>
</tr>
</thead>
<tbody>
<tr>
<td>T_DIS_IN_GEN</td>
<td>- No</td>
</tr>
<tr>
<td>T_DIS_IN_STD</td>
<td>- No</td>
</tr>
<tr>
<td>T_DIS_EVT</td>
<td>- No</td>
</tr>
<tr>
<td>T_DIS_OUT_GEN</td>
<td>- No</td>
</tr>
<tr>
<td>T_DIS_OUT_STD</td>
<td>- No</td>
</tr>
<tr>
<td>T_DIS_OUT_REFLEX</td>
<td>- No</td>
</tr>
</tbody>
</table>

## Analog application

The following table lists the IODDTs of the analog application for Premium PLCs.

<table>
<thead>
<tr>
<th>IODDT families</th>
<th>Can be used in a DFB</th>
</tr>
</thead>
<tbody>
<tr>
<td>T_ANA_IN_GEN</td>
<td>- No</td>
</tr>
<tr>
<td>T_ANA_IN_STD</td>
<td>- No</td>
</tr>
<tr>
<td>T_ANA_IN_CTRL</td>
<td>Yes -</td>
</tr>
<tr>
<td>T_ANA_IN_EVT</td>
<td>Yes -</td>
</tr>
<tr>
<td>T_ANA_OUT_GEN</td>
<td>- No</td>
</tr>
<tr>
<td>T_ANA_OUT_STD</td>
<td>- No</td>
</tr>
</tbody>
</table>

## Counting application

The following table lists the IODDTs of the counting application for Premium PLCs.

<table>
<thead>
<tr>
<th>IODDT families</th>
<th>Can be used in a DFB</th>
</tr>
</thead>
<tbody>
<tr>
<td>T_COUNT_ACQ</td>
<td>Yes -</td>
</tr>
<tr>
<td>T_COUNT_HIGH_SPEED</td>
<td>Yes -</td>
</tr>
<tr>
<td>T_COUNT_STD</td>
<td>Yes -</td>
</tr>
</tbody>
</table>
## Use of DFBs

### Electronic cam application

The following table lists the IODDTs of the electronic cam application for Premium PLCs.

<table>
<thead>
<tr>
<th>IODDT families</th>
<th>Can be used in a DFB</th>
</tr>
</thead>
<tbody>
<tr>
<td>T_CCY_GROUP0</td>
<td>-</td>
</tr>
<tr>
<td>T_CCY_GROUP1_2_3</td>
<td>-</td>
</tr>
</tbody>
</table>

### Axis control application

The following table lists the IODDTs of the axis control application for Premium PLCs.

<table>
<thead>
<tr>
<th>IODDT families</th>
<th>Can be used in a DFB</th>
</tr>
</thead>
<tbody>
<tr>
<td>T_AXIS_AUTO</td>
<td>Yes -</td>
</tr>
<tr>
<td>T_AXIS_STD</td>
<td>Yes -</td>
</tr>
<tr>
<td>T_INTERPO_STD</td>
<td>Yes -</td>
</tr>
<tr>
<td>T_STEPPER_STD</td>
<td>Yes -</td>
</tr>
</tbody>
</table>

### Sercos application

The following table lists the IODDTs of the Sercos application.

<table>
<thead>
<tr>
<th>IODDT families</th>
<th>Can be used in a DFB</th>
</tr>
</thead>
<tbody>
<tr>
<td>T_CSY_CMD</td>
<td>Yes -</td>
</tr>
<tr>
<td>T_CSY_TRF</td>
<td>Yes -</td>
</tr>
<tr>
<td>T_CSY_RING</td>
<td>Yes -</td>
</tr>
<tr>
<td>T_CSY_IND</td>
<td>Yes -</td>
</tr>
<tr>
<td>T_CSY_FOLLOW</td>
<td>Yes -</td>
</tr>
<tr>
<td>T_CSY_COORD</td>
<td>Yes -</td>
</tr>
<tr>
<td>T_CSY_CAM</td>
<td>Yes -</td>
</tr>
</tbody>
</table>

### Communication application

The following table lists the IODDTs of the communication application for Premium PLCs.

<table>
<thead>
<tr>
<th>IODDT families</th>
<th>Can be used in a DFB</th>
</tr>
</thead>
<tbody>
<tr>
<td>T_COM_STS_GEN</td>
<td>Yes -</td>
</tr>
<tr>
<td>T_COM_UTW_M</td>
<td>Yes -</td>
</tr>
<tr>
<td>T_COM_UTW_S</td>
<td>Yes -</td>
</tr>
<tr>
<td>T_COM_MB</td>
<td>Yes -</td>
</tr>
<tr>
<td>T_COM_CHAR</td>
<td>Yes -</td>
</tr>
<tr>
<td>T_COM_FPW</td>
<td>Yes -</td>
</tr>
<tr>
<td>T_COM_MBP</td>
<td>Yes -</td>
</tr>
</tbody>
</table>
### Adjustment application

The following table lists the IODDTs of the adjustment application.

<table>
<thead>
<tr>
<th>IODDT families</th>
<th>Can be used in a DFB</th>
</tr>
</thead>
<tbody>
<tr>
<td>T_COM_INET</td>
<td>Yes -</td>
</tr>
<tr>
<td>T_COM_ASI</td>
<td>Yes -</td>
</tr>
<tr>
<td>T_COM_ETY_1X0</td>
<td>Yes -</td>
</tr>
<tr>
<td>T_COM_ETY_210</td>
<td>Yes -</td>
</tr>
<tr>
<td>T_COM_IBS_128</td>
<td>Yes -</td>
</tr>
<tr>
<td>T_COM_IBS_242</td>
<td>Yes -</td>
</tr>
<tr>
<td>T_COM_PBY</td>
<td>Yes -</td>
</tr>
<tr>
<td>T_COM_CFP100</td>
<td>Yes -</td>
</tr>
<tr>
<td>T_COM_ETYX103</td>
<td>Yes -</td>
</tr>
<tr>
<td>T_COM_ETHCOPRO</td>
<td>Yes -</td>
</tr>
</tbody>
</table>

### Weighing application

The following table lists the IODDTs of the weighing application for Premium PLCs.

<table>
<thead>
<tr>
<th>IODDT families</th>
<th>Can be used in a DFB</th>
</tr>
</thead>
<tbody>
<tr>
<td>T_WEIGHING_ISPY101</td>
<td>Yes -</td>
</tr>
</tbody>
</table>

### Common to all applications

The following table lists the IODDTs common to all the applications for Premium PLCs.

<table>
<thead>
<tr>
<th>IODDT families</th>
<th>Can be used in a DFB</th>
</tr>
</thead>
<tbody>
<tr>
<td>T_GEN_MOD</td>
<td>- No</td>
</tr>
</tbody>
</table>

### Analog application

The following table lists the IODDTs of the analog application for Quantum PLCs.

<table>
<thead>
<tr>
<th>IODDT families</th>
<th>Can be used in a DFB</th>
</tr>
</thead>
<tbody>
<tr>
<td>T_ANA_IN_VE</td>
<td>- No</td>
</tr>
<tr>
<td>T_ANA_IN_VWE</td>
<td>- No</td>
</tr>
</tbody>
</table>
The following table lists the IODDTs of the counting application for Quantum PLCs.

<table>
<thead>
<tr>
<th>IODDT families</th>
<th>Can be used in a DFB</th>
</tr>
</thead>
<tbody>
<tr>
<td>T_ANA_BI_VWE</td>
<td>- No</td>
</tr>
<tr>
<td>T_ANA_BI_IN_VWE</td>
<td>- No</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>IODDT families</th>
<th>Can be used in a DFB</th>
</tr>
</thead>
<tbody>
<tr>
<td>T_CNT_105</td>
<td>- No</td>
</tr>
</tbody>
</table>
Use of a DFB in a Ladder language program

Principle

In Ladder language, there are two possible ways of calling a DFB function block:
- via a textual call in an operation block in which the syntax and constraints on the parameters are identical to those of Structured Text language
- via a graphic call

The inputs of the function blocks may be connected or assigned a value, an object or an expression. In any case, the type of external element (value, evaluation of the expression, etc.) must be identical to that of the input parameter.

A DFB block must have at least one connected Boolean input and an output (if necessary). For this you may use the EN input parameters and the ENO output parameter (see the description of these parameters below).

It is compulsory to connect or assign the ANY_ARRAY-type inputs, the generic data-type outputs (ANY,...) and the input/outputs of a DFB block.

Graphic representation of a DFB block

The following illustration shows a simple DFB programming example:

Elements of the DFB block

The following table lists the different elements of the DFB block, labeled in the above illustration:

<table>
<thead>
<tr>
<th>Label</th>
<th>Element</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Name of the DFB (instance)</td>
</tr>
<tr>
<td>2</td>
<td>Name of the DFB type</td>
</tr>
<tr>
<td>3</td>
<td>Input assigned by an expression</td>
</tr>
<tr>
<td>4</td>
<td>Input assigned by a value</td>
</tr>
</tbody>
</table>
### Use of DFBs

<table>
<thead>
<tr>
<th>Label</th>
<th>Element</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>Connected input</td>
</tr>
<tr>
<td>6</td>
<td>Input assigned by an object (address or symbol)</td>
</tr>
<tr>
<td>7</td>
<td>Input parameters</td>
</tr>
<tr>
<td>8</td>
<td>Output parameters</td>
</tr>
<tr>
<td>9</td>
<td>Input/output parameters</td>
</tr>
</tbody>
</table>

### Use of EN|ENO parameters

See *EN and ENO parameters, p. 527*
Use of DFBs

Use of a DFB in a Structured Text language program

Principle
In Structured Text, a user function block is called by a DFB call: name of the DFB instance followed by a list of arguments. Arguments are displayed in the list between brackets and separated by commas.

The DFB call can be of one of two types:
- a formal call, when arguments are assignments (parameter = value). In this case, the order in which the arguments are entered in the list is not important.
- an informal call, when arguments are values (expression, object or an immediate value). In this case, the order in which the arguments are entered in the list must follow the order of the DFB input parameters, including for non-assigned inputs (the argument is an empty field).

It is not possible to use EN and ENO parameters.

DFB_Name (argument 1,argument 2,...,argument n)

Note: The ANY_ARRAY-type inputs, generic data-type outputs (ANY_...) and input/outputs of a DFB must be assigned.

Use of EN\ENO parameters
See EN and ENO parameters, p. 527

Example of a DFB
The following simple example explains the different DFB calls in Structured Text language. This is the instance Cpt_1 of the Cpt_parts: type DFB

<table>
<thead>
<tr>
<th>Cpt_1</th>
<th>Cpt_parts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clear</td>
<td>Reset</td>
</tr>
<tr>
<td>Presel</td>
<td>Count</td>
</tr>
<tr>
<td>100</td>
<td>%Q1.2.1</td>
</tr>
</tbody>
</table>

Note:
In Structured Text, a user function block is called by a DFB call: name of the DFB instance followed by a list of arguments. Arguments are displayed in the list between brackets and separated by commas.

The DFB call can be of one of two types:
- a formal call, when arguments are assignments (parameter = value). In this case, the order in which the arguments are entered in the list is not important.
- an informal call, when arguments are values (expression, object or an immediate value). In this case, the order in which the arguments are entered in the list must follow the order of the DFB input parameters, including for non-assigned inputs (the argument is an empty field).

It is not possible to use EN and ENO parameters.

DFB_Name (argument 1,argument 2,...,argument n)

Note: The ANY_ARRAY-type inputs, generic data-type outputs (ANY_...) and input/outputs of a DFB must be assigned.

Use of EN\ENO parameters
See EN and ENO parameters, p. 527

Example of a DFB
The following simple example explains the different DFB calls in Structured Text language. This is the instance Cpt_1 of the Cpt_parts: type DFB

<table>
<thead>
<tr>
<th>Cpt_1</th>
<th>Cpt_parts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clear</td>
<td>Reset</td>
</tr>
<tr>
<td>Presel</td>
<td>Count</td>
</tr>
<tr>
<td>100</td>
<td>%Q1.2.1</td>
</tr>
</tbody>
</table>
Formal DFB call

The formal DFB call \texttt{Cpt\_1} is performed with the following syntax:

\[
\texttt{Cpt\_1 (Reset:=Clear, Presel:=P\_Select, Count:=100, Done=>\%Q1.2.1)};
\]

Where the input parameters assigned by a value (expression, object or immediate value) are entered in the list of arguments, the syntax is:

\[
\texttt{Cpt\_1 (Reset:=Clear, Presel:=P\_Select, Count:=100)};
\]

\[
\ldots
\]

\[
\%Q1.2.1:=\texttt{Cpt\_1\_Done};
\]

Elements of the sequence

The following table lists the different elements of the program sequence, when a formal DFB call is made:

<table>
<thead>
<tr>
<th>Element</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{Cpt_1}</td>
<td>Name of the DFB instance</td>
</tr>
<tr>
<td>Reset, Presel, Count</td>
<td>Input parameters</td>
</tr>
<tr>
<td>:=</td>
<td>Assignment symbol of an input</td>
</tr>
<tr>
<td>Clear</td>
<td>Assignment object of an input (symbol)</td>
</tr>
<tr>
<td>100</td>
<td>Assignment value of an input</td>
</tr>
<tr>
<td>Done</td>
<td>Output parameter</td>
</tr>
<tr>
<td>=&gt;</td>
<td>Assignment symbol of an output</td>
</tr>
<tr>
<td>%Q1.2.1</td>
<td>Assignment object of an output (address)</td>
</tr>
<tr>
<td>;</td>
<td>End of sequence symbol</td>
</tr>
<tr>
<td>,</td>
<td>Argument separation symbol</td>
</tr>
</tbody>
</table>

Informal DFB call

The informal DFB call \texttt{Cpt\_1} is performed with the following syntax:

\[
\texttt{Cpt\_1 (Clear, \%MD10, , 100)};
\]

\[
\ldots
\]

\[
\%Q1.2.1:=\texttt{Cpt\_1\_Done};
\]
Use of DFBs

Elements of the sequence

The following table lists the different elements of the program sequence, when a formal DFB call is made:

<table>
<thead>
<tr>
<th>Element</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cpt_1</td>
<td>Name of the DFB instance</td>
</tr>
<tr>
<td>Clear, %MD10, ,100</td>
<td>Assignment object or value of the inputs. Non-assigned inputs are represented by an empty field</td>
</tr>
<tr>
<td>;</td>
<td>End of sequence symbol</td>
</tr>
<tr>
<td>,</td>
<td>Argument separation symbol</td>
</tr>
</tbody>
</table>
Use of a DFB in an Instruction List program

Principle

In Instruction List, a user function block is called by a CAL instruction, followed by the name of the DFB instance as an operand and a list of arguments (optional). Arguments are displayed in the list between brackets and separated by commas.

In Instruction List, there are three possible ways of calling a DFB:

- the instruction `CAL DFB_Name` is followed by a list of arguments that are assignments (parameter = value). In this case, the order in which the arguments are entered in the list is not important. The EN input parameter can be used to control the execution of the function block.
- the instruction `CAL DFB_Name` is followed by a list of arguments that are values (expression, object or immediate value). In this case, the order in which the arguments are entered in the list must follow the order of the DFB input parameters, including for non-assigned inputs (the argument is an empty field). It is not possible to use EN and ENO parameters.
- The instruction `CAL DFB_Name` is not followed by a list of arguments. In this case, this instruction must be preceded by the assignment of the input parameters, via a register: loading of the value (Load) then assignment to the input parameter (Store). The order of assignment of the parameters (LD/ST) is not important; however, you must assign all the required input parameters before executing the CAL command. It is not possible to use EN and ENO parameters.

CAL DFB_Name (argument 1,argument 2,...,argument n)

or

LD Value 1
ST Parameter 1
...
LD Value n
ST Parameter n
CAL DFB_Name

Note: The ANY_ARRAY-type inputs, generic data-type outputs (ANY_...) and input/outputs of a DFB must be assigned.
Use of DFBs

Example of a DFB

The following example explains the different calls of a DFB in Instruction List. This is the instance Cpt_1 of the Cpt_parts type DFB.

DFB call when the arguments are assignments

When the arguments are assignments, the DFB call Cpt_1 is performed with the following syntax:

```plaintext
CAL Cpt_1 (Reset:=Clear, Presel:=%MD10, Count:=100, Done=>%Q1.2.1)
```

Where the input parameters assigned by a value (expression, object or immediate value) are entered in the list of arguments, the syntax is:

```plaintext
CAL Cpt_1 (Reset:=Clear, Presel:=%MD10, Count:=100)
```

```
... 
LD Cpt_1.Done 
ST %Q1.2.1 
```

In order to make your application program more legible, you can enter a carriage return after the commas that separate the arguments. The sequence then takes the following syntax:

```plaintext
CAL Cpt_1( 
Reset:=Clear, 
Presel:=%MD10, 
Count:=100, 
Done=>%Q1.2.1)
```

See EN and ENO parameters, p. 527.
Use of DFBs

Elements of the DFB call program

The following table lists the different elements of the DFB call program:

<table>
<thead>
<tr>
<th>Element</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAL</td>
<td>DFB call instruction</td>
</tr>
<tr>
<td>Cpt_1</td>
<td>Name of the DFB instance</td>
</tr>
<tr>
<td>Reset, Presel, Count</td>
<td>Input parameters</td>
</tr>
<tr>
<td>:=</td>
<td>Assignment symbol of an input</td>
</tr>
<tr>
<td>Clear, %MD10, 100</td>
<td>Assignment object or value of the inputs</td>
</tr>
<tr>
<td>Done</td>
<td>Output parameter</td>
</tr>
<tr>
<td>=&gt;</td>
<td>Assignment symbol of an output</td>
</tr>
<tr>
<td>%Q1.2.1</td>
<td>Assignment object of an output</td>
</tr>
<tr>
<td>,</td>
<td>Argument separation symbol</td>
</tr>
</tbody>
</table>

DFB call when the arguments are values

When the arguments are values, the DFB call \texttt{Cpt\_1} is performed with the following syntax:

\begin{verbatim}
CAL Cpt_1 (Clear, %MD10,, 100)
...
LD Cpt_1.Done
ST %Q1.2.1
\end{verbatim}

Elements of the DFB call program

The following table lists the different elements of the DFB call program:

<table>
<thead>
<tr>
<th>Element</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAL</td>
<td>DFB call instruction</td>
</tr>
<tr>
<td>Cpt_1</td>
<td>Name of the DFB instance</td>
</tr>
<tr>
<td>Clear, %MD10, 100</td>
<td>Assignment object or value of the inputs</td>
</tr>
<tr>
<td>,</td>
<td>Argument separation symbol</td>
</tr>
</tbody>
</table>
DFB call with no argument

When there is no argument, the DFB call Cpt_1 is performed with the following syntax:

LD Clear
ST Cpt_1.Reset
LD %MD10
ST Cpt_1.Presel
LD 100
ST Cpt_1.Count
CAL Cpt_1(
...
LD Cpt_1.Done
ST %Q1.2.1

Elements of the DFB call program

The following table lists the different elements of the DFB call program:

<table>
<thead>
<tr>
<th>Element</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD Clear</td>
<td>Load instruction to load the Clear value into a register</td>
</tr>
<tr>
<td>ST Cpt_1.Reset</td>
<td>Assign instruction to assign the contents of the register to the input parameter Cpt_1.Reset</td>
</tr>
<tr>
<td>CAL Cpt_1()</td>
<td>Call instruction for the DFB Cpt_1</td>
</tr>
</tbody>
</table>
Use of a DFB in a program in Function Block Diagram language

**Principle**

In FBD (Function Block Diagram) language, the user function blocks are represented in the same way as in Ladder language and are called graphically.

The inputs of the user function blocks may be connected or assigned a value, an immediate object or an expression. In any case, the type of external element must be identical to that of the input parameter.

Only one object can be assigned (link to another block with the same variable) to an input of the DFB. However, several objects may be connected to a single output.

A DFB block must have at least one connected Boolean input and an output (if necessary). For this, you can use an EN input parameter and an ENO output parameter.

It is compulsory to connect or assign the ANY_ARRAY-type inputs, the generic data-type outputs (ANY_...) and the input/outputs of a DFB block.

**Graphic representation of a DFB block**

The following illustration shows a simple DFB programming example.
Elements of the DFB block

The following table lists the different elements of the DFB block, labeled in the above illustration:

<table>
<thead>
<tr>
<th>Label</th>
<th>Element</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Name of the DFB (instance)</td>
</tr>
<tr>
<td>2</td>
<td>Name of the DFB type</td>
</tr>
<tr>
<td>3</td>
<td>Input assigned by an object (symbol)</td>
</tr>
<tr>
<td>4</td>
<td>Input assigned by a value</td>
</tr>
<tr>
<td>5</td>
<td>Connected input</td>
</tr>
<tr>
<td>6</td>
<td>Input parameters</td>
</tr>
<tr>
<td>7</td>
<td>Output parameter</td>
</tr>
<tr>
<td>8</td>
<td>Input assigned by an object (address)</td>
</tr>
</tbody>
</table>

Use of EN\ENO parameters

See EN and ENO parameters, p. 527.
User diagnostics DFB

At a Glance

Subject of this Chapter
This chapter describes how to create and use User Diagnostic Function Blocks on Premium, Atrium and Quantum PLCs.

What's in this Chapter?
This chapter contains the following topics:

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Presentation of user diagnostic DFBs</td>
<td>558</td>
</tr>
<tr>
<td>How to create a user diagnostic DFB</td>
<td>559</td>
</tr>
</tbody>
</table>
Presentation of user diagnostic DFBs

General
The Unity Pro application is used to create your own diagnostic DFBs (See Unity Pro, Operating Modes Manual, Project diagnostic). These diagnostic DFBs are standard DFBs that you will have configured beforehand with the Diagnostic property and in which you will have used the following two functions:
- REGDFB (See Unity Pro, Diagnostic Library, REGDFB: Alarm saving and dating) to save the alarm date.
- DEREG (See Unity Pro, Diagnostic Library, DEREG: Alarm de-registration) to de-register the alarm.

These DFBs enable you to monitor your process. They will automatically report the information you will have chosen in the Viewer. You can thus monitor changes in state or variations in your process.

Advantages
The main advantages inherent in this service are as follows:
- The diagnostic is integrated in the project, and can thus be conceived during development and therefore better meets the user's requirements.
- The error dating and recording system is done at the source (in the PLC), which means the information exactly represents the state of the process.
- You can connect a number of Viewers (Unity Pro, Magelis, Factory Cast) which will transcribe the exact state of the process to the user. Each Viewer is independent, and any action performed on one (for example, an acknowledgement) is automatically viewed on the others.
How to create a user diagnostic DFB

At a Glance

You can create a diagnostic DFB in one of two ways.

- You can create a standard DFB and then define its values and program it so that it is a diagnostic DFB.
- You can use the USER_DIAG_ST_MODEL (See Unity Pro, Diagnostic Library, USER_DIAG_ST_MODEL : description) model that we will provide you with, or use a model you already have.

These two procedures are described below.

Note: to understand how user diagnostic DFBs work, refer to the USER_DIAG_ST_MODEL (See Unity Pro, Diagnostic Library, USER_DIAG_ST_MODEL : description) model described in the diagnostic library.

Creating a diagnostic DFB without a model

The following table describes the procedure to follow to create a diagnostic DFB without a predefined model.

<table>
<thead>
<tr>
<th>Step</th>
<th>Action</th>
</tr>
</thead>
</table>
| 1    | In the structure view of the Project Browser, right-click on the directory Derived FB Types.  
     | Result: A shortcut menu is opened. |
| 2    | Activate the Open command.  
     | Result: The Data Editor window is opened. It is positioned directly under the DFB Types tab and shows the list of the project’s DFBs. |
| 3    | Select the first empty cell in the Name column (indicated by an arrow) and enter the name of the new DFB type, then confirm using Enter.  
     | Result: The empty structure of the DFB type is now created. This new DFB is added to the list of existing DFBs. It also appears in the Derived FB Types directory tree. |
| 4    | In the Data Editor, right-click on the type of DFB you have just created.  
     | Result: the contextual menu appears. |
| 5    | Choose the Properties command.  
     | Result: the Data Properties window appears. |
| 6    | Click on the Diag button.  
     | Result: the checkbox is checked in red. This type of DFB is now a diagnostic DFB. If, in the Project Settings (See Unity Pro, Operating Modes Manual, Choice of configuration parameters for project generation), the application diagnostic has been requested, the project’s generation will incorporate the project diagnostic service in the PLC. |
Creating a diagnostic DFB with the USER_DIAG_ST_MODEL model

The following table describes the procedure to follow to create a diagnostic DFB using the USER_DIAG_ST_MODEL predefined model.

<table>
<thead>
<tr>
<th>Step</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>In the structure view of the Project Browser, right-click on the directory Derived FB Types. Result: A shortcut menu is opened.</td>
</tr>
<tr>
<td>2</td>
<td>Choose the Get from Library command. Result: the Library Browser window appears.</td>
</tr>
<tr>
<td>3</td>
<td>Select the USER_DIAG_ST_MODEL diagnostic DFB model in the diagnostic library, then click on the &gt;&gt; button to transfer it to your project.</td>
</tr>
<tr>
<td>4</td>
<td>Confirm the transfer with OK. Result: the model appears in the Project Browser.</td>
</tr>
<tr>
<td>5</td>
<td>Replace the USER_DIAG_ST_MODEL name with the name of the type of user DFB you want to create.</td>
</tr>
<tr>
<td>6</td>
<td>Modify and create the inputs, outputs, inputs/outputs and the internal variables of the DFB. Note: in the input's properties, remember to check the Diag checkbox for inputs that will be monitored by the DFB.</td>
</tr>
<tr>
<td>7</td>
<td>Write your DFB's code using the model. Note: refer to the description of the code of the user diagnostic DFB model (See Unity Pro, Diagnostic Library, USER_DIAG_ST_MODEL : description) if you are not familiar with this type of DFB.</td>
</tr>
</tbody>
</table>
Run a project analysis to validate your new type of DFB.

**Result:** your new type of DFB is created. You can now create and insert instances of this type of DFB in your program. When your PLC is in RUN mode, the Viewer (See Unity Pro, Operating Modes Manual, Diagnostics Viewer) will display the alarms according to the operation you have programmed.

**Note:** the message displayed by the Viewer is the comment associated with the instance of the DFB. You must therefore remember to fill in this comment for each instance created.

<table>
<thead>
<tr>
<th>Step</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>Run a project analysis to validate your new type of DFB. <strong>Result:</strong> your new type of DFB is created. You can now create and insert instances of this type of DFB in your program. When your PLC is in RUN mode, the Viewer (See Unity Pro, Operating Modes Manual, Diagnostics Viewer) will display the alarms according to the operation you have programmed. <strong>Note:</strong> the message displayed by the Viewer is the comment associated with the instance of the DFB. You must therefore remember to fill in this comment for each instance created.</td>
</tr>
</tbody>
</table>
Appendices

Introduction

At a Glance
The appendix contains additional information.

What’s in this Appendix?
The appendix contains the following chapters:

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Chapter Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>EFB Error Codes and Values</td>
<td>565</td>
</tr>
<tr>
<td>B</td>
<td>IEC Compliance</td>
<td>601</td>
</tr>
</tbody>
</table>
Overview

Introduction

The following tables show the error codes and error values created for the EFBs sort by library and family.

What's in this Chapter?

This chapter contains the following topics:

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tables of Error Codes for the Base Library</td>
<td>566</td>
</tr>
<tr>
<td>Tables of Error Codes for the Diagnostics Library</td>
<td>568</td>
</tr>
<tr>
<td>Tables of Error Codes for the Communication Library</td>
<td>569</td>
</tr>
<tr>
<td>Tables of Error Codes for the IO Management Library</td>
<td>573</td>
</tr>
<tr>
<td>Tables of Error Codes for the CONT_CTL Library</td>
<td>582</td>
</tr>
<tr>
<td>Tables of Error Codes for the Motion Library</td>
<td>590</td>
</tr>
<tr>
<td>Tables of Error Codes for the Obsolete Library</td>
<td>592</td>
</tr>
<tr>
<td>Common Floating Point Errors</td>
<td>599</td>
</tr>
</tbody>
</table>
# Tables of Error Codes for the Base Library

## Introduction
The following tables show the error codes and error values created for the EFBs of the Base Library.

## Date & Time
Table of error codes and errors values created for EFBs of the Date & Time family.

<table>
<thead>
<tr>
<th>EFB name</th>
<th>Error code</th>
<th>ENO state in case of error</th>
<th>Error value in Dec</th>
<th>Error value in Hex</th>
<th>Error description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIVTIME</td>
<td>E_DIVIDE_BY_ZERO</td>
<td>F</td>
<td>-30176</td>
<td>16#8A20</td>
<td>Divide by zero</td>
</tr>
<tr>
<td>DIVTIME</td>
<td>E_NEGATIVE_INPUT_FOR_TIME_OPERATION</td>
<td>F</td>
<td>-30177</td>
<td>16#8A1F</td>
<td>A negative value cannot be converted to data type TIME</td>
</tr>
<tr>
<td>DIVTIME</td>
<td>E_ARITHMETIC_ERROR</td>
<td>F</td>
<td>-30170</td>
<td>16#8A26</td>
<td>Arithmetic error</td>
</tr>
<tr>
<td>DIVTIME</td>
<td>E_ERR_ARITHMETIC</td>
<td>F</td>
<td>-30003</td>
<td>16#8ACD</td>
<td>Arithmetic overflow (%S18 set)</td>
</tr>
<tr>
<td>DIVTIME</td>
<td>FP_ERROR</td>
<td>F</td>
<td>-</td>
<td>-</td>
<td>See table Common Floating Point Errors, p. 599</td>
</tr>
<tr>
<td>MULTIME</td>
<td>E_ERR_ARITHMETIC</td>
<td>F</td>
<td>-30003</td>
<td>16#8ACD</td>
<td>Arithmetic overflow (%S18 set)</td>
</tr>
<tr>
<td>MULTIME</td>
<td>E_ARITHMETIC_ERROR_MUL_OV</td>
<td>F</td>
<td>-30172</td>
<td>16#8A24</td>
<td>Arithmetic error / Multiplication overflow</td>
</tr>
<tr>
<td>MULTIME</td>
<td>E_ARITHMETIC_ERROR_ADD_OV</td>
<td>F</td>
<td>-30173</td>
<td>16#8A23</td>
<td>Arithmetic error / Addition overflow</td>
</tr>
<tr>
<td>MULTIME</td>
<td>E_ARITHMETIC_ERROR_BIG_PAR</td>
<td>F</td>
<td>-30171</td>
<td>16#8A25</td>
<td>Arithmetic error / Parameter exceeds range</td>
</tr>
<tr>
<td>MULTIME</td>
<td>E_NEGATIVE_INPUT_FOR_TIME_OPERATION</td>
<td>F</td>
<td>-30177</td>
<td>16#8A1F</td>
<td>A negative value cannot be converted to data type TIME</td>
</tr>
<tr>
<td>MULTIME</td>
<td>FP_ERROR</td>
<td>F</td>
<td>-</td>
<td>-</td>
<td>See table Common Floating Point Errors, p. 599</td>
</tr>
</tbody>
</table>
**Table of error codes and errors values created for EFBs of the Statistical family.**

<table>
<thead>
<tr>
<th>EFB name</th>
<th>Error code</th>
<th>ENO state in case of error</th>
<th>Error value in Dec</th>
<th>Error value in Hex</th>
<th>Error description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AVE</td>
<td>E_INPUT_VALUE_OUT_OF_RANGE</td>
<td>F</td>
<td>-30183</td>
<td>16#8A19</td>
<td>Input value is out of range</td>
</tr>
<tr>
<td>AVE</td>
<td>E_DIVIDE_BY_ZERO</td>
<td>F</td>
<td>-30176</td>
<td>16#8A20</td>
<td>Divide by zero</td>
</tr>
<tr>
<td>AVE</td>
<td>FP_ERROR</td>
<td>F</td>
<td>-</td>
<td>-</td>
<td>See table Common Floating Point Errors, p. 599</td>
</tr>
<tr>
<td>AVE</td>
<td>E_ARITHMETIC_ERROR</td>
<td>F</td>
<td>-30170</td>
<td>16#8A26</td>
<td>Arithmetic error</td>
</tr>
<tr>
<td>AVE</td>
<td>E_FP_STATUS_FAILED</td>
<td>F</td>
<td>-30150</td>
<td>16#8A3A</td>
<td>Illegal floating point operation</td>
</tr>
<tr>
<td>AVE</td>
<td>E_ARITHMETIC_ERROR_MUL_OV</td>
<td>F</td>
<td>-30172</td>
<td>16#8A24</td>
<td>Arithmetic error / Multiplication overflow</td>
</tr>
<tr>
<td>AVE</td>
<td>E_ARITHMETIC_ERROR_ADD_OV</td>
<td>F</td>
<td>-30173</td>
<td>16#8A23</td>
<td>Arithmetic error / Addition overflow</td>
</tr>
<tr>
<td>AVE</td>
<td>E_ARITHMETIC_ERROR_BIG_PAR</td>
<td>F</td>
<td>-30171</td>
<td>16#8A25</td>
<td>Arithmetic error / Parameter exceeds range</td>
</tr>
<tr>
<td>AVE</td>
<td>E_ARITHMETIC_ERROR_UNSIGNED_OV</td>
<td>F</td>
<td>-30174</td>
<td>16#8A22</td>
<td>Arithmetic error / Unsigned overflow</td>
</tr>
<tr>
<td>MAX</td>
<td>FP_ERROR</td>
<td>F</td>
<td>-</td>
<td>-</td>
<td>See table Common Floating Point Errors, p. 599</td>
</tr>
<tr>
<td>MIN</td>
<td>FP_ERROR</td>
<td>F</td>
<td>-</td>
<td>-</td>
<td>See table Common Floating Point Errors, p. 599</td>
</tr>
<tr>
<td>MUX</td>
<td>E_SELECTOR_OUT_OF_RANGE</td>
<td>F</td>
<td>-30175</td>
<td>16#8A21</td>
<td>Selector is out of range</td>
</tr>
</tbody>
</table>
# Tables of Error Codes for the Diagnostics Library

## Introduction

The following tables show the error codes and error values created for the EFBs of the Diagnostics Library.

## Diagnostics

Table of error codes and error values created for EFBs of the Diagnostics family.

<table>
<thead>
<tr>
<th>EFB name</th>
<th>Error code</th>
<th>ENO state in case of error</th>
<th>Error value in Dec</th>
<th>Error value in Hex</th>
<th>Error description</th>
</tr>
</thead>
</table>
| ONLEVT   | E_EFB_ONLEVT | T/F                        | -30196             | 16#8AOC            | Error of EFB ONLEVT ENO states  
  - True = Error registration OK  
  - False = Error registration failed |
## Tables of Error Codes for the Communication Library

### Introduction

The following tables show the error codes and error values created for the EFBs of the Communication Library.

### Extended

Table of error codes and error values created for EFBs of the *Extended family*.

<table>
<thead>
<tr>
<th>EFB name</th>
<th>Error code</th>
<th>ENO state in case of error</th>
<th>Error value in Dec</th>
<th>Error value in Hex</th>
<th>Error description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CREAD_REG</td>
<td>E_EFB_MSTR_ERROR</td>
<td>F</td>
<td>-30191</td>
<td>16#8A11</td>
<td>MSTR communication error</td>
</tr>
<tr>
<td>CREAD_REG</td>
<td>E_EFB_NOT_STATE_RAM_4X</td>
<td>F</td>
<td>-30531</td>
<td>16#88BD</td>
<td>Variable not mapped to % MW (4x) area</td>
</tr>
<tr>
<td>CREAD_REG</td>
<td></td>
<td>F</td>
<td>8195</td>
<td>16#2003</td>
<td>Value displayed in status word. (Comes together with E_EFB_MSTR_ERROR)</td>
</tr>
<tr>
<td>CREAD_REG</td>
<td></td>
<td>F</td>
<td>8206</td>
<td>16#200E</td>
<td>Value displayed in status word. Comes together with E_EFB_NOT_STATE_RAM_4X</td>
</tr>
<tr>
<td>CWRITE_REG</td>
<td>E_EFB_MSTR_ERROR</td>
<td>F</td>
<td>-30191</td>
<td>16#8A11</td>
<td>MSTR communication error</td>
</tr>
<tr>
<td>CWRITE_REG</td>
<td></td>
<td>F</td>
<td>8195</td>
<td>16#2003</td>
<td>Value displayed in status word. Comes together with E_EFB_MSTR_ERROR</td>
</tr>
<tr>
<td>CWRITE_REG</td>
<td></td>
<td>F</td>
<td>8206</td>
<td>16#200E</td>
<td>Value displayed in status word. Comes together with E_EFB_NOT_STATE_RAM_4X</td>
</tr>
<tr>
<td>EFB name</td>
<td>Error code</td>
<td>ENO state in case of error</td>
<td>Error value in Dec</td>
<td>Error value in Hex</td>
<td>Error description</td>
</tr>
<tr>
<td>------------------</td>
<td>---------------------</td>
<td>-----------------------------</td>
<td>--------------------</td>
<td>-------------------</td>
<td>------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>CWRITE_REG</td>
<td>-</td>
<td>F</td>
<td>-</td>
<td>-</td>
<td>See tables of:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>- Modbus Plus and SY/MAX EtherNet Error Codes (See Unity Pro, Communication Library, Modbus Plus und SY/MAX EtherNet Fehlercodes)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>- SY/MAX specific Error Codes (See Unity Pro, Communication Library, SY/MAX-specific error codes)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>- TCP/IP EtherNet Error Codes (See Unity Pro, Communication Library, TCP/IP Ethernet error codes)</td>
</tr>
<tr>
<td>MBP_MSTR</td>
<td>E_EFB_OUT_OF_RANGE</td>
<td>F</td>
<td>-30192</td>
<td>16#8A10</td>
<td>Internal error: EFB has detected a violation e.g. write exceeds %MW (4x) boundaries</td>
</tr>
<tr>
<td>MBP_MSTR</td>
<td>E_EFB_NOT_STATE_RAM_4X</td>
<td>F</td>
<td>-30531</td>
<td>16#88BD</td>
<td>Variable not mapped to %MW (4x) area</td>
</tr>
<tr>
<td>MBP_MSTR</td>
<td>-</td>
<td>F</td>
<td>8195</td>
<td>16#2003</td>
<td>Value displayed in status word Comes together with E_EFB_MSTR_ERROR in status of control block</td>
</tr>
<tr>
<td>MBP_MSTR</td>
<td>-</td>
<td>F</td>
<td>8206</td>
<td>16#200E</td>
<td>Value displayed in status word Comes together with E_EFB_NOT_STATE_RAM_4X in status of control block</td>
</tr>
<tr>
<td>MBP_MSTR</td>
<td>-</td>
<td>F</td>
<td>-</td>
<td>-</td>
<td>See tables of:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>- Modbus Plus and SY/MAX EtherNet Error Codes (See Unity Pro, Communication Library, Modbus Plus und SY/MAX EtherNet Fehlercodes)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>- SY/MAX specific Error Codes (See Unity Pro, Communication Library, SY/MAX-specific error codes)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>- TCP/IP EtherNet Error Codes (See Unity Pro, Communication Library, TCP/IP Ethernet error codes)</td>
</tr>
<tr>
<td>EFB name</td>
<td>Error code</td>
<td>ENO state in case of error</td>
<td>Error value in Dec</td>
<td>Error value in Hex</td>
<td>Error description</td>
</tr>
<tr>
<td>----------</td>
<td>-----------------</td>
<td>---------------------------</td>
<td>--------------------</td>
<td>--------------------</td>
<td>----------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>READ_REG</td>
<td>W_WARN_OUT_OF_</td>
<td>F</td>
<td>30110</td>
<td>16#759E</td>
<td>Parameter out of range</td>
</tr>
<tr>
<td></td>
<td>RANGE</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>READ_REG</td>
<td>E_EFB_NOT_STATE_</td>
<td>F</td>
<td>-30531</td>
<td>16#88BD</td>
<td>Variable not mapped to %MW (4x) area</td>
</tr>
<tr>
<td></td>
<td>RAM_4X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>READ_REG</td>
<td>E_EFB_MSTR_ERROR</td>
<td>F</td>
<td>-30191</td>
<td>16#8A11</td>
<td>MSTR communication error</td>
</tr>
<tr>
<td>READ_REG</td>
<td>-</td>
<td>F</td>
<td>8195</td>
<td>16#2003</td>
<td>Value displayed in status word</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Comes together with W_WARN_OUT_OF_RANGE</td>
</tr>
<tr>
<td>READ_REG</td>
<td>MBPUNLOC</td>
<td>F</td>
<td>8206</td>
<td>16#200E</td>
<td>Value displayed in status word</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Comes together with E_EFB_NOT_STATE_RAM_4X</td>
</tr>
<tr>
<td>WRITE_REG</td>
<td>W_WARN_OUT_OF_</td>
<td>F</td>
<td>30110</td>
<td>16#759E</td>
<td>Parameter out of range</td>
</tr>
<tr>
<td></td>
<td>RANGE</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WRITE_REG</td>
<td>E_EFB_NOT_STATE_</td>
<td>F</td>
<td>-30531</td>
<td>16#88BD</td>
<td>Variable not mapped to %MW (4x) area</td>
</tr>
<tr>
<td></td>
<td>RAM_4X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WRITE_REG</td>
<td>E_EFB_MSTR_ERROR</td>
<td>F</td>
<td>-30191</td>
<td>16#8A11</td>
<td>MSTR communication error</td>
</tr>
<tr>
<td>WRITE_REG</td>
<td>-</td>
<td>F</td>
<td>8195</td>
<td>16#2003</td>
<td>Value displayed in status word</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Comes together with W_WARN_OUT_OF_RANGE</td>
</tr>
<tr>
<td>WRITE_REG</td>
<td>-</td>
<td>F</td>
<td>8206</td>
<td>16#200E</td>
<td>Value displayed in status word</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Comes together with E_EFB_NOT_STATE_RAM_4X</td>
</tr>
</tbody>
</table>
# EFB Error Codes and Values

<table>
<thead>
<tr>
<th>EFB name</th>
<th>Error code</th>
<th>ENO state in case of error</th>
<th>Error value in Dec</th>
<th>Error value in Hex</th>
<th>Error description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WRITE_REG</td>
<td>-</td>
<td>F</td>
<td>-</td>
<td>-</td>
<td>See tables of:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Modbus Plus and SY/MAX EtherNet Error Codes (See Unity Pro, Communication Library, Modbus Plus und SY/MAX EtherNet Fehlercodes)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• SY/MAX specific Error Codes (See Unity Pro, Communication Library, SY/MAX-specific error codes)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• TCP/IP EtherNet Error Codes (See Unity Pro, Communication Library, TCP/IP Ethernet error codes)</td>
</tr>
</tbody>
</table>
# Tables of Error Codes for the IO Management Library

**Introduction**

The following tables show the error codes and error values created for the EFBs of the IO Management Library.

## Analog I/O Configuration

Table of error codes and error values created for EFBs of the Analog I/O Configuration family.

<table>
<thead>
<tr>
<th>EFB name</th>
<th>Error code</th>
<th>ENO state in case of error</th>
<th>Error value in Dec</th>
<th>Error value in Hex</th>
<th>Error description</th>
</tr>
</thead>
<tbody>
<tr>
<td>I_FILTER</td>
<td>E_EFB_NOT_CONFIGURED</td>
<td>F</td>
<td>-30188</td>
<td>16#8A14</td>
<td>EFB configuration does not match hardware configuration</td>
</tr>
<tr>
<td>I_SET</td>
<td>E_EFB_USER_ERROR_1</td>
<td>F</td>
<td>-30200</td>
<td>16#8A08</td>
<td>The input IN_REG is not connected with the number of an input word (%IW).</td>
</tr>
<tr>
<td>I_SET</td>
<td>E_EFB_USER_ERROR_2</td>
<td>F</td>
<td>-30201</td>
<td>16#8A07</td>
<td>The input IN_REG is connected with an invalid number of an input word (%IW).</td>
</tr>
<tr>
<td>I_SET</td>
<td>E_EFB_USER_ERROR_3</td>
<td>F</td>
<td>-30202</td>
<td>16#8A06</td>
<td>MN_RAW MX_RAW</td>
</tr>
<tr>
<td>I_SET</td>
<td>E_EFB_USER_ERROR_4</td>
<td>F</td>
<td>-30203</td>
<td>16#8A05</td>
<td>Unknown value for MN_PHYS</td>
</tr>
<tr>
<td>I_SET</td>
<td>E_EFB_USER_ERROR_5</td>
<td>F</td>
<td>-30204</td>
<td>16#8A04</td>
<td>Unknown value for MX_PHYS</td>
</tr>
<tr>
<td>I_SET</td>
<td>E_EFB_USER_ERROR_11</td>
<td>F</td>
<td>-30210</td>
<td>16#89FE</td>
<td>ST_REG not entered</td>
</tr>
<tr>
<td>I_SET</td>
<td>E_EFB_USER_ERROR_12</td>
<td>F</td>
<td>-30211</td>
<td>16#89FD</td>
<td>ST_REG too large</td>
</tr>
<tr>
<td>I_SET</td>
<td>E_EFB_USER_ERROR_13</td>
<td>F</td>
<td>-30212</td>
<td>16#89FC</td>
<td>ST_CH not entered</td>
</tr>
<tr>
<td>O_FILTER</td>
<td>E_EFB_NOT_CONFIGURED</td>
<td>F</td>
<td>-30188</td>
<td>16#8A14</td>
<td>EFB configuration does not match hardware configuration</td>
</tr>
<tr>
<td>O_SET</td>
<td>E_EFB_USER_ERROR_1</td>
<td>F</td>
<td>-30200</td>
<td>16#8A08</td>
<td>The input OUT_REG is not connected with the number of an output word (%MW).</td>
</tr>
<tr>
<td>O_SET</td>
<td>E_EFB_USER_ERROR_2</td>
<td>F</td>
<td>-30201</td>
<td>16#8A07</td>
<td>The input OUT_REG is connected with an invalid number of an output word (%MW).</td>
</tr>
<tr>
<td>O_SET</td>
<td>E_EFB_USER_ERROR_3</td>
<td>F</td>
<td>-30202</td>
<td>16#8A06</td>
<td>MN_RAW MX_RAW</td>
</tr>
<tr>
<td>O_SET</td>
<td>E_EFB_USER_ERROR_4</td>
<td>F</td>
<td>-30203</td>
<td>16#8A05</td>
<td>Unknown value for MN_PHYS</td>
</tr>
<tr>
<td>O_SET</td>
<td>E_EFB_USER_ERROR_5</td>
<td>F</td>
<td>-30204</td>
<td>16#8A04</td>
<td>Unknown value for MX_PHYS</td>
</tr>
<tr>
<td>EFB name</td>
<td>Error code</td>
<td>ENO state in case of error</td>
<td>Error value in Dec</td>
<td>Error value in Hex</td>
<td>Error description</td>
</tr>
<tr>
<td>----------</td>
<td>------------</td>
<td>---------------------------</td>
<td>--------------------</td>
<td>-------------------</td>
<td>-------------------------</td>
</tr>
<tr>
<td>O_SET</td>
<td>E_EFB_USER_ERROR_11</td>
<td>F</td>
<td>-30210</td>
<td>16#89FE</td>
<td>ST_REG not entered</td>
</tr>
<tr>
<td>O_SET</td>
<td>E_EFB_USER_ERROR_12</td>
<td>F</td>
<td>-30211</td>
<td>16#89FD</td>
<td>ST_REG too large</td>
</tr>
<tr>
<td>O_SET</td>
<td>E_EFB_USER_ERROR_13</td>
<td>F</td>
<td>-30212</td>
<td>16#89FC</td>
<td>ST_CH not entered</td>
</tr>
</tbody>
</table>
## Analog I/O Scaling

Table of error codes and errors values created for EFBs of the Analog I/O Scaling family.

<table>
<thead>
<tr>
<th>EFB name</th>
<th>Error code</th>
<th>ENO state in case of error</th>
<th>Error value in Dec</th>
<th>Error value in Hex</th>
<th>Error description</th>
</tr>
</thead>
<tbody>
<tr>
<td>I_NORM</td>
<td>E_EFB_NEG_OVER_RANGE</td>
<td>F</td>
<td>-30187</td>
<td>16#8A15</td>
<td>Negative overflow</td>
</tr>
<tr>
<td>I_NORM</td>
<td>E_EFB_NOT_CONFIGURED</td>
<td>F</td>
<td>-30188</td>
<td>16#8A14</td>
<td>EFB configuration does not match hardware configuration</td>
</tr>
<tr>
<td>I_NORM_WARN</td>
<td>E_EFB_NO_WARNING_STATUS_AVailable</td>
<td>F</td>
<td>-30189</td>
<td>16#8A13</td>
<td>Module delivers no warning status</td>
</tr>
<tr>
<td>I_NORM_WARN</td>
<td>E_EFB_POS_OVER_RANGE</td>
<td>F</td>
<td>-30186</td>
<td>16#8A16</td>
<td>Positive overflow</td>
</tr>
<tr>
<td>I_NORM_WARN</td>
<td>E_EFB_NEG_OVER_RANGE</td>
<td>F</td>
<td>-30187</td>
<td>16#8A15</td>
<td>Negative overflow</td>
</tr>
<tr>
<td>I_NORM_WARN</td>
<td>E_EFB_NOT_CONFIGURED</td>
<td>F</td>
<td>-30188</td>
<td>16#8A14</td>
<td>EFB configuration does not match hardware configuration</td>
</tr>
<tr>
<td>I_PHYS</td>
<td>E_EFB_NO_WARNING_STATUS_AVailable</td>
<td>F</td>
<td>-30189</td>
<td>16#8A13</td>
<td>Module delivers no warning status</td>
</tr>
<tr>
<td>I_PHYS</td>
<td>E_INPUT_VALUE_OUT_OF_RANGE</td>
<td>F</td>
<td>-30183</td>
<td>16#8A19</td>
<td>Input value is out of range</td>
</tr>
<tr>
<td>I_PHYS</td>
<td>E_EFB_NO_MEASURING_RANGE</td>
<td>F</td>
<td>-30185</td>
<td>16#8A17</td>
<td>Internal error</td>
</tr>
<tr>
<td>I_PHYS</td>
<td>E_EFB_POS_OVER_RANGE</td>
<td>F</td>
<td>-30186</td>
<td>16#8A16</td>
<td>Positive overflow</td>
</tr>
<tr>
<td>I_PHYS</td>
<td>E_EFB_NEG_OVER_RANGE</td>
<td>F</td>
<td>-30187</td>
<td>16#8A15</td>
<td>Negative overflow</td>
</tr>
<tr>
<td>I_PHYS</td>
<td>E_EFB_NOT_CONFIGURED</td>
<td>F</td>
<td>-30188</td>
<td>16#8A14</td>
<td>EFB configuration does not match hardware configuration</td>
</tr>
<tr>
<td>I_PHYS_WARN</td>
<td>E_EFB_NO_WARNING_STATUS_AVailable</td>
<td>F</td>
<td>-30189</td>
<td>16#8A13</td>
<td>Module delivers no warning status</td>
</tr>
<tr>
<td>I_PHYS_WARN</td>
<td>E_EFB_FILTER_SQRT_NOT_AVAIL</td>
<td>F</td>
<td>-30195</td>
<td>16#8A0D</td>
<td>Filter SQRT is not available</td>
</tr>
<tr>
<td>I_PHYS_WARN</td>
<td>E_INPUT_VALUE_OUT_OF_RANGE</td>
<td>F</td>
<td>-30183</td>
<td>16#8A19</td>
<td>Input value is out of range</td>
</tr>
<tr>
<td>I_PHYS_WARN</td>
<td>E_EFB_NO_MEASURING_RANGE</td>
<td>F</td>
<td>-30185</td>
<td>16#8A17</td>
<td>Internal error</td>
</tr>
</tbody>
</table>
## EFB Error Codes and Values

<table>
<thead>
<tr>
<th>EFB name</th>
<th>Error code</th>
<th>ENO state in case of error</th>
<th>Error value in Dec</th>
<th>Error value in Hex</th>
<th>Error description</th>
</tr>
</thead>
<tbody>
<tr>
<td>I_PHYS_WARN</td>
<td>E_EFB_POS_OVER_RANGE</td>
<td>F</td>
<td>-30186</td>
<td>16#8A16</td>
<td>Positive overflow</td>
</tr>
<tr>
<td>I_PHYS_WARN</td>
<td>E_EFB_NEG_OVER_RANGE</td>
<td>F</td>
<td>-30187</td>
<td>16#8A15</td>
<td>Negative overflow</td>
</tr>
<tr>
<td>I_PHYS_WARN</td>
<td>E_EFB_NOT_CONFIGURED</td>
<td>F</td>
<td>-30188</td>
<td>16#8A14</td>
<td>EFB configuration does not match hardware configuration</td>
</tr>
<tr>
<td>I_RAW</td>
<td>E_EFB_OUT_OF_RANGE</td>
<td>F</td>
<td>-30192</td>
<td>16#8A10</td>
<td>Internal error: EFB has detected a violation e.g. write exceeds %MW (4x) boundaries</td>
</tr>
<tr>
<td>I_RAW</td>
<td>E_EFB_NOT_CONFIGURED</td>
<td>F</td>
<td>-30188</td>
<td>16#8A14</td>
<td>EFB configuration does not match hardware configuration</td>
</tr>
<tr>
<td>I_RAWSIM</td>
<td>E_EFB_NOT_CONFIGURED</td>
<td>F</td>
<td>-30188</td>
<td>16#8A14</td>
<td>EFB configuration does not match hardware configuration</td>
</tr>
<tr>
<td>I_SCALE</td>
<td>E_EFB_POS_OVER_RANGE</td>
<td>F</td>
<td>-30186</td>
<td>16#8A16</td>
<td>Positive overflow</td>
</tr>
<tr>
<td>I_SCALE</td>
<td>E_EFB_NEG_OVER_RANGE</td>
<td>F</td>
<td>-30187</td>
<td>16#8A15</td>
<td>Negative overflow</td>
</tr>
<tr>
<td>I_SCALE</td>
<td>E_EFB_NOT_CONFIGURED</td>
<td>F</td>
<td>-30188</td>
<td>16#8A14</td>
<td>EFB configuration does not match hardware configuration</td>
</tr>
<tr>
<td>I_SCALE</td>
<td>E_EFB_NO_WARNING_STATUS_AVAILABLE</td>
<td>F</td>
<td>-30189</td>
<td>16#8A13</td>
<td>Module delivers no warning status</td>
</tr>
<tr>
<td>I_SCALE_WARN</td>
<td>E_EFB_POS_OVER_RANGE</td>
<td>F</td>
<td>-30186</td>
<td>16#8A16</td>
<td>Positive overflow</td>
</tr>
<tr>
<td>I_SCALE_WARN</td>
<td>E_EFB_NEG_OVER_RANGE</td>
<td>F</td>
<td>-30187</td>
<td>16#8A15</td>
<td>Negative overflow</td>
</tr>
<tr>
<td>I_SCALE_WARN</td>
<td>E_EFB_NOT_CONFIGURED</td>
<td>F</td>
<td>-30188</td>
<td>16#8A14</td>
<td>EFB configuration does not match hardware configuration</td>
</tr>
<tr>
<td>O_NORM</td>
<td>E_EFB_POS_OVER_RANGE</td>
<td>F</td>
<td>-30186</td>
<td>16#8A16</td>
<td>Positive overflow</td>
</tr>
<tr>
<td>O_NORM</td>
<td>E_EFB_NEG_OVER_RANGE</td>
<td>F</td>
<td>-30187</td>
<td>16#8A15</td>
<td>Negative overflow</td>
</tr>
<tr>
<td>O_NORM</td>
<td>E_EFB_NOT_CONFIGURED</td>
<td>F</td>
<td>-30188</td>
<td>16#8A14</td>
<td>EFB configuration does not match hardware configuration</td>
</tr>
<tr>
<td>EFB name</td>
<td>Error code</td>
<td>ENO state in case of error</td>
<td>Error value in Dec</td>
<td>Error value in Hex</td>
<td>Error description</td>
</tr>
<tr>
<td>-------------</td>
<td>-------------------------</td>
<td>----------------------------</td>
<td>--------------------</td>
<td>--------------------</td>
<td>-----------------------------------------------------------</td>
</tr>
<tr>
<td>O_NORM_WARN</td>
<td>E_EFB_POS_OVER_RANGE</td>
<td>F</td>
<td>-30186</td>
<td>16#8A16</td>
<td>Positive overflow</td>
</tr>
<tr>
<td>O_NORM_WARN</td>
<td>E_EFB_NEG_OVER_RANGE</td>
<td>F</td>
<td>-30187</td>
<td>16#8A15</td>
<td>Negative overflow</td>
</tr>
<tr>
<td>O_NORM_WARN</td>
<td>E_EFB_NOT_CONFIGURED</td>
<td>F</td>
<td>-30188</td>
<td>16#8A14</td>
<td>EFB configuration does not match hardware configuration</td>
</tr>
<tr>
<td>O_PHYS</td>
<td>E_EFB_NO_MEASURING_RANGE</td>
<td>F</td>
<td>-30185</td>
<td>16#8A17</td>
<td>Internal error</td>
</tr>
<tr>
<td>O_PHYS</td>
<td>E_EFB_POS_OVER_RANGE</td>
<td>F</td>
<td>-30186</td>
<td>16#8A16</td>
<td>Positive overflow</td>
</tr>
<tr>
<td>O_PHYS</td>
<td>E_EFB_NEG_OVER_RANGE</td>
<td>F</td>
<td>-30187</td>
<td>16#8A15</td>
<td>Negative overflow</td>
</tr>
<tr>
<td>O_PHYS</td>
<td>E_EFB_NOT_CONFIGURED</td>
<td>F</td>
<td>-30188</td>
<td>16#8A14</td>
<td>EFB configuration does not match hardware configuration</td>
</tr>
<tr>
<td>O_PHYS_WARN</td>
<td>E_EFB_NO_MEASURING_RANGE</td>
<td>F</td>
<td>-30185</td>
<td>16#8A17</td>
<td>Internal error</td>
</tr>
<tr>
<td>O_PHYS_WARN</td>
<td>E_EFB_POS_OVER_RANGE</td>
<td>F</td>
<td>-30186</td>
<td>16#8A16</td>
<td>Positive overflow</td>
</tr>
<tr>
<td>O_PHYS_WARN</td>
<td>E_EFB_NEG_OVER_RANGE</td>
<td>F</td>
<td>-30187</td>
<td>16#8A15</td>
<td>Negative overflow</td>
</tr>
<tr>
<td>O_PHYS_WARN</td>
<td>E_EFB_NOT_CONFIGURED</td>
<td>F</td>
<td>-30188</td>
<td>16#8A14</td>
<td>EFB configuration does not match hardware configuration</td>
</tr>
<tr>
<td>O_RAW</td>
<td>E_EFB_NEG_OVER_RANGE</td>
<td>F</td>
<td>-30187</td>
<td>16#8A15</td>
<td>Negative overflow</td>
</tr>
<tr>
<td>O_RAW</td>
<td>E_EFB_NOT_CONFIGURED</td>
<td>F</td>
<td>-30188</td>
<td>16#8A14</td>
<td>EFB configuration does not match hardware configuration</td>
</tr>
<tr>
<td>O_SCALE</td>
<td>E_INPUT_VALUE_OUT_OF_RANGE</td>
<td>F</td>
<td>-30183</td>
<td>16#8A19</td>
<td>Input value is out of range</td>
</tr>
<tr>
<td>O_SCALE</td>
<td>E_EFB_POS_OVER_RANGE</td>
<td>F</td>
<td>-30186</td>
<td>16#8A16</td>
<td>Positive overflow</td>
</tr>
<tr>
<td>O_SCALE</td>
<td>E_EFB_NEG_OVER_RANGE</td>
<td>F</td>
<td>-30187</td>
<td>16#8A15</td>
<td>Negative overflow</td>
</tr>
<tr>
<td>O_SCALE</td>
<td>E_EFB_NOT_CONFIGURED</td>
<td>F</td>
<td>-30188</td>
<td>16#8A14</td>
<td>EFB configuration does not match hardware configuration</td>
</tr>
<tr>
<td>EFB name</td>
<td>Error code</td>
<td>ENO state in case of error</td>
<td>Error value in Dec</td>
<td>Error value in Hex</td>
<td>Error description</td>
</tr>
<tr>
<td>-----------------------</td>
<td>-----------------------------</td>
<td>---------------------------</td>
<td>--------------------</td>
<td>--------------------</td>
<td>--------------------------------------------------------</td>
</tr>
<tr>
<td>O_SCALE_WARN</td>
<td>E_INPUT_VALUE_OUT_OF_RANGE</td>
<td>F</td>
<td>-30183</td>
<td>16#8A19</td>
<td>Input value is out of range</td>
</tr>
<tr>
<td>O_SCALE_WARN</td>
<td>E_EFB_POS OVER RANGE</td>
<td>F</td>
<td>-30186</td>
<td>16#8A16</td>
<td>Positive overflow</td>
</tr>
<tr>
<td>O_SCALE_WARN</td>
<td>E_EFB_NEG OVER RANGE</td>
<td>F</td>
<td>-30187</td>
<td>16#8A15</td>
<td>Negative overflow</td>
</tr>
<tr>
<td>O_SCALE_WARN</td>
<td>E_EFB_NOT CONFIGURED</td>
<td>F</td>
<td>-30188</td>
<td>16#8A14</td>
<td>EFB configuration does not match hardware configuration</td>
</tr>
</tbody>
</table>

**Immediate I/O**  
Table of error codes and errors values created for EFBs of the **Immediate I/O** family.

<table>
<thead>
<tr>
<th>EFB name</th>
<th>Error code</th>
<th>ENO state in case of error</th>
<th>Error value in Dec</th>
<th>Error value in Hex</th>
<th>Error description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IMIO_IN</td>
<td>-</td>
<td>F</td>
<td>0000</td>
<td>0000</td>
<td>Operation OK</td>
</tr>
<tr>
<td>IMIO_IN</td>
<td>-</td>
<td>F</td>
<td>8193</td>
<td>2001</td>
<td>invalid operation type (e.g. the I/O module addressed is not an input module)</td>
</tr>
<tr>
<td>IMIO_IN</td>
<td>-</td>
<td>F</td>
<td>8194</td>
<td>2002</td>
<td>Invalid rack or slot number (I/O map in the configurator contains no module entry for this slot)</td>
</tr>
<tr>
<td>IMIO_IN</td>
<td>-</td>
<td>F</td>
<td>8195</td>
<td>2003</td>
<td>invalid slot number</td>
</tr>
<tr>
<td>IMIO_IN</td>
<td>-</td>
<td>F</td>
<td>-4095</td>
<td>F001</td>
<td>Module not OK</td>
</tr>
<tr>
<td>IMIO_OUT</td>
<td>-</td>
<td>F</td>
<td>0000</td>
<td>0000</td>
<td>Operation OK</td>
</tr>
<tr>
<td>IMIO_OUT</td>
<td>-</td>
<td>F</td>
<td>8193</td>
<td>2001</td>
<td>invalid operation type (e.g. the I/O module addressed is not an input module)</td>
</tr>
<tr>
<td>IMIO_OUT</td>
<td>-</td>
<td>F</td>
<td>8194</td>
<td>2002</td>
<td>Invalid rack or slot number (I/O map in the configurator contains no module entry for this slot)</td>
</tr>
<tr>
<td>IMIO_OUT</td>
<td>-</td>
<td>F</td>
<td>8195</td>
<td>2003</td>
<td>invalid slot number</td>
</tr>
<tr>
<td>IMIO_OUT</td>
<td>-</td>
<td>F</td>
<td>-4095</td>
<td>F001</td>
<td>Module not OK</td>
</tr>
</tbody>
</table>
# EFB Error Codes and Values

## Quantum I/O Configuration

Table of error codes and errors values created for EFBs of the Quantum I/O Configuration family.

<table>
<thead>
<tr>
<th>EFB name</th>
<th>Error code</th>
<th>ENO state in case of error</th>
<th>Error value in Dec</th>
<th>Error value in Hex</th>
<th>Error description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACI030</td>
<td>E_EFB_NOT_CONFIGURED</td>
<td>F</td>
<td>-30188</td>
<td>16#8A14</td>
<td>EFB configuration does not match hardware configuration</td>
</tr>
<tr>
<td>ACI040</td>
<td>E_EFB_NOT_CONFIGURED</td>
<td>F</td>
<td>-30188</td>
<td>16#8A14</td>
<td>EFB configuration does not match hardware configuration</td>
</tr>
<tr>
<td>ACI040</td>
<td>E_EFB_CURRENT_MODE_NOT_ALLOWED</td>
<td>F</td>
<td>-30197</td>
<td>16#8A0B</td>
<td>EFB error: Current mode is not allowed</td>
</tr>
<tr>
<td>ACO020</td>
<td>E_EFB_NOT_CONFIGURED</td>
<td>F</td>
<td>-30188</td>
<td>16#8A14</td>
<td>EFB configuration does not match hardware configuration</td>
</tr>
<tr>
<td>ACO130</td>
<td>E_EFB_NOT_CONFIGURED</td>
<td>F</td>
<td>-30188</td>
<td>16#8A14</td>
<td>EFB configuration does not match hardware configuration</td>
</tr>
<tr>
<td>ACO130</td>
<td>E_EFB_CURRENT_MODE_NOT_ALLOWED</td>
<td>F</td>
<td>-30197</td>
<td>16#8A0B</td>
<td>EFB error: Current mode is not allowed</td>
</tr>
<tr>
<td>AII330</td>
<td>E_EFB_NOT_CONFIGURED</td>
<td>F</td>
<td>-30188</td>
<td>16#8A14</td>
<td>EFB configuration does not match hardware configuration</td>
</tr>
<tr>
<td>AII330</td>
<td>E_EFB_ILLEGAL_CONFIG_DATA</td>
<td>F</td>
<td>-30198</td>
<td>16#8A0A</td>
<td>EFB error: Illegal configuration data</td>
</tr>
<tr>
<td>AII33010</td>
<td>E_EFB_NOT_CONFIGURED</td>
<td>F</td>
<td>-30188</td>
<td>16#8A14</td>
<td>EFB configuration does not match hardware configuration</td>
</tr>
<tr>
<td>AII33010</td>
<td>E_EFB_CURRENT_MODE_NOT_ALLOWED</td>
<td>F</td>
<td>-30197</td>
<td>16#8A0B</td>
<td>EFB error: Current mode is not allowed</td>
</tr>
<tr>
<td>AIO330</td>
<td>E_EFB_NOT_CONFIGURED</td>
<td>F</td>
<td>-30188</td>
<td>16#8A14</td>
<td>EFB configuration does not match hardware configuration</td>
</tr>
<tr>
<td>AIO330</td>
<td>E_EFB_CURRENT_MODE_NOT_ALLOWED</td>
<td>F</td>
<td>-30197</td>
<td>16#8A0B</td>
<td>EFB error: Current mode is not allowed</td>
</tr>
<tr>
<td>AMM090</td>
<td>E_EFB_NOT_CONFIGURED</td>
<td>F</td>
<td>-30188</td>
<td>16#8A14</td>
<td>EFB configuration does not match hardware configuration</td>
</tr>
<tr>
<td>ARI030</td>
<td>E_EFB_NOT_CONFIGURED</td>
<td>F</td>
<td>-30188</td>
<td>16#8A14</td>
<td>EFB configuration does not match hardware configuration</td>
</tr>
<tr>
<td>ARI030</td>
<td>E_EFB_ILLEGAL_CONFIG_DATA</td>
<td>F</td>
<td>-30198</td>
<td>16#8A0A</td>
<td>EFB error: Illegal configuration data</td>
</tr>
<tr>
<td>ATI030</td>
<td>E_EFB_NOT_CONFIGURED</td>
<td>F</td>
<td>-30188</td>
<td>16#8A14</td>
<td>EFB configuration does not match hardware configuration</td>
</tr>
<tr>
<td>EFB name</td>
<td>Error code</td>
<td>ENO state in case of error</td>
<td>Error value in Dec</td>
<td>Error value in Hex</td>
<td>Error description</td>
</tr>
<tr>
<td>------------------</td>
<td>--------------------</td>
<td>-----------------------------</td>
<td>--------------------</td>
<td>--------------------</td>
<td>--------------------------------------------------------</td>
</tr>
<tr>
<td>AVI030</td>
<td>E_EFB_NOT_CONFIGURED</td>
<td>F</td>
<td>-30188</td>
<td>16#8A14</td>
<td>EFB configuration does not match hardware configuration</td>
</tr>
<tr>
<td>AVO020</td>
<td>E_EFB_NOT_CONFIGURED</td>
<td>F</td>
<td>-30188</td>
<td>16#8A14</td>
<td>EFB configuration does not match hardware configuration</td>
</tr>
<tr>
<td>DROP</td>
<td>E_EFB_NOT_CONFIGURED</td>
<td>F</td>
<td>-30188</td>
<td>16#8A14</td>
<td>EFB configuration does not match hardware configuration</td>
</tr>
<tr>
<td>ERT_854_10</td>
<td>ES_WRONG_SLOT</td>
<td>F</td>
<td>20480</td>
<td>16#5000</td>
<td>-</td>
</tr>
<tr>
<td>ERT_854_10</td>
<td>E_WRONG_SLOT</td>
<td>F</td>
<td>-30215</td>
<td>16#89F9</td>
<td>Defined as E_EFB_USER_ERROR_16</td>
</tr>
<tr>
<td>ERT_854_10</td>
<td>ES_HEALTHBIT</td>
<td>F</td>
<td>24576</td>
<td>16#6000</td>
<td>-</td>
</tr>
<tr>
<td>ERT_854_10</td>
<td>E_HEALTHBIT</td>
<td>F</td>
<td>-30216</td>
<td>16#89F8</td>
<td>Defined as E_EFB_USER_ERROR_17</td>
</tr>
<tr>
<td>ERT_854_10</td>
<td>ES_TIMEOUT</td>
<td>F</td>
<td>32768</td>
<td>16#8000</td>
<td>-</td>
</tr>
<tr>
<td>ERT_854_10</td>
<td>E_TIMEOUT</td>
<td>F</td>
<td>-30210</td>
<td>16#89FE</td>
<td>Defined as E_EFB_USER_ERROR_11</td>
</tr>
<tr>
<td>ERT_854_10</td>
<td>E_ERT_BASIC - values</td>
<td>F</td>
<td>-30199</td>
<td>16#8A09</td>
<td>Defined as E_EFB_USER_ERROR_1 + 1</td>
</tr>
<tr>
<td>ERT_854_10</td>
<td>E_WRONG_ANSW</td>
<td>F</td>
<td>-30211</td>
<td>16#89FD</td>
<td>Defined as E_EFB_USER_ERROR_12</td>
</tr>
<tr>
<td>ERT_854_10</td>
<td>ES_CBUF_OFLOW</td>
<td>F</td>
<td>28672</td>
<td>16#7000</td>
<td>-</td>
</tr>
<tr>
<td>ERT_854_10</td>
<td>E_CBUF_OFLOW</td>
<td>F</td>
<td>-30217</td>
<td>16#89F7</td>
<td>Defined as E_EFB_USER_ERROR_18</td>
</tr>
<tr>
<td>ERT_854_10</td>
<td>ES_WRONG_PAKET</td>
<td>F</td>
<td>8192</td>
<td>16#2000</td>
<td>-</td>
</tr>
<tr>
<td>ERT_854_10</td>
<td>E_WRONG_FELD</td>
<td>F</td>
<td>-30212</td>
<td>16#89FC</td>
<td>Defined as E_EFB_USER_ERROR_13</td>
</tr>
<tr>
<td>ERT_854_10</td>
<td>ES_WRONG_FELD</td>
<td>F</td>
<td>12288</td>
<td>16#3000</td>
<td>-</td>
</tr>
<tr>
<td>ERT_854_10</td>
<td>E_WRONG_PAKET</td>
<td>F</td>
<td>-30213</td>
<td>16#89FB</td>
<td>Defined as E_EFB_USER_ERROR_14</td>
</tr>
<tr>
<td>QUANTUM</td>
<td>E_EFB_NOT_CONFIGURED</td>
<td>F</td>
<td>-30188</td>
<td>16#8A14</td>
<td>EFB configuration does not match hardware configuration</td>
</tr>
<tr>
<td>QUANTUM</td>
<td>E_EFB_UNKNOWN_DROP</td>
<td>F</td>
<td>-30190</td>
<td>16#8A12</td>
<td>Unknown drop / No Quantum traffic cop</td>
</tr>
<tr>
<td>XBE</td>
<td>E_EFB_NOT_CONFIGURED</td>
<td>F</td>
<td>-30188</td>
<td>16#8A14</td>
<td>EFB configuration does not match hardware configuration</td>
</tr>
<tr>
<td>XBE</td>
<td>E_EFB_UNKNOWN_DROP</td>
<td>F</td>
<td>-30190</td>
<td>16#8A12</td>
<td>Unknown drop / No Quantum traffic cop</td>
</tr>
<tr>
<td>XDROP</td>
<td>E_EFB_NOT_CONFIGURED</td>
<td>F</td>
<td>-30188</td>
<td>16#8A14</td>
<td>EFB configuration does not match hardware configuration</td>
</tr>
</tbody>
</table>
Note: For details about ERT_854_10, please refer to the ERT_854_10 description (See Unity Pro, I/O Management Manual, Data Flow) in the IO Management Library.
## Tables of Error Codes for the CONT_CTL Library

**Introduction**
The following tables show the error codes and error values created for the EFBs of the CONT_CTL Library.

**Conditioning**
Table of error codes and error values created for EFBs of the Conditioning family.

<table>
<thead>
<tr>
<th>EFB name</th>
<th>Error code</th>
<th>ENO state in case of error</th>
<th>Error value in Dec</th>
<th>Error value in Hex</th>
<th>Error description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DTIME</td>
<td>W_WARN_OUT_OF_RANGE</td>
<td>T</td>
<td>30110</td>
<td>16#759E</td>
<td>Parameter out of range</td>
</tr>
<tr>
<td>DTIME</td>
<td>FP_ERROR</td>
<td>F</td>
<td>-</td>
<td>-</td>
<td>See table Common Floating Point Errors, p. 599</td>
</tr>
<tr>
<td>DTIME</td>
<td>Status word values</td>
<td>T/F</td>
<td>-</td>
<td>-</td>
<td>For details about the DTIME status word refer to the DTIME description (see Unity Pro, Control, Block Library)</td>
</tr>
<tr>
<td>INTEGRATOR</td>
<td>E_ERR_DEN</td>
<td>F</td>
<td>-30152</td>
<td>16#8A38</td>
<td>Not a valid floating point number</td>
</tr>
<tr>
<td>INTEGRATOR</td>
<td>E_ERR_IB_MAX_MIN</td>
<td>F</td>
<td>-30102</td>
<td>16#8A6A</td>
<td>YMAX &lt; YMIN</td>
</tr>
<tr>
<td>INTEGRATOR</td>
<td>FP_ERROR</td>
<td>F</td>
<td>-</td>
<td>-</td>
<td>See table Common Floating Point Errors, p. 599</td>
</tr>
<tr>
<td>LAG_FILTER</td>
<td>E_ERR_DEN</td>
<td>F</td>
<td>-30152</td>
<td>16#8A38</td>
<td>Not a valid floating point number</td>
</tr>
<tr>
<td>LAG_FILTER</td>
<td>FP_ERROR</td>
<td>F</td>
<td>-</td>
<td>-</td>
<td>See table Common Floating Point Errors, p. 599</td>
</tr>
<tr>
<td>LDLG</td>
<td>E_ERR_DEN</td>
<td>F</td>
<td>-30152</td>
<td>16#8A38</td>
<td>Not a valid floating point number</td>
</tr>
<tr>
<td>LDLG</td>
<td>FP_ERROR</td>
<td>F</td>
<td>-</td>
<td>-</td>
<td>See table Common Floating Point Errors, p. 599</td>
</tr>
<tr>
<td>LEAD</td>
<td>E_ERR_DEN</td>
<td>F</td>
<td>-30152</td>
<td>16#8A38</td>
<td>Not a valid floating point number</td>
</tr>
<tr>
<td>LEAD</td>
<td>FP_ERROR</td>
<td>F</td>
<td>-</td>
<td>-</td>
<td>See table Common Floating Point Errors, p. 599</td>
</tr>
<tr>
<td>MFLOW</td>
<td>W_WARN_OUT_OF_RANGE</td>
<td>T</td>
<td>30110</td>
<td>16#759E</td>
<td>Parameter out of range</td>
</tr>
<tr>
<td>MFLOW</td>
<td>FP_ERROR</td>
<td>F</td>
<td>-</td>
<td>-</td>
<td>See table Common Floating Point Errors, p. 599</td>
</tr>
<tr>
<td>MFLOW</td>
<td>Status word values</td>
<td>T/F</td>
<td>-</td>
<td>-</td>
<td>For details about the MFLOW status word refer to the MFLOW description (see Unity Pro, Control, Block Library)</td>
</tr>
<tr>
<td>EFB name</td>
<td>Error code</td>
<td>ENO state in case of error</td>
<td>Error value in Dec</td>
<td>Error value in Hex</td>
<td>Error description</td>
</tr>
<tr>
<td>-------------</td>
<td>------------------</td>
<td>----------------------------</td>
<td>--------------------</td>
<td>--------------------</td>
<td>-----------------------------------------------------------</td>
</tr>
<tr>
<td>QDTIME</td>
<td>E_ERR_DEN</td>
<td>F</td>
<td>-30152</td>
<td>16#8A38</td>
<td>Not a valid floating point number</td>
</tr>
<tr>
<td>SCALING</td>
<td>E_ERR_NULL_INPUT_SCA</td>
<td>F</td>
<td>-30121</td>
<td>16#8A57</td>
<td>Null input scale: max and min limit must be different</td>
</tr>
<tr>
<td>SCALING</td>
<td>FP_ERROR</td>
<td>F</td>
<td>-</td>
<td>-</td>
<td>See table Common Floating Point Errors, p. 599</td>
</tr>
<tr>
<td>SCALING</td>
<td>Status word values</td>
<td>T/F</td>
<td>-</td>
<td>-</td>
<td>For details about the SCALING status word refer to the SCALING description (see Unity Pro, Control, Block Library)</td>
</tr>
<tr>
<td>TOTALIZER</td>
<td>W_WARN_OUT_OF_RANGE</td>
<td>T</td>
<td>30110</td>
<td>16#759E</td>
<td>Parameter out of range</td>
</tr>
<tr>
<td>TOTALIZER</td>
<td>FP_ERROR</td>
<td>F</td>
<td>-</td>
<td>16#759E</td>
<td>See table Common Floating Point Errors, p. 599</td>
</tr>
<tr>
<td>TOTALIZER</td>
<td>W_WARN_TOTALIZER_CTER_MAX</td>
<td>T</td>
<td>30113</td>
<td>16#75A1</td>
<td>Maximum value of cter has been reached</td>
</tr>
<tr>
<td>TOTALIZER</td>
<td>Status word values</td>
<td>T/F</td>
<td>-</td>
<td>-</td>
<td>For details about the TOTALIZER status word refer to the TOTALIZER description (see Unity Pro, Control, Block Library)</td>
</tr>
<tr>
<td>VEL_LIM</td>
<td>E_ERR_DEN</td>
<td>F</td>
<td>-30152</td>
<td>16#8A38</td>
<td>Not a valid floating point number</td>
</tr>
<tr>
<td>VEL_LIM</td>
<td>E_ERR_AB1_MAX_MIN</td>
<td>F</td>
<td>-30101</td>
<td>16#8A6B</td>
<td>YMAX &lt; YMIN</td>
</tr>
<tr>
<td>VEL_LIM</td>
<td>FP_ERROR</td>
<td>F</td>
<td>-</td>
<td>-</td>
<td>See table Common Floating Point Errors, p. 599</td>
</tr>
</tbody>
</table>
## EFB Error Codes and Values

Table of error codes and errors values created for EFBs of the **Controller** family.

<table>
<thead>
<tr>
<th>EFB name</th>
<th>Error code</th>
<th>ENO state in case of error</th>
<th>Error value in Dec</th>
<th>Error value in Hex</th>
<th>Error description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AUTOTUNE</td>
<td>W_WARN_OUT_OF_RANGE</td>
<td>T</td>
<td>30110</td>
<td>16#759E</td>
<td>Parameter out of range</td>
</tr>
<tr>
<td>AUTOTUNE</td>
<td>E_ERR_NULL_INPUT_SCALE</td>
<td>F</td>
<td>-30121</td>
<td>16#8A57</td>
<td>Null input scale: max and min limit must be different</td>
</tr>
<tr>
<td>AUTOTUNE</td>
<td>W_WARN_AUTOTUNE_FAILED</td>
<td>T</td>
<td>30111</td>
<td>16#759F</td>
<td>AUTOTUNE has failed</td>
</tr>
<tr>
<td>AUTOTUNE</td>
<td>FP_ERROR</td>
<td>F</td>
<td>-</td>
<td>-</td>
<td>See table Common Floating Point Errors, p. 599</td>
</tr>
<tr>
<td>AUTOTUNE</td>
<td>E_ERR_AUTOTUNE_ID_UNKNOWN</td>
<td>F</td>
<td>-30120</td>
<td>16#8A58</td>
<td>The tuned EFB is not allowed or has not yet been called</td>
</tr>
<tr>
<td></td>
<td>Status word values</td>
<td>T/F</td>
<td>-</td>
<td>-</td>
<td>For details about the AUTOTUNE status word refer to the AUTOTUNE description (see Unity Pro, Control, Block Library)</td>
</tr>
<tr>
<td>PI_B</td>
<td>W_WARN_OUT_OF_RANGE</td>
<td>T</td>
<td>30110</td>
<td>16#759E</td>
<td>Parameter out of range</td>
</tr>
<tr>
<td>PI_B</td>
<td>E_ERR_NULL_INPUT_SCALE</td>
<td>F</td>
<td>-30121</td>
<td>16#8A57</td>
<td>Null input scale: max and min limit must be different</td>
</tr>
<tr>
<td>PI_B</td>
<td>FP_ERROR</td>
<td>F</td>
<td>-</td>
<td>-</td>
<td>See table Common Floating Point Errors, p. 599</td>
</tr>
<tr>
<td>PI_B</td>
<td>Status word values</td>
<td>T/F</td>
<td>-</td>
<td>-</td>
<td>For details about the PI_B status word refer to the PI_B description (see Unity Pro, Control, Block Library)</td>
</tr>
<tr>
<td>PIDFF</td>
<td>W_WARN_OUT_OF_RANGE</td>
<td>T</td>
<td>30110</td>
<td>16#759E</td>
<td>Parameter out of range</td>
</tr>
<tr>
<td>PIDFF</td>
<td>E_ERR_NULL_INPUT_SCALE</td>
<td>F</td>
<td>-30121</td>
<td>16#8A57</td>
<td>Null input scale: max and min limit must be different</td>
</tr>
<tr>
<td>PIDFF</td>
<td>FP_ERROR</td>
<td>F</td>
<td>-</td>
<td>-</td>
<td>See table Common Floating Point Errors, p. 599</td>
</tr>
<tr>
<td>PIDFF</td>
<td>Status word values</td>
<td>T/F</td>
<td>-</td>
<td>-</td>
<td>For details about the PIDFF status word refer to the PIDFF description (see Unity Pro, Control, Block Library)</td>
</tr>
<tr>
<td>SAMPLETM</td>
<td>E_EFB_SAMPLE_TIME_OVERFLOW</td>
<td>F</td>
<td>-30184</td>
<td>16#8A18</td>
<td>Internal error</td>
</tr>
<tr>
<td>STEP2</td>
<td>W_WARN_OUT_OF_RANGE</td>
<td>T</td>
<td>30110</td>
<td>16#759E</td>
<td>Parameter out of range</td>
</tr>
</tbody>
</table>

---

584

35006144 04 June 2006
<table>
<thead>
<tr>
<th>EFB name</th>
<th>Error code</th>
<th>ENO state in case of error</th>
<th>Error value in Dec</th>
<th>Error value in Hex</th>
<th>Error description</th>
</tr>
</thead>
<tbody>
<tr>
<td>STEP2</td>
<td>FP_ERROR</td>
<td>F</td>
<td>-</td>
<td>-</td>
<td>See table Common Floating Point Errors, p. 599</td>
</tr>
<tr>
<td>STEP2</td>
<td>Status word values</td>
<td>T/F</td>
<td>-</td>
<td>-</td>
<td>For details about the STEP2 status word refer to the STEP2 description (see Unity Pro, Control, Block Library)</td>
</tr>
<tr>
<td>STEP3</td>
<td>W_WARN_OUT_OF_RANGE</td>
<td>T</td>
<td>30110</td>
<td>16#759E</td>
<td>Parameter out of range</td>
</tr>
<tr>
<td>STEP3</td>
<td>FP_ERROR</td>
<td>F</td>
<td>-</td>
<td>-</td>
<td>See table Common Floating Point Errors, p. 599</td>
</tr>
<tr>
<td>STEP3</td>
<td>Status word values</td>
<td>T/F</td>
<td>-</td>
<td>-</td>
<td>For details about the STEP3 status word refer to the STEP3 description (see Unity Pro, Control, Block Library)</td>
</tr>
</tbody>
</table>
### Mathematics

Table of error codes and error values created for EFBs of the Mathematics family.

<table>
<thead>
<tr>
<th>EFB name</th>
<th>Error code</th>
<th>ENO state in case of error</th>
<th>Error value in Dec</th>
<th>Error value in Hex</th>
<th>Error description</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMP_DB</td>
<td>W_WARN_OUT_OF_RANGE</td>
<td>T</td>
<td>30110</td>
<td>16#759E</td>
<td>Parameter out of range</td>
</tr>
<tr>
<td>COMP_DB</td>
<td>FP_ERROR</td>
<td>F</td>
<td>-</td>
<td>-</td>
<td>See table Common Floating Point Errors, p. 599</td>
</tr>
<tr>
<td>K_SQRT</td>
<td>W_WARN_OUT_OF_RANGE</td>
<td>T</td>
<td>30110</td>
<td>16#759E</td>
<td>Parameter out of range</td>
</tr>
<tr>
<td>K_SQRT</td>
<td>FP_ERROR</td>
<td>F</td>
<td>-</td>
<td>-</td>
<td>See table Common Floating Point Errors, p. 599</td>
</tr>
<tr>
<td>MULDIV_W</td>
<td>FP_ERROR</td>
<td>F</td>
<td>-</td>
<td>-</td>
<td>See table Common Floating Point Errors, p. 599</td>
</tr>
<tr>
<td>SUM_W</td>
<td>FP_ERROR</td>
<td>F</td>
<td>-</td>
<td>-</td>
<td>See table Common Floating Point Errors, p. 599</td>
</tr>
</tbody>
</table>
**Measurement**

Table of error codes and errors values created for EFBs of the Measurement family.

<table>
<thead>
<tr>
<th>EFB name</th>
<th>Error code</th>
<th>ENO state in case of error</th>
<th>Error value in Dec</th>
<th>Error value in Hex</th>
<th>Error description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AVGMV</td>
<td>E_ERR_DEN</td>
<td>F</td>
<td>-30152</td>
<td>16#8A38</td>
<td>Not a valid floating point number</td>
</tr>
<tr>
<td>AVGMV</td>
<td>W_WARN_AVGMV</td>
<td>T</td>
<td>30108</td>
<td>16#759C</td>
<td>AVGMV: N must be &lt;= 50</td>
</tr>
<tr>
<td>AVGMV</td>
<td>FP_ERROR</td>
<td>F</td>
<td>-</td>
<td>-</td>
<td>See table <a href="#">Common Floating Point Errors, p. 599</a></td>
</tr>
<tr>
<td>AVGMV_K</td>
<td>E_ERR_DEN</td>
<td>F</td>
<td>-30152</td>
<td>16#8A38</td>
<td>Not a valid floating point number</td>
</tr>
<tr>
<td>AVGMV_K</td>
<td>W_WARN_AVGMV_K</td>
<td>T</td>
<td>30109</td>
<td>16#759D</td>
<td>AVGMV_K: N must be &lt;= 10000</td>
</tr>
<tr>
<td>AVGMV_K</td>
<td>FP_ERROR</td>
<td>F</td>
<td>-</td>
<td>-</td>
<td>See table <a href="#">Common Floating Point Errors, p. 599</a></td>
</tr>
<tr>
<td>DEAD_ZONE</td>
<td>E_ERR_DEN</td>
<td>F</td>
<td>-30152</td>
<td>16#8A38</td>
<td>Not a valid floating point number</td>
</tr>
<tr>
<td>DEAD_ZONE</td>
<td>E_ERR_DZONE</td>
<td>F</td>
<td>-30119</td>
<td>16#8A59</td>
<td>DZONE: DZ must be &gt;= 0</td>
</tr>
<tr>
<td>DEAD_ZONE</td>
<td>FP_ERROR</td>
<td>F</td>
<td>-</td>
<td>-</td>
<td>See table <a href="#">Common Floating Point Errors, p. 599</a></td>
</tr>
<tr>
<td>LOOKUP_TABLE1</td>
<td>E_ERR_DEN</td>
<td>F</td>
<td>-30152</td>
<td>16#8A38</td>
<td>Not a valid floating point number</td>
</tr>
<tr>
<td>LOOKUP_TABLE1</td>
<td>E_ERR_POLY_ANZAHL</td>
<td>F</td>
<td>-30107</td>
<td>16#8A65</td>
<td>number of inputs not even</td>
</tr>
<tr>
<td>LOOKUP_TABLE1</td>
<td>E_ERR_POLY_FOLGE</td>
<td>F</td>
<td>-30108</td>
<td>16#8A64</td>
<td>base point x(i) &lt;= x(i-1)</td>
</tr>
<tr>
<td>LOOKUP_TABLE1</td>
<td>FP_ERROR</td>
<td>F</td>
<td>-</td>
<td>-</td>
<td>See table <a href="#">Common Floating Point Errors, p. 599</a></td>
</tr>
</tbody>
</table>
## Table of error codes and error values created for EFBs of the Output Processing family.

<table>
<thead>
<tr>
<th>EFB name</th>
<th>Error code</th>
<th>ENO state in case of error</th>
<th>Error value in Dec</th>
<th>Error value in Hex</th>
<th>Error description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MS</td>
<td>W_WARN_OUT_OF_RANGE</td>
<td>T</td>
<td>30110</td>
<td>16#759E</td>
<td>Parameter out of range</td>
</tr>
<tr>
<td>MS</td>
<td>FP_ERROR</td>
<td>F</td>
<td>-</td>
<td>-</td>
<td>See table Common Floating Point Errors, p. 599</td>
</tr>
<tr>
<td>MS</td>
<td>Status word values</td>
<td>T/F</td>
<td>-</td>
<td>-</td>
<td>For details about the MS status word refer to the MS description (see Unity Pro, Control, Block Library)</td>
</tr>
<tr>
<td>PWM1</td>
<td>WAF_PBM_TMINMAX</td>
<td>F</td>
<td>-30113</td>
<td>16#8A5F</td>
<td>t_min &lt; t_max</td>
</tr>
<tr>
<td>PWM1</td>
<td>FP_ERROR</td>
<td>F</td>
<td>-</td>
<td>-</td>
<td>See table Common Floating Point Errors, p. 599</td>
</tr>
<tr>
<td>SERVO</td>
<td>FP_ERROR</td>
<td>F</td>
<td>-</td>
<td>-</td>
<td>See table Common Floating Point Errors, p. 599</td>
</tr>
<tr>
<td>SERVO</td>
<td>Status word values</td>
<td>T/F</td>
<td>-</td>
<td>-</td>
<td>For details about the SERVO status word refer to the SERVO description (see Unity Pro, Control, Block Library)</td>
</tr>
<tr>
<td>SPLRG</td>
<td>W_WARN_OUT_OF_RANGE</td>
<td>T</td>
<td>30110</td>
<td>16#759E</td>
<td>Parameter out of range</td>
</tr>
<tr>
<td>SPLRG</td>
<td>E_ERR_NULL_INPUT_SCALE</td>
<td>F</td>
<td>-30121</td>
<td>16#8A57</td>
<td>Null input scale: max and min limit must be different</td>
</tr>
<tr>
<td>SPLRG</td>
<td>FP_ERROR</td>
<td>F</td>
<td>-</td>
<td>-</td>
<td>See table Common Floating Point Errors, p. 599</td>
</tr>
<tr>
<td>SPLRG</td>
<td>Status word values</td>
<td>T/F</td>
<td>-</td>
<td>-</td>
<td>For details about the SPLRG status word refer to the SPLRG description (see Unity Pro, Control, Block Library)</td>
</tr>
</tbody>
</table>
## Setpoint Management

Table of error codes and errors values created for EFBs of the Setpoint Management family.

<table>
<thead>
<tr>
<th>EFB name</th>
<th>Error code</th>
<th>ENO state in case of error</th>
<th>Error value in Dec</th>
<th>Error value in Hex</th>
<th>Error description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAMP</td>
<td>W_WARN_OUT_OF_RANGE</td>
<td>T</td>
<td>30110</td>
<td>16#759E</td>
<td>Parameter out of range</td>
</tr>
<tr>
<td>RAMP</td>
<td>FP_ERROR</td>
<td>F</td>
<td>-</td>
<td>-</td>
<td>See table Common Floating Point Errors, p. 599</td>
</tr>
<tr>
<td>RAMP</td>
<td>Status word values</td>
<td>T/F</td>
<td>-</td>
<td>-</td>
<td>For details about the RAMP status word refer to the RAMP description (see Unity Pro, Control, Block Library)</td>
</tr>
<tr>
<td>RATIO</td>
<td>FP_ERROR</td>
<td>F</td>
<td>-</td>
<td>-</td>
<td>See table Common Floating Point Errors, p. 599</td>
</tr>
<tr>
<td>RATIO</td>
<td>Status word values</td>
<td>T/F</td>
<td>-</td>
<td>-</td>
<td>For details about the RATIO status word refer to the RATIO description (see Unity Pro, Control, Block Library)</td>
</tr>
<tr>
<td>SP_SEL</td>
<td>W_WARN_OUT_OF_RANGE</td>
<td>T</td>
<td>30110</td>
<td>16#759E</td>
<td>Parameter out of range</td>
</tr>
<tr>
<td>SP_SEL</td>
<td>FP_ERROR</td>
<td>F</td>
<td>-</td>
<td>-</td>
<td>See table Common Floating Point Errors, p. 599</td>
</tr>
<tr>
<td>SP_SEL</td>
<td>Status word values</td>
<td>T/F</td>
<td>-</td>
<td>-</td>
<td>For details about the SP_SEL status word refer to the SP_SEL description (see Unity Pro, Control, Block Library)</td>
</tr>
</tbody>
</table>
# Tables of Error Codes for the Motion Library

## Introduction

The following tables show the error codes and error values created for the EFBs of the Motion Library.

## MMF Start

Table of error codes and error values created for EFBs of the **MMF Start** family.

<table>
<thead>
<tr>
<th>EFB name</th>
<th>Error code</th>
<th>ENO state in case of error</th>
<th>Error value in Dec</th>
<th>Error value in Hex</th>
<th>Error description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CFG_CP_F</td>
<td>BAD_REVISION</td>
<td>F</td>
<td>-30200</td>
<td>16#8A08</td>
<td>defined as E_EFB_USER_ERROR_1</td>
</tr>
<tr>
<td>CFG_CP_F</td>
<td>MMF_BAD_4X</td>
<td>T</td>
<td>9010</td>
<td>16#2332</td>
<td></td>
</tr>
<tr>
<td>CFG_CP_V</td>
<td>BAD_REVISION</td>
<td>F</td>
<td>-30200</td>
<td>16#8A08</td>
<td>defined as E_EFB_USER_ERROR_1</td>
</tr>
<tr>
<td>CFG_CP_V</td>
<td>MMF_BAD_4X</td>
<td>T</td>
<td>9010</td>
<td>16#2332</td>
<td></td>
</tr>
<tr>
<td>CFG_CS</td>
<td>BAD_REVISION</td>
<td>F</td>
<td>-30200</td>
<td>16#8A08</td>
<td>defined as E_EFB_USER_ERROR_1</td>
</tr>
<tr>
<td>CFG_CS</td>
<td>MMF_ABORT_SUB</td>
<td>T</td>
<td>7004</td>
<td>16#1B5C</td>
<td>SubNum/SubNumEcho handshake error</td>
</tr>
<tr>
<td>CFG_FS</td>
<td>BAD_REVISION</td>
<td>F</td>
<td>-30200</td>
<td>16#8A08</td>
<td>defined as E_EFB_USER_ERROR_1</td>
</tr>
<tr>
<td>CFG_FS</td>
<td>MMF_ABORT_SUB</td>
<td>T</td>
<td>7004</td>
<td>16#1B5C</td>
<td>SubNum/SubNumEcho handshake error</td>
</tr>
<tr>
<td>CFG_RA</td>
<td>BAD_REVISION</td>
<td>F</td>
<td>-30200</td>
<td>16#8A08</td>
<td>defined as E_EFB_USER_ERROR_1</td>
</tr>
<tr>
<td>CFG_RA</td>
<td>MMF_ABORT_SUB</td>
<td>T</td>
<td>7004</td>
<td>16#1B5C</td>
<td>SubNum/SubNumEcho handshake error</td>
</tr>
<tr>
<td>CFG_SA</td>
<td>BAD_REVISION</td>
<td>F</td>
<td>-30200</td>
<td>16#8A08</td>
<td>defined as E_EFB_USER_ERROR_1</td>
</tr>
<tr>
<td>CFG_SA</td>
<td>MMF_ABORT_SUB</td>
<td>T</td>
<td>7004</td>
<td>16#1B5C</td>
<td>SubNum/SubNumEcho handshake error</td>
</tr>
<tr>
<td>DRV_DNLD</td>
<td>BAD_REVISION</td>
<td>F</td>
<td>-30200</td>
<td>16#8A08</td>
<td>defined as E_EFB_USER_ERROR_1</td>
</tr>
<tr>
<td>DRV_DNLD</td>
<td>MMF_ABORT_SUB</td>
<td>T</td>
<td>7004</td>
<td>16#1B5C</td>
<td>SubNum/SubNumEcho handshake error</td>
</tr>
<tr>
<td>DRV_UPLD</td>
<td>BAD_REVISION</td>
<td>F</td>
<td>-30200</td>
<td>16#8A08</td>
<td>defined as E_EFB_USER_ERROR_1</td>
</tr>
<tr>
<td>DRV_UPLD</td>
<td>MMF_ABORT_SUB</td>
<td>T</td>
<td>7004</td>
<td>16#1B5C</td>
<td>SubNum/SubNumEcho handshake error</td>
</tr>
<tr>
<td>IDN_CHK</td>
<td>BAD_REVISION</td>
<td>F</td>
<td>-30200</td>
<td>16#8A08</td>
<td>defined as E_EFB_USER_ERROR_1</td>
</tr>
<tr>
<td>IDN_CHK</td>
<td>MMF_ABORT_SUB</td>
<td>T</td>
<td>7004</td>
<td>16#1B5C</td>
<td>SubNum/SubNumEcho handshake error</td>
</tr>
<tr>
<td>IDN_XFER</td>
<td>BAD_REVISION</td>
<td>F</td>
<td>-30200</td>
<td>16#8A08</td>
<td>defined as E_EFB_USER_ERROR_1</td>
</tr>
<tr>
<td>IDN_XFER</td>
<td>MMF_ABORT_SUB</td>
<td>T</td>
<td>7004</td>
<td>16#1B5C</td>
<td>SubNum/SubNumEcho handshake error</td>
</tr>
</tbody>
</table>
## EFB Error Codes and Values

<table>
<thead>
<tr>
<th>EFB name</th>
<th>Error code</th>
<th>ENO state in case of error</th>
<th>Error value in Dec</th>
<th>Error value in Hex</th>
<th>Error description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MMF_BITS</td>
<td>BAD_REVISION</td>
<td>F</td>
<td>-30200</td>
<td>16#8A08</td>
<td>defined as E_EFB_USER_ERROR_1</td>
</tr>
<tr>
<td>MMF_ESUB</td>
<td>BAD_REVISION</td>
<td>F</td>
<td>-30200</td>
<td>16#8A08</td>
<td>defined as E_EFB_USER_ERROR_1</td>
</tr>
<tr>
<td>MMF_ESUB</td>
<td>MMF_ABORT_SUB</td>
<td>T</td>
<td>7004</td>
<td>16#1B5C</td>
<td>SubNum/SubNumEcho handshake error</td>
</tr>
<tr>
<td>MMF_IDNX</td>
<td>BAD_REVISION</td>
<td>F</td>
<td>-30200</td>
<td>16#8A08</td>
<td>defined as E_EFB_USER_ERROR_1</td>
</tr>
<tr>
<td>MMF_IDNX</td>
<td>MMF_ABORT_SUB</td>
<td>T</td>
<td>7004</td>
<td>16#1B5C</td>
<td>SubNum/SubNumEcho handshake error</td>
</tr>
<tr>
<td>MMF_JOG</td>
<td>BAD_REVISION</td>
<td>F</td>
<td>-30200</td>
<td>16#8A08</td>
<td>defined as E_EFB_USER_ERROR_1</td>
</tr>
<tr>
<td>MMF_JOG</td>
<td>MMF_ABORT_SUB</td>
<td>T</td>
<td>7004</td>
<td>16#1B5C</td>
<td>SubNum/SubNumEcho handshake error</td>
</tr>
<tr>
<td>MMF_JOG</td>
<td>MMF_SUB_TIMEOUT</td>
<td>T</td>
<td>7005</td>
<td>16#1B5D</td>
<td>Subroutine does not complete in time</td>
</tr>
<tr>
<td>MMF_MOVE</td>
<td>BAD_REVISION</td>
<td>F</td>
<td>-30200</td>
<td>16#8A08</td>
<td>defined as E_EFB_USER_ERROR_1</td>
</tr>
<tr>
<td>MMF_MOVE</td>
<td>MMF_ABORT_SUB</td>
<td>T</td>
<td>7004</td>
<td>16#1B5C</td>
<td>SubNum/SubNumEcho handshake error</td>
</tr>
<tr>
<td>MMF_RST</td>
<td>BAD_REVISION</td>
<td>F</td>
<td>-30200</td>
<td>16#8A08</td>
<td>defined as E_EFB_USER_ERROR_1</td>
</tr>
<tr>
<td>MMF_SUB</td>
<td>BAD_REVISION</td>
<td>F</td>
<td>-30200</td>
<td>16#8A08</td>
<td>defined as E_EFB_USER_ERROR_1</td>
</tr>
<tr>
<td>MMF_SUB</td>
<td>MMF_ABORT_SUB</td>
<td>T</td>
<td>7004</td>
<td>16#1B5C</td>
<td>SubNum/SubNumEcho handshake error</td>
</tr>
<tr>
<td>MMF_USUB</td>
<td>BAD_REVISION</td>
<td>F</td>
<td>-30200</td>
<td>16#8A08</td>
<td>defined as E_EFB_USER_ERROR_1</td>
</tr>
<tr>
<td>MMF_USUB</td>
<td>MMF_ABORT_SUB</td>
<td>T</td>
<td>7004</td>
<td>16#1B5C</td>
<td>SubNum/SubNumEcho handshake error</td>
</tr>
</tbody>
</table>

**Note:** For details about MMF error codes and error values, please refer to the Faults and Error Reporting (see Unity Pro, Drive control, Block Library) description in the Motion Library.
# Tables of Error Codes for the Obsolete Library

## Introduction

The following tables show the error codes and error values created for the EFBs of the Obsolete Library.

## CLC

Table of error codes and error values created for EFBs of the CLC family.

<table>
<thead>
<tr>
<th>EFB name</th>
<th>Error code</th>
<th>ENO state in case of error</th>
<th>Error value in Dec</th>
<th>Error value in Hex</th>
<th>Error description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DELAY</td>
<td>E_ERR_DEN</td>
<td>F</td>
<td>-30152</td>
<td>16#8A38</td>
<td>Not a valid floating point number</td>
</tr>
<tr>
<td>INTEGRATOR1</td>
<td>E_ERR_DEN</td>
<td>F</td>
<td>-30152</td>
<td>16#8A38</td>
<td>Not a valid floating point number</td>
</tr>
<tr>
<td>INTEGRATOR1</td>
<td>E_ERR_IB_MAX_MIN</td>
<td>F</td>
<td>-30102</td>
<td>16#8A6A</td>
<td>YMAX &lt; YMIN</td>
</tr>
<tr>
<td>INTEGRATOR1</td>
<td>FP_ERROR</td>
<td>F</td>
<td>-</td>
<td>-</td>
<td>See table Common Floating Point Errors, p. 599</td>
</tr>
<tr>
<td>LAG1</td>
<td>E_ERR_DEN</td>
<td>F</td>
<td>-30152</td>
<td>16#8A38</td>
<td>Not a valid floating point number</td>
</tr>
<tr>
<td>LAG1</td>
<td>FP_ERROR</td>
<td>F</td>
<td>-</td>
<td>-</td>
<td>See table Common Floating Point Errors, p. 599</td>
</tr>
<tr>
<td>LEAD_LAG1</td>
<td>E_ERR_DEN</td>
<td>F</td>
<td>-30152</td>
<td>16#8A38</td>
<td>Not a valid floating point number</td>
</tr>
<tr>
<td>LEAD_LAG1</td>
<td>FP_ERROR</td>
<td>F</td>
<td>-</td>
<td>-</td>
<td>See table Common Floating Point Errors, p. 599</td>
</tr>
<tr>
<td>LIMV</td>
<td>E_ERR_DEN</td>
<td>F</td>
<td>-30152</td>
<td>16#8A38</td>
<td>Not a valid floating point number</td>
</tr>
<tr>
<td>LIMV</td>
<td>E_ERR_AB1_MAX_MIN</td>
<td>F</td>
<td>-30101</td>
<td>16#8A6B</td>
<td>YMAX &lt; YMIN</td>
</tr>
<tr>
<td>LIMV</td>
<td>FP_ERROR</td>
<td>F</td>
<td>-</td>
<td>-</td>
<td>See table Common Floating Point Errors, p. 599</td>
</tr>
<tr>
<td>PI1</td>
<td>E_ERR_DEN</td>
<td>F</td>
<td>-30152</td>
<td>16#8A38</td>
<td>Not a valid floating point number</td>
</tr>
<tr>
<td>PI1</td>
<td>E_ERR_PI_MAX_MIN</td>
<td>F</td>
<td>-30103</td>
<td>16#8A69</td>
<td>YMAX &lt; YMIN</td>
</tr>
<tr>
<td>PI1</td>
<td>FP_ERROR</td>
<td>F</td>
<td>-</td>
<td>-</td>
<td>See table Common Floating Point Errors, p. 599</td>
</tr>
<tr>
<td>PID1</td>
<td>E_ERR_DEN</td>
<td>F</td>
<td>-30152</td>
<td>16#8A38</td>
<td>Not a valid floating point number</td>
</tr>
<tr>
<td>PID1</td>
<td>E_ERR_PID_MAX_MIN</td>
<td>F</td>
<td>-30104</td>
<td>16#8A68</td>
<td>YMAX &lt; YMIN</td>
</tr>
<tr>
<td>PID1</td>
<td>FP_ERROR</td>
<td>F</td>
<td>-</td>
<td>-</td>
<td>See table Common Floating Point Errors, p. 599</td>
</tr>
<tr>
<td>PIDP1</td>
<td>E_ERR_DEN</td>
<td>F</td>
<td>-30152</td>
<td>16#8A38</td>
<td>Not a valid floating point number</td>
</tr>
<tr>
<td>PIDP1</td>
<td>E_ERR_PID_MAX_MIN</td>
<td>F</td>
<td>-30104</td>
<td>16#8A68</td>
<td>YMAX &lt; YMIN</td>
</tr>
<tr>
<td>EFB name</td>
<td>Error code</td>
<td>ENO state in case of error</td>
<td>Error value in Dec</td>
<td>Error value in Hex</td>
<td>Error description</td>
</tr>
<tr>
<td>------------------</td>
<td>------------</td>
<td>----------------------------</td>
<td>--------------------</td>
<td>--------------------------</td>
<td>--------------------------------------------------------</td>
</tr>
<tr>
<td>PIDP1</td>
<td>FP_ERROR</td>
<td>F</td>
<td>-</td>
<td>-</td>
<td>See table Common Floating Point Errors, p. 599</td>
</tr>
<tr>
<td>SMOOTH_RATE</td>
<td>E_ERR_DEN</td>
<td>F</td>
<td>-30152</td>
<td>16#8A38</td>
<td>Not a valid floating point number</td>
</tr>
<tr>
<td>SMOOTH_RATE</td>
<td>FP_ERROR</td>
<td>F</td>
<td>-</td>
<td>-</td>
<td>See table Common Floating Point Errors, p. 599</td>
</tr>
<tr>
<td>THREE_STEP_CON1</td>
<td>E_ERR_DEN</td>
<td>F</td>
<td>-30152</td>
<td>16#8A38</td>
<td>Not a valid floating point number</td>
</tr>
<tr>
<td>THREE_STEP_CON1</td>
<td>W_WARN_DSR_TN</td>
<td>T</td>
<td>30101</td>
<td>16#7595</td>
<td>TN = 0</td>
</tr>
<tr>
<td>THREE_STEP_CON1</td>
<td>W_WARN_DSR_TSN</td>
<td>T</td>
<td>30102</td>
<td>16#7596</td>
<td>TSN = 0</td>
</tr>
<tr>
<td>THREE_STEP_CON1</td>
<td>W_WARN_DSR_KP</td>
<td>T</td>
<td>30103</td>
<td>16#7597</td>
<td>KP &lt;= 0</td>
</tr>
<tr>
<td>THREE_STEP_CON1</td>
<td>E_ERR_DSR_HYS</td>
<td>F</td>
<td>-30152</td>
<td>16#8A67</td>
<td>2 *</td>
</tr>
<tr>
<td>THREEPOINT_CON1</td>
<td>E_ERR_ERROR</td>
<td>F</td>
<td>-</td>
<td>-</td>
<td>See table Common Floating Point Errors, p. 599</td>
</tr>
<tr>
<td>THREEPOINT_CON1</td>
<td>W_WARN_ZDR_XRR</td>
<td>F</td>
<td>30105</td>
<td>16#7599</td>
<td>DR: XRR &lt; -100 or XRR &gt; 100</td>
</tr>
<tr>
<td>THREEPOINT_CON1</td>
<td>W_WARN_ZDR_T1T2</td>
<td>F</td>
<td>30104</td>
<td>16#7598</td>
<td>T2 &gt; T1</td>
</tr>
<tr>
<td>THREEPOINT_CON1</td>
<td>E_ERR_ERROR</td>
<td>F</td>
<td>-</td>
<td>-</td>
<td>See table Common Floating Point Errors, p. 599</td>
</tr>
<tr>
<td>THREEPOINT_CON1</td>
<td>E_ERR_ZDR_HYS</td>
<td>F</td>
<td>-30166</td>
<td>16#8A66</td>
<td>2 *</td>
</tr>
<tr>
<td>TWOPOINT_CON1</td>
<td>E_ERR_DEN</td>
<td>F</td>
<td>-30152</td>
<td>16#8A38</td>
<td>Not a valid floating point number</td>
</tr>
<tr>
<td>TWOPOINT_CON1</td>
<td>W_WARN_ZDR_XRR</td>
<td>F</td>
<td>30105</td>
<td>16#7599</td>
<td>DR: XRR &lt; -100 or XRR &gt; 100</td>
</tr>
<tr>
<td>TWOPOINT_CON1</td>
<td>W_WARN_ZDR_T1T2</td>
<td>F</td>
<td>30104</td>
<td>16#7598</td>
<td>T2 &gt; T1</td>
</tr>
<tr>
<td>TWOPOINT_CON1</td>
<td>FP_ERROR</td>
<td>F</td>
<td>-</td>
<td>-</td>
<td>See table Common Floating Point Errors, p. 599</td>
</tr>
<tr>
<td>TWOPOINT_CON1</td>
<td>E_ERR_ZDR_HYS</td>
<td>F</td>
<td>-30106</td>
<td>16#8A66</td>
<td>2 *</td>
</tr>
</tbody>
</table>
## CLC_PRO

Table of error codes and error values created for EFBs of the CLC_PRO family.

<table>
<thead>
<tr>
<th>EFB name</th>
<th>Error code</th>
<th>ENO state in case of error</th>
<th>Error value in Dec</th>
<th>Error value in Hex</th>
<th>Error description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALIM</td>
<td>E_ERR_DEN</td>
<td>F</td>
<td>-30152</td>
<td>16#8A38</td>
<td>Not a valid floating point number</td>
</tr>
<tr>
<td>ALIM</td>
<td>WAF_AB2_VMAX</td>
<td>F</td>
<td>-30111</td>
<td>16#8A61</td>
<td>vmax &lt;= 0</td>
</tr>
<tr>
<td>ALIM</td>
<td>WAF_AB2_BMAX</td>
<td>F</td>
<td>-30112</td>
<td>16#8A60</td>
<td>bmax &lt;= 0</td>
</tr>
<tr>
<td>ALIM</td>
<td>FP_ERROR</td>
<td>F</td>
<td>-</td>
<td>-</td>
<td>See table Common Floating Point Errors, p. 599</td>
</tr>
<tr>
<td>COMP_PID</td>
<td>E_ERR_DEN</td>
<td>F</td>
<td>-30152</td>
<td>16#8A38</td>
<td>Not a valid floating point number</td>
</tr>
<tr>
<td>COMP_PID</td>
<td>WAF_KPID_KUZ</td>
<td>F</td>
<td>-30110</td>
<td>16#8A62</td>
<td>gain_red &lt; 0 or gain_red &gt; 1</td>
</tr>
<tr>
<td>COMP_PID</td>
<td>WAF_KPID_OGUG</td>
<td>F</td>
<td>-30104</td>
<td>16#8A68</td>
<td>YMAX &lt; YMIN</td>
</tr>
<tr>
<td>COMP_PID</td>
<td>WAF_KPID_UZ</td>
<td>F</td>
<td>-30109</td>
<td>16#8A63</td>
<td>db &lt; 0</td>
</tr>
<tr>
<td>COMP_PID</td>
<td>FP_ERROR</td>
<td>F</td>
<td>-</td>
<td>-</td>
<td>See table Common Floating Point Errors, p. 599</td>
</tr>
<tr>
<td>DEADTIME</td>
<td>E_ERR_DEN</td>
<td>F</td>
<td>-30152</td>
<td>16#8A38</td>
<td>Not a valid floating point number</td>
</tr>
<tr>
<td>DERIV</td>
<td>E_ERR_DEN</td>
<td>F</td>
<td>-30152</td>
<td>16#8A38</td>
<td>Not a valid floating point number</td>
</tr>
<tr>
<td>DERIV</td>
<td>FP_ERROR</td>
<td>F</td>
<td>-</td>
<td>-</td>
<td>See table Common Floating Point Errors, p. 599</td>
</tr>
<tr>
<td>FGEN</td>
<td>E_ERR_DEN</td>
<td>F</td>
<td>-30152</td>
<td>16#8A38</td>
<td>Not a valid floating point number</td>
</tr>
<tr>
<td>FGEN</td>
<td>WAF_SIG_TV_MAX</td>
<td>F</td>
<td>-30116</td>
<td>16#8A5C</td>
<td>t_acc &gt; t_rise / 2</td>
</tr>
<tr>
<td>FGEN</td>
<td>WAF_SIG_TH_MAX</td>
<td>F</td>
<td>-30117</td>
<td>16#8A5B</td>
<td>t_rise too big</td>
</tr>
<tr>
<td>FGEN</td>
<td>WAF_SIG_TA_MAX</td>
<td>T</td>
<td>30106</td>
<td>16#759A</td>
<td>t_off &gt;= halfperiod</td>
</tr>
<tr>
<td>FGEN</td>
<td>WAF_SIG_T1_MIN</td>
<td>T</td>
<td>30107</td>
<td>16#759B</td>
<td>t_max &lt;= t_min</td>
</tr>
<tr>
<td>FGEN</td>
<td>WAF_SIG_FKT</td>
<td>F</td>
<td>-30118</td>
<td>16#8A5A</td>
<td>func_no &lt;= 0 or func_no &gt; 8</td>
</tr>
<tr>
<td>FGEN</td>
<td>FP_ERROR</td>
<td>F</td>
<td>-</td>
<td>-</td>
<td>See table Common Floating Point Errors, p. 599</td>
</tr>
<tr>
<td>INTEG</td>
<td>E_ERR_DEN</td>
<td>F</td>
<td>-30152</td>
<td>16#8A38</td>
<td>Not a valid floating point number</td>
</tr>
<tr>
<td>INTEG</td>
<td>E_ERR_IB_MAX_MIN</td>
<td>F</td>
<td>-30102</td>
<td>16#8A6A</td>
<td>YMAX &lt; YMIN</td>
</tr>
<tr>
<td>INTEG</td>
<td>FP_ERROR</td>
<td>F</td>
<td>-</td>
<td>-</td>
<td>See table Common Floating Point Errors, p. 599</td>
</tr>
<tr>
<td>LAG</td>
<td>E_ERR_DEN</td>
<td>F</td>
<td>-30152</td>
<td>16#8A38</td>
<td>Not a valid floating point number</td>
</tr>
<tr>
<td>LAG</td>
<td>FP_ERROR</td>
<td>F</td>
<td>-</td>
<td>-</td>
<td>See table Common Floating Point Errors, p. 599</td>
</tr>
<tr>
<td>LAG2</td>
<td>E_ERR_DEN</td>
<td>F</td>
<td>-30152</td>
<td>16#8A38</td>
<td>Not a valid floating point number</td>
</tr>
<tr>
<td>EFB name</td>
<td>Error code</td>
<td>ENO state in case of error</td>
<td>Error value in Dec</td>
<td>Error value in Hex</td>
<td>Error description</td>
</tr>
<tr>
<td>----------</td>
<td>--------------</td>
<td>----------------------------</td>
<td>--------------------</td>
<td>--------------------</td>
<td>----------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>LAG2</td>
<td>FP_ERROR</td>
<td>F</td>
<td>-</td>
<td>-</td>
<td>See table Common Floating Point Errors, p. 599</td>
</tr>
<tr>
<td>LEAD_LAG</td>
<td>E_ERR_DEN</td>
<td>F</td>
<td>-30152</td>
<td>16#8A38</td>
<td>Not a valid floating point number</td>
</tr>
<tr>
<td>LEAD_LAG</td>
<td>FP_ERROR</td>
<td>F</td>
<td>-</td>
<td>-</td>
<td>See table Common Floating Point Errors, p. 599</td>
</tr>
<tr>
<td>PCON2</td>
<td>E_ERR_DEN</td>
<td>F</td>
<td>-30152</td>
<td>16#8A38</td>
<td>Not a valid floating point number</td>
</tr>
<tr>
<td>PCON2</td>
<td>W_WARN_ZDR_XRR</td>
<td>T</td>
<td>30105</td>
<td>16#7599</td>
<td>DR: XRR &lt; -100 or XRR &gt; 100</td>
</tr>
<tr>
<td>PCON2</td>
<td>W_WARN_ZDR_T1T2</td>
<td>T</td>
<td>30104</td>
<td>16#7598</td>
<td>T2 &gt; T1</td>
</tr>
<tr>
<td>PCON2</td>
<td>FP_ERROR</td>
<td>F</td>
<td>-</td>
<td>-</td>
<td>See table Common Floating Point Errors, p. 599</td>
</tr>
<tr>
<td>PCON3</td>
<td>E_ERR_DEN</td>
<td>F</td>
<td>-30152</td>
<td>16#8A38</td>
<td>Not a valid floating point number</td>
</tr>
<tr>
<td>PCON3</td>
<td>W_WARN_ZDR_XRR</td>
<td>T</td>
<td>30105</td>
<td>16#7599</td>
<td>DR: XRR &lt; -100 or XRR &gt; 100</td>
</tr>
<tr>
<td>PCON3</td>
<td>W_WARN_ZDR_T1T2</td>
<td>T</td>
<td>30104</td>
<td>16#7598</td>
<td>T2 &gt; T1</td>
</tr>
<tr>
<td>PCON3</td>
<td>FP_ERROR</td>
<td>F</td>
<td>-</td>
<td>-</td>
<td>See table Common Floating Point Errors, p. 599</td>
</tr>
<tr>
<td>PD_OR_PI</td>
<td>E_ERR_DEN</td>
<td>F</td>
<td>-30152</td>
<td>16#8A38</td>
<td>Not a valid floating point number</td>
</tr>
<tr>
<td>PD_OR_PI</td>
<td>WAF_PDPI_OG_UG</td>
<td>F</td>
<td>-30103</td>
<td>16#8A69</td>
<td>YMAX &lt; YMIN</td>
</tr>
<tr>
<td>PD_OR_PI</td>
<td>FP_ERROR</td>
<td>F</td>
<td>-</td>
<td>-</td>
<td>See table Common Floating Point Errors, p. 599</td>
</tr>
<tr>
<td>PDM</td>
<td>PDM_TMAX_TMIN</td>
<td>F</td>
<td>-30115</td>
<td>16#8A5D</td>
<td>t_max &lt;= t_min</td>
</tr>
<tr>
<td>PDM</td>
<td>PDM_OG_UG</td>
<td>F</td>
<td>-30114</td>
<td>16#8A69</td>
<td></td>
</tr>
<tr>
<td>PDM</td>
<td>FP_ERROR</td>
<td>F</td>
<td>-</td>
<td>-</td>
<td>See table Common Floating Point Errors, p. 599</td>
</tr>
<tr>
<td>PI</td>
<td>E_ERR_DEN</td>
<td>F</td>
<td>-30152</td>
<td>16#8A38</td>
<td>Not a valid floating point number</td>
</tr>
<tr>
<td>PI</td>
<td>E_ERR_PI_MAX_MIN</td>
<td>F</td>
<td>-30103</td>
<td>16#8A69</td>
<td>YMAX &lt; YMIN</td>
</tr>
<tr>
<td>PI</td>
<td>FP_ERROR</td>
<td>F</td>
<td>-</td>
<td>-</td>
<td>See table Common Floating Point Errors, p. 599</td>
</tr>
<tr>
<td>PID</td>
<td>E_ERR_DEN</td>
<td>F</td>
<td>-30152</td>
<td>16#8A38</td>
<td>Not a valid floating point number</td>
</tr>
<tr>
<td>PID</td>
<td>E_ERR_PID_MAX_MIN</td>
<td>F</td>
<td>-30104</td>
<td>16#8A69</td>
<td>YMAX &lt; YMIN</td>
</tr>
<tr>
<td>PID</td>
<td>FP_ERROR</td>
<td>F</td>
<td>-</td>
<td>-</td>
<td>See table Common Floating Point Errors, p. 599</td>
</tr>
<tr>
<td>EFB name</td>
<td>Error code</td>
<td>ENO state in case of error</td>
<td>Error value in Dec</td>
<td>Error value in Hex</td>
<td>Error description</td>
</tr>
<tr>
<td>----------</td>
<td>------------------</td>
<td>----------------------------</td>
<td>--------------------</td>
<td>--------------------</td>
<td>----------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>PID_P</td>
<td>E_ERR_DEN</td>
<td>F</td>
<td>-30152</td>
<td>16#8A38</td>
<td>Not a valid floating point number</td>
</tr>
<tr>
<td>PID_P</td>
<td>E_ERR_PID_MAX_MIN</td>
<td>F</td>
<td>-30104</td>
<td>16#8A68</td>
<td>YMAX &lt; YMIN</td>
</tr>
<tr>
<td>PID_P</td>
<td>FP_ERROR</td>
<td>F</td>
<td>-</td>
<td>-</td>
<td>See table Common Floating Point Errors, p. 599</td>
</tr>
<tr>
<td>PIP</td>
<td>E_ERR_DEN</td>
<td>F</td>
<td>-30152</td>
<td>16#8A38</td>
<td>Not a valid floating point number</td>
</tr>
<tr>
<td>PIP</td>
<td>E_ERR_PI_MAX_MIN</td>
<td>F</td>
<td>-30103</td>
<td>16#8A69</td>
<td>YMAX &lt; YMIN</td>
</tr>
<tr>
<td>PPI</td>
<td>E_ERR_DEN</td>
<td>F</td>
<td>-30152</td>
<td>16#8A38</td>
<td>Not a valid floating point number</td>
</tr>
<tr>
<td>PPI</td>
<td>E_ERR_PI_MAX_MIN</td>
<td>F</td>
<td>-30103</td>
<td>16#8A69</td>
<td>YMAX &lt; YMIN</td>
</tr>
<tr>
<td>PPI</td>
<td>FP_ERROR</td>
<td>F</td>
<td>-</td>
<td>-</td>
<td>See table Common Floating Point Errors, p. 599</td>
</tr>
<tr>
<td>PWM</td>
<td>WAF_PBM_TMINMAX</td>
<td>F</td>
<td>-30113</td>
<td>16#8A5F</td>
<td>t_min &lt; t_max</td>
</tr>
<tr>
<td>PWM</td>
<td>FP_ERROR</td>
<td>F</td>
<td>-</td>
<td>-</td>
<td>See table Common Floating Point Errors, p. 599</td>
</tr>
<tr>
<td>QPWM</td>
<td>WAF_PBM_TMINMAX</td>
<td>F</td>
<td>-30113</td>
<td>16#8A5F</td>
<td>t_min &lt; t_max</td>
</tr>
<tr>
<td>QPWM</td>
<td>FP_ERROR</td>
<td>F</td>
<td>-</td>
<td>-</td>
<td>See table Common Floating Point Errors, p. 599</td>
</tr>
<tr>
<td>SCON3</td>
<td>E_ERR_DEN</td>
<td>F</td>
<td>-30152</td>
<td>16#8A38</td>
<td>Not a valid floating point number</td>
</tr>
<tr>
<td>SCON3</td>
<td>W_WARN_DSR_TN</td>
<td>T</td>
<td>30101</td>
<td>16#7595</td>
<td>TN = 0</td>
</tr>
<tr>
<td>SCON3</td>
<td>W_WARN_DSR_TSN</td>
<td>T</td>
<td>30102</td>
<td>16#7596</td>
<td>TSN = 0</td>
</tr>
<tr>
<td>SCON3</td>
<td>W_WARN_DSR_KP</td>
<td>T</td>
<td>30103</td>
<td>16#7597</td>
<td>KP &lt;= 0</td>
</tr>
<tr>
<td>SCON3</td>
<td>E_ERR_DSR_HYS</td>
<td>F</td>
<td>-30105</td>
<td>16#8A67</td>
<td>2 *</td>
</tr>
<tr>
<td>SCON3</td>
<td>FP_ERROR</td>
<td>F</td>
<td>-</td>
<td>-</td>
<td>See table Common Floating Point Errors, p. 599</td>
</tr>
<tr>
<td>VLIM</td>
<td>E_ERR_DEN</td>
<td>F</td>
<td>-30152</td>
<td>16#8A38</td>
<td>Not a valid floating point number</td>
</tr>
<tr>
<td>VLIM</td>
<td>E_ERR_AB1_MAX_MIN</td>
<td>F</td>
<td>-30101</td>
<td>16#8A6B</td>
<td>YMAX &lt; YMIN</td>
</tr>
<tr>
<td>VLIM</td>
<td>FP_ERROR</td>
<td>F</td>
<td>-</td>
<td>-</td>
<td>See table Common Floating Point Errors, p. 599</td>
</tr>
</tbody>
</table>
## Extension/ Compatibility

Table of error codes and errors values created for EFBs of the Extension/ Compatibility family.

<table>
<thead>
<tr>
<th>EFB name</th>
<th>Error code</th>
<th>ENO state in case of error</th>
<th>Error value in Dec</th>
<th>Error value in Hex</th>
<th>Error description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AKF_TA</td>
<td>E_AKFEFB_TIMEBASE_IS_ZERO</td>
<td>F</td>
<td>-30482</td>
<td>16#88EE</td>
<td>Time base is zero</td>
</tr>
<tr>
<td>AKF_TE</td>
<td>E_AKFEFB_TIMEBASE_IS_ZERO</td>
<td>F</td>
<td>-30482</td>
<td>16#88EE</td>
<td>Time base is zero</td>
</tr>
<tr>
<td>AKF_TI</td>
<td>E_AKFEFB_TIMEBASE_IS_ZERO</td>
<td>F</td>
<td>-30482</td>
<td>16#88EE</td>
<td>Time base is zero</td>
</tr>
<tr>
<td>AKF_TS</td>
<td>E_AKFEFB_TIMEBASE_IS_ZERO</td>
<td>F</td>
<td>-30482</td>
<td>16#88EE</td>
<td>Time base is zero</td>
</tr>
<tr>
<td>AKF_TV</td>
<td>E_AKFEFB_TIMEBASE_IS_ZERO</td>
<td>F</td>
<td>-30482</td>
<td>16#88EE</td>
<td>Time base is zero</td>
</tr>
<tr>
<td>FIFO</td>
<td>E_INPUT_VALUE_OUT_OF_RANGE</td>
<td>F</td>
<td>-30183</td>
<td>16#8A19</td>
<td>Input value is out of range</td>
</tr>
<tr>
<td>GET_3X</td>
<td>E_INPUT_VALUE_OUT_OF_RANGE</td>
<td>F</td>
<td>-30183</td>
<td>16#8A19</td>
<td>Input value is out of range</td>
</tr>
<tr>
<td>GET_4X</td>
<td>E_INPUT_VALUE_OUT_OF_RANGE</td>
<td>F</td>
<td>-30183</td>
<td>16#8A19</td>
<td>Input value is out of range</td>
</tr>
<tr>
<td>GET_BIT</td>
<td>E_INPUT_VALUE_OUT_OF_RANGE</td>
<td>F</td>
<td>-30183</td>
<td>16#8A19</td>
<td>Input value is out of range</td>
</tr>
<tr>
<td>IEC_BMDI</td>
<td>E_EFB_USER_ERROR_1</td>
<td>F</td>
<td>-30200</td>
<td>16#8A08</td>
<td>Input value is invalid register type (SourceTable).</td>
</tr>
<tr>
<td>IEC_BMDI</td>
<td>E_EFB_USER_ERROR_2</td>
<td>F</td>
<td>-30201</td>
<td>16#8A07</td>
<td>The input offset (OffsetInSourceTable) selects an address outside acceptable limits.</td>
</tr>
<tr>
<td>IEC_BMDI</td>
<td>E_EFB_USER_ERROR_3</td>
<td>F</td>
<td>-30202</td>
<td>16#8A06</td>
<td>The input offset (OFF_IN) is not 1 or a multiple of 16+1.</td>
</tr>
<tr>
<td>IEC_BMDI</td>
<td>E_EFB_USER_ERROR_4</td>
<td>F</td>
<td>-30203</td>
<td>16#8A05</td>
<td>Output value is invalid register type (DestinationTable).</td>
</tr>
<tr>
<td>IEC_BMDI</td>
<td>E_EFB_USER_ERROR_5</td>
<td>F</td>
<td>-30204</td>
<td>16#8A04</td>
<td>The output offset (OffsetInDestinationTable) selects an address outside acceptable limits.</td>
</tr>
<tr>
<td>EFB name</td>
<td>Error code</td>
<td>ENO state in case of error</td>
<td>Error value in Dec</td>
<td>Error value in Hex</td>
<td>Error description</td>
</tr>
<tr>
<td>------------</td>
<td>-----------------------------</td>
<td>-----------------------------</td>
<td>--------------------</td>
<td>-------------------</td>
<td>-----------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>IEC_BMDI</td>
<td>E_EFB_USER_ERROR_6</td>
<td>F</td>
<td>-30205</td>
<td>16#8A03</td>
<td>The output offset (OffsetInDestinationTable) is not 1 or a multiple of 16+1.</td>
</tr>
<tr>
<td>IEC_BMDI</td>
<td>E_EFB_USER_ERROR_7</td>
<td>F</td>
<td>-30206</td>
<td>16#8A02</td>
<td>The value for (NumberOfElements) is 0.</td>
</tr>
<tr>
<td>IEC_BMDI</td>
<td>E_EFB_USER_ERROR_8</td>
<td>F</td>
<td>-30207</td>
<td>16#8A01</td>
<td>The value for (NumberOfElements) addresses more than 1600 bits.</td>
</tr>
<tr>
<td>IEC_BMDI</td>
<td>E_EFB_USER_ERROR_9</td>
<td>F</td>
<td>-30208</td>
<td>16#8A00</td>
<td>The value for (NumberOfElements) addresses more than 100 words.</td>
</tr>
<tr>
<td>IEC_BMDI</td>
<td>E_EFB_USER_ERROR_10</td>
<td>F</td>
<td>-30209</td>
<td>16#89FF</td>
<td>The value for (NumberOfElements) selects a source address outside the acceptable limits.</td>
</tr>
<tr>
<td>IEC_BMDI</td>
<td>E_EFB_USER_ERROR_11</td>
<td>F</td>
<td>-30210</td>
<td>16#89FE</td>
<td>The value for (NumberOfElements) selects a destination address outside the acceptable limits.</td>
</tr>
<tr>
<td>IEC_BMDI</td>
<td>E_EFB_USER_ERROR_12</td>
<td>F</td>
<td>-30211</td>
<td>16#89FD</td>
<td>The value for (NumberOfElements) is not a multiple of 16.</td>
</tr>
<tr>
<td>IEC_BMDI</td>
<td>E_EFB_USER_ERROR_13</td>
<td>F</td>
<td>-30212</td>
<td>16#89FC</td>
<td>Warning: Address overlap of input and output addresses.</td>
</tr>
<tr>
<td>LIFO</td>
<td>E_INPUT_VALUE_OUT_OF_RANGE</td>
<td>F</td>
<td>-30183</td>
<td>16#8A19</td>
<td>Input value is out of range</td>
</tr>
<tr>
<td>PUT_4X</td>
<td>E_INPUT_VALUE_OUT_OF_RANGE</td>
<td>F</td>
<td>-30183</td>
<td>16#8A19</td>
<td>Input value is out of range</td>
</tr>
<tr>
<td>MUX_DINTARR_125</td>
<td>E_SELECTOR_OUT_OF_RANGE</td>
<td>F</td>
<td>-30175</td>
<td>16#8A21</td>
<td>Selector is out of range</td>
</tr>
<tr>
<td>SET_BIT</td>
<td>E_INPUT_VALUE_OUT_OF_RANGE</td>
<td>F</td>
<td>-30183</td>
<td>16#8A19</td>
<td>Input value is out of range</td>
</tr>
</tbody>
</table>
Common Floating Point Errors

Introduction The following table shows the common error codes and error values created for floating point errors.

<table>
<thead>
<tr>
<th>Error codes</th>
<th>Error value in Dec</th>
<th>Error value in Hex</th>
<th>Error description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP_ERROR</td>
<td>-30150</td>
<td>16#8A3A</td>
<td>Base value (not appearing as an error value)</td>
</tr>
<tr>
<td>E_FP_STATUS_FAILED_IE</td>
<td>-30151</td>
<td>16#8A39</td>
<td>Illegal floating point operation</td>
</tr>
<tr>
<td>E_FP_STATUS_FAILED_DE</td>
<td>-30152</td>
<td>16#8A38</td>
<td>Operand is denormalized - not a valid REAL number</td>
</tr>
<tr>
<td>E_FP_STATUS_FAILED_ZE</td>
<td>-30154</td>
<td>16#8A36</td>
<td>Illegal divide by zero</td>
</tr>
<tr>
<td>E_FP_STATUS_FAILED_ZE_IE</td>
<td>-30155</td>
<td>16#8A35</td>
<td>Illegal floating point operation / Divide by zero</td>
</tr>
<tr>
<td>E_FP_STATUS_FAILED_OE</td>
<td>-30158</td>
<td>16#8A32</td>
<td>Floating point overflow</td>
</tr>
<tr>
<td>E_FP_STATUS_FAILED_OE_IE</td>
<td>-30159</td>
<td>16#8A31</td>
<td>Illegal floating point operation / Overflow</td>
</tr>
<tr>
<td>E_FP_STATUS_FAILED_OE_ZE</td>
<td>-30162</td>
<td>16#8A2E</td>
<td>Floating point overflow / Divide by zero</td>
</tr>
<tr>
<td>E_FP_STATUS_FAILED_OE_ZE_IE</td>
<td>-30163</td>
<td>16#8A2D</td>
<td>Illegal floating point operation / Overflow / Divide by zero</td>
</tr>
<tr>
<td>E_FP_NOT_COMPARABLE</td>
<td>-30166</td>
<td>16#8A2A</td>
<td>Internal error</td>
</tr>
</tbody>
</table>
IEC Compliance

At a Glance

Overview

This chapter contains the compliance tables required by IEC 61131-3.

What’s in this Chapter?

This chapter contains the following sections:

<table>
<thead>
<tr>
<th>Section</th>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>B.1</td>
<td>General Information regarding IEC 61131-3</td>
<td>602</td>
</tr>
<tr>
<td>B.2</td>
<td>IEC Compliance Tables</td>
<td>604</td>
</tr>
<tr>
<td>B.3</td>
<td>Extensions of IEC 61131-3</td>
<td>628</td>
</tr>
<tr>
<td>B.4</td>
<td>Textual language syntax</td>
<td>630</td>
</tr>
</tbody>
</table>
B.1 General Information regarding IEC 61131-3

General information about IEC 61131-3 Compliance

At a Glance

The IEC 61131-3 Standard (cf. its subclause 1.4) specifies the syntax and semantics of a unified suite of programming languages for programmable controllers. These consist of two textual languages, IL (Instruction List) and ST (Structured Text), and two graphical languages, LD (Ladder Diagram) and FBD (Function Block Diagram). Additionally, Sequential Function Chart (SFC) language elements are defined for structuring the internal organization of programmable controller programs and function blocks. Also, configuration elements are defined which support the installation of programmable controller programs into programmable controller systems.

Note: Unity Pro uses the English acronyms for the programming languages.

Furthermore, features are defined which facilitate communication among programmable controllers and other components of automated systems.

Unity Pro compliance to IEC 61131-3

The current version of the Unity Pro Programming System supports a compliant subset of the language elements defined in the Standard.

Compliance in this context means the following:

- The Standard allows an implementer of an IEC Programming System to choose or to drop specific language features or even complete languages out of the Feature Tables which form an inherent part of the specifications; a system claiming compliance to the Standard just has to implement the chosen features according to the specifications given in the Standard.
- Further on, the Standard allows an implementer to use the defined programming language elements in an interactive programming environment. Since the Standard explicitly states that the specification of such environments is beyond its scope, the implementer has certain degrees of freedom in providing optimized presentation and handling procedures for specific language elements to the benefit of the user.
- Unity Pro uses these degrees of freedom e.g. by introducing the notion of "Project" for the combined handling of the IEC language elements "Configuration" and "Resource". It also uses them e.g. in the mechanisms provided for handling variable declarations or function block instantiations.
| IEC standards tables | The supported features and other implementation specific information is given in the following compliance statement and the subsequent tables as required by the Standard. |
IEC Compliance

B.2 IEC Compliance Tables

At a Glance

Overview
This system complies with the requirements of IEC 61131-3 for the language and feature listed in the following tables.

What's in this Section?
This section contains the following topics:

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Common elements</td>
<td>605</td>
</tr>
<tr>
<td>IL language elements</td>
<td>617</td>
</tr>
<tr>
<td>ST language elements</td>
<td>619</td>
</tr>
<tr>
<td>Common graphical elements</td>
<td>620</td>
</tr>
<tr>
<td>LD language elements</td>
<td>621</td>
</tr>
<tr>
<td>Implementation-dependent parameters</td>
<td>622</td>
</tr>
<tr>
<td>Error Conditions</td>
<td>626</td>
</tr>
</tbody>
</table>
### IEC Compliance Table for Common Elements

<table>
<thead>
<tr>
<th>Table No.</th>
<th>Feature No.</th>
<th>Description of Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>Lower case characters</td>
</tr>
<tr>
<td></td>
<td>3a</td>
<td>Number sign (#)</td>
</tr>
<tr>
<td></td>
<td>4a</td>
<td>Dollar sign ($)</td>
</tr>
<tr>
<td></td>
<td>5a</td>
<td>Vertical bar (</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>Upper case and numbers</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>Upper and lower case, numbers, embedded underlines</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>Upper and lower case, numbers, leading or embedded underlines</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>Comments</td>
</tr>
<tr>
<td></td>
<td>3a</td>
<td>Pragmas</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>Integer literals</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>Real literals</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>Real literals with exponents</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>Base 2 literals</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>Base 8 literals</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>Base 16 literals</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>Boolean zero and one</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>Boolean <code>FALSE</code> and <code>TRUE</code></td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>Typed literals</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>Single-byte character strings</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>Single-byte typed string literals</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>Dollar sign</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>Single quote</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>Line feed</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>New line</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>Form feed (page)</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>Carriage return</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>Tab</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>Double quote</td>
</tr>
<tr>
<td>Table No.</td>
<td>Feature No.</td>
<td>Description of Feature</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
<td>-------------------------------------------------------------</td>
</tr>
<tr>
<td>7</td>
<td>1a</td>
<td>Duration literals without underlines: short prefix</td>
</tr>
<tr>
<td></td>
<td>1b</td>
<td>long prefix</td>
</tr>
<tr>
<td></td>
<td>2a</td>
<td>Duration literals with underlines: short prefix</td>
</tr>
<tr>
<td></td>
<td>2b</td>
<td>long prefix</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>Date literals (long prefix)</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>Date literals (short prefix)</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>Time of day literals (long prefix)</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>Time of day literals (short prefix)</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>Date and time literals (long prefix)</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>Date and time literals (short prefix)</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>Data type BOOL</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>Data type INT</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>Data type DINT</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>Data type UINT</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>Data type UDINT</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>Data type REAL</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>Data type TIME</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>Data type DATE</td>
</tr>
<tr>
<td></td>
<td>14</td>
<td>Data type TIME_OF_DAY or TOD</td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>Data type DATE_AND_TIME or DT</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>Data type STRING</td>
</tr>
<tr>
<td></td>
<td>17</td>
<td>Data type BYTE</td>
</tr>
<tr>
<td></td>
<td>18</td>
<td>Data type WORD</td>
</tr>
<tr>
<td></td>
<td>19</td>
<td>Data type DWORD</td>
</tr>
<tr>
<td>12</td>
<td>4</td>
<td>Array data types</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>Structured data types</td>
</tr>
<tr>
<td>14</td>
<td>4</td>
<td>Initialization of array data types</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>Initialization of derived structured data types</td>
</tr>
<tr>
<td>Table No.</td>
<td>Feature No.</td>
<td>Description of Feature</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
<td>------------------------</td>
</tr>
<tr>
<td>15</td>
<td>1</td>
<td>Input location</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>Output location</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>Memory location</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>Single bit size (X Prefix)</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>Single bit size (No Prefix)</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>Word (16 bits) size</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>Double word (32 bits) size</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>Long (quad) word (64 bits) size</td>
</tr>
<tr>
<td>17</td>
<td>3</td>
<td>Declaration of locations of symbolic variables (Note 5, p. 614)</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>Array location assignment (Note 5, p. 614)</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>Automatic memory allocation of symbolic variables</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>Array declaration (Note 11, p. 616)</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>Retentive array declaration (Note 11, p. 616)</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>Declaration for structured variables</td>
</tr>
<tr>
<td>18</td>
<td>1</td>
<td>Initialization of directly represented variables (Note 11, p. 616)</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>Location and initial value assignment to symbolic variables</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>Array location assignment and initialization</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>Initialization of symbolic variables</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>Array initialization (Note 11, p. 616)</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>Retentive array declaration and initialization (Note 11, p. 616)</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>Initialization of structured variables</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>Initialization of constants</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>Initialization of function block instances</td>
</tr>
<tr>
<td>19</td>
<td>1</td>
<td>Negated input</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>Negated output</td>
</tr>
<tr>
<td>19a</td>
<td>1</td>
<td>Formal function / function block call</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>Non-formal function / function block call</td>
</tr>
<tr>
<td>20</td>
<td>1</td>
<td>Use of \textbf{EN} and \textbf{ENO} shown in LD</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>Usage without \textbf{EN} and \textbf{ENO} shown in FBD</td>
</tr>
<tr>
<td>20a</td>
<td>1</td>
<td>In-out variable declaration (textual)</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>In-out variable declaration (graphical)</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>Graphical connection of in-out variable to different variables (graphical)</td>
</tr>
<tr>
<td>21</td>
<td>1</td>
<td>Overloaded functions</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>Typed functions</td>
</tr>
<tr>
<td>Table No.</td>
<td>Feature No.</td>
<td>Description of Feature</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------</td>
<td>------------------------</td>
</tr>
<tr>
<td>22</td>
<td>1</td>
<td><em><em>.TO</em>.</em>* (Note 1, p. 613)</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>TRUNC (Note 2, p. 613)</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td><em><em>.BCD.TO</em>.</em>* (Note 3, p. 614)</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td><strong><em>.TO</em>.BCD_.</strong> (Note 3, p. 614)</td>
</tr>
<tr>
<td>23</td>
<td>1</td>
<td>ABS function</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>SQRT function</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>LN function</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>LOG function</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>EXP function</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>SIN function</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>COS function</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>TAN function</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>ASIN function</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>ACOS function</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>ATAN function</td>
</tr>
<tr>
<td>24</td>
<td>12</td>
<td>ADD function</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>MUL function</td>
</tr>
<tr>
<td></td>
<td>14</td>
<td>SUB function</td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>DIV function</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>MOD function</td>
</tr>
<tr>
<td></td>
<td>17</td>
<td>EXPT function</td>
</tr>
<tr>
<td></td>
<td>18</td>
<td>MOVE function</td>
</tr>
<tr>
<td>25</td>
<td>1</td>
<td>SHL function</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>SHR function</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>ROR function</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>ROL function</td>
</tr>
<tr>
<td>26</td>
<td>5</td>
<td>AND function</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>OR function</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>XOR function</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>NOT function</td>
</tr>
<tr>
<td>Table No.</td>
<td>Feature No.</td>
<td>Description of Feature</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------</td>
<td>------------------------</td>
</tr>
<tr>
<td>27</td>
<td>1</td>
<td>SEL function</td>
</tr>
<tr>
<td></td>
<td>2a</td>
<td>MAX function</td>
</tr>
<tr>
<td></td>
<td>2b</td>
<td>MIN function</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>LIMIT function</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>MUX function</td>
</tr>
<tr>
<td>28</td>
<td>5</td>
<td>GT function</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>GE function</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>EQ function</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>LE function</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>LT function</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>NE function</td>
</tr>
<tr>
<td>29</td>
<td>1</td>
<td>LEN function (Note 4, p. 614)</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>LEFT function (Note 4, p. 614)</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>RIGHT function (Note 4, p. 614)</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>MID function (Note 4, p. 614)</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>INSERT function (Note 4, p. 614)</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>DELETE function (Note 4, p. 614)</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>REPLACE function (Note 4, p. 614)</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>FIND function (Note 4, p. 614)</td>
</tr>
<tr>
<td>30</td>
<td>1a</td>
<td>ADD function (Note 6, p. 615)</td>
</tr>
<tr>
<td></td>
<td>1b</td>
<td>ADD_TIME function</td>
</tr>
<tr>
<td></td>
<td>2b</td>
<td>ADD_TOD_TIME function</td>
</tr>
<tr>
<td></td>
<td>3b</td>
<td>ADD_DT_TIME function</td>
</tr>
<tr>
<td></td>
<td>4a</td>
<td>SUB function (Note 6, p. 615)</td>
</tr>
<tr>
<td></td>
<td>4b</td>
<td>SUB_TIME function</td>
</tr>
<tr>
<td></td>
<td>5b</td>
<td>SUB_DATE_DATE function</td>
</tr>
<tr>
<td></td>
<td>6b</td>
<td>SUB_TOD_TIME function</td>
</tr>
<tr>
<td></td>
<td>7b</td>
<td>SUB_TOD_TOD function</td>
</tr>
<tr>
<td></td>
<td>8b</td>
<td>SUB_DT_TIME function</td>
</tr>
<tr>
<td></td>
<td>9b</td>
<td>SUB_DT_DT function</td>
</tr>
<tr>
<td></td>
<td>10a</td>
<td>MUL function (Note 6, p. 615)</td>
</tr>
<tr>
<td></td>
<td>10b</td>
<td>MULTIME function</td>
</tr>
<tr>
<td></td>
<td>11a</td>
<td>DIV function (Note 6, p. 615)</td>
</tr>
<tr>
<td></td>
<td>11b</td>
<td>DIVTIME function</td>
</tr>
</tbody>
</table>
### Table No. Feature No. Description of Feature

<table>
<thead>
<tr>
<th>Table No.</th>
<th>Feature No.</th>
<th>Description of Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>33</td>
<td>1a</td>
<td>RETAIN qualifier for internal variables <em>(Note 11, p. 616)</em></td>
</tr>
<tr>
<td></td>
<td>2a</td>
<td>RETAIN qualifier for output variables <em>(Note 11, p. 616)</em></td>
</tr>
<tr>
<td></td>
<td>2b</td>
<td>RETAIN qualifier for input variables <em>(Note 11, p. 616)</em></td>
</tr>
<tr>
<td></td>
<td>3a</td>
<td>RETAIN qualifier for internal function blocks <em>(Note 11, p. 616)</em></td>
</tr>
<tr>
<td></td>
<td>4a</td>
<td>VAR IN_OUT declaration (textual)</td>
</tr>
<tr>
<td></td>
<td>4b</td>
<td>VAR_IN_OUT declaration and usage (graphical)</td>
</tr>
<tr>
<td></td>
<td>4c</td>
<td>VAR IN_OUT declaration with assignment to different variables (graphical)</td>
</tr>
<tr>
<td>34</td>
<td>1</td>
<td>Bistable Function Block (set dominant)</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>Bistable Function Block (reset dominant)</td>
</tr>
<tr>
<td>35</td>
<td>1</td>
<td>Rising edge detector</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>Falling edge detector</td>
</tr>
<tr>
<td>36</td>
<td>1a</td>
<td>CTU (Up-counter) function block</td>
</tr>
<tr>
<td></td>
<td>1b</td>
<td>CTU_DINT function block</td>
</tr>
<tr>
<td></td>
<td>1d</td>
<td>CTU_UDINT function block</td>
</tr>
<tr>
<td></td>
<td>2a</td>
<td>CTD (Down-counter) function block</td>
</tr>
<tr>
<td></td>
<td>2b</td>
<td>CTD_DINT function block</td>
</tr>
<tr>
<td></td>
<td>2d</td>
<td>CTD_UDINT function block</td>
</tr>
<tr>
<td></td>
<td>3a</td>
<td>CTUD (Up-down-counter) function block</td>
</tr>
<tr>
<td></td>
<td>3b</td>
<td>CTUD_DINT function block</td>
</tr>
<tr>
<td></td>
<td>3d</td>
<td>CTUD_UDINT function block</td>
</tr>
<tr>
<td>37</td>
<td>1</td>
<td>TP (Pulse) function block</td>
</tr>
<tr>
<td></td>
<td>2a</td>
<td>TON (On delay) function block</td>
</tr>
<tr>
<td></td>
<td>3a</td>
<td>TOF (Off delay) function block</td>
</tr>
<tr>
<td>39</td>
<td>19</td>
<td>Use of directly represented variables</td>
</tr>
<tr>
<td>40</td>
<td>1</td>
<td>Step and initial step - Graphical form with directed links</td>
</tr>
<tr>
<td></td>
<td>3a</td>
<td>Step flag – General form</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>Step elapsed time– General form</td>
</tr>
<tr>
<td>Table No.</td>
<td>Feature No.</td>
<td>Description of Feature</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------</td>
<td>------------------------</td>
</tr>
<tr>
<td>41</td>
<td>7</td>
<td>Use of transition name</td>
</tr>
<tr>
<td></td>
<td>7a</td>
<td>Transition condition linked through transition name using LD language</td>
</tr>
<tr>
<td></td>
<td>7b</td>
<td>Transition condition linked through transition name using FBD language</td>
</tr>
<tr>
<td></td>
<td>7c</td>
<td>Transition condition linked through transition name using IL language</td>
</tr>
<tr>
<td></td>
<td>7d</td>
<td>Transition condition linked through transition name using ST language</td>
</tr>
<tr>
<td>42</td>
<td>1</td>
<td>Any Boolean variable declared in a <code>VAR</code> or <code>VAR_OUTPUT</code> block, or their graphical equivalents, can be an action</td>
</tr>
<tr>
<td></td>
<td>2l</td>
<td>Graphical declaration of action in LD language</td>
</tr>
<tr>
<td></td>
<td>2f</td>
<td>Graphical declaration of action in FBD language</td>
</tr>
<tr>
<td></td>
<td>3s</td>
<td>Textual declaration of action in ST language</td>
</tr>
<tr>
<td></td>
<td>3i</td>
<td>Textual declaration of action in IL language</td>
</tr>
<tr>
<td>43</td>
<td>1</td>
<td>Action block physically or logically adjacent to the step (Note 7, p. 615)</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>Concatenated action blocks physically or logically adjacent to the step (Note 8, p. 615)</td>
</tr>
<tr>
<td>44</td>
<td>1</td>
<td>Action qualifier in action block supported</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>Action name in action block supported</td>
</tr>
<tr>
<td>45</td>
<td>1</td>
<td>None - no qualifier</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>Qualifier N</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>Qualifier R</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>Qualifier S</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>Qualifier L</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>Qualifier D</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>Qualifier P</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>Qualifier DS</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>Qualifier P1</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>Qualifier P0</td>
</tr>
<tr>
<td>45a</td>
<td>2</td>
<td>Action control without &quot;final scan&quot;</td>
</tr>
</tbody>
</table>
46  | Single sequence
   | Divergence of sequence selection: left-to-right priority of transition evaluations
   | Convergence of sequence selection
   | Simultaneous sequences - divergence and convergence
   | Sequence skip: left-to-right priority of transition evaluations
   | Sequence loop: left-to-right priority of transition evaluations

49  | CONFIGURATION...END_CONFIGURATION construction (Note 12, p. 616)
   | Periodic TASK construction
   | Non-periodic TASK construction
   | WITH construction for PROGRAM to TASK association (Note 9, p. 615)
   | PROGRAM declaration with no TASK association (Note 10, p. 616)

50  | Non-preemptive scheduling (Note 13, p. 616)
   | Preemptive scheduling (Note 14, p. 616)
Note 1

List of type conversion functions:

- BOOL_TO_BYTE, BOOL_TO_DINT, BOOL_TO_INT, BOOL_TO_REAL,
  BOOL_TO_TIME, BOOL_TO_UDINT, BOOL_TO_UINT, BOOL_TO_WORD,
  BOOL_TO_DWORD
- BYTE_TO_BOOL, BYTE_TO_DINT, BYTE_TO_INT, BYTE_TO_REAL,
  BYTE_TO_TIME, BYTE_TO_UDINT, BYTE_TO_UINT, BYTE_TO_WORD,
  BYTE_TO_DWORD, BYTE_TO_BIT
- DINT_TO_BOOL, DINT_TO_BYTE, DINT_TO_INT, DINT_TO_REAL,
  DINT_TO_TIME, DINT_TO_UDINT, DINT_TO_UINT, DINT_TO_WORD,
  DINT_TO_DWORD, DINT_TO_DBCD, DINT_TO_STRING
- INT_TO_BOOL, INT_TO_BYTE, INT_TO_DINT, INT_TO_REAL, INT_TO_TIME,
  INT_TO_UDINT, INT_TO_UINT, INT_TO_WORD, INT_TO_BCD, INT_TO_DBCD,
  INT_TO_DWORD, INT_TO_STRING
- REAL_TO_BOOL, REAL_TO_BYTE, REAL_TO_DINT, REAL_TO_INT,
  REAL_TO_TIME, REAL_TO_UDINT, REAL_TO_UINT, REAL_TO_WORD,
  REAL_TO_DWORD, REAL_TO_STRING
- TIME_TO_BOOL, TIME_TO_BYTE, TIME_TO_DINT, TIME_TO_INT,
  TIME_TO_REAL, TIME_TO_UDINT, TIME_TO_UINT, TIME_TO_WORD,
  TIME_TO_DWORD, TIME_TO_STRING
- UDINT_TO_BOOL, UDINT_TO_BYTE, UDINT_TO_DINT, UDINT_TO_INT,
  UDINT_TO_REAL, UDINT_TO_TIME, UDINT_TO_UINT, UDINT_TO_WORD,
  UDINT_TO_DWORD
- UINT_TO_BOOL, UINT_TO_BYTE, UINT_TO_DINT, UINT_TO_INT,
  UINT_TO_REAL, UINT_TO_TIME, UINT_TO_UDINT, UINT_TO_UINT,
  UINT_TO_DINT, UINT_TO_DWORD,
- WORD_TO_BOOL, WORD_TO_BYTE, WORD_TO_DINT, WORD_TO_INT,
  WORD_TO_REAL, WORD_TO_TIME, WORD_TO_UDINT, WORD_TO_UINT,
  WORD_TO_BIT, WORD_TO_DWORD
- DWORD_TO_BOOL, DWORD_TO_BYTE, DWORD_TO_DINT, DWORD_TO_INT,
  DWORD_TO_REAL, DWORD_TO_TIME, DWORD_TO_UDINT, DWORD_TO_UINT,
  DWORD_TO_BIT,

The effects of each conversion are described in the help text supplied with the Base Library.

Note 2

List of types for truncate function:

- REAL_TRUNC_DINT, REAL_TRUNC_INT, REAL_TRUNC_UDINT,
  REAL_TRUNC_UINT

The effects of each conversion are described in the help text supplied with the Base Library.
**Note 3**

List of types for BCD conversion function:
- `BCD_TO_INT`, `DBCD_TO_INT`, `DBCD_TO_DINT`

List of types for BCD conversion function:
- `INT_TO_BCD`, `INT_TO_DBCD`, `DINT_TO_DBCD`

The effects of each conversion are described in the help text supplied with the Base Library.

---

**Note 4**

List of types for String functions:
- `LEN_INT`, `LEFT_INT`, `RIGHT_INT`, `MID_INT`, `INSERT_INT`, `DELETE_INT`, `REPLACE_INT`, `FIND_INT`

---

**Note 5**

A variable can be mapped to a directly represented variable if they strictly have the same type.

This means that a variable of type `INT` can only be mapped on a directly represented variable of type `INT`.

But there is one exception to this rule: for internal word (%MW<i>), Flat word (%IW<i>) and constant word (%KW<i>) memory variables any declared variable type is allowed.

Allowed mappings:

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Data type</th>
<th>Allowed variable types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Internal bit</td>
<td>%M&lt;i&gt; or %MX&lt;i&gt;</td>
<td>EBOOL</td>
</tr>
<tr>
<td>Internal word</td>
<td>%MW&lt;i&gt;</td>
<td>INT</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Internal double word</td>
<td>%MD&lt;i&gt;</td>
<td>DINT</td>
</tr>
<tr>
<td>Internal real</td>
<td>%MF&lt;i&gt;</td>
<td>REAL</td>
</tr>
<tr>
<td>Constant word</td>
<td>%KW&lt;i&gt;</td>
<td>INT</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Constant double word</td>
<td>%KD&lt;i&gt;</td>
<td>DINT</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Constant real</td>
<td>%KF&lt;i&gt;</td>
<td>REAL</td>
</tr>
<tr>
<td>System bit</td>
<td>%S&lt;i&gt; or %SX&lt;i&gt;</td>
<td>EBOOL</td>
</tr>
<tr>
<td>System word</td>
<td>%SW&lt;i&gt;</td>
<td>INT</td>
</tr>
<tr>
<td>System double word</td>
<td>%SD&lt;i&gt;</td>
<td>DINT</td>
</tr>
<tr>
<td>Flat bit</td>
<td>%I&lt;i&gt;</td>
<td>EBOOL</td>
</tr>
<tr>
<td>Flat word</td>
<td>%IW&lt;i&gt;</td>
<td>INT</td>
</tr>
<tr>
<td>Common word</td>
<td>%NWi.j.k</td>
<td>INT</td>
</tr>
<tr>
<td>Topological variables</td>
<td>%I...,%Q..., ...</td>
<td>Same Type (On some digital I/O modules it is allowed to map arrays of EBOOL on %IX&lt;topo&gt; and %QX&lt;topo&gt; objects.)</td>
</tr>
<tr>
<td>Extract bits</td>
<td>%MWi.j,...</td>
<td>BOOL</td>
</tr>
</tbody>
</table>

**Note 6**  
Only operator "+" (for ADD), "-" (for SUB), "*" (for MUL) or "/" (for DIV) in ST language.

**Note 7**  
This feature is only presented in the "expanded view" of the chart.

**Note 8**  
This feature is presented in the "expanded view" of the chart, but not as concatenated blocks, but as a scrollable list of action names with associated qualifiers inside one single block symbol.

**Note 9**  
There is only a one-to-one mapping of program instance to task. The textual format is replaced by a property dialog.
### IEC Compliance

**Note 10**  
The textual format is replaced by a property dialog.

**Note 11**  
All variables are retentive (*RETAIN* qualifier implicitly assumed in variable declarations).

**Note 12**  
The textual format is replaced by the project browser representation.

**Note 13**  
Using Mask-IT instruction, the user is able to get a non-preemptive behaviour. You will find *MASKEVT* (Global EVT masking) and *UNMASKEVT* (Global EVT unmasking) in the System functions of the libset.

**Note 14**  
By default, the multi-task system is preemptive.
## IL language elements

### IEC compliance table for IL language elements:

<table>
<thead>
<tr>
<th>Table No.</th>
<th>Feature No.</th>
<th>Feature description</th>
</tr>
</thead>
<tbody>
<tr>
<td>51b</td>
<td>1</td>
<td>Parenthesized expression beginning with explicit operator</td>
</tr>
<tr>
<td>51b</td>
<td>2</td>
<td>Parenthesized expression (short form)</td>
</tr>
<tr>
<td>52</td>
<td>1</td>
<td>LD operator (with modifier &quot;n&quot;)</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>ST operator (with modifier &quot;n&quot;)</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>S, R operator</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>AND operator (with modifiers &quot;+&quot;, &quot;n&quot;)</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>OR operator (with modifiers &quot;+&quot;, &quot;n&quot;)</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>XOR operator (with modifiers &quot;+&quot;, &quot;n&quot;)</td>
</tr>
<tr>
<td></td>
<td>7a</td>
<td>NOT operator</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>ADD operator (with modifier &quot;+&quot;)</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>SUB operator (with modifier &quot;+&quot;)</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>MUL operator (with modifier &quot;+&quot;)</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>DIV operator (with modifier &quot;+&quot;)</td>
</tr>
<tr>
<td></td>
<td>11a</td>
<td>MOD operator (with modifier &quot;+&quot;)</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>GT operator (with modifier &quot;+&quot;)</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>GE operator (with modifier &quot;+&quot;)</td>
</tr>
<tr>
<td></td>
<td>14</td>
<td>EQ operator (with modifier &quot;+&quot;)</td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>NE operator (with modifier &quot;+&quot;)</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>LE operator (with modifier &quot;+&quot;)</td>
</tr>
<tr>
<td></td>
<td>17</td>
<td>LT operator (with modifier &quot;+&quot;)</td>
</tr>
<tr>
<td></td>
<td>18</td>
<td>JMP operator (with modifiers &quot;C&quot;, &quot;n&quot;)</td>
</tr>
<tr>
<td></td>
<td>19</td>
<td>CAL operator (with modifiers &quot;C&quot;, &quot;n&quot;)</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>RET operator (with modifiers &quot;C&quot;, &quot;n&quot;) (Note, p. 618)</td>
</tr>
<tr>
<td></td>
<td>21</td>
<td>(evaluate deferred operation)</td>
</tr>
<tr>
<td>53</td>
<td>1a</td>
<td>CAL of Function Block with non-formal argument list</td>
</tr>
<tr>
<td></td>
<td>1b</td>
<td>CAL of Function Block with formal argument list</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>CAL of Function Block with load/store of arguments</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>Function invocation with formal argument list</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>Function invocation with non-formal argument list</td>
</tr>
</tbody>
</table>
Note

In DFB only.
ST language elements

IEC compliance table for ST language elements:

<table>
<thead>
<tr>
<th>Table No.</th>
<th>Feature No.</th>
<th>Feature description</th>
</tr>
</thead>
<tbody>
<tr>
<td>55</td>
<td>1</td>
<td>Parenthesization (expression)</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>Function evaluation: functionName(listOfArguments)</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>Exponentiation: **</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>Negation: -</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>Complement: NOT</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>Multiply: *</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>Divide: /</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>Modulo: MOD</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>Add: +</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>Subtract: -</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>Comparison: &lt;, &gt;, &lt;=, &gt;=</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>Equality: =</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>Inequality: &lt;&gt;</td>
</tr>
<tr>
<td></td>
<td>14</td>
<td>Boolean AND: &amp;</td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>Boolean AND: AND</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>Boolean Exclusive OR: XOR</td>
</tr>
<tr>
<td></td>
<td>17</td>
<td>Boolean OR: OR</td>
</tr>
<tr>
<td>56</td>
<td>1</td>
<td>Assignment</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>Function block invocation and function block output usage</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>RETURN statement (Note, p. 619)</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>IF statement</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>CASE statement</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>FOR statement</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>WHILE statement</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>REPEAT statement</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>EXIT statement</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>Empty statement</td>
</tr>
</tbody>
</table>

Note
In DFB only.
IEC compliance table for common graphical elements:

<table>
<thead>
<tr>
<th>Table No.</th>
<th>Feature No.</th>
<th>Feature description</th>
</tr>
</thead>
<tbody>
<tr>
<td>57</td>
<td>2</td>
<td>Horizontal lines: Graphic or semigraphic</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>Vertical lines: Graphic or semigraphic</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>Horizontal/vertical connection: Graphic or semigraphic</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>Line crossings without connection: Graphic or semigraphic</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>Connected and non-connected corners: Graphic or semigraphic</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>Blocks with connecting lines: Graphic or semi-graphic</td>
</tr>
<tr>
<td>58</td>
<td>1</td>
<td>Unconditional Jump: FBD Language</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>Unconditional Jump: LD Language</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>Conditional Jump: FBD Language</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>Conditional Jump: LD Language</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>Conditional Return: LD Language (Note, p. 620)</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>Conditional Return: FBD Language (Note, p. 620)</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>Unconditional Return from function or function block (Note, p. 620)</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>Unconditional Return: LD Language (Note, p. 620)</td>
</tr>
</tbody>
</table>

Note: In DFB only.
IEC compliance table for LD language elements:

<table>
<thead>
<tr>
<th>Table No.</th>
<th>Feature No.</th>
<th>Feature description</th>
</tr>
</thead>
<tbody>
<tr>
<td>59</td>
<td>1</td>
<td>Left power rail</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>Right power rail</td>
</tr>
<tr>
<td>60</td>
<td>1</td>
<td>Horizontal link</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>Vertical link</td>
</tr>
<tr>
<td>61</td>
<td>1</td>
<td>Normally open contact (vertical bar) <em>(Note, p. 621)</em></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>Normally closed contact (vertical bar) <em>(Note, p. 621)</em></td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>Positive transition-sensing contact (vertical bar) <em>(Note, p. 621)</em></td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>Negative transition-sensing contact (vertical bar) <em>(Note, p. 621)</em></td>
</tr>
<tr>
<td>62</td>
<td>1</td>
<td>Coil</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>Negated coil</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>SET (latch) coil</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>RESET (un latch) coil</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>Positive transition-sensing coil</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>Negative transition-sensing coil</td>
</tr>
</tbody>
</table>

**Note**

Only graphical representation.
## Implementation-dependent parameters

IEC compliance table for implementation-dependent parameters:

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Limitations/Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum length of identifiers</td>
<td>32 characters</td>
</tr>
<tr>
<td>Maximum comment length</td>
<td>Within the Unity Pro: 1024 characters for each editor object. Import: limited by XML constraints or UDBString usage in the persistent layer.</td>
</tr>
<tr>
<td>Syntax and semantics of pragmas</td>
<td>Unity V1.0 only implements 1 pragma, used for legacy convertor:</td>
</tr>
<tr>
<td></td>
<td>{ ConvError (' error text'); }</td>
</tr>
<tr>
<td></td>
<td>any other pragma construct is ignored (a warning message is given)</td>
</tr>
<tr>
<td>Syntax and semantics for the use of the double-quote character when a particular implementation supports Feature #4 but not Feature #2 of Table 5.</td>
<td>(#2 of table 5 is supported)</td>
</tr>
<tr>
<td>Range of values and precision of representation for variables of type TIME, DATE, TIME_OF_DAY and DATE_AND_TIME</td>
<td>for TIME: t#0ms .. t#4294967295ms (=t#49D_17H_2M_47S_295MS)</td>
</tr>
<tr>
<td></td>
<td>for DATE: D#1990-01-01 .. D#2099-12-31</td>
</tr>
<tr>
<td></td>
<td>for TOD: TOD#00:00:00 .. TOD#23:59:59</td>
</tr>
<tr>
<td>Precision of representation of seconds in types TIME, TIME_OF_DAY and DATE_AND_TIME</td>
<td>TIME: precision 1 ms</td>
</tr>
<tr>
<td></td>
<td>TIME_OF_DAY: precision 1 s</td>
</tr>
<tr>
<td>Maximum number of enumerated values</td>
<td>Not applicable</td>
</tr>
<tr>
<td>Maximum number of array subscripts</td>
<td>6</td>
</tr>
<tr>
<td>Maximum array size</td>
<td>64 kbytes</td>
</tr>
<tr>
<td>Maximum number of structure elements</td>
<td>no limit</td>
</tr>
<tr>
<td>Maximum structure size</td>
<td>64 kbytes</td>
</tr>
<tr>
<td>Maximum range of subscript values</td>
<td>DINT range</td>
</tr>
<tr>
<td>Maximum number of levels of nested structures</td>
<td>10</td>
</tr>
<tr>
<td>Default maximum length of STRING and WSTRING variables</td>
<td>16 characters</td>
</tr>
<tr>
<td>Maximum allowed length of STRING and WSTRING variables</td>
<td>64 kbytes</td>
</tr>
</tbody>
</table>
### Parameters

| Maximum number of hierarchical levels Logical or physical mapping | Premium: physical mapping (5 levels) Quantum: logical mapping (1 level) |
| Maximum number of inputs of extensible functions | The number of all input parameters (including in-out parameters) is limited to 32. The number of all output parameters (including in-out parameters) is also limited to 32. Thus the limit for extensible input parameters is (32 - number of input parameters - number of in-out parameters). The limit for extensible output parameters is (32 - number of output parameters - number of in-out parameters). |

### Effects of type conversions on accuracy

- See online help.

### Error conditions during type conversions

- Error conditions are described in the online-help. Globally %S18 is set for overflow errors. ENO is also set. The result is depending on the specific function.

### Accuracy of numerical functions

- INTEL floating point processing or emulation.

### Effects of type conversions between time data types and other data types not defined in Table 30

- See online help.

### Maximum number of function block specifications and instantiations

- Only limited by the maximum size of a section.

### Function block input variable assignment when EN is FALSE

- No assignment

### Parameters Limitations/Behavior

<table>
<thead>
<tr>
<th>Pvmin, Pvmax of counters</th>
</tr>
</thead>
<tbody>
<tr>
<td>INT base counters:</td>
</tr>
<tr>
<td>● Pvmin=32768 (0x8000)</td>
</tr>
<tr>
<td>● Pvmax=32767 (0x7FFF)</td>
</tr>
<tr>
<td>UINT base counters:</td>
</tr>
<tr>
<td>● Pvmin=0 (0x0)</td>
</tr>
<tr>
<td>● Pvmax=65535 (0xFFFF)</td>
</tr>
<tr>
<td>DINT base counters:</td>
</tr>
<tr>
<td>● Pvmin=-2147483648 (0x80000000)</td>
</tr>
<tr>
<td>● Pvmax=2147483647 (0x7FFFFFFF)</td>
</tr>
<tr>
<td>UDINT base counters:</td>
</tr>
<tr>
<td>● Pvmin=0 (0x0)</td>
</tr>
<tr>
<td>● Pvmax=4294967295 (0xFFFFFFFF)</td>
</tr>
</tbody>
</table>
### Parameters

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Limitations/Behavior</th>
</tr>
</thead>
</table>
| Maximum number of hierarchical levels | Premium: physical mapping (5 levels)  
Quantum: logical mapping (1 level) |
| Maximum number of inputs of extensible functions | The number of all input parameters (including in-out parameters) is limited to 32. The number of all output parameters (including in-out parameters) is also limited to 32. Thus the limit for extensible input parameters is (32 - number of input parameters - number of in-out parameters).  
The limit for extensible output parameters is (32 - number of output parameters - number of in-out parameters). |
| Effects of type conversions on accuracy | See online help. |
| Error conditions during type conversions | Error conditions are described in the online-help. Globally $S18$ is set for overflow errors.  
$ENO$ is also set. The result is depending on the specific function. |
| Accuracy of numerical functions | INTEL floating point processing or emulation. |
| Effects of type conversions between time data types and other data types not defined in Table 30 | See online help. |
| Maximum number of function block specifications and instantiations | Only limited by the maximum size of a section. |
| Function block input variable assignment when EN $\equiv$ FALSE | No assignment |

### Pvmin, Pvmx of counters

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Pvmin</th>
<th>Pvmax</th>
</tr>
</thead>
<tbody>
<tr>
<td>INT base counters:</td>
<td>$-32768$ ($0x8000$)</td>
<td>$32767$ ($0x7FFF$)</td>
</tr>
<tr>
<td>UINT base counters:</td>
<td>$0$ ($0x0$)</td>
<td>$65535$ ($0xFFFF$)</td>
</tr>
<tr>
<td>DINT base counters:</td>
<td>$-2147483648$ ($0x80000000$)</td>
<td>$2147483647$ ($0x7FFFFFFF$)</td>
</tr>
<tr>
<td>UDINT base counters:</td>
<td>$0$ ($0x0$)</td>
<td>$4294967295$ ($0xFFFFFFFF$)</td>
</tr>
</tbody>
</table>
**Parameters** | **Limitations/Behavior**
---|---
Effect of a change in the value of a PT input during a timing operation | The new PT values are immediately taken into account, even during a running timing operation immediately works with the new values.
Program size limitations | Depends on controller type and memory
Precision of step elapsed time | 10 ms
Maximum number of steps per SFC | 1024 steps per SFC section
Maximum number of transitions per SFC and per step | Limited by the available area for entering steps/transitions and by the maximum number of steps per SFC section (1024 Steps). 32 transition per step. Limited by the available area for entering Alternative/Parallel branches, maximum is 32 rows.
Maximum number of action blocks per step | 20
Access to the functional equivalent of the □ or △ outputs | Not applicable
Transition clearing time | Target dependent; always < 100 micro-seconds
Maximum width of diverge/converge constructs | 32
Contents of RESOURCE libraries | Not applicable
Effect of using READ_WRITE access to function block outputs | Not applicable
Maximum number of tasks | Depends on controller type. Maximum on most powerful controller: 9 tasks
Task interval resolution | 10 ms
Maximum length of expressions | Practically no limit
Maximum length of statements | Practically no limit
Maximum number of CASE selections | Practically no limit
Value of control variable upon termination of FOR loop | Undefined
Restrictions on network topology | No restrictions
Evaluation order of feedback loops | The block connected to the feedback variable is executed first
## Error Conditions

### IEC standards table for error conditions:

<table>
<thead>
<tr>
<th>Error conditions</th>
<th>Treatment (see Note, p. 627)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nested comments</td>
<td>2) error is reported during programming</td>
</tr>
<tr>
<td>Value of a variable exceeds the specified subrange</td>
<td>Not applicable</td>
</tr>
<tr>
<td>Missing configuration of an incomplete address specification (** notation)</td>
<td>Not applicable</td>
</tr>
<tr>
<td>Attempt by a program organization unit to modify a variable which has been declared CONSTANT</td>
<td>2) error is reported during programming</td>
</tr>
<tr>
<td>Improper use of directly represented or external variables in functions</td>
<td>Not applicable</td>
</tr>
<tr>
<td>A <code>VAR_IN_OUT</code> variable is not &quot;properly mapped&quot;</td>
<td>2) error is reported during programming</td>
</tr>
<tr>
<td>Type conversion errors</td>
<td>4) error is reported during execution</td>
</tr>
<tr>
<td>Numerical result exceeds range for data type</td>
<td>4) error is reported during execution</td>
</tr>
<tr>
<td>Division by zero</td>
<td>4) error is reported during execution</td>
</tr>
<tr>
<td>Mixed input data types to a selection function</td>
<td>2) error is reported during programming</td>
</tr>
<tr>
<td>Result exceeds range for data type</td>
<td>4) error is reported during execution</td>
</tr>
<tr>
<td>No value specified for an in-out variable</td>
<td>2) error is reported during programming</td>
</tr>
<tr>
<td>Zero or more than one initial steps in SFC network</td>
<td></td>
</tr>
<tr>
<td>User program attempts to modify step state or time</td>
<td></td>
</tr>
<tr>
<td>Side effects in evaluation of transition condition</td>
<td>3) error is reported during analyzing/loading/linking</td>
</tr>
<tr>
<td>Action control contention error</td>
<td>Not applicable</td>
</tr>
<tr>
<td>Simultaneously true, non-prioritized transitions in a selection divergence</td>
<td>Not applicable</td>
</tr>
<tr>
<td>Unsafe or unreachable SFC</td>
<td>3) error is reported during analyzing/loading/linking</td>
</tr>
<tr>
<td>Data type conflict in <code>VAR_ACCESS</code></td>
<td>Not applicable</td>
</tr>
<tr>
<td>A task fails to be scheduled or to meet its execution deadline</td>
<td>4) error is reported during execution</td>
</tr>
<tr>
<td>Numerical result exceeds range for data type</td>
<td>4) error is reported during execution</td>
</tr>
</tbody>
</table>
Note

Idenifications for the treatment of error conditions according to IEC 61131-3, subclause 1.5.1, d):

- 1) error is not reported
- 2) error is reported during programming
- 3) error is reported during analyzing/loading/linking
- 4) error is reported during execution

<table>
<thead>
<tr>
<th>Error conditions</th>
<th>Treatment (see Note, p. 627)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Current result and operand not of same data type</td>
<td>2) error is reported during programming</td>
</tr>
<tr>
<td>Division by zero</td>
<td>4) error is reported during execution</td>
</tr>
<tr>
<td>Numerical result exceeds range for data type</td>
<td>4) error is reported during execution</td>
</tr>
<tr>
<td>Invalid data type for operation</td>
<td>4) error is reported during execution</td>
</tr>
<tr>
<td>Return from function without value assigned</td>
<td>Not applicable</td>
</tr>
<tr>
<td>Iteration fails to terminate</td>
<td>4) error is reported during execution</td>
</tr>
<tr>
<td>Same identifier used as connector label and element name</td>
<td>Not applicable</td>
</tr>
<tr>
<td>Uninitialized feedback variable</td>
<td>1) error is not reported</td>
</tr>
</tbody>
</table>
B.3 Extensions of IEC 61131-3

Extensions of IEC 61131-3, 2nd Edition

At a Glance

In addition to the standardized IEC features listed in the (See IEC Compliance Tables, p. 604), the Unity Pro programming environment inherited a number of features from the PL7 programming environment. These extensions are optionally provided; they can be checked or not in a corresponding options dialog. The dialog and the features are described in detail in a chapter of the online help titled Data and Languages (See Unity Pro, Operating Modes Manual, Dates and Languages).

Not included in the options dialog is another extension, which is inherited both from the PL7 and the Concept programming environments: Unity Pro provides the construct of the so-called Section in all programming languages, which allows to subdivide a Program Organization Unit (POU). This construct introduces the possibility to mix several languages (e.g. FBD sections, SFC sections) in a POU body, a feature which, if used for this purpose, constitutes an extension of the IEC syntax. A compliant POU body should contain a single section only. Sections do not create a distinct name scope; the name scope for all language elements is the POU.

Purpose of Sections

Sections serve different purposes:

- Sections allow to subdivide large POU bodies according to functional aspects: the user has the possibility to subdivide his POU body into functionally meaningful parts. The list of sections represents a kind of functional table of contents of a large, otherwise unstructured POU body.

- Sections allow to subdivide large POU bodies according to graphical aspects: the user has the possibility to design substructures of a large POU body according to an intended graphical presentation. He can create small or large graphical sections according to his taste.

- The subdivision of large POU bodies allows quick online changes: in Unity Pro, the Section serves as the unit for online change. If a POU body is modified during runtime at different locations, automatically all sections affected by the changes are downloaded on explicit request.

- Sections allow to rearrange the execution order of specific, labeled parts of a POU body: the section name serves as a label of that part of the body which is contained inside the section, and by ordering these labels the execution order of those parts is manageable.
Sections allow to use different languages in parallel in the same POU: this feature is a major extension of the IEC syntax, which allows only one single IEC language to be used for a POU body. In a compliant body, SFC has to be used to manage different languages inside a body (each transition and action may be formulated in its own language).
B.4 Textual language syntax

Textual Language Syntax

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Unity Pro V1.0 programming environment does not yet provide support for an import or export of text files complying with the full textual language syntax as specified in Annex B of IEC 61131-3, 2nd Edition. However, the textual syntax of the IL and ST languages, as specified in Annex B.2 and B.3 of IEC 61131-3, 2nd Edition, including all directly and indirectly referenced productions out of Annex B.1, is supported in textual language sections. Those syntax productions in Annex B of IEC 61131-3, 2nd Edition belonging to features which are not supported by Unity Pro according to the compliance tables (See IEC Compliance Tables, p. 604) are not implemented.</td>
</tr>
</tbody>
</table>
Glossary

%I
According to the IEC standard, \%I indicates a discrete input-type language object.

%IW
According to the IEC standard, \%IW indicates an analog input-type language object.

%KW
According to the IEC standard, \%KW indicates a constant word-type language object.

%M
According to the IEC standard, \%M indicates a memory bit-type language object.

%MW
According to the IEC standard, \%MW indicates a memory word-type language object.

%Q
According to the IEC standard, \%Q indicates a discrete output-type language object.

%QW
According to the IEC standard, \%QW indicates an analog output-type language object.

Animating the links
This is also called power flow, and refers to a type of animation used with Ladder language and the function blocks. The links are displayed in red, green or black according to the variables connected.

ANY
There is a hierarchy between the different types of data. In the DFB, it is sometimes possible to declare which variables can contain several types of values. Here, we use ANY_xxx types.
The following diagram shows the hierarchically-ordered structure:

![Diagram showing the hierarchically-ordered structure of data types]

**ARRAY**

An **ARRAY** is a table of elements of the same type.

The syntax is as follows: **ARRAY [ <terminals> ] OF <Type>**

Example:

**ARRAY [1..2] OF BOOL** is a one-dimensional table made up of two BOOL-type elements.

**ARRAY [1..10, 1..20] OF INT** is a two-dimensional table made up of 10x20 INT-type elements.
ASCII is the abbreviation of American Standard Code for Information Interchange. This is an American code (but which has become an international standard) that uses 7 bits to define every alphanumerical character used in English, punctuation symbols, certain graphic characters and other miscellaneous commands.

Auxiliary tasks: Optional periodic tasks used to process procedures that do not require fast processing: measurement, adjustment, diagnostic aid, etc.

---

**ASCII**

This is an American code (but which has become an international standard) that uses 7 bits to define every alphanumerical character used in English, punctuation symbols, certain graphic characters and other miscellaneous commands.

**Auxiliary tasks**

Optional periodic tasks used to process procedures that do not require fast processing: measurement, adjustment, diagnostic aid, etc.

---

**Base 10 literals**

A literal value in base 10 is used to represent a decimal integer value. This value can be preceded by the signs "+" and "-". If the character "_" is employed in this literal value, it is not significant.

Example:

-12, 0, 123_456, +986

**Base 16 literals**

An literal value in base 16 is used to represent an integer in hexadecimal. The base is determined by the number "16" and the sign ".#". The signs "+" and "-" are not allowed. For greater clarity when reading, you can use the sign "._" between bits.

Example:

16#F_F or 16#FF (in decimal 255)
16#F_F or 16#FF (in decimal 224)

**Base 2 literals**

A literal value in base 2 is used to represent a binary integer. The base is determined by the number "2" and the sign ".#". The signs "+" and "-" are not allowed. For greater clarity when reading, you can use the sign "._" between bits.

Example:

2#1111_1111 or 2#11111111 (in decimal 255)
2#1110_0000 or 2#11100000 (in decimal 224)

**Base 8 literals**

A literal value in base 8 is used to represent an octal integer. The base is determined by the number "8" and the sign ".#". The signs "+" and "-" are not allowed. For greater clarity when reading, you can use the sign "._" between bits.

Example:

8#3_77 or 8#377 (in decimal 255)
8#34_0 or 8#340 (in decimal 224)

**BCD**

The Binary Coded Decimal (BCD) format is used to represent decimal numbers between 0 and 9 using a group of four bits (half-byte). In this format, the four bits used to code the decimal numbers have a range of unused combinations.
Example of BCD coding:
- the number 2450
- is coded: 0010 0100 0101 0000

**BIT**

This is a binary unit for a quantity of information which can represent two distinct values (or statuses): 0 or 1.

**BOOL**

BOOL is the abbreviation of Boolean type. This is the elementary data item in computing. A BOOL type variable has a value of either: 0 (FALSE) or 1 (TRUE). A BOOL type word extract bit, for example: %MN10.4.

**Break point**

Used in the "debug" mode of the application. It is unique (one at a time) and, when reached, signals to the processor to stop the program run.

Used in connected mode, it can be positioned in one of the following program elements:
- LD network,
- Structured Text Sequence or Instruction List,
- Structured Text Line (Line mode).

**BYTE**

When 8 bits are put together, this is called a BYTE. A BYTE is either entered in binary, or in base 8. The BYTE type is coded in an 8 bit format, which, in hexadecimal, ranges from 16#00 to 16#FF

**C**

**Constants**

An INT, DINT or REAL type variable located in the constant field (%K), or variables used in direct addressing (%KW, %KD or %KF). The contents of these cannot be modified by the program during execution.

**CPU**

Is the abbreviation of Control Processing Unit. This is the microprocessor. It is made up of the control unit combined with the arithmetic unit. The aim of the control unit is to extract the instruction to be executed and the data needed to execute this instruction from the central memory, to establish electrical connections in the arithmetic unit and logic, and to run the processing of this data in this unit. We can sometimes find ROM or RAM memories included in the same chip, or even I/O interfaces or buffers.
Cyclic execution

The master task is executed either cyclically or periodically. Cyclic execution consists of stringing cycles together one after the other with no waiting time between the cycles.

DATE

The DATE type coded in BCD in 32 bit format contains the following information:
- the year coded in a 16-bit field,
- the month coded in an 8-bit field,
- the day coded in an 8-bit field.

The DATE type is entered as follows: D#<Year>-<Month>-<Day>

This table shows the lower/upper limits in each field:

<table>
<thead>
<tr>
<th>Field</th>
<th>Limits</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Year</td>
<td>[1990,2099]</td>
<td>Year</td>
</tr>
<tr>
<td>Month</td>
<td>[01,12]</td>
<td>The left 0 is always displayed, but can be omitted at the time of entry</td>
</tr>
<tr>
<td>Day</td>
<td>[01,31]</td>
<td>For the months 01/03/05/07/08/10/12</td>
</tr>
<tr>
<td></td>
<td>[01,30]</td>
<td>For the months 04/06/09/11</td>
</tr>
<tr>
<td></td>
<td>[01,29]</td>
<td>For the month 02 (leap years)</td>
</tr>
<tr>
<td></td>
<td>[01,28]</td>
<td>For the month 02 (non leap years)</td>
</tr>
</tbody>
</table>

DATE_AND_TIME  see DT

DBCD

Representation of a Double BCD-format double integer.
The Binary Coded Decimal (BCD) format is used to represent decimal numbers between 0 and 9 using a group of four bits.
In this format, the four bits used to code the decimal numbers have a range of unused combinations.
Example of DBCD coding:
- the number 78993016
- is coded: 0111 1000 1001 1001 0011 0000 0000 0010

DDT

DDT is the abbreviation of Derived Data Type.
A derived data type is a set of elements of the same type (ARRAY) or of various types (structure)

DFB

DFB is the abbreviation of Derived Function Block.
DFB types are function blocks that can be programmed by the user ST, IL, LD or FBD.
By using DFB types in an application, it is possible to:
- simplify the design and input of the program,
- increase the legibility of the program,
- facilitate the debugging of the program,
- reduce the volume of the generated code.

DFB instance
A DFB type instance occurs when an instance is called from a language editor. The instance possesses a name, input/output interfaces, the public and private variables are duplicated (one duplication per instance, the code is not duplicated). A DFB type can have several instances.

DINT
DINT is the abbreviation of Double Integer format (coded on 32 bits). The lower and upper limits are as follows: -(2 to the power of 31) to (2 to the power of 31) - 1.
Example:
-2147483648, 2147483647, 16#FFFFFFFF.

Documentation
Contains all the information of the project. The documentation is printed once compiled and used for maintenance purposes. The information contained in the documentation cover:
- the hardware and software configuration,
- the program,
- the DFB types,
- the variables and animation tables,
- the cross-references.
- ...
When building the documentation file, you can include all or some of these items.

Driver
A program indicating to your computer's operating system the presence and characteristics of a peripheral device. We also use the term peripheral device driver. The best-known drivers are printer drivers. To make a PLC communicate with a PC, communication drivers need to be installed (Uni-Telway, XIP, Fipway, etc.).

DT
DT is the abbreviation of Date and Time. The DT type coded in BCD in 64 bit format contains the following information:
- The year coded in a 16-bit field,
- the month coded in an 8-bit field,
- the day coded in an 8-bit field,
- the hour coded in a 8-bit field,
- the minutes coded in an 8-bit field,
- the seconds coded in an 8-bit field.
**Note:** The 8 least significant bits are unused.

The **DT** type is entered as follows:

```
DT#<Year>-<Month>-<Day>-<Hour>:<Minutes>:<Seconds>
```

This table shows the lower/upper limits in each field:

<table>
<thead>
<tr>
<th>Field</th>
<th>Limits</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Year</td>
<td>[1990,2099]</td>
<td>Year</td>
</tr>
<tr>
<td>Month</td>
<td>[01,12]</td>
<td>The left 0 is always displayed, but can be omitted at the time of entry</td>
</tr>
<tr>
<td>Day</td>
<td>[01,31]</td>
<td>For the months 01\03\05\07\08\10\12</td>
</tr>
<tr>
<td></td>
<td>[01,30]</td>
<td>For the months 04\06\09\11</td>
</tr>
<tr>
<td></td>
<td>[01,29]</td>
<td>For the month 02 (leap years)</td>
</tr>
<tr>
<td></td>
<td>[01,28]</td>
<td>For the month 02 (non leap years)</td>
</tr>
<tr>
<td>Hour</td>
<td>[00,23]</td>
<td>The left 0 is always displayed, but can be omitted at the time of entry</td>
</tr>
<tr>
<td>Minute</td>
<td>[00,59]</td>
<td>The left 0 is always displayed, but can be omitted at the time of entry</td>
</tr>
<tr>
<td>Second</td>
<td>[00,59]</td>
<td>The left 0 is always displayed, but can be omitted at the time of entry</td>
</tr>
</tbody>
</table>

**DWORD**

**DWORD** is the abbreviation of Double Word.
The **DWORD** type is coded in 32 bit format.
This table shows the lower/upper limits of the bases which can be used:

<table>
<thead>
<tr>
<th>Base</th>
<th>Lower limit</th>
<th>Upper limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hexadecimal</td>
<td>16#0</td>
<td>16#FFFFFFFF</td>
</tr>
<tr>
<td>Octal</td>
<td>8#0</td>
<td>8#37777777777</td>
</tr>
<tr>
<td>Binary</td>
<td>2#0</td>
<td>2#11111111111111111111111111</td>
</tr>
</tbody>
</table>

Representation examples:

<table>
<thead>
<tr>
<th>Data content</th>
<th>Representation in one of the bases</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000000000010101110011011110</td>
<td>16#ADCDE</td>
</tr>
<tr>
<td>00000000000101011100100000000000</td>
<td>8#200000</td>
</tr>
<tr>
<td>0000000000010101110010111110</td>
<td>2#10101011110011011110</td>
</tr>
</tbody>
</table>
EBOOL

EBOOL is the abbreviation of Extended Boolean type. A EBOOL type variable brings a value (0 (FALSE) or 1 (TRUE)) but also rising or falling edges and forcing capabilities.

An EBOOL type variable takes up one byte of memory.

The byte split up into:
- one bit for the value,
- one bit for the history bit (each time the state’s object changes, the value is copied inside the history bit),
- one bit for the forcing bit (equals to 0 if the object isn’t forced, equal to 1 if the bit is forced.

The default type value of each bit is 0 (FALSE).

EDT

EDT is the abbreviation of Elementary Data Type.

These types are as follows:
- BOOL,
- EBOOL,
- WORD,
- DWORD,
- INT,
- DINT,
- UINT,
- UDINT,
- REAL,
- DATE,
- TOD,
- DT.

EF

EF is the abbreviation of Elementary Function.

This is a block which is used in a program, and which performs a predefined software function.

A function has no internal status information. Multiple invocations of the same function using the same input parameters always supply the same output values.

Details of the graphic form of the function invocation can be found in the “[Functional block (instance)]”. In contrast to the invocation of the function blocks, function invocations only have a single unnamed output, whose name is the same as the function. In FBD each invocation is denoted by a unique [number] via the graphic block, this number is automatically generated and can not be altered.

You position and set up these functions in your program in order to carry out your application.

You can also develop other functions using the SDKC development kit.
EFB

Is the abbreviation for Elementary Function Block.
This is a block which is used in a program, and which performs a predefined software function.
EFBs have internal statuses and parameters. Even where the inputs are identical, the output values may be different. For example, a counter has an output which indicates that the preselection value has been reached. This output is set to 1 when the current value is equal to the preselection value.

Elementary Function

see EF

EN / ENO (enable / error notification)

EN means Enable, this is an optional block input.
If EN = 0, the block is not activated, its internal program is not executed and ENO is set to 0.
If EN = 1, the internal program of the block is executed, and ENO is set to 1 by the system. If an error occurs, ENO is set to 0.
ENO means Error Notification, this is the output associated to the optional input EN.
If ENO is set to 0 (caused by EN=0 or in case of an execution error),
- the outputs of function blocks remain in the status they were in for the last correct executed scanning cycle and
- the output(s) of functions and procedures are set to "0".

Note: If EN is not connected, it is automatically set to 1.

Event processing

Event processing 1 is a program section launched by an event. The instructions programmed in this section are executed when a software application event (Timer) or a hardware event (application specific module) is received by the processor.
Event processes take priority over other tasks, and are executed the moment the event is detected.
The event process EVT0 is of highest priority. All others have the same level of priority.

Fast task

Task launched periodically (setting of the period in the PC configuration) used to carry out a part of the application having a superior level of priority to the Mast task (master).

FBD

FBD is the abbreviation of Function Block Diagram.
Glossary

FBD is a graphic programming language that operates as a logic diagram. In addition to the simple logic blocks (AND, OR, etc.), each function or function block of the program is represented using this graphic form. For each block, the inputs are located to the left and the outputs to the right. The outputs of the blocks can be linked to the inputs of other blocks to form complex expressions.

FFB
Collective term for EF (Elementary Function), EFB (Elementary Function Block) and DFB (Derived Function block)

Flash Eprom
PCMCIA memory card containing the program and constants of the application.

FNES
FNES is the abbreviation of Fichiers Neutres d'Entrées Sorties (Neutral I/O Documentation).
FNES format describes using a tree structure the PLCs in terms of rack, cards and channels.
It is based on the CNOMO standard (comité de normalisation des outillages de machines outils).

Function
see EF

Function block
see EFB

Function view
View making it possible to see the program part of the application through the functional modules created by the user (see Functional module definition).

Functional Module
A functional module is a group of program elements (sections, sub-programs, macro steps, animation tables, runtime screen, etc.) whose purpose is to perform an automation device function.
A functional module may itself be separated into lower-level functional modules, which perform one or more sub-functions of the main function of the automation device.

GRAY
Gray or "reflected binary" code is used to code a numerical value being developed into a chain of binary configurations that can be differentiated by the change in status of one and only one bit.
This code can be used, for example, to avoid the following random event: in pure binary, the change of the value 0111 to 1000 can produce random numbers between 0 and 1000, as the bits do not change value altogether simultaneously.
Equivalence between decimal, BCD and Gray:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decimal</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>BCD</td>
<td>0000</td>
<td>0001</td>
<td>0010</td>
<td>0011</td>
<td>0100</td>
<td>0101</td>
<td>0110</td>
<td>0111</td>
<td>1000</td>
<td>1001</td>
</tr>
<tr>
<td>Gray</td>
<td>0000</td>
<td>0001</td>
<td>0011</td>
<td>0010</td>
<td>0110</td>
<td>0111</td>
<td>0101</td>
<td>0100</td>
<td>1100</td>
<td>1101</td>
</tr>
</tbody>
</table>

Hyperlink

The hyperlink function enables links to be created between your project and external documents. You can create hyperlinks in all the elements of the project directory, in the variables, in the processing screen objects, etc. The external documents can be web pages, files (xls, pdf, wav, mp3, jpg, gif, etc.).

I/O Object

An I/O object is an implicit or explicit language object for an expert function module or a I/O device on a fieldbus. They are of the following types: %Ch, %I, %IW, %ID, %IF, %Q, %QW, %QD, %QF, %KW, %KD, %KF, %MW, %MD, and %MF. The objects' topological address depends on the module's position on the rack or the device's position on the bus.

IEC 61131-3


IL

IL is the abbreviation of Instruction List. This language is a series of basic instructions. This language is very close to the assembly language used to program processors. Each instruction is composed of an instruction code and an operand.

INF

Used to indicate that a number overruns the allowed limits.
For a number of Integers, the value ranges (shown in gray) are as follows:

-3.402824e+38  -1.1754944e-38  0.0  1.1754944e-38  3.402824e+38

When a calculation result is:
- less than -3.402824e+38, the symbol \(-\text{INF}\) (for \(-\text{infinite}\)) is displayed,
- greater than +3.402824e+38, the symbol \(+\text{INF}\) (for \(+\text{infinite}\)) is displayed.

**Instanciate**

To instanciate an object is to allocate a memory space whose size depends on the type of object to be instanciated. When an object is instanciated, it exists and can be manipulated by the program.

**INT**

\text{INT} is the abbreviation of single integer format (coded on 16 bits).
The lower and upper limits are as follows: \((-2 \text{ to the power of } 31)\) to \((2 \text{ to the power of } 31) - 1\).
Example:
\(-32768, 32767, 2\#1111110001001001, 16\#9FA4\).

**Integer literals**

Integer literal are used to enter integer values in the decimal system. The values can have a preceding sign (+/-). Individual underlines (_) between numbers are not significant.
Example:
\(-12, 0, 123_456, +986\)

**IODDT**

\text{IODDT} is the abbreviation of Input/Output Derived Data Type.
The term IODDT designates a structured data type representing a module or a channel of a PLC module. Each application expert module possesses its own IODDTs.

**Keyword**

A keyword is a unique combination of characters used as a syntactical programming language element (See annex B definition of the IEC standard 61131-3. All the keywords used in Unity Proand of this standard are listed in annex C of the IEC standard 61131-3. These keywords cannot be used as identifiers in your program (names of variables, sections, DFB types, etc.)).
LD
LD is the abbreviation of Ladder Diagram.
LD is a programming language, representing the instructions to be carried out in the form of graphic diagrams very close to a schematic electrical diagram (contacts, coils, etc.).

Located variable
A located variable is a variable for which it is possible to know its position in the PLC memory. For example, the variable `Water_pressure`, is associated with `%MW102`. `Water_pressure` is said to be localized.

Macro step
A macro step is the symbolic representation of a unique set of steps and transitions, beginning with an input step and ending with an output step.
A macro step can call another macro step.

Master task
Main program task.
It is obligatory and is used to carry out sequential processing of the PLC.

Mono Task
An application comprising a single task, and so necessarily the Master task.

Multi task
Application comprising several tasks (Mast, Fast, Auxiliary, event processing).
The order of priority for the execution of tasks is defined by the operating system of the PLC.

Multiple token
Operating mode of an SFC. In multitoken mode, the SFC may possess several active steps at the same time.
Glossary

N

Naming convention (identifier)

An identifier is a sequence of letters, numbers and underlines beginning with a letter or underline (e.g. name of a function block type, an instance, a variable or a section). Letters from national character sets (e.g: ö, ü, é, õ) can be used except in project and DFB names. Underlines are significant in identifiers; e.g. A_BCD and AB_CD are interpreted as different identifiers. Multiple leading underlines and consecutive underlines are invalid.

Identifiers cannot contain spaces. Not case sensitive; e.g. ABCD and abcd are interpreted as the same identifier.

According to IEC 61131-3 leading digits are not allowed in identifiers. Nevertheless, you can use them if you activate in dialog Tools → Project settings in tab Language extensions the check box Leading digits.

Identifiers cannot be keywords.

NAN

Used to indicate that a result of an operation is not a number (NAN = Not A Number). Example: calculating the square root of a negative number.

Note: The IEC 559 standard defines two classes of NAN: quiet NAN (QNAN) and signaling NaN (SNaN) QNAN is a NAN with the most significant fraction bit set and a SNaN is a NAN with the most significant fraction bit clear (Bit number 22). QNANs are allowed to propagate through most arithmetic operations without signaling an exception. SNaNs generally signal an invalid-operation exception whenever they appear as operands in arithmetic operations (See %SW17 and %S18).

Network

Mainly used in communication, a network is a group of stations which communicate among one another. The term network is also used to define a group of interconnected graphic elements. This group forms then a part of a program which may be composed of a group of networks.

O

Operator screen

This is an editor that is integrated into Unity Pro, which is used to facilitate the operation of an automated process. The user regulates and monitors the operation of the installation, and, in the event of any problems, can act quickly and simply.
Periodic execution

The master task is executed either cyclically or periodically. In periodic mode, you determine a specific time (period) in which the master task must be executed. If it is executed under this time, a waiting time is generated before the next cycle. If it is executed over this time, a control system indicates the overrun. If the overrun is too high, the PLC is stopped.

Procedure

Procedures are functions view technically. The only difference to elementary functions is that procedures can take up more than one output and they support data type VAR_IN_OUT. To the eye, procedures are no different than elementary functions.

Protection

Option preventing the contents of a program element to be read (read protected), or to write or modify the contents of a program element (read/write protected). The protection is confirmed by a password.

REAL

Real type is a coded type in 32 bits.

The ranges of possible values are illustrated in gray in the following diagram:

When a calculation result is:

- between -1.1754944e-38 and 1.1754944e-38 it is considered as a \( \text{DEN} \),
- less than -3.402824e+38, the symbol \( -\text{INF} \) (for - infinite) is displayed,
- greater than +3.402824e+38, the symbol \( \text{INF} \) (for +infinite) is displayed,
- undefined (square root of a negative number), the symbol \( \text{NAN} \) or \( \text{NAN} \) is displayed.
Glossary

**Real literals**
An literal real value is a number expressed in one or more decimals.
Example:
-12.0, 0.0, +0.456, 3.14159_26

**Real literals with exponent**
An literal decimal value can be expressed using standard scientific notation. The representation is as follows: mantissa + exponential.
Example:
-1.34E-12 or -1.34e-12
1.0E+6 or 1.0e+6
1.234E6 or 1.234e6

**Note:** The IEC 559 standard defines two classes of **NAN**: quiet **NAN (QNAN)** and signaling **NAN (SNaN)**. QNAN is a **NAN** with the most significant fraction bit set and a **SNaN** is a **NAN** with the most significant fraction bit clear (Bit number 22). QNANS are allowed to propagate through most arithmetic operations without signaling an exception. **SNaN** generally signal an invalid-operation exception whenever they appear as operands in arithmetic operations (See %SW17 and %S18).

**RS 232C**
Serial communication standard which defines the voltage of the following service:
- a signal of +12 V indicates a logical 0,
- a signal of -12 V indicates a logical 1.
There is, however, in the case of any attenuation of the signal, detection provided up to the limits -3 V and +3 V. Between these two limits, the signal will be considered as invalid.
RS 232 connections are quite sensitive to interference. The standard specifies not to exceed a distance of 15 m or a maximum of 9600 bauds (bits/s).

**RS 485**
Serial connection standard that operates in 10 V/+5 V differential. It uses two wires for send/receive. Their "3 states" outputs enable them to switch to listen mode when the transmission is terminated.

**RUN**
Function enabling the startup of the application program of the PLC.

**RUN Auto**
Function enabling the execution of the PLC application program to be started automatically in the case of a cold start.

**Note:** when an operand is a **DEN** (Denormalized number) the result is not significant.
Rung
A rung is the equivalent of a sequence in LD; other related terms are "Ladder network" or, more generally, "Network". A rung is inscribed between two potential bars of an LD editor and is composed of a group of graphic elements interconnected by means of horizontal or vertical connections. The dimensions of a rung are 17 to 256 lines and 11 to 64 columns maximum.

Section
Program module belonging to a task which can be written in the language chosen by the programmer (FBD, LD, ST, IL, or SFC). A task can be composed of several sections, the order of execution of the sections corresponding to the order in which they are created, and being modifiable.

SFC
SFC is the abbreviation of Sequential Function Chart. SFC enables the operation of a sequential automation device to be represented graphically and in a structured manner. This graphic description of the sequential behavior of an automation device, and the various situations which result from it, is performed using simple graphic symbols.

SFC objects
An SFC object is a data structure representing the status properties of an action or transition of a sequential chart.

Single token
Operating mode of an SFC chart for which only a single step can be active at any one time.

ST
ST is the abbreviation of Structured Text language. Structured Text language is an elaborated language close to computer programming languages. It enables you to structure series of instructions.

STRING
A variable of the type STRING is an ASCII standard character string. A character string has a maximum length of 65534 characters.

Structure
View in the project navigator with represents the project structure.

Subroutine
Program module belonging to a task (Mast, Fast) which can be written in the language chosen by the programmer (FBD, LD, ST, or IL). A subroutine may only be called by a section or by another subroutine belonging to the task in which it is declared.
**Task**
A group of sections and subroutines, executed cyclically or periodically for the MAST task, or periodically for the FAST task. A task possesses a level of priority and is linked to inputs and outputs of the PLC. These I/O are refreshed in consequence.

**TIME**
The type `TIME` expresses a duration in milliseconds. Coded in 32 bits, this type makes it possible to obtain periods from 0 to \((2^{32})-1\) milliseconds.

**Time literals**
The units of type `TIME` are the following: the days \((d)\), the hours \((h)\), the minutes \((m)\), the seconds \((s)\) and the milliseconds \((ms)\). A literal value of the type `TIME` is represented by a combination of previous types preceded by `T#`, `t#`, `TIME#` or `time#`.

**Examples:** `T#25h15m`, `t#14.7S`, `TIME#5d10h23m45s3ms`

**Time Out**
In communication projects, The Time out is a delay after which the communication is stopped if there is no answer of the target device.

**TIME_OF_DAY**
See TOD

**TOD**
TOD is the abbreviation of Time of Day. The TOD type coded in BCD in 32 bit format contains the following information:
- the hour coded in a 8-bit field,
- the minutes coded in an 8-bit field,
- the seconds coded in an 8-bit field.

**Note:** The 8 least significant bits are unused.

The Time of Day type is entered as follows: `TOD#<Hour>:<Minutes>:<Seconds>`

This table shows the lower/upper limits in each field:

<table>
<thead>
<tr>
<th>Field</th>
<th>Limits</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hour</td>
<td>[00,23]</td>
<td>The left 0 is always displayed, but can be omitted at the time of entry</td>
</tr>
<tr>
<td>Minute</td>
<td>[00,59]</td>
<td>The left 0 is always displayed, but can be omitted at the time of entry</td>
</tr>
<tr>
<td>Second</td>
<td>[00,59]</td>
<td>The left 0 is always displayed, but can be omitted at the time of entry</td>
</tr>
</tbody>
</table>
**Glossary**

**Example:** TOD#23:59:45.

**Token**
An active step of an SFC is known as a token.

**UDINT**
UDINT is the abbreviation of Unsigned Double Integer format (coded on 32 bits) unsigned. The lower and upper limits are as follows: 0 to (2 to the power of 32) - 1. Example: 0, 4294967295, 2#11111111111111111111111111111111, 8#37777777777, 16#FFFFFFF.

**UINT**
UINT is the abbreviation of Unsigned integer format (coded on 16 bits). The lower and upper limits are as follows: 0 to (2 to the power of 16) - 1. Example: 0, 65535, 2#1111111111111111, 8#177777, 16#FFFF.

**Unlocated variable**
An unlocated variable is a variable for which it is impossible to know its position in the PLC memory. A variable which have no address assigned is said to be unlocated.

**Variable**
Memory entity of the type BOOL, WORD, DWORD, etc., whose contents can be modified by the program during execution.

**Visualization window**
This window, also called a watch window, displays the variables that cannot be animated in the language editors. Only those variables that are visible at a given time in the editor are displayed.

**Watch point**
Used in the "debug" mode of the application. It enables the display of animated variables to be synchronized with the execution of a program element (containing the watch point) in order to ascertain their values at this precise point of the program.
**WORD**

The **WORD** type is coded in 16 bit format and is used to carry out processing on bit strings.

This table shows the lower/upper limits of the bases which can be used:

<table>
<thead>
<tr>
<th>Base</th>
<th>Lower limit</th>
<th>Upper limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hexadecimal</td>
<td>16#0</td>
<td>16#FFFF</td>
</tr>
<tr>
<td>Octal</td>
<td>8#0</td>
<td>8#177777</td>
</tr>
<tr>
<td>Binary</td>
<td>2#0</td>
<td>2#111111111111111</td>
</tr>
</tbody>
</table>

Representation examples:

<table>
<thead>
<tr>
<th>Data content</th>
<th>Representation in one of the bases</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000000011010011</td>
<td>16#D3</td>
</tr>
<tr>
<td>1010101010101010</td>
<td>8#125252</td>
</tr>
<tr>
<td>0000000011010011</td>
<td>2#11010011</td>
</tr>
</tbody>
</table>
Symbols

%S0, 132
%S1, 132
%S10, 133
%S100, 144
%S11, 133
%S118, 144
%S119, 144
%S12, 133
%S120, 144
%S121, 144
%S122, 144
%S13, 133
%S15, 135
%S16, 135
%S17, 135
%S18, 136
%S19, 136
%S20, 137
%S21, 137
%S30, 138
%S31, 138
%S32, 138
%S33, 138
%S34, 138
%S35, 138
%S38, 138
%S39, 138
%S4, 132
%S40, 139
%S41, 139
%S42, 139
%S43, 139
%S44, 139
%S45, 139
%S46, 139
%S47, 139
%S48, 139
%S49, 139
%S5, 132
%S50, 139
%S51, 140
%S59, 140
%S6, 132
%S60, 141
%S67, 141
%S68, 141
%S7, 132
%S75, 142
%S76, 142
%S77, 142
%S78, 142
%S80, 143
%S9, 133
%S90, 143
%S91, 143
%S92, 143
%S94, 143
%SW0, 146
%SW1, 146
%SW10, 147
%SW108, 158
%SW109, 158
%SW11, 147
%SW116, 158
%SW12, 149
| %SW124, 159 | %SW3, 146 |
| %SW125, 159 | %SW30, 151 |
| %SW126, 160 | %SW31, 151 |
| %SW127, 160 | %SW32, 151 |
| %SW128 | %SW33, 151 |
| Quantum, 173 | %SW34, 151 |
| %SW128...143 | %SW340, 176 |
| Premium, 162 | %SW341 to %SW404, 177 |
| %SW129, 173 | %SW35, 151 |
| %SW13, 149 | %SW36, 152 |
| %SW130, 173 | %SW37, 152 |
| %SW132, 173 | %SW38, 152 |
| %SW133, 173 | %SW39, 152 |
| %SW134, 173 | %SW4, 146 |
| %SW135, 173 | %SW40, 152 |
| %SW136, 173 | %SW405, 177 |
| %SW137, 173 | %SW406 to %SW469, 178 |
| %SW138, 173 | %SW41, 152 |
| %SW139, 174 | %SW42, 152 |
| %SW14, 149 | %SW43, 152 |
| %SW140, 174 | %SW44, 152 |
| %SW141, 174 | %SW45, 152 |
| %SW142, 174 | %SW46, 152 |
| %SW143, 174 | %SW47, 152 |
| %SW144, 163, 174 | %SW470, 178 |
| %SW145, 164, 174 | %SW471 to %SW534, 179 |
| %SW146, 164, 174 | %SW48, 153 |
| %SW147, 165, 174 | %SW49, 153 |
| %SW148, 165, 174 | %SW5, 146 |
| %SW149, 174 | %SW50, 153 |
| %SW15, 149 | %SW51, 153 |
| %SW150, 165, 174 | %SW52, 153 |
| %SW151, 165, 175 | %SW53, 153 |
| %SW152, 165 | %SW535, 180 |
| %SW153, 166 | %SW36 to %SW538, 181 |
| %SW154, 168 | %SW539 to %SW541, 181 |
| %SW155, 169 | %SW54, 154 |
| %SW16, 149 | %SW542 to %SW544, 182 |
| %SW160 to %SW167, 175 | %SW545 to %SW547, 184 |
| %SW160 to %SW168, 169 | %SW545 to %SW640, 183 |
| %SW168 to %SW171, 175 | %SW55, 154 |
| %SW17, 150 | %SW56, 154 |
| %SW18, 150 | %SW57, 154 |
| %SW180 to %SW339, 176 | %SW58, 154 |
| %SW19, 150 | %SW59, 154 |
| %SW2, 146 | %SW60, 171 |
%SW61, 172
%SW62, 172
%SW63, 172
%SW70, 155
%SW71, 155
%SW75, 155
%SW76, 155
%SW77, 155
%SW78, 156
%SW8, 147
%SW80, 156
%SW81, 156
%SW82, 156
%SW83, 156
%SW84, 156
%SW85, 156
%SW86, 156
%SW87, 156
%SW88, 156
%SW89, 156
%SW9, 147
%SW90, 157
%SW99, 157

Addition
   IL, 431
   ST, 472
Addressing, 263
   Syntax, 263
ADJDATETIME, 154
Alternative Branch
   SFC, 385
Alternative Joint
   SFC, 385
Alternative Sequence
   SFC, 385
Alternative String
   SFC, 391, 402
AND
   IL, 429
   ST, 474
ANY, 245
APPservcnt, 156
ARRAY, 225
ASNSERVcnt, 156
Assignment
   ST, 478
Asymmetric Parallel String Selection
   SFC, 397
Auto RUN, 127
Automatic RUN, 127
AUX, 73
AUX0ACT, 138
AUX0CURRTIME, 152
AUX0MAXTIME, 152
AUX0MINTIME, 152
AUX0PERIOD, 146
AUX1ACT, 138
AUX1CURRTIME, 152
AUX1MAXTIME, 152
AUX1MINTIME, 152
AUX1PERIOD, 146
AUX2ACT, 138
AUX2CURRTIME, 152
AUX2MAXTIME, 152
AUX2MINTIME, 152
AUX2PERIOD, 146
AUX3ACT, 138
AUX3CURRTIME, 152
AUX3MAXTIME, 152
Index

AUX3MINTIME, 152
AUX3PERIOD, 146

B
BACKUPCHVOV, 141
BAOPMOD, 163
BAPARAM, 164
BASTATUS, 164
Bit
forced, 199
BLKERRTYPE, 159
Block Call
IL, 442
ST, 495
BOARD_STS, 175
BOOL, 199
BOOL/EBOOL
compatibility, 203
Brackets
ST, 471
BW_GLBD_IOS, 174
BW_OTHER_MSG, 174
BYTE, 222

C
C
IL, 424
CAERRCNTI, 181
CAL
IL, 435
Call coil
LD, 318
CARRY, 135
CASE...OF...END_CASE
ST, 483
CBERRCNTI, 181
Change Execution Sequence
FBD, 304, 347
CN
IL, 424
Coil
LD, 317
Coil for detecting negative transitions
LD, 317
Coil for detecting positive transitions
LD, 317
Cold start, 121
COLDSTART, 132
Comment
FBD, 301
IL, 440
LD, 335
SFC, 388
ST, 493
Compatibility, 25
Compatibility between data types, 250
Complement
ST, 471
COMRFSH, 143
Connector
FBD, 300
LD, 334
Contact
LD, 315
Contact for detecting negative transitions
LD, 316
Contact for detecting positive transitions
LD, 316
Contact to recognize negative transitions
LD, 336
Contact to recognize positive transitions
LD, 336
Control element
LD, 329
Control Elements
FBD, 298
CPUERR, 159
Cycle
Master task, 82
Cyclic, 84

D
Data Flow
LD, 343
Data flow
FBD, 302
Data Instances
Direct Addressing, 263
## Index

### Data instances
- attributes, 260
- located, 257
- Overview, 191
- unlocated, 256

### Data references
- by address, 277
- by name, 274
- by value, 272
- Overview, 192

### Data types
- Boolean types, 199
- data types belonging to the Sequential Function Chart family, 248
- data types in BCD format, 208
- Derived Data Types, 229
- Function Block types, 238
- General overview, 189
- Generic Data Types, 245
- Input/Output Derived Data Types, 235
- integer types, 204
- Real type, 214
- the bit string type, 222
- the character string type, 217
- the Date and Time (DT) type, 212
- the Date type, 210
- the Time of Day (TOD) type, 211
- the Time type, 206

<table>
<thead>
<tr>
<th>DATE, 210</th>
<th>DATE AND TIME, 212</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDATEWEEK, 153</td>
<td>DDT, 228</td>
</tr>
<tr>
<td>Memory occupancy, 232</td>
<td></td>
</tr>
</tbody>
</table>

### Delay Time
- SFC, 362

### Derived Function Block
- FBD, 290
- IL, 447
- LD, 321
- ST, 501

### Detect positive transition
- LD, 316

### Detecting negative transitions
- LD, 316, 317

### Detecting positive transitions
- LD, 317

### DFB, 238
- Code section, 529
- FBD, 290
- Function Block Diagram language, 555
- IL, 447
- Instance, 532
- Instruction List, 551
- Internal Data, 524
- IODD, 542
- Ladder language, 546
- LD, 321
- Parameters, 526
- Rules of use, 540
- ST, 501
- Structured Text, 548
- Variables, 528

### DFB Call
- FBD, 290
- IL, 447
- LD, 321
- ST, 501

### DFB instance
- Execution, 534
- DIAGBUFFCONF, 142
- DIAGBUFFFULL, 142
- Diagnostics DFB, 557
- DINT, 204
- DIOERRNO1, 144
- DIOERRNO2, 144
- DIOERRPLC, 144
- DIV
- IL, 431
- Division
- IL, 431
- ST, 472
- DLASTDEREG, 155
- DLASTREG, 155
- DNBERRBUFF, 156
- DWORD, 222

### E
- EBOOL, 199
- Edge recognition
- LD, 336
Index

EF
FBD, 288
IL, 442
LD, 319
ST, 495

EFB, 238
FBD, 288
IL, 447
LD, 320
ST, 501

EFB Call
IL, 447
ST, 501

EFB call
FBD, 288
LD, 320

Elementary Function
FBD, 288
IL, 442
LD, 319
ST, 495

Elementary Function Block
FBD, 288
IL, 447
LD, 320
ST, 501

ELSE
ST, 481

ELSIF...THEN
ST, 482

Empty Instruction
ST, 491

EN
FBD, 294
IL, 446, 454, 461
LD, 326
ST, 499, 506, 512

ENO
FBD, 294
IL, 446, 454, 461
LD, 326
ST, 499, 506, 512

EQ
IL, 433

Equal to
IL, 433

Equality
ST, 474

ERRADDRi, 160
ERRORCNXi, 162

Event processing
Programming, 106

EVTi, 75
EVTOVR, 138
EXCHGTIME, 143

Execution
Cyclic, 84
mono task, 82
Periodic, 85

Execution Sequence
LD, 343
SFC, 390, 401

Execution sequence
FBD, 302
IL, 421
ST, 468
EXIT
ST, 488

Exponentiation
ST, 471

F
FAST, 72
FASTACT, 138
FASTCURRTIME, 151
FASTMAXTIME, 151
FASTMINTIME, 151
FASTPERIOD, 146
FBD, 285
FFB
FBD, 288
IL, 441
LD, 319
ST, 494
FipioERR0, 166
FipioERR1, 168
Floating point, 214
Floating point number, 214
FLOATSTAT, 150
FOR...TO...BY...DO...END_FOR
ST, 484
FORCEDANA, 158
FORCEDIOIM, 158

Formal Parameters
FBD, 291
IL, 442, 447, 457
LD, 322
ST, 495, 501, 509

Function
FBD, 288
IL, 442
LD, 319
ST, 495

Function Block
FBD, 288
IL, 447
LD, 320
ST, 501

Function Block Call
FBD, 288
IL, 447
LD, 320
ST, 501

Function Block language, 285

Function Call
FBD, 288
IL, 442
LD, 319
ST, 471, 495

FUNCTIONNAME
IL, 435

FW_VERSION, 174

H
HALT, 128
HALT mode, 128
HALTIFERROR, 142
Horizontal Matching Block
LD, 331
HOURMIN, 153
HSB_CMD, 171
HSBSTS, 172
HSBY_REVERSEi, 172

I
I/O Addressing
Syntax, 263
Identifier
SFC, 373
IEC Compliance, 601
IEVTNB, 153
IF...THEN...END_IF
ST, 480
IL, 417
INDEXOVF, 137
Inequality
ST, 474
Initial Step, 361
Input Step
SFC, 365
INPUTADR/SWAP, 157
Inputs/Outputs
syntax, 267
Instruction
IL, 419
ST, 476, 477
Instruction list, 417
INT, 204
IODDT, 235
IOERR, 133
IOERRCNT1, 184
IOERRCNTi, 183
IOERRTSK, 135
IOHEALTHij, 176
IORETRY1, 184
IORETRYi, 183
IP_ADDR, 174
Index

IP_GATEWAY, 174
IP_NETMASK, 174

J

JMP
   IL, 436
   ST, 492
Jump
   FBD, 298
   IL, 436, 438
   LD, 329
   SFC, 382
   ST, 492
Jump into a Parallel String
   SFC, 409
Jump out of a Parallel String
   SFC, 411

K

KEY_SWITCH, 155

L

Label
   FBD, 298
   LD, 329
   ST, 492
Labels
   ST, 492
Ladder Diagram, 311
LOCASYNREQ, 143
LD, 311
   IL, 427
LE
   IL, 434
left bracket
   IL, 425
Less than comparison
   IL, 434
   ST, 473
Less than or equal to comparison
   IL, 434
   ST, 473

Link
   FBD, 299
   LD, 332
   SFC, 383
Link rules
   SFC, 359
LOCIOERR, 144
Logical AND
   IL, 429
   ST, 474
Logical Exclusive OR
   ST, 475
Logical exclusive OR
   IL, 429
Logical OR
   IL, 429
   ST, 475
Loops
   FBD, 309
   LD, 345
LT
   IL, 434

M

MAC_ADDRi, 174
Macro Section
   SFC, 365
Macro Step
   SFC, 365
MAST, 71
MASTACT, 138
MASTCURRTIME, 151
Master, 71
MASTMINTIME, 151
MASTMAXTIME, 151
MASTPERIOD, 146
Maximum Supervision Time
   SFC, 362
MAXREQNB, 157
MB+DIOSLOT, 176
MB+IOHEALTHi, 177
Memory
   Atrium, 110
   Premium, 110
   Quantum, 113
Minimum Supervision Time
SFC, 362
MOD
IL, 432
Modifier
IL, 424
Modulo
IL, 432
ST, 472
MODUNHEALTH1, 184
MODUNHEALTHi, 183
Mono task, 82
MONTHDAY, 153
MSGCNT0, 156
MSGCNT1, 156
MSGCNT2, 156
MSGCNT3, 156
MSGCNT4, 156
MSGCNT5, 156
MSGCNT6, 156
MSTSERVCNT, 156
MUL
IL, 431
Multi Token
SFC, 357
Multiplication
IL, 431
ST, 472
Multitasking, 89, 91
Multi-Token
SFC, 401, 402, 405, 409, 411

N
N
IL, 424
NB_DENIED_CNX, 173
NB_IOS_CNX, 173
NB_IOS_MSG, 173
NB_P502_CNX, 173
NB_P502_REF, 173
NB_RCV_MSG, 173
NB_SENT_MSG, 173
NBEXPLFIP, 169
NBFRRREC, 165
NBFRRSENT, 165
NBRESENTMSG, 165
NE
IL, 434
negated coil
LD, 317
Negation
ST, 471
Nested DDT, 228
NOM1DIO SLOT, 177
NOM1HEALTHi, 178
NOM2DIOHEALTHi, 179
NOM2DIO SLOT, 178
Normally closed
LD, 316
Normally open
LD, 316
NOT
IL, 430
ST, 471
Not equal to
IL, 434

O
Operand
IL, 422
ST, 469
Operation block
LD, 330
Operator
IL, 426
ST, 471
OR
IL, 429
ST, 475
OSCOMMPATCH, 149
OSCOMMVERS, 149
OSINTVERS, 149
OUTDIS, 133
Output Step
SFC, 366
OVERFLOW, 136
OVERRUN, 136
Index

P
Parallel Branch
   SFC, 387, 409, 411
Parallel Joint
   SFC, 387, 409, 411
Parallel Sequence
   SFC, 387
Parallel String
   SFC, 397
Parallel Strings
   SFC, 395, 405
PCMCIABAT0, 141
PCMCIABAT1, 142
Periodic, 85
PLCBAT, 141
PLCRUNNING, 133
Power outage, 118
Power restoral, 118
PREMRACK0 to PREMRACK7, 169
Procedure
   FBD, 290
   IL, 457
   LD, 321
   ST, 509
Procedure Call
   IL, 457
   ST, 509
PROCEDURENAME
   IL, 435
Processing
   Event, 105
   Events, 75
PROTTERINL, 144
Public Variables
   FBD, 293
   IL, 448
   LD, 325
   ST, 502
RACK2ERR, 139
RACK3ERR, 139
RACK4ERR, 139
RACK5ERR, 139
RACK6ERR, 139
RACK7ERR, 139
REAL, 214
Recognizing negative transitions
   LD, 336
Recognizing positive transitions
   LD, 336
REFRESH _IO, 175
REMOERR, 144, 158
Repeat Instruction
   ST, 484, 486, 487, 488
REPEAT...UNTIL...END_REPEAT
   ST, 487
Reset coil
   LD, 318
RET
   IL, 436
RETURN
   ST, 490
Return
   FBD, 298
   IL, 436
   LD, 329
   ST, 490
right bracket
   IL, 436
RIOERRSTAT, 180
RSTMSGCNT, 143
RTCERR, 140
RTCTUNING, 140
RTCWRITE, 139
Rules
   data naming, 281
   Syntax of typenstance names, 193

S
S
   IL, 428
SAVECURRVAL, 143
SEC, 153
Section, 77
Index

Select Instruction
  ST, 480, 481, 482, 483
Sequence Jump
  SFC, 392
Sequence Language, 353
Sequence Loop
  SFC, 393
Set coil
  LD, 318
  SFC, 353
  Action, 370
SFC Section, 79
SFCCHART_STATE
  SFC, 357
SFCSTEP_STATE, 364
SFCSTEP_TIMES, 363
Signal Flow
  LD, 343
  SFC, 390, 401
Signal flow
  FBD, 302
Single Token
  SFC, 357
Single-Token
  SFC, 390, 391, 392, 395, 397
ST, 465
  IL, 427
Step
  SFC, 361
Step Dwell Time
  SFC, 362
Step Variable, 364
Stop coil
  LD, 318
STOPDAY, 154
STOPHM, 154
STOPMD, 154
STOPSEC, 154
STOPYEAR, 154
STRING, 217
STRING type, 217
STRINGERROR, 135
Structured Text, 465
Structures, 228
SUB
  IL, 431
SUB
  IL, 431
Subroutine, 80
Subroutine Call
  FBD, 297
  IL, 437
  ST, 489
Subroutine call
  LD, 318
Subtraction
  IL, 431
  ST, 472
Syntax, 263
T
  Table, 225
Tag
  IL, 438
  LD, 329
Task
  Fast, 72
  Master, 71
Tasks
  Auxiliary, 73
TB100MS, 132
TB10MS, 132
TB1MIN, 132
TB1SEC, 132
TCRFAST, 165
TCRMAST, 165
Text Object
  FBD, 301
  LD, 335
  SFC, 388
TIME, 206
TIME OF DAY, 211
TIMEREVTNB, 155
Token
  SFC, 357
Topologic Addressing, 263
Transition, 378
SFC, 378
Transition Section
  SFC, 380
Transition Variable
  SFC, 381
TSKINHIBIN, 147
Index

TSKIHIBOUT, 147
TSKINIT, 147

U
UDINT, 204
UINT, 204
User function block
  DFB, 518
  UTPORTADDR, 149

V
VALID_GD, 175
VAR_IN_OUT Variable
  IL, 455, 462
VAR_IN_OUT-Variable
  FBD, 296
    LD, 328
    ST, 507, 513

W
Warm restart, 125
WARMSTART, 132
Watchdog, 86
WDG, 133
WDVALUE, 147
WEEKOFYEAR, 155
WHILE...DO...END_WHILE
  ST, 486
WORD, 222

X
XOR
  IL, 429
    ST, 475
  XWAYNETWADDR, 149

Y
YEAR, 153