



DPC: 22 / 30434346 DC

BSI Group Headquarters

389 Chiswick High Road London W4 4AL

Tel: +44 (0)20 8996 9000

Fax: +44 (0)20 8996 7400

www.bsigroup.com

Date: 21 June 2022
Origin: International**Latest date for receipt of comments: 20 July 2022**

Project No. 2021/00478

Responsible committee: GEL/65 Measurement and control

Interested committees:

Title: BS EN IEC 61131-3 Programmable controllers

Part 3: Programming languages

Please notify the secretary if you are aware of any keywords that might assist in classifying or identifying the standard or if the content of this standard

- i) has any issues related to 3rd party IPR, patent or copyright
- ii) affects other national standard(s)
- iii) requires additional national guidance or information

**WARNING: THIS IS A DRAFT AND MUST NOT BE REGARDED OR USED AS A BRITISH STANDARD.
THIS DRAFT IS NOT CURRENT BEYOND 20 July 2022**

This draft is issued to allow comments from interested parties; all comments will be given consideration prior to publication. No acknowledgement will normally be sent. **See overleaf for information on the submission of comments.**

No copying is allowed, in any form, without prior written permission from BSI except as permitted under the Copyright, Designs and Patent Act 1988 or for circulation within a nominating organization for briefing purposes. Electronic circulation is limited to dissemination by e-mail within such an organization by committee members.

Further copies of this draft may be purchased from BSI Shop <http://shop.bsigroup.com> or from BSI Customer Services, Tel: +44(0) 20 8996 9001 or email cservices@bsigroup.com. British, International and foreign standards are also available from BSI Customer Services.

Information on the co-operating organizations represented on the committees referenced above may be obtained from <http://standardsdevelopment.bsigroup.com>

Responsible Committee Secretary: Ms Charmaine Mukherjee (BSI)
Direct tel:
E-mail: charmaine.mukherjee@bsigroup.com

Introduction

This draft standard is based on national and international discussions. Your comments on this draft are invited and will assist in the preparation of the consequent standard.

For international standards, comments will be reviewed by the relevant UK national committee before sending the consensus UK vote and comments to the international committee, which will then decide appropriate action. If the international standard is approved, it is usual for the text to be published as a British Standard.

For national standards, comments will be reviewed by the relevant UK national committee and the resulting standards published as a British Standard.

UK Vote

Please indicate whether you consider the UK should submit a negative (with supporting technical reasons) or positive vote on this draft. Please indicate if you are aware of any reason why this draft standard should not be published as a British Standard.

Submission of Comments

- **Annotated drafts are not acceptable and will be rejected.**
- All comments should be submitted online at <https://standardsdevelopment.bsigroup.com/>. You will need to register in order to comment.

Template for comments and secretariat observations

Date: xx/xx/20xx	Document: ISO/DIS xxxx
------------------	------------------------

1	2	(3)	4	5	(6)	7
MB	Clause No./ Subclause No./Annex (e.g. 3.1)	Paragraph/Figure/ Table/Note	Type of comment	Commend (justification for change) by the MB	Proposed change by the MB	Secretariat observations on each comment submitted
	3.1	Definition 1	ed	Definition is ambiguous and needs clarifying.	Amend to read '...so that the mains connector to which no connection...'	
	6.4	Paragraph 2	te	The use of the UV photometer as an alternative cannot be supported as serious problems have been encountered in its use in the UK.	Delete reference to UV photometer.	



PROJECT NUMBER: IEC 61131-3 ED4	
DATE OF CIRCULATION: 2022-05-27	CLOSING DATE FOR COMMENTS: 2022-08-19
SUPERSEDES DOCUMENTS: 65B/1195/RR	

IEC SC 65B : MEASUREMENT AND CONTROL DEVICES	
SECRETARIAT: United States of America	SECRETARY: Mr Wallie Zoller
OF INTEREST TO THE FOLLOWING COMMITTEES:	PROPOSED HORIZONTAL STANDARD: ☐ Other TC/SCs are requested to indicate their interest, if any, in this CD to the secretary.
FUNCTIONS CONCERNED: ☐ EMC ☐ ENVIRONMENT ☐ QUALITY ASSURANCE ☐ SAFETY	

This document is still under study and subject to change. It should not be used for reference purposes.

Recipients of this document are invited to submit, with their comments, notification of any relevant patent rights of which they are aware and to provide supporting documentation.

TITLE: Programmable controllers - Part 3: Programming languages

NOTE FROM TC/SC OFFICERS:

Copyright © 2022 International Electrotechnical Commission, IEC. All rights reserved. It is permitted to download this electronic file, to make a copy and to print out the content for the sole purpose of preparing National Committee positions. You may not copy or "mirror" the file or printed version of the document, or any part of it, for any other purpose without permission in writing from IEC.

CONTENTS

1		
2		
3	FOREWORD.....	7
4	1 Scope.....	9
5	2 Normative references	9
6	3 Terms and definitions	9
7	4 Architectural models	17
8	4.1 Software model	17
9	4.2 Communication model	19
10	4.3 Programming model	20
11	5 Compliance	22
12	5.1 General	22
13	5.2 Feature tables	22
14	5.3 Implementer's compliance statement	22
15	6 Common elements	24
16	6.1 Use of printed characters	24
17	6.1.1 Character set.....	24
18	6.1.2 Identifiers	24
19	6.1.3 Keywords	24
20	6.1.4 Use of white space	25
21	6.1.5 Comments	25
22	6.2 Pragma	26
23	6.3 Literals – External representation of data	26
24	6.3.1 General	26
25	6.3.2 Numeric literals and string literals	26
26	6.3.3 Character string literals	27
27	6.3.4 Duration literal	29
28	6.3.5 Date and time of day literal	30
29	6.4 Data types	30
30	6.4.1 General	30
31	6.4.2 Elementary data types (BOOL, INT, REAL, STRING, etc.)	31
32	6.4.3 Generic data types	35
33	6.4.4 User-defined data types	36
34	6.5 Variables	49
35	6.5.1 Declaration and initialization of variables	49
36	6.5.2 Variable sections	51
37	6.5.3 Variable length ARRAY variables	53
38	6.5.4 Constant variables	55
39	6.5.5 Directly represented variables (%)	56
40	6.5.6 Retentive variables (RETAIN, NON_RETAIN)	58
41	6.6 Program organization units (POUs)	60
42	6.6.1 Common features for POU's	60
43	6.6.2 Functions	74
44	6.6.3 Function blocks	105
45	6.6.4 Programs	126
46	6.6.5 Classes	126

47	6.6.6	Interface	147
48	6.6.7	Object-oriented features for function block types	156
49	6.6.8	Polymorphism	162
50	6.7	Sequential Function Chart (SFC) elements	165
51	6.7.1	General	165
52	6.7.2	Steps	165
53	6.7.3	Transitions	167
54	6.7.4	Actions	170
55	6.7.5	Rules of evolution	177
56	6.8	Configuration elements	185
57	6.8.1	General	185
58	6.8.2	Tasks	189
59	6.9	Namespaces	195
60	6.9.1	General	195
61	6.9.2	Declaration	195
62	6.9.3	Usage	201
63	6.9.4	Namespace directive USING	202
64	7	Textual languages	205
65	7.1	Common elements	205
66	7.2	~~Instruction list (IL)~~	~~205~~
67	7.2.1	~~General~~	~~205~~
68	7.2.2	~~Instructions~~	~~205~~
69	7.2.3	~~Operators, modifiers and operands~~	~~205~~
70	7.2.4	~~Functions and function blocks~~	~~205~~
71	7.3	Structured Text (ST)	205
72	7.3.1	General	205
73	7.3.2	Expressions	206
74	7.3.3	Statements	207
75	8	Graphical languages	213
76	8.1	Common elements	213
77	8.1.1	General	213
78	8.1.2	Representation of variables and instances	213
79	8.1.3	Representation of lines and blocks	215
80	8.1.4	Direction of flow in networks	216
81	8.1.5	Evaluation of networks	217
82	8.1.6	Execution control elements	218
83	8.2	Ladder diagram (LD)	219
84	8.2.1	General	219
85	8.2.2	Power rails	220
86	8.2.3	Link elements and states	220
87	8.2.4	Contacts	220
88	8.2.5	Coils	222
89	8.2.6	Functions and function blocks	223
90	8.2.7	Order of network evaluation	223
91	8.3	Function Block Diagram (FBD)	223
92	8.3.1	General	223
93	8.3.2	Combination of elements	223
94	8.3.3	Order of network evaluation	224
95	Annex A (normative)	Formal specification of the languages elements	225

96	Annex B (informative) List of major changes and extensions of the 4 th edition.....	232
97	Bibliography.....	233
98	Figure 1 – Software model.....	18
99	Figure 2 – Communication model.....	20
100	Figure 3 – Combination of programmable controller language elements.....	21
101	Figure 4 – Implementer's compliance statement (Example).....	23
102	Figure 5 – Hierarchy of the generic data types	35
103	Figure 6 – Initialization by literals and constant expressions (Rules)	36
104	Figure 7 – Variable declaration keywords (Summary).....	52
105	Figure 8 – Usage of <code>VAR_GLOBAL</code> , <code>VAR_EXTERNAL</code> and <code>CONSTANT</code> (Rules).....	53
106	Figure 9 – Conditions for the initial value of a variable (Rules).....	59
107	Figure 10 – Formal and non-formal representation of call (Examples)	65
108	Figure 11 – Data type conversion rules – implicit and/or explicit (Summary)	69
109	Figure 12 – Implicit type conversions overview	70
110	Figure 13 – Usage of function block input and output parameters (Rules)	116
111	Figure 14 – Usage of function block input and output parameters (Illustration of rules)	117
112	Figure 15 – Standard timer function blocks – timing diagrams (Rules)	124
113	Figure 16 – Recommended implementation in pseudo code for timer	126
114	Figure 17 – Overview of inheritance and interface implementation	129
115	Figure 18 – Inheritance of classes (Illustration)	138
116	Figure 19 – Interface with derived classes (Illustration)	149
117	Figure 20 – Inheritance of interface and class (Illustration)	153
118	Figure 21 – Function block types with optional body and methods (Illustration)	159
119	Figure 22 – Inheritance of function block body with <code>SUPER()</code> (Example).....	161
120	Figure 23 – <code>ACTION_CONTROL</code> function block – External interface (Summary).....	174
121	Figure 24 – <code>ACTION_CONTROL</code> function block body (Summary)	175
122	Figure 25 – Action control (Example)	177
123	Figure 26 – SFC evolution (Rules)	183
124	Figure 27 – SFC errors (Example)	184
125	Figure 28 – Configuration (Example).....	186
126	Figure 29 – <code>CONFIGURATION</code> and <code>RESOURCE</code> declaration (Example)	189
127	Figure 30 – Accessibility using namespaces (Rules)	198
128	Figure 31 – Common textual elements (Summary)	205
129	Table 1 – Character set	24
130	Table 2 – Identifiers	24
131	Table 3 – Comments.....	25
132	Table 4 – Pragma	26
133	Table 5 – Numeric literals	27
134	Table 6 – Character string literals	28
135	Table 7 – Two-character combinations in character strings	29
136	Table 8 – Duration literals.....	30
137	Table 9 – Date and time of day literals.....	30
138	Table 10 – Elementary data types.....	

139	Table 11 – Declaration of user-defined data types and initialization	37
140	Table 12 – Reference operations	48
141	Table 13 – Declaration of variables	50
142	Table 14 – Initialization of variables	51
143	Table 15 – Variable-length <code>ARRAY</code> variables	54
144	Table 16 – Directly represented variables	56
145	Table 17 – Partial access of <code>ANY_BIT</code> variables	62
146	Table 18 – Execution control graphically using <code>EN</code> and <code>ENO</code>	67
147	Table 19 – Function declaration	75
148	Table 20 – Function call	77
149	Table 21 – Typed and overloaded functions	79
150	Table 22 – Data type conversion function	82
151	Table 23 – Data type conversion of numeric data types	83
152	Table 24 – Data type conversion of bit data types	86
153	Table 25 – Data type conversion of bit and numeric types	88
154	Table 26 – Data type conversion of date and time types	90
155	Table 27 – Data type conversion of character types	91
156	Table 28 – Numerical and arithmetic functions	92
157	Table 29 – Arithmetic functions	93
158	Table 30 – Bit shift functions	94
159	Table 31 – Bitwise Boolean functions	94
160	Table 32 – Selection functions ^d	95
161	Table 33 – Comparison functions	96
162	Table 34 – Character string functions	97
163	Table 35 – Numerical functions of time and duration data types	98
164	Table 36 – Additional functions of time data types <code>CONCAT</code> and <code>SPLIT</code>	100
165	Table 37 – Function for endianness conversion	104
166	Table 38 – Functions of enumerated data types	104
167	Table 39 – Validate functions	105
168	Table 40 – Function block type declaration	106
169	Table 41 – Function block instance declaration	111
170	Table 42 – Function block call	113
171	Table 43 – Standard bistable function blocks ^a	120
172	Table 44 – Standard edge detection function blocks	121
173	Table 45 – Standard counter function blocks	121
174	Table 46 – Standard timer function blocks	123
175	Table 47 – Program declaration	127
176	Table 48 – Class	130
177	Table 49 – Class instance declaration	132
178	Table 50 – Textual call of methods – Formal and non-formal parameter list	135
179	Table 51 – Interface	148
180	Table 52 – Assignment attempt	156
181	Table 53 – Object-oriented function block type	156

182	Table 54 – SFC step	166
183	Table 55 – SFC transition and transition condition	168
184	Table 56 – SFC declaration of actions	170
185	Table 57 – Step/action association	171
186	Table 58 – Action block.....	172
187	Table 59 – Action qualifiers.....	172
188	Table 60 – Action control features.....	177
189	Table 61 – Sequence evolution – graphical.....	178
190	Table 62 – Configuration and resource declaration	188
191	Table 63 – Task.....	191
192	Table 64 – Namespace	200
193	Table 65 – Nested namespace declaration options	201
194	Table 66 – Namespace directive <code>USING</code>	204
195	Table 67 – Operators of the ST language.....	207
196	Table 68 – ST language statements	208
197	Table 69 – Graphical execution control elements	219
198	Table 70 – Power rails and link elements	220
199	Table 71 – Contacts.....	221
200	Table 72 – Coils.....	222
201		
202		

INTERNATIONAL ELECTROTECHNICAL COMMISSION

PROGRAMMABLE CONTROLLERS –

Part 3: Programming languages

FOREWORD

- 1) The International Electrotechnical Commission (IEC) is a worldwide organization for standardization comprising all national electrotechnical committees (IEC National Committees). The object of IEC is to promote international co-operation on all questions concerning standardization in the electrical and electronic fields. To this end and in addition to other activities, IEC publishes International Standards, Technical Specifications, Technical Reports, Publicly Available Specifications (PAS) and Guides (hereafter referred to as “IEC Publication(s)”). Their preparation is entrusted to technical committees; any IEC National Committee interested in the subject dealt with may participate in this preparatory work. International, governmental and non-governmental organizations liaising with the IEC also participate in this preparation. IEC collaborates closely with the International Organization for Standardization (ISO) in accordance with conditions determined by agreement between the two organizations.
- 2) The formal decisions or agreements of IEC on technical matters express, as nearly as possible, an international consensus of opinion on the relevant subjects since each technical committee has representation from all interested IEC National Committees.
- 3) IEC Publications have the form of recommendations for international use and are accepted by IEC National Committees in that sense. While all reasonable efforts are made to ensure that the technical content of IEC Publications is accurate, IEC cannot be held responsible for the way in which they are used or for any misinterpretation by any end user.
- 4) In order to promote international uniformity, IEC National Committees undertake to apply IEC Publications transparently to the maximum extent possible in their national and regional publications. Any divergence between any IEC Publication and the corresponding national or regional publication shall be clearly indicated in the latter.
- 5) IEC itself does not provide any attestation of conformity. Independent certification bodies provide conformity assessment services and, in some areas, access to IEC marks of conformity. IEC is not responsible for any services carried out by independent certification bodies.
- 6) All users should ensure that they have the latest edition of this publication.
- 7) No liability shall attach to IEC or its directors, employees, servants or agents including individual experts and members of its technical committees and IEC National Committees for any personal injury, property damage or other damage of any nature whatsoever, whether direct or indirect, or for costs (including legal fees) and expenses arising out of the publication, use of, or reliance upon, this IEC Publication or any other IEC Publications.
- 8) Attention is drawn to the Normative references cited in this publication. Use of the referenced publications is indispensable for the correct application of this publication.
- 9) Attention is drawn to the possibility that some of the elements of this IEC Publication may be the subject of patent rights. IEC shall not be held responsible for identifying any or all such patent rights.

This edition of IEC 61131-3 follows the third edition, published in 2013 and it constitutes a technical revision. The main extension is the inclusion of UTF-8 strings and their associated functions.

See Annex B for a comprehensive list of features that have been added, removed or deprecated in comparison to IEC 61131-3:2013.

251

252 This publication has been drafted in accordance with the ISO/IEC Directives, Part 2.

253 A list of all the parts in the IEC 61131 series, published under the general title *Programmable*
254 *controllers* can be found on the IEC website.

255 The committee has decided that the contents of this publication will remain unchanged until
256 the stability date indicated on the IEC web site under "<http://webstore.iec.ch>" in the data re-
257 lated to the specific publication. At this date, the publication will be

- 258 • reconfirmed,
- 259 • withdrawn,
- 260 • replaced by a revised edition, or
- 261 • amended.

262

263

PROGRAMMABLE CONTROLLERS –

Part 3: Programming languages

1 Scope

This part of IEC 61131 specifies the syntax and semantics of programming languages for programmable controllers as defined in Part 1 of IEC 61131.

This part of IEC 61131 specifies the syntax and semantics of a unified suite of programming languages for programmable controllers (PCs). This suite consists of the textual language Structured Text (ST), and the graphical languages, Ladder Diagram (LD) and Function Block Diagram (FBD).

An additional set of graphical and equivalent textual elements named Sequential Function Chart (SFC) is defined for structuring the internal organization of programs and function blocks. Also, configuration elements are defined which support the installation of programmable controller programs into programmable controller systems.

In addition, features are defined which facilitate communication among programmable controllers and other components of automated systems.

2 Normative references

The following documents, in whole or in part, are normatively referenced in this document and are indispensable for its application. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

IEC 61131-1, *Programmable controllers – Part 1: General information*

IEC 61131-5, *Programmable controllers – Part 5: Communications*

ISO/IEC 10646, *Information technology – Universal Coded Character Set (UCS)*

ISO/IEC/IEEE 60559, *Information technology – Microprocessor Systems – Floating-Point arithmetic*

3 Terms and definitions

For the purposes of this document, the terms and definitions given in IEC 61131-1 and the following apply.

3.1

access path

association of a symbolic name with a variable for the purpose of open communication

3.2

action

Boolean variable or a collection of operations to be performed, together with an associated control structure

- 303 **3.3**
 304 **action block**
 305 graphical language element which utilizes a Boolean input variable to determine the value of a
 306 Boolean output variable or the enabling condition for an action, according to a predetermined
 307 control structure
- 308 **3.4**
 309 **aggregate**
 310 structured collection of data objects forming a data type
- 311 **3.5**
 312 **array**
 313 aggregate that consists of data objects, with identical attributes, each of which may be
 314 uniquely referenced by subscripting
- 315 **3.6**
 316 **assignment**
 317 mechanism to give a value to a variable or to an aggregate
- 318 **3.7**
 319 **base type**
 320 data type, function block type or class from which further types are inherited/derived
- 321 **3.8**
 322 **based number**
 323 number represented in a specified base other than ten
- 324 **3.9**
 325 **binary coded decimal**
 326 BCD
 327 encoding for decimal numbers in which each digit is represented by its own binary sequence
- 328 **3.10**
 329 **bistable function block**
 330 function block with two stable states controlled by one or more inputs
- 331 **3.11**
 332 **bit string**
 333 data element consisting of one or more bits
- 334 **3.12**
 335 **bit string literal**
 336 literal that directly represents a bit string value of data type `BOOL`, `BYTE`, `WORD`, `DWORD`, or
 337 `LWORD`
- 338 **3.13**
 339 **body**
 340 set of operations of a program organization unit
- 341 **3.14**
 342 **call**
 343 language construct causing the execution of a function, function block, or method
- 344 **3.15**
 345 **character string**
 346 aggregate that consists of an ordered sequence of characters

- 347 **3.16**
348 **character string literal**
349 literal that directly represents a character or character string value of data type CHAR, WCHAR,
350 UCHAR, STRING, WSTRING or USTRING
- 351 **3.17**
352 **class**
353 program organization unit consisting of:
354 • the definition of a data structure,
355 • a set of methods to be performed upon the data structure
- 356 **3.18**
357 **comment**
358 language construct for the inclusion of text having no impact on the execution of the program
- 359 **3.19**
360 **configuration**
361 language element corresponding to a programmable controller system
- 362 **3.20**
363 **constant**
364 language element which declares a data element with a fixed value
- 365 **3.21**
366 **counter function block**
367 function block which accumulates a value for the number of changes sensed at one or more
368 specified inputs
- 369 **3.22**
370 **data type**
371 language element which allows a set of values together with a set of permitted operations
- 372 **3.23**
373 **date and time**
374 date within the year and the time of day represented as a single language element
- 375 **3.24**
376 **declaration**
377 mechanism for establishing the definition of a language element
- 378 **3.25**
379 **delimiter**
380 character or combination of characters used to separate program language elements
- 381 **3.26**
382 **derived class**
383 class that extends another class by inheritance; derived class is also named extended class
384 or child class
- 385 **3.27**
386 **directly derived data type**
387 data type that is declared as an alias for another data type
- 388 **3.28**
389 **derived function block type**
390 function block type created by inheritance from another function block type or class type

- 391 **3.29**
 392 **direct representation**
 393 means of representing a variable in a programmable controller program from which an imple-
 394 mentation-specified correspondence to a physical or logical location may be determined di-
 395 rectly
- 396 **3.30**
 397 **double word**
 398 data element containing 32 bits
- 399 **3.31**
 400 **dynamic binding**
 401 situation in which the implementation of a method is retrieved during runtime according to the
 402 actual type of the instance
- 403 **3.32**
 404 **evaluation**
 405 process of establishing a value for an expression or a function, or for the outputs of a network
 406 or function block instance, during program execution
- 407 **3.33**
 408 **execution control element**
 409 language element which controls the flow of program execution
- 410 **3.34**
 411 **falling edge**
 412 change from TRUE (1) to FALSE (0) of a Boolean variable
- 413 **3.35**
 414 **function**
 415 program organization unit that does not retain state and which, when executed, typically
 416 yields one result and possibly additional output variables
- 417 **3.36**
 418 **function block instance**
 419 instance of a function block type
- 420 **3.37**
 421 **function block type**
 422 language element consisting of:
- 423 – the definition of a data structure partitioned into input, output, and internal variables; and
 - 424 – a set of operations or a set of methods to be performed upon the elements of the data
 - 425 structure when an instance of the function block type is called
- 426 **3.38**
 427 **function block diagram**
 428 network in which the nodes are function block instances, graphically represented functions or
 429 method calls, variables, literals, and labels
- 430 **3.39**
 431 **generic data type**
 432 data type which represents more than one type of data
- 433 **3.40**
 434 **global variable**
 435 variable whose scope is global

- 436 **3.41**
437 **hierarchical addressing**
438 direct representation of a data element as a member of a physical or logical hierarchy
- 439 EXAMPLE A point within a module which is contained in a rack, which in turn is contained in a cubicle, etc.
- 440 **3.42**
441 **identifier**
442 combination of letters, numbers, and underscore characters which begins with a letter or un-
443 derscore and which names a language element
- 444 **3.43**
445 **implementation**
446 product version of a PLC or the programming and debugging tool provided by the implementer
- 447 **3.44**
448 **implementer**
449 manufacturer of the PLC or the programming and debugging tool provided to the user to pro-
450 gram a PLC application
- 451 **3.45**
452 **inheritance**
453 creation of a new class, function block type or interface based on an existing class, function
454 block type or interface, respectively
- 455 **3.46**
456 **initial value**
457 value assigned to a variable at system start-up
- 458 **3.47**
459 **in-out variable**
460 variable which is used to supply a value to a program organization unit and which is addition-
461 ally used to return a value from the program organization unit
- 462 **3.48**
463 **input variable**
464 variable which is used to supply a value to a program organization unit or method
- 465 **3.49**
466 **instance**
467 individual, named copy of the data structure associated with a function block type, class, or
468 program type
- 469 **3.50**
470 **instance name**
471 identifier associated with a specific instance
- 472 **3.51**
473 **instantiation**
474 creation of an instance
- 475 **3.52**
476 **integer literal**
477 literal which directly represents an integer value

478 **3.53**
479 **interface**
480 language element in the context of object-oriented programming containing a set of method
481 prototypes

482 **3.54**
483 **keyword**
484 lexical unit reserved for use in a programming language

485 **3.55**
486 **label**
487 language construction naming an instruction, network, or group of networks, and including an
488 identifier

489 **3.56**
490 **language element**
491 any item identified by a symbol on the left-hand side of a production rule in the formal specifi-
492 cation

493 **3.57**
494 **literal**
495 lexical unit that directly represents a value

496 **3.58**
497 **logical location**
498 location of a hierarchically addressed variable in a schema which may or may not bear any
499 relation to the physical structure of the programmable controller's inputs, outputs, and
500 memory

501 **3.59**
502 **method**
503 is equal to a function except that it is called on an instance and its operations can access this
504 instance. Methods can only be defined in the scope of a function block type or class type.

505 **3.60**
506 **method prototype**
507 language element consisting only of the name and the signature of a method

508 **3.61**
509 **named element**
510 element of a structure which is named by its associated identifier

511 **3.62**
512 **network**
513 arrangement of nodes and interconnecting branches

514 **3.63**
515 **numeric literal**
516 literal which directly represents a numeric value i.e. an integer literal or real literal

517 **3.64**
518 **operation**
519 language element that represents an elementary functionality belonging to a program organi-
520 zation unit or method

521 **3.65**
522 **operand**
523 language element on which an operation is performed

- 524 **3.66**
525 **operator**
526 symbol that represents the action to be performed in an operation
- 527 **3.67**
528 **override**
529 keyword used with a method in a derived class or function block type for a method with the
530 same name and signature as the method of the base class or function block type using a new
531 method body
- 532 **3.68**
533 **output variable**
534 variable which is used to return a value from the program organization unit or method
- 535 **3.69**
536 **parameter**
537 variable which is used to provide a value to a program organization unit or method (as input
538 or in-out parameter) or a variable which is used to return a value from a program organization
539 unit or method (as output or in-out parameter)
- 540 **3.70**
541 **reference**
542 user-defined data containing the location address to a variable or to an instance of a function
543 block of a specified type
- 544 **3.71**
545 **power flow**
546 symbolic flow of electrical power in a ladder diagram, used to denote the progression of a log-
547 ic solving algorithm
- 548 **3.72**
549 **pragma**
550 language construct for the inclusion of text in a program organization unit which may affect
551 the preparation of the program for execution
- 552 **3.73**
553 **program**
554 to design, write, and test user programs
- 555 **3.74**
556 **program organization unit**
557 function, function block, class, or program
- 558 **3.75**
559 **real literal**
560 literal directly representing a value of type `REAL` or `LREAL`
- 561 **3.76**
562 **resource**
563 language element corresponding to a “signal processing function” and its “man-machine inter-
564 face” and “sensor and actuator interface functions”, if any
- 565 **3.77**
566 **result**
567 value which is returned as an outcome of a function or method

- 568 **3.78**
 569 **return**
 570 language construction within a program organization unit designating an end to the execution
 571 sequences in the unit
- 572 **3.79**
 573 **rising edge**
 574 change from `FALSE` (0) to `TRUE` (1) of a Boolean variable
- 575 **3.80**
 576 **scope**
 577 set of program organization units within which a declaration or label applies
- 578 **3.81**
 579 **semantics**
 580 relationships between the symbolic elements of a programming language and their meanings,
 581 interpretation and use
- 582 **3.82**
 583 **semigraphic representation**
 584 representation of graphical information by the use of a limited set of characters
- 585 **3.83**
 586 **signature**
 587 set of information defining a `FUNCTION`- or `METHOD`-interface, consisting of an ordered list of
 588 parameters and the return type (if any), where each parameter is defined by its name, its type
 589 and its variable-kind (input, output or in-out)
- 590 **3.84**
 591 **single-element variable**
 592 variable which represents a single data element
- 593 **3.85**
 594 **static variable**
 595 variable whose value is stored from one call to the next one
- 596 **3.86**
 597 **step**
 598 situation in which the behavior of a program organization unit with respect to its inputs and
 599 outputs follows a set of rules defined by the associated actions of the step
- 600 **3.87**
 601 **structured data type**
 602 aggregate data type which has been declared using a `STRUCT` or `FUNCTION_BLOCK` declara-
 603 tion
- 604 **3.88**
 605 **subscripting**
 606 mechanism for referencing an array element by means of an array reference and one or more
 607 expressions that, when evaluated, denote the position of the element
- 608 **3.89**
 609 **task**
 610 execution control element providing for periodic or triggered execution of a group of associat-
 611 ed program organization units

612 **3.90**
613 **time literal**
614 literal representing data of type `TIME`, `LTIME`, `DATE`, `LDATE`, `TIME_OF_DAY`, `LTIME_OF_DAY`,
615 `DATE_AND_TIME` or `LDATE_AND_TIME`

616 **3.91**
617 **transition**
618 condition whereby control passes from one or more predecessor steps to one or more suc-
619 cessor steps along a directed link

620 **3.92**
621 **unsigned integer**
622 integer which can only have a value greater than or equal to zero number which may contain
623 positive and zero values

624 **3.93**
625 **unsigned integer literal**
626 integer literal not containing a leading plus (+) or minus (-) sign representing an unsigned in-
627 teger

628 **3.94**
629 **user-defined data type**
630 data type defined by the user

631 EXAMPLE Enumeration, array or structure.

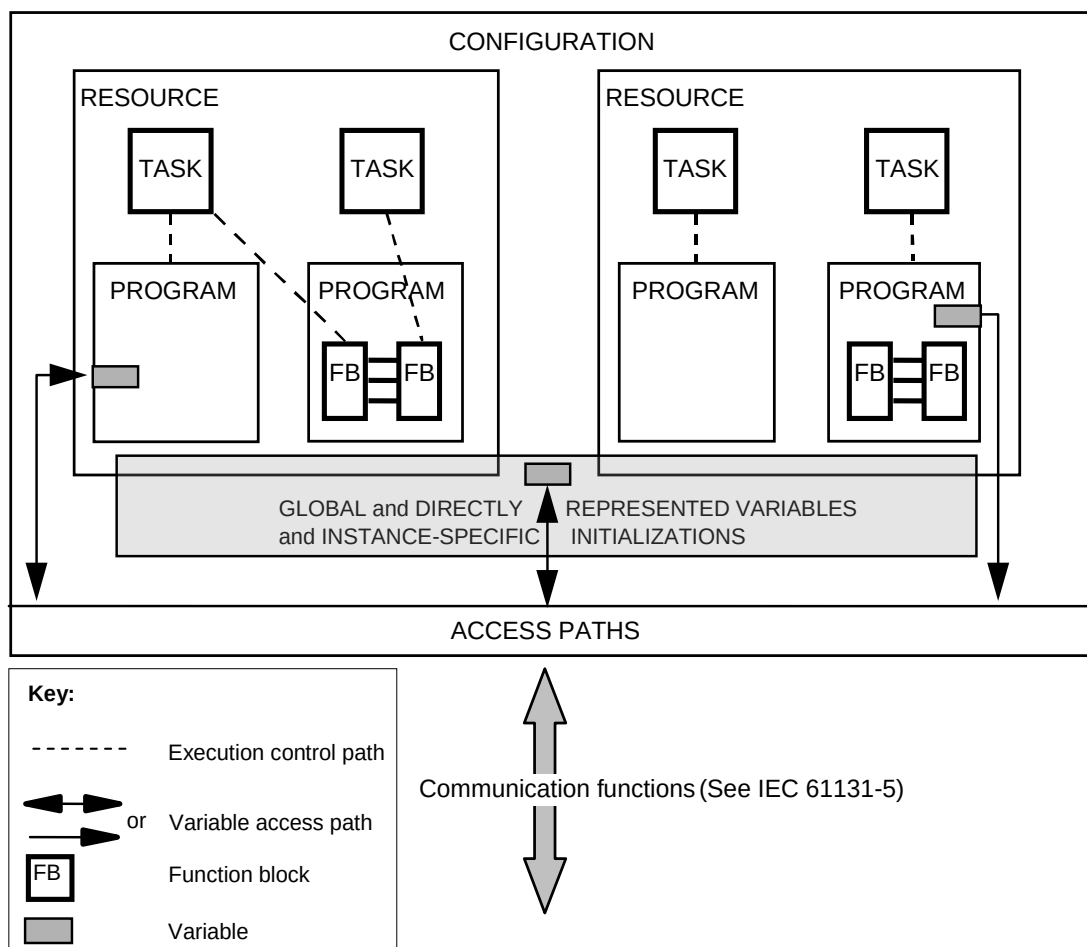
632 **3.95**
633 **variable**
634 software entity that may take different values, one at a time

635 **4 Architectural models**

636 **4.1 Software model**

637 The basic high-level language elements and their interrelationships are illustrated in Figure 1.

638 These consist of elements which are programmed using the languages defined in this stand-
639 ard, that is, programs and function block types, classes, functions, and configuration ele-
640 ments, namely, configurations, resources, tasks, global variables, access paths, and instance-
641 specific initializations, which support the installation of programmable controller programs into
642 programmable controller systems.



NOTE 1 Figure 1 is illustrative only. The graphical representation is not normative.

NOTE 2 In a configuration with a single resource, the resource need not be explicitly represented.

NOTE 3 The use of access paths to make data available for transfer is deprecated.

NOTE 4 The association of tasks directly to function block instances is deprecated.

643

Figure 1 – Software model

644 A configuration is the language element which corresponds to a programmable controller sys-
 645 tem as defined in IEC 61131-1. A resource corresponds to a “signal processing function” and
 646 its “man-machine interface” and “sensor and actuator interface” functions (if any) as defined
 647 in IEC 61131-1.

648 A configuration contains one or more resources, each of which contains one or more pro-
 649 grams executed under the control of zero or more tasks.

650 A program may contain zero or more function block instances or other language elements as
 651 defined in this part of IEC 61131.

652 A task is capable of causing, e.g. on a periodic basis, the execution of a set of programs and
 653 function block instances.

654 Configurations and resources can be started and stopped via the “operator interface”, “pro-
 655 gramming, testing, and monitoring”, or “operating system” functions defined in IEC 61131-1.
 656 The starting of a configuration shall cause the initialization of its global variables, followed by
 657 the starting of all the resources in the configuration. The starting of a resource shall cause the
 658 initialization of all the variables in the resource, followed by the enabling of all the tasks in the

659 resource. The stopping of a resource shall cause the disabling of all its tasks, while the stop-
660 ping of a configuration shall cause the stopping of all its resources.

661 Mechanisms for the control of tasks are defined in 6.8.2, while mechanisms for the starting
662 and stopping of configurations and resources via communication functions are defined in
663 IEC 61131-5.

664 Programs, resources, global variables, access paths (and their corresponding access privi-
665 leges), and configurations can be loaded or deleted by the “communication function” defined
666 in IEC 61131-1. The loading or deletion of a configuration or resource shall be equivalent to
667 the loading or deletion of all the elements it contains.

668 The mapping of the language elements onto communication objects shall be as defined in
669 IEC 61131-5.

670 **4.2 Communication model**

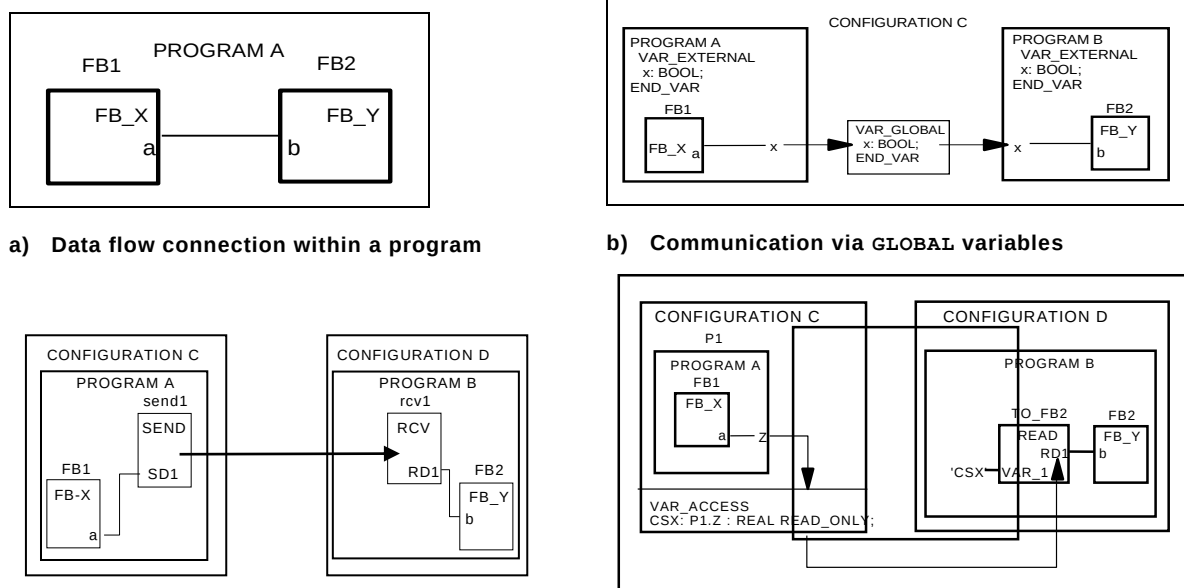
671 Figure 2 illustrates the ways that values of variables can be communicated among software
672 elements.

673 As shown in Figure 2a), variable values within a program can be communicated directly by
674 connection of the output of one program element to the input of another. This connection is
675 shown explicitly in graphical languages and implicitly in textual languages.

676 Variable values can be communicated between programs in the same configuration via global
677 variables such as the variable `x` illustrated in Figure 2b). These variables shall be declared as
678 `GLOBAL` in the configuration, and as `EXTERNAL` in the programs.

679 As illustrated in Figure 2c), the values of variables can be communicated between different
680 parts of a program, between programs in the same or different configurations, or between a
681 programmable controller program and a non-programmable controller system, using the com-
682 munication function blocks defined in IEC 61131-5.

683 In addition, programmable controllers or non-programmable controller systems can transfer
684 data which is made available by access paths, as illustrated in Figure 2d), using the mecha-
685 nisms defined in IEC 61131-5. This functionality is deprecated.



a) Data flow connection within a program

b) Communication via GLOBAL variables

c) Communication function blocks

d) Communication via access paths

NOTE 1 Figure 2 is illustrative only. The graphical representation is not normative.

NOTE 2 In these examples, configurations C and D are each considered to have a single resource.

NOTE 3 The details of the communication function blocks are not shown in Figure 2.

NOTE 4 Access paths can be declared on directly represented variables, global variables, or input, output, or internal variables of programs or function block instances.

NOTE 5 IEC 61131-5 specifies the means by which both PC and non-PC systems can use access paths for reading and writing of variables.

NOTE 6 The feature 9 VAR_ACCESS in Table 47 is deprecated and may not be included in the next edition of this part of IEC 61131

Figure 2 – Communication model

4.3 Programming model

Figure 3 illustrates the relationships between the different PLC language elements. The combination of these elements shall obey the following rules:

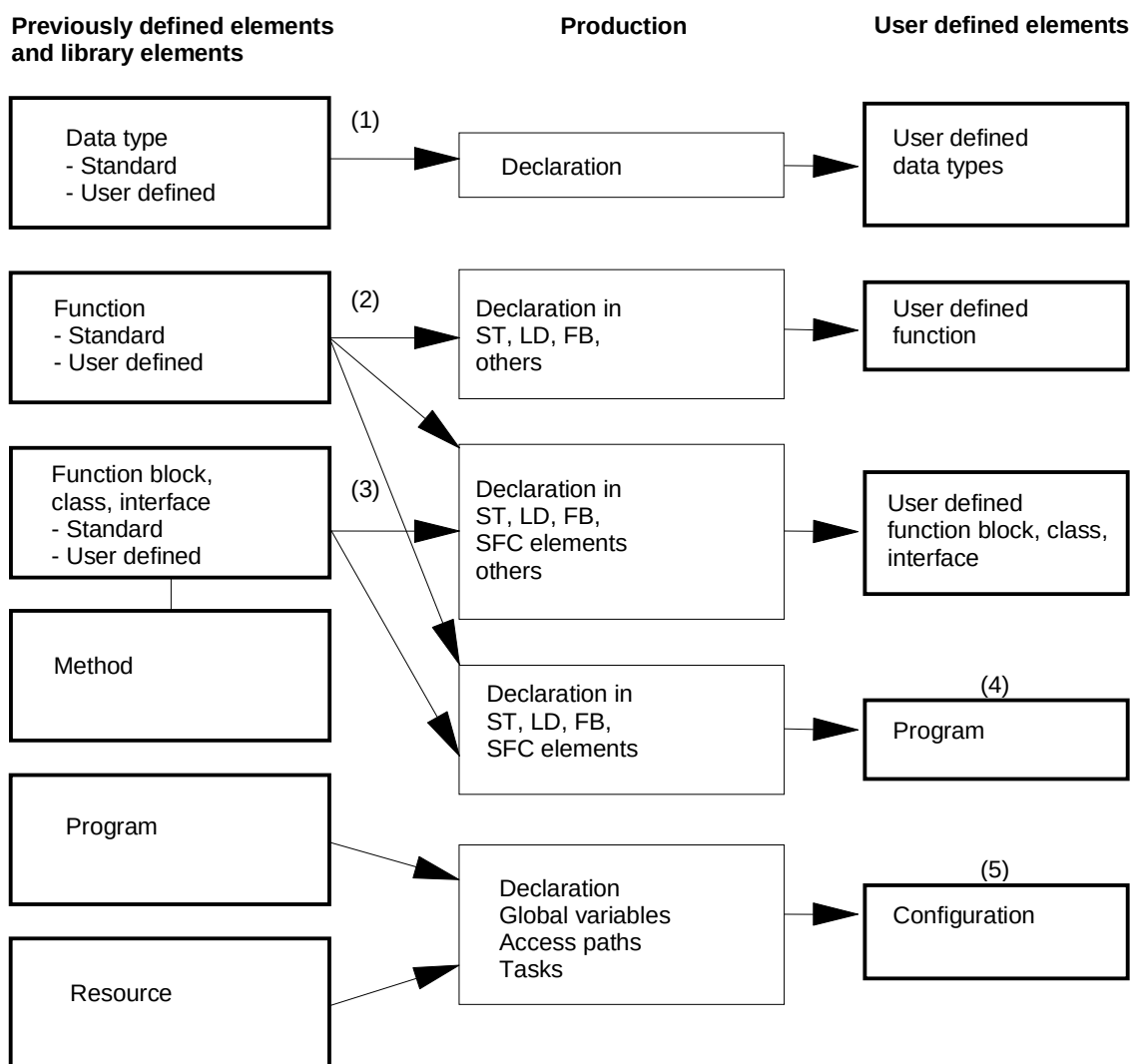
1. Data types shall be declared, using the standard data types and any previously defined data types.
2. Functions can be declared using standard or user-defined data types, the standard functions and any previously defined functions.
This declaration shall use the mechanisms defined for the ST, LD or FBD language.
3. Function block types can be declared using standard and user-defined data types, functions, standard function block types and any previously defined function block types.
These declarations shall use the mechanisms defined for the ST, LD, or FBD language, and can include Sequential Function Chart (SFC) elements.
Optionally, one may define object-oriented function block types or classes which can contain methods. Object-oriented function block types and classes may also be derived and can implement interfaces (not shown).
4. A program shall be declared using standard or user-defined data types, functions, function blocks and classes.

704 This declaration shall use the mechanisms defined for the ST, LD, or FBD language, and
 705 can include Sequential Function Chart (SFC) elements.

706 5. Programs can be combined into configurations using the configuration elements, that is,
 707 global variables, resources, tasks, and access paths.

708 Reference to “previously defined” data types, functions, and function blocks in the above rules
 709 is intended to imply that once such a previously defined element has been declared, its defini-
 710 tion is available, for example, in a “library” of previously defined elements, for use in further
 711 definitions.

712 A programming language other than one of those defined in this standard may be used for
 713 programming of a function, function block type, program or method.



LD: Ladder Diagram

FBD: Function Block Diagram

ST: Structured Text

Others: Other programming languages

NOTE 1 The parenthesized numbers (1) to (5) refer to the corresponding paragraphs 1) through 5) above.

NOTE 2 Data types are used in all productions. For clarity, the corresponding linkages are omitted in this figure.

714

Figure 3 – Combination of programmable controller language elements

5 Compliance

5.1 General

A PLC programming and debugging tool (PADT), as defined in IEC 61131-1, which claims to comply, wholly or partially, with the requirements of this part of IEC 61131 shall do only as described below.

- a) shall provide a subset of the features and provide the corresponding “Implementer’s compliance statement” as defined below.
- b) shall not require the inclusion of substitute or additional language elements to accomplish any of the features.
- c) shall provide a document that specifies all implementer specific extensions. These are any features accepted by the system that are prohibited or not specified.
- d) shall provide a document that specifies all implementer specific dependencies. This includes the implementation dependencies explicitly designated in this part of IEC 61131 and the limiting parameters like maximum length, number, size, and range of value which are not explicitly specified here.
- e) shall provide a document that specifies all errors that are detectable and reported by the implementation. This includes the errors explicitly designated in this part and the errors detectable during preparation of the program for execution and during execution of the program.

NOTE Errors occurring during execution of the program are only partially specified in this part of IEC 61131.

- f) shall not use any of the standard names of data types, function or function block names defined in this standard for implementation-defined features whose functionality differs from that described in this part of IEC 61131.

5.2 Feature tables

The tables in this part of IEC 61131 list the features defined in this part of IEC 61131 and share the same basic structure. The first column contains the “feature number”, the second column gives the “feature description”, the following columns may contain examples or further information. The same table structure is preferably used in the implementer’s compliance statement.

5.3 Implementer’s compliance statement

The implementer may define any consistent subset of the features listed in the feature tables and shall declare the provided subset in the “Implementer’s compliance statement”.

The “Implementer’s compliance statement” shall be included in the documentation accompanying the system or shall be produced by the system itself.

The “Implementer’s compliance statement” shall provide the following information. Figure 4 shows an example.

- The general information including the implementer name and address, the product name and version, the controller type and version and the date of issue.
- For each implemented feature the number of the corresponding feature table, the feature number, and the applicable programming languages.
Optionally, the title and subtitle of the feature table, the feature description, examples, implementer’s note and other relevant information may be provided.

Not implemented tables and features may be omitted from the “Implementer’s compliance statement”.

IEC 61131-3 “PLC Programming Languages”						
Implementer: Company name, address, etc.						
Product: Product name, version, etc. Controller type specific subset, etc.						
Date: 2022-05-01						
This Product complies with the requirements of the standard for the following language features:						
Feature No.	Table number and title / Feature description	Compliantly implemented in the language (✓)				Implementer's note
		LD	FB D	ST		
	Table 1 – Character set					
1	ISO/IEC 10646:2012, Information technology – Universal Coded Character Set (UCS)	✓	✓	✓		
2a	Lower case characters a: a, b, c, ...	✓	✓	✓		No “ß, ü, ä, ö”
2b	Number sign: # See Table 5	✓				
2c	Dollar sign: \$ See Table 6		✓			
	Table 2 – Identifiers					
1	Upper case letters and numbers: I W 2 1 5					
2	Upper and lower case letters, numbers, embedded under-score					
3	Upper and lower case, numbers, leading or embedded under-score					
	Table 3 – Comments					
1	Single-line comment //...					
2a	Multi-line comment (* ... *)					
2b	Multi-line comment /* ... */					
3a	Nested comment (* .. (* .. *) ..*)					
3b	Nested comment /* .. /* .. */ .. */					
	Table 4 – Pragma					
1	Pragma with curly brackets { ... }					
	Table 5 – Numeric literals					
1	Integer literal: -12					
2	Real literal: -12.0					
3	Real literals with exponent: -1.34E-12					
4	Binary literal: 2#1111_1111					
5	Hexadecimal literal: 16#FF					
6	Boolean zero and one					
7	Boolean FALSE and TRUE					
8	Typed literal: INT#-123					
	Etc.					

759

Figure 4 – Implementer's compliance statement (Example)

6 Common elements

6.1 Use of printed characters

6.1.1 Character set

Table 1 shows the character set of the textual languages and textual elements of graphical languages. The characters are represented in terms of the ISO/IEC 10646.

Table 1 – Character set

No.	Description
1	ISO/IEC 10646
2a	Lower case characters ^a : a, b, c
2b	Number sign: # See Table 5
2c	Dollar sign: \$ See Table 6
^a When lower-case letters are supported, the case of letters shall not be significant in language elements except within comments as defined in 6.1.5, string literals as defined in 6.3.3, and variables of type <code>STRING</code> , <code>WSTRING</code> and <code>USTRING</code> as defined in 6.3.3.	

6.1.2 Identifiers

An identifier is a string of letters, digits, and underscores which shall begin with a letter or underscore character.

The case of letters shall not be significant in identifiers, for example, the identifiers `abcd`, `ABCD`, and `aBCd` shall be interpreted identically.

The underscore character shall be significant in identifiers, for example, `A_BCD` and `AB_CD` shall be interpreted as different identifiers. Multiple leading or multiple embedded underlines are not allowed; for example, the character sequences `__LIM_SW5` and `LIM__SW5` are not valid identifiers. Trailing underscores are not allowed; for example, the character sequence `LIM_SW5_` is not a valid identifier.

At least six characters of uniqueness shall be supported in all systems which support the use of identifiers, for example, `ABCDE1` shall be interpreted as different from `ABCDE2` in all such systems. The maximum number of characters allowed in an identifier is an implementer specific dependency.

Identifier features and examples are shown in Table 2.

Table 2 – Identifiers

No.	Description	Examples
1	Upper case letters and numbers: IW215	IW215 IW215Z QX75 IDENT
2	Upper and lower case letters, numbers, embedded underscore	All the above plus: <code>LIM_SW_5</code> <code>LimSw5</code> <code>abcd</code> <code>ab_Cd</code>
3	Upper and lower case, numbers, leading or embedded underscore	All the above plus: <code>_MAIN</code> <code>_12V7</code>

6.1.3 Keywords

Keywords are unique combinations of characters utilized as individual syntactic elements. Keywords shall not contain embedded spaces. The case of characters shall not be significant

in keywords; for instance, the keywords `FOR` and `for` are syntactically equivalent. They shall not be used for any other purpose, for example, variable names or extensions.

6.1.4 Use of white space

The user shall be allowed to insert one or more characters of “white space” anywhere in the text of programmable controller programs except within keywords, literals, enumerated values, identifiers, directly represented variables or delimiter combinations for example, for comments. “White space” is defined as the SPACE character with encoded value 32 decimal, as well as non-printing characters such as tab, newline, etc. for which no encoding is given in IEC/ISO 10646.

6.1.5 Comments

There are different kinds of user comments listed in Table 3:

1. Single line comments start with the character combination `//` and end at the next following line feed, new line, form feed (page), or carriage return.

In single-line comments the character combinations `( * and * )` or `/* and */` have no special meaning.

2. Multi-line comments shall be delimited at the beginning and end by the special character combinations `( * and * )`, respectively.

An alternative multi-line comment may be provided using the character combinations `/*` and `*/`.

In multi-line comments the special character combination `//` has no special meaning.

Comments shall be permitted anywhere in the program where spaces are allowed, except within character string literals.

Comments shall have no syntactic or semantic significance in any of the languages defined in this standard. They are treated like a white space.

Nested comments use corresponding

- pairs of `( * , * )`, e.g. `( * ... ( * NESTED * ) ... * )` or
- pairs of `/* , */`, e.g. `/* ... /* NESTED */ ... */`.

Table 3 – Comments

No.	Description	Examples
1	Single-line comment with <code>// ...</code>	<code>X := 13; // comment for one line</code> <code>// a single line comments can start at</code> <code>// the first character position.</code>
2a	Multi-line comment with <code>(* ... *)</code>	<code>(* comment *)</code> <code>(*****</code> <code> A framed comment on three line</code> <code>*****)</code>
2b	Multi-line comment with <code>/* ... */</code>	<code>/* comment in one</code> <code>or more lines */</code>
3a	Nested comment with <code>(* .. (* .. *) .. *)</code>	<code>(* (* NESTED *) *)</code>
3b	Nested comment with <code>/* .. /* .. */ .. */</code>	<code>/* /* NESTED */ */</code>

6.2 Pragma

As illustrated in Table 4, pragmas shall be delimited at the beginning and end by curly brackets { and }, respectively. The syntax and semantics of particular pragma constructions are implementer specific. Pragmas shall be permitted anywhere in the program where spaces are allowed, except within character string literals.

Table 4 – Pragma

No.	Description	Examples
1	Pragma with { ... } curly brackets	<pre>{VERSION 2.0} {AUTHOR JHC} {x := 256, y := 384}</pre>

6.3 Literals – External representation of data

6.3.1 General

External representations of data in the various programmable controller programming languages shall consist of numeric literals, character string literals, and time literals.

The need to provide external representations for two distinct types of time-related data is recognized:

- duration data for measuring or controlling the elapsed time of a control event,
- and time of day data which may also include date information for synchronizing the beginning or end of a control event to an absolute time reference.

6.3.2 Numeric literals and string literals

There are two kinds of numeric literals: integer literals and real literals. A numeric literal is defined as a decimal number or a based number. The maximum number of digits for each kind of numeric literal shall be sufficient to express the entire range and precision of values of all the data types which are represented by the literal in a given implementation.

Single underscore characters “_” inserted between the digits of a numeric literal shall not be significant. No other use of underscore characters in numeric literals is allowed.

Decimal literals shall be represented in conventional decimal notation. Real literals shall be distinguished by the presence of a decimal point. An exponent indicates the integer power of ten by which the preceding number is to be multiplied to obtain the value represented. Decimal literals and their exponents can contain a preceding sign “+” or “-”.

Literals can also be represented in base 2, or 16. The base shall be in decimal notation. For base 16, an extended set of digits consisting of the letters A through F shall be used, with the conventional significance of decimal 10 through 15, respectively. Based numbers shall not contain a leading sign “+” or “-”. They are interpreted as bit string literals.

Numeric literals which represent a positive integer may be used as bit string literals.

Boolean data shall be represented by integer literals with the value zero (0) or one (1), or the keywords FALSE or TRUE, respectively.

Numeric literal features and examples are shown in Table 5.

851 The data type of a Boolean or numeric literal can be specified by adding a type prefix to the
 852 literal, consisting of the name of an elementary data type and the “#” sign. For examples, see
 853 feature 9 in Table 5.

854

Table 5 – Numeric literals

No.	Description	Examples	Explanation
1	Integer literal	-12, 0, 123_4, +986	
2	Real literal	0.0, 0.4560, 3.14159_26	
3	Real literals with exponent	-1.34E-12, -1.34e-12 1.0E+6, 1.0e+6 1.234E6, 1.234e6	
4	Binary literal	2#1111_1111 2#1110_0000	Base 2 literal 255 decimal 224 decimal
5	Feature 5 (octal literals) is deleted !		
6	Hexadecimal literal	16#FF or 16#ff 16#E0 or 16#e0	Base 16 literal 255 decimal 224 decimal
7	Boolean zero and one	0 or 1	
8	Boolean FALSE and TRUE	FALSE TRUE	
9	Typed literal	INT#-123	INT representation of the decimal value -123
		INT#16#7FFF	INT representation of the decimal value 32767
		WORD#16#AFF	WORD representation of the hexadecimal value 0AFF
		WORD#1234	WORD representation of the decimal value 1234=16#4D2
		UINT#16#89AF	UINT representation of the hexadecimal value 89AF
		BOOL#0	
		BOOL#1	
		BOOL#FALSE	
		BOOL#TRUE	

855

856 6.3.3 Character string literals

857 Character string literals include single-byte or double-byte or Unicode characters.

- 858 • A single-byte character string literal is a sequence of zero or more characters from the
 859 Basic Latin and Latin-1 Supplement (code point 0 to 255) of the ISO/IEC 10646 character
 860 set. It is prefixed and terminated by the single quote character ('). In single-byte charac-
 861 ter strings, the three-character combination of the dollar sign (\$) followed by two hexadec-
 862 imal digits shall be interpreted as the hexadecimal representation of the eight-bit charac-
 863 ter code, as shown in feature 1g of Table 6.

- A double-byte character string literal is a sequence of zero or more characters from the Basic Multilingual Plane (code point 0 to 65535) of the ISO/IEC 10646 character set. It is prefixed and terminated by the double quote character ("). In double-byte character strings, the five-character combination of the dollar sign "\$" followed by four hexadecimal digits shall be interpreted as the hexadecimal representation of the sixteen-bit character code, as shown in feature 2h of Table 6.
- A Unicode character string literal is a sequence of zero or more characters from the entire ISO/IEC 10646 character set. It is prefixed by the characters U# and the single quote character (') and terminated by the single quote character ('). Additionally, the prefix characters USTRING# may be used in place of U# by the implementer. In these literals, the seven-character combination of the dollar sign "\$" followed by six hexadecimal digits shall be interpreted as the hexadecimal representation of the 24-bit character code, as shown in feature 5e of Table 6.

NOTE Relation of ISO/IEC 10646 and Unicode:

Although the character codes and encoding forms are synchronized between Unicode and ISO/IEC 10646, the Unicode Standard imposes additional constraints on implementations to ensure that they treat characters uniformly across platforms and applications. To this end, it supplies an extensive set of functional character specifications, character data, algorithms and substantial background material that is not in ISO/IEC 10646.

Two-character combinations beginning with the dollar sign shall be interpreted as shown in Table 7 when they occur in character strings.

Table 6 – Character string literals

No.	Description	Examples
Single-byte characters or character strings with ' '		
1a	Empty string (length zero)	' '
1b	String of length one or character CHAR containing a single character	'A'
1c	String of length one or character CHAR containing the "space" character	' '
1d	String of length one or character CHAR containing the "single quote" character	'\$''
1e	String of length one or character CHAR containing the "double quote" character	'\"'
1f	Support of two character combinations of Table 7	'\$R\$L'
1g	Support of a character representation with '\$' and two hexadecimal digits	'\$0A'
Double-byte characters or character strings with " " (NOTE)		
2a	Empty string (length zero)	" "
2b	String of length one or character WCHAR containing a single character	"A"
2c	String of length one or character WCHAR containing the "space" character	" "
2d	String of length one or character WCHAR containing the "single quote" character	"' "
2e	String of length one or character WCHAR containing the "double quote" character	"\$\""
2f	Support of two character combinations of Table 7	"\$R\$L"
2h	Support of a character representation with '\$' and four hexadecimal digits	"\$00C4"
Single-byte typed characters or string literals with #		
3a	Typed string	STRING#'OK'
3b	Typed character	CHAR#'X'
Double-byte typed string literals with # (NOTE)		
4a	Typed double-byte string (using "double quote" character)	WSTRING#"OK"
4b	Typed double-byte character (using "double quote" character)	WCHAR#"X"
4c	Typed double-byte string (using "single quote" character)	WSTRING#'OK'

No.	Description	Examples
4d	Typed double-byte character (using “single quote” character)	WCHAR# 'X'
String literals supporting entire Unicode character set		
5a	Unicode character (using U# prefix and single quotes)	U# 'X'
5b	Unicode string (using U# prefix and single quotes)	U# 'OK'
5c	Unicode character (using UCHAR# prefix and single quotes)	UCHAR# 'X'
5d	Unicode string (using USTRING# prefix and single quotes)	USTRING# 'OK'
5e	Support of a character representation with '\$' and six hexadecimal digits in a literal according to 5a to 5d	U# '\$002192'
5f	Support of a character representation with '\$' and 1 to 6 hexadecimal digits between '{ }' in a literal according to 5a to 5d	U# '\$ {2192}'
NOTE If a particular implementation supports feature 4 but not feature 2, the implementer may specify implementer specific syntax and semantics for the use of the double-quote character.		

886

887

Table 7 – Two-character combinations in character strings

No.	Description	Combinations
1	Dollar sign	\$\$
2	Single quote	\$'
3	Line feed	\$L or \$l
4	Newline	\$N or \$n
5	Form feed (page)	\$P or \$p
6	Carriage return	\$R or \$r
7	Tabulator	\$T or \$t
8	Double quote	\$"
NOTE 1 The “newline” character provides an implementation-independent means of defining the end of a line of data; for printing, the effect is that of ending a line of data and resuming printing at the beginning of the next line.		
NOTE 2 The '\$' combination is only valid inside single quoted string literals.		
NOTE 3 The '\$" combination is only valid inside double quoted string literals.		

888

6.3.4 Duration literal

Duration data shall be delimited on the left by one of the keyword prefixes T#, LT#, TIME# or LTIME#. The representation of duration data in terms of days, hours, minutes, seconds, and fraction of a second, or any combination thereof, shall be supported as shown in Table 8. The least significant time unit can be written in real notation without an exponent.

The units of duration literals can be separated by underscore characters.

“Overflow” of the most significant unit of a duration literal is permitted, for example, the notation T#25h_15m is permitted.

Time units, for example, seconds, milliseconds, etc., can be represented in upper- or lower-case letters.

As illustrated in Table 8, both positive and negative values are allowed for durations.

900

Table 8 – Duration literals

No.	Description	Examples
	Duration abbreviations	
1a	d	Day
1b	h	Hour
1c	m	Minute
1d	s	Second
1e	ms	Millisecond
1f	us	Microsecond
1g	ns	Nanoseconds
	Duration literals without underscore	
2a	short prefix	T#14ms T#-14ms LT#14.7s T#14.7m T#14.7h t#14.7d t#25h15m lt#5d14h12m18s3.5ms t#12h4m34ms230us400ns
2b	long prefix	TIME#14ms TIME#-14ms time#14.7s
	Duration literals with underscore	
3a	short prefix	t#25h_15m t#5d_14h_12m_18s_3.5ms LTIME#5m_30s_500ms_100.1us
3b	long prefix	TIME#25h_15m ltime#5d_14h_12m_18s_3.5ms LTIME#34s_345ns

901

902 6.3.5 Date and time of day literal

903 Prefix keywords for time of day and date literals shall be as shown in **Table 9**.

904 **Table 9 – Date and time of day literals**

No.	Description	Examples
1a	Date literal (long prefix)	DATE#1984-06-25, date#2010-09-22
1b	Date literal (short prefix)	D#1984-06-25
2a	Long date literal (long prefix)	LDATE#2012-02-29
2b	Long date literal (short prefix)	LD#1984-06-25
3a	Time of day literal (long prefix)	TIME_OF_DAY#15:36:55.36
3b	Time of day literal (short prefix)	TOD#15:36:55.36
4a	Long time of day literal (short prefix)	LTOD#15:36:55.36
4b	Long time of day literal (long prefix)	LTIME_OF_DAY#15:36:55.36
5a	Date and time literal (long prefix)	DATE_AND_TIME#1984-06-25-15:36:55.360227400
5b	Date and time literal (short prefix)	DT#1984-06-25-15:36:55.360_227_400
6a	Long date and time literal (long prefix)	LDATE_AND_TIME#1984-06-25-15:36:55.360_227_400
6b	Long date and time literal (short prefix)	LDT#1984-06-25-15:36:55.360_227_400

905

906 6.4 Data types

907 6.4.1 General

908 A data type is a classification which defines for literals and variables the possible values, the
909 operations that can be done, and the way the values are stored.

910 **6.4.2 Elementary data types (BOOL, INT, REAL, STRING, etc.)**

911 **6.4.2.1 Specification of elementary data types**

912 A set of (pre-defined) elementary data types is specified by this standard.

913 The elementary data types, keyword for each data type, number of bits per data element, and
914 range of values for each elementary data type shall be as shown in Table 10.

915

Table 10 – Elementary data types

No.	Description	Keyword	Default initial value	N (bits) ^a
1	Boolean	BOOL	0, FALSE	1 ^h
2	Short integer	SINT	0	8 ^c
3	Integer	INT	0	16 ^c
4	Double integer	DINT	0	32 ^c
5	Long integer	LINT	0	64 ^c
6	Unsigned short integer	USINT	0	8 ^d
7	Unsigned integer	UINT	0	16 ^d
8	Unsigned double integer	UDINT	0	32 ^d
9	Unsigned long integer	ULINT	0	64 ^d
10	Real numbers	REAL	0.0	32 ^e
11	Long reals	LREAL	0.0	64 ^f
12a	Duration	TIME	T#0s	-- ^b
12b	Long Duration	LTIME	LTIME#0s	64 ^{m, q, s}
13a	Date (only)	DATE		-- ^{b, r}
13b	Long Date (only)	LDATE	LDATE#1970-01-01	64 ^{n, s}
14a	Time of day (only)	TIME_OF_DAY or TOD	TOD#00:00:00	-- ^b
14b	Long Time of day (only)	LTIME_OF_DAY or LTOD	LTOD#00:00:00	64 ^{p, q, s}
15a	Date and time of Day	DATE_AND_TIME or DT		-- ^{b, r}
15b	Long Date and time of Day	LDATE_AND_TIME or LDT	LDT#1970-01-01-00:00:00	64 ^{o, q, s}
16a	Variable-length single-byte character string	STRING	' ' (empty)	8 ^{i, g, k, l}
16b	Variable-length double-byte character string	WSTRING	" " (empty)	16 ^{i, g, k, l}
16c	Variable-length UTF-8 encoded character string	USTRING	U# ' ' (empty)	-- ^{g, k, l, t}
17a	Single-byte character	CHAR	'\$00'	8 ^{g, l}
17b	Double-byte character	WCHAR	"\$0000"	16 ^{g, l}
17c	Four-byte character	UCHAR	U# '\$000000'	32 ^{g, l}
18	Bit string of length 8	BYTE	16#00	8 ^{j, g}
19	Bit string of length 16	WORD	16#0000	16 ^{j, g}
20	Bit string of length 32	DWORD	16#0000_0000	32 ^{j, g}
21	Bit string of length 64	LWORD	16#0000_0000_0000_0000	64 ^{j, g}

No.	Description	Keyword	Default initial value	N (bits) ^a
a	Entries in this column shall be interpreted as specified in the table footnotes.			
b	The range of values and precision of representation in these data types is implementer specific.			
c	The range of values for variables of this data type is from $-(2^{N-1})$ to $(2^{N-1}) - 1$.			
d	The range of values for variables of this data type is from 0 to $(2^N) - 1$.			
e	The range of values for variables of this data type shall be as defined in IEC 60559 for the basic single width floating-point format. Results of arithmetic instructions with denormalized values, infinity, or not-a-number values are implementer specific.			
f	The range of values for variables of this data type shall be as defined in IEC 60559 for the basic double width floating-point format. Results of arithmetic instructions with denormalized values, infinity, or not-a-number values are implementer specific.			
g	A numeric range of values does not apply to this data type.			
h	The possible values of variables of this data type shall be 0 and 1, corresponding to the keywords <code>FALSE</code> and <code>TRUE</code> , respectively.			
i	The value of N indicates the number of bits/character for this data type.			
j	The value of N indicates the number of bits in the bit string for this data type.			
k	The maximum allowed length of <code>STRING</code> , <code>WSTRING</code> and <code>USTRING</code> variables is implementer specific.			
l	For <code>CHAR</code> , <code>STRING</code> , <code>WCHAR</code> , <code>WSTRING</code> , <code>UCHAR</code> , and <code>USTRING</code> , characters are assigned code-points (character encoding) according to ISO/IEC 10646 (see 6.4.2.2).			
m	The data type <code>LTIME</code> is a signed 64-bit integer with unit of nanoseconds.			
n	The data type <code>LDATE</code> is a signed 64-bit integer with unit of nanoseconds with starting time at midnight 1970-01-01.			
o	The data type <code>LDT</code> is a signed 64-bit integer with unit of nanoseconds with starting time at midnight 1970-01-01.			
p	The data type <code>LTOD</code> is a signed 64-bit integer with unit of nanoseconds with starting time midnight with <code>TOD#00:00:00</code> .			
q	The update accuracy of the values of this time format is implementer specific, i.e. the value is given in nanoseconds, but it may be updated every microsecond or millisecond.			
r	The default initial value of this data type is implementer specific.			
s	Long time formats do not count the leap seconds i.e. a day always consists of 86 400 s, a non-leap year consists of 365 days, a leap year consists of 366 days. Gregorian calendar is used.			
t	Characters are binary represented according to UTF-8 as defined in ISO/IEC 10646 (see NOTE).			
NOTE	A single character occupies 1 to 4 code-units, depending on the character's code-point. A code-point from the following ranges occupies 1,2,3, or 4 code-units respectively: 0..127, 128..2047, 2048..65535, 65536..1114111 (0..16#7F, 16#80..16#7FF, 16#800..16#FFFF, 16#10000..16#10FFFF). A code-unit has a size of one byte.			

916

917 6.4.2.2 Elementary data type strings (`STRING`, `WSTRING`, `USTRING`)

918 The supported maximum length of values of type `STRING`, `WSTRING` and `USTRING` shall be
919 implementer specific.

920 The explicit maximum length n is specified in square brackets “[n]” in the associated declara-
921 tion. The explicit maximum length n shall not exceed the implementer specific supported max-
922 imum value.

923 **NOTE** This is the number of ASCII characters that can be represented by the string. For type `USTRING`, the sup-
924 ported maximum length and the explicit maximum length denote the strings capacity in terms of code-units.

925

926 Access to single characters of a string using elements of the data type `CHAR`, `WCHAR` or
927 `UCHAR` shall be supported using square brackets and the position of the character in the
928 string, starting with position 1.

929 Let U be a variable of type `USTRING[n]` that contains the sequence of characters C1, C2, ...,
930 Ci, ..., Cm.

931 The assignment `U[i] := X` denotes the replacement of `Ci` by `X`. If the resulting character
932 sequence does not fit in `U`, a minimal number of characters at the end of the string is truncat-
933 ed.

934

EXAMPLE 1 `STRING`, `WSTRING` and `CHAR`, `WCHAR`

a) Declaration

```
VAR
    String1:   STRING[10] := 'ABCD';
    String2:   STRING[10] := '';
    aWStrings: ARRAY [0..1] OF WSTRING := ["1234", "5678"];
    UString1:  USTRING[10] := U#'A+Ä-$01F68D'; //5 UCHAR with 9 code-units:'A+Ä-🚗'
    Char1:     CHAR;
    WChar1:    WCHAR;
    UChar1:    UCHAR := U#'$1F600'; // Unicode character '😄'
END_VAR
```

b) Usage of `STRING` and `CHAR`

```
Char1 := String1[2];           //is equivalent to Char1 := 'B';
String1[3] := Char1;           //results in String1 := 'ABBD '
String1[4] := 'B';             //results in String1 := 'ABBB'
String1[1] := String1[4];       //results in String1 := 'BBBB'
String2 := String1[2]; (*results in String2 := 'B'
                        if implicit conversion CHAR_TO_STRING has been implemented*)
```

c) Usage of `WSTRING` and `WCHAR`

```
WChar1 := aWStrings[1][2];     //is equivalent to WChar1 := "6";
aWStrings[1][3] := WChar1;      //results in aWStrings[1] := "5668"
aWStrings[1][4] := "6"; //results in aWStrings[1] := "5666"
aWStrings[1][1] := aWStrings[1][4]; //results in String1 := "6666"
aWStrings[0] := aWStrings[1][4]; (* results in aWStrings[0] := "6";
if implicit conversion WCHAR_TO_WSTRING has been implemented *)
```

d) Usage of `USTRING` and `UCHAR`

```
UChar1 := UString1[4]; //results in UChar1 = U#'- '
UString1[2] := UChar1; //results in UString1 = U#'A-Ä-🚗'
UString1[5] := U#'$1F600'; //results in UString1 = U#'A-Ä-😄'
UString1[5] := USTRING#'$1F600'; //equivalent to previous line
UString1[1] := USTRING#'$01F68D'; (* results in UString1 = U#'🚗-Ä-'
To fit in USTRING[10], the string is truncated from 12 code-units to 8 code-units
by dropping the last character *)
UString1 := "ABCÄ"; // results in UString1 = U#'ABCÄ' (implicit conversion)
```

e) Equivalent functions (see 6.6.2.5.11)

```
Char1 := String1[2];
is equivalent to
Char1 := STRING_TO_CHAR(Mid(IN := String1, L := 1, P := 2));
aWStrings[1][3] := WChar1;
is equivalent to
REPLACE(IN1 := aWStrings[1], IN2 := WChar1, L := 1, P := 3 );
```

f) Error cases

```
String1[2] := WChar1; //invalid implicit conversion
String1[2] := String2;
//requires implicit conversion STRING_TO_CHAR which is not allowed
```

935 **NOTE** The data types for single characters (`CHAR` and `WCHAR`) can only contain one character. Strings can con-
936 tain several characters; therefore, strings may require additional management information which is not
937 needed for single characters.

EXAMPLE 2

```
TYPE STR10: STRING[10] := 'ABCDEF'; END_TYPE
```

Here the type STR10 is declared by the maximum length of STR10 is 10 characters, default initial value is 'ABCDEF' and the initial length of data elements of type STR10 is 6 characters.

```
VAR
```

```
    UString10: USTRING[10] := U#'A+Ä-Š01F68D'; //5 UCHAR with 9 code-units:'A+Ä-Š'
END_VAR
```

Here the variable UString10 is defined by the maximum length of 10 code-units, default initial value is 'A+Ä-Š' containing 5 Unicode characters and 9 code-units.

938 6.4.3 Generic data types

939 In addition to the elementary data types shown in Table 10, the hierarchy of generic data
 940 types shown in Figure 5 can be used in the specification of inputs and outputs of standard
 941 functions and function blocks. Generic data types are identified by the prefix “ANY”.

942 The use of generic data types is subject to the following rules:

- 943 1. The generic type of a directly derived type shall be the same as the generic type of the
 944 elementary type from which it is derived.
- 945 2. The generic type of a subrange type shall be ANY_INT.
- 946 3. The generic type of all other derived types defined in Table 11 shall be ANY_DERIVED.

947 The usage of generic data types in user-declared program organization units is beyond the
 948 scope of this standard.

Generic data types		Groups of elementary data types
ANY		
ANY_DERIVED		
ANY_ELEMENTARY		
ANY_MAGNITUDE		
ANY_DURATION		TIME, LTIME
ANY_NUM		
ANY_REAL		REAL, LREAL
ANY_INT		
	ANY_UNSIGNED	USINT, UINT, UDINT, ULINT
	ANY_SIGNED	SINT, INT, DINT, LINT
ANY_BIT		BOOL, BYTE, WORD, DWORD, LWORD
ANY_CHARS		
ANY_STRING		STRING, WSTRING, USTRING
ANY_CHAR		CHAR, WCHAR, UCHAR
ANY_DATE		DATE_AND_TIME (DT), LONG_DATE_AND_TIME (LDT), DATE, LDATE, TIME_OF_DAY (TOD), LONG_TIME_OF_DAY (LTOD)

949 **Figure 5 – Hierarchy of the generic data types**

6.4.4 User-defined data types

6.4.4.1 Declaration (TYPE)

6.4.4.1.1 General

The purpose of the user-defined data types is to be used in the declaration of other data types and in the variable declarations.

A user-defined type can be used anywhere, where a base type can be used.

User-defined data types are declared using the `TYPE . . . END_TYPE` textual construct.

A user-defined type declaration consists of

- the name of the type
- a ':' (colon)
- the declaration of the type itself as defined in the following clauses.

EXAMPLE Type declaration

```
TYPE
    myDatatype1: <data type declaration with optional initialization>;
END_TYPE
```

6.4.4.1.2 Initialization

The initial value is given by the default initial value, unless a user-defined value is specified.

The user-defined initialization follows the type declaration and starts with the assignment operator ":@" followed by the initial value(s).

Literals (e.g. -123, 1.55, "abc") or constant expressions (e.g. 12*24) may be used. The initial values used shall be of a compatible type i.e. the same type or a type which can be converted using implicit type conversion.

The rules according to Figure 6 shall apply for the initialization of data types.

Generic Data Type	Initialized by literal	Result
ANY_UNSIGNED	Non-negative integer literal or non-negative constant expression	Non-negative integer value
ANY_SIGNED	Integer literal or constant expression	Integer value
ANY_REAL	Numeric literal or constant expression	Numeric value
ANY_BIT	Unsigned integer literal or unsigned constant expression	Unsigned integer value
ANY_STRING	String literal	String value
ANY_DATE	Date and Time of Day literal	Date and Time of Day value
ANY_DURATION	Duration literal	Duration value

Figure 6 – Initialization by literals and constant expressions (Rules)

Table 11 defines the features of the declaration of user-defined data types and initialization.

971

Table 11 – Declaration of user-defined data types and initialization

No.	Description	Example	Explanation
1a 1b	Enumerated data types	<pre> TYPE ANALOG_SIGNAL_RANGE: (BIPOLAR_10V, UNIPOLAR_10V, UNIPOLAR_1_5V, UNIPOLAR_0_5V, UNIPOLAR_4_20_MA, UNIPOLAR_0_20_MA) := UNIPOLAR_1_5V; END_TYPE </pre>	Initialization
2a 2b	Data types with named values	<pre> TYPE Colors: DWORD (Red := 16#00FF0000, Green:= 16#0000FF00, Blue := 16#000000FF, White:= Red OR Green OR Blue, Black:= Red AND Green AND Blue) := Green; END_TYPE </pre>	Initialization
3a 3b	Subrange data types	<pre> TYPE ANALOG_DATA: INT(-4095 .. 4095) := 0; END_TYPE </pre>	
4a 4b	Array data types	<pre> TYPE ANALOG_16_INPUT_DATA: ARRAY [1..16] OF ANALOG_DATA := [8(-4095), 8(4095)]; END_TYPE </pre>	ANALOG_DATA see above. Initialization
5a 5b	FB types and classes as array elements	<pre> TYPE TONS: ARRAY[1..50] OF TON := [50((PT :=T#100ms))]; END_TYPE </pre>	FB TON as array element Initialization
6a 6b	Structured data type	<pre> TYPE ANALOG_CHANNEL_CONFIGURATION: STRUCT RANGE: ANALOG_SIGNAL_RANGE; MIN_SCALE: ANALOG_DATA := -4095; MAX_SCALE: ANALOG_DATA := 4095; END_STRUCT; END_TYPE </pre>	ANALOG_SIGNAL_RANGE see above
7a 7b	FB types and classes as structure elements	<pre> TYPE Cooler: STRUCT Temp: INT; Cooling: TOF := (PT := T#100ms); END_STRUCT; END_TYPE </pre>	FB TOF as structure element
8a 8b	Structured data type with relative addressing AT	<pre> TYPE Com1_data: STRUCT head AT %B0: INT; length AT %B2: USINT := 26; flag1 AT %X3.0: BOOL; end AT %B25: BYTE; END_STRUCT; END_TYPE </pre>	Explicit layout without overlapping
9a	Structured data type with relative addressing AT and OVERLAP	<pre> TYPE Com2_data: STRUCT OVERLAP head AT %B0: INT; length AT %B2: USINT; flag2 AT %X3.3: BOOL; data1 AT %B5: BYTE; data2 AT %B5: REAL; end AT %B19: BYTE; END_STRUCT; END_TYPE </pre>	Explicit layout with overlapping

No.	Description	Example	Explanation
10a 10b	Directly represented elements of a structure – partly specified using “*”	<pre> TYPE HW_COMP: STRUCT; IN AT %I*: BOOL; OUT_VAR AT %Q*: WORD := 200; ITNL_VAR: REAL := 123.0; // not located END_STRUCT; END_TYPE </pre>	Assigns the components of a structure to not yet located inputs and outputs
11a 11b	Directly derived data types	<pre> TYPE CNT: UINT; FREQ: REAL := 50.0; ANALOG_CHANNEL_CONFIG: ANALOG_CHANNEL_CONFIGURATION := (MIN_SCALE := 0, MAX_SCALE := 4000); END_TYPE </pre>	Initialization new initialization
12	Initialization using constant expressions	<pre> TYPE PIx2: REAL := 2 * 3.1416; END_TYPE </pre>	Uses a constant expression
13a 13b	Reference data type	<pre> TYPE ref1 : REF_TO INT; ref2 : REF_TO INT := NULL; st : STRUCT a : INT; ref3 : REF_TO INT := REF(a); END_STRUCT END_TYPE VAR_GLOBAL g : ARRAY [0..7] OF INT; END_VAR PROGRAM A VAR_EXTERNAL g : ARRAY [0..7] OF INT; END_VAR VAR var1 : REF_TO INT := REF(g[3]); END_VAR ; END_PROGRAM </pre>	Initialization Initialization Initialization
The declaration of data type is possible without initialization (feature a) or with (feature b) initialization. If only feature (a) is supported, the data type is initialized with the default initial value. If feature (b) is supported, the data type shall be initialized with the given value or default initial value, if no initial value is given. Variables with directly represented elements of a structure – partly specified using “*” may not be used in the VAR_INPUT, VAR_IN_OUT or VAR_TEMP sections.			

972 6.4.4.2 Enumerated data type

973 6.4.4.2.1 General

974 The declaration of an enumerated data type specifies that the value of any data element of
 975 that type can only take one of the values given in the associated list of identifiers, as illustrat-
 976 ed in Table 11.

977 An enumerated identifier must not be assigned a value, as it is mandatory for a datatype with
 978 named values. An enumerated data type T is not related to any other type, except for directly
 979 derived datatypes of (transitive) base type T.

980 The enumeration list defines an ordered set of enumerated values, starting with the first iden-
 981 tifier of the list, and ending with the last one. Different enumerated data types may use the
 982 same identifiers for enumerated values. The maximum allowed number of enumerated values
 983 is implementer specific.

984 To enable unique identification when used in a particular context, enumerated literals may be
 985 qualified by a prefix consisting of their associated data type name and the hash sign (number
 986 sign) '#', similar to typed literals. Such a prefix shall not be used in an enumeration list.

987 It is an error if sufficient information is not provided in an enumerated literal to determine its
 988 value unambiguously (see example below).

989 The definition of an enumeration may include a base type. If the base type is not given, it is
 990 implementer specific. The base type must be a one of the `ANY_INT` or `ANY_BIT` types without
 991 `BOOL`.

992 NOTE The definition of a base type enables a user to determine the memory size of respective enumeration varia-
 993 bles.

994 An enumeration type is unrelated to its base type with respect to the type system. In particu-
 995 lar, the types are not equal and neither is a subtype of the other.

996 Each enumeration value is represented by a dedicated value of the base type.

997 It is an error, if the enumeration defines more values than can be represented by its base
 998 type.

999 NOTE The user has no means to determine the base type value that has been assigned to an enumeration value.

1000 NOTE An expression of the enumeration's type can only evaluate to values of the enumeration.

EXAMPLE Enumerated data type

```

TYPE
  Traffic_light:    (Red, Amber, Green);
  Painting_colors: (Red, Yellow, Green, Blue) := Blue;
END_TYPE

VAR
  My_Traffic_light: Traffic_light := Amber;
END_VAR

IF My_Traffic_light = Traffic_light#Amber THEN ...      // OK
IF My_Traffic_light = Traffic_light#Red THEN ...       // OK
IF My_Traffic_light = Amber THEN ...                   // OK - Amber is unique
IF My_Traffic_light = Red THEN ...                     // ERROR - Red is not unique

```

1001 6.4.4.2.2 Initialization

1002 The default initial value of an enumerated data type shall be the first identifier in the associat-
 1003 ed enumeration list.

1004 The user can initialize the data type with a user-defined value out of the list of its enumerated
 1005 values. This initialization has priority.

1006 As shown in Table 11 for `ANALOG_SIGNAL_RANGE`, the user-defined default initial value of
 1007 the enumerated data type is the assigned value `UNIPOLAR_1_5V`.

1008 The user-defined assignment of the initial value of the data type is a feature in Table 11.

1009 6.4.4.3 Data type with named values

1010 6.4.4.3.1 General

1011 Related to the enumeration data type – where the values of enumerated identifiers are not
 1012 known by the user – is an enumerated data type with named values. The declaration specifies
 1013 the data type and assigns the values of the named values, as illustrated in Table 11.

1014 Declaring named values does not limit the use of the value range of variables of these data
 1015 types; i.e. other constants can be assigned, or can arise through calculations.

1016 In addition to the enumeration data type the same operations on named values shall be avail-
1017 able like on the base type of the named values.

1018 To enable unique identification when used in a particular context, named values may be quali-
1019 fied by a prefix consisting of their associated data type name and the hash sign (number sign)
1020 '#', similar to typed literals.

1021 Such a prefix shall not be used in a declaration list. It is an error if sufficient information is not
1022 provided in an enumerated literal to determine its value unambiguously (see example below).

EXAMPLE Data type with named values

```

TYPE
  Traffic_light: INT (Red := 1, Amber := 2, Green := 3):= Green;
  Painting_colors: INT (Red := 1, Yellow := 2, Green := 3, Blue := 4):= Blue;
  BaseColors: DWORD (Red := 16#00FF0000, Green := 16#0000FF00, Blue := 16#000000FF);
  Colors: DWORD (White := BaseColors#Red OR BaseColors#Green OR BaseColors#Blue,
                  Black := BaseColors#Red AND BaseColors#Green AND BaseColors#Blue);
END_TYPE

VAR
  My_Traffic_light: Traffic_light;
END_VAR

My_Traffic_light := 27;           // Assignment from a constant
My_Traffic_light := Amber + 1;   // Assignment from an expression
                                // Note: This is not possible for enumerated values
My_Traffic_light := Traffic_light#Red + 1;

IF My_Traffic_light = 123 THEN ...           // OK
IF My_Traffic_light = Traffic_light#Amber THEN ... // OK
IF My_Traffic_light = Traffic_light#Red THEN ... // OK
IF My_Traffic_light = Amber THEN ...         // OK because Amber is unique
IF My_Painting_Color = Amber THEN ...         // OK
IF My_Traffic_light = Red THEN ...            // Error because Red is not unique
My_Traffic_light := Painting_colors#Red;      // OK because it is the same data type
My_Traffic_light := Painting_colors#Blue-1; // OK
My_Traffic_light := My_Painting_colors;      // OK

```

1023 6.4.4.3.2 Initialization

1024 The default value for a data type with named values is the first data element in the enumera-
1025 tion list. In the example above for `Traffic_light` this element is `Red`.

1026 The user can initialize the data type with a user-defined value. The initialization is not restrict-
1027 ed to named values, any value from within the range of the base data type may be used. This
1028 initialization has priority.

1029 In the example, the user-defined initial value of the named value data type for `Traf-`
1030 `fic_light` is `Green`.

1031 The user-defined assignment of the initial value of the data type is a feature in Table 11.

1032 6.4.4.4 Subrange data type

1033 6.4.4.4.1 General

1034 A subrange is a subtype of its base type, which is limited to a user-defined range of values.
1035 An expression of the base type can be given, where a value of the subrange type is expected.
1036 It is an error, if a data element of a subrange type is assigned a value that is not a value of
1037 the subrange type, i.e. does not lie in its range which includes the specified upper and lower
1038 limit.

1039 NOTE It is not an error, if an intermediate value (e.g. during expression evaluation) exceeds the subrange's
 1040 bounds, as long as the bounds of its base type are not exceeded. The limits of a subrange shall be literals
 1041 or constant expressions and must be values of the base type.

```

1042
1043 EXAMPLE:
1044 TYPE
1045     ANALOG_DATA: INT (-4096..4095) := 0;
1046 END_TYPE
1047
1048 PROGRAM prog
1049 VAR
1050     i : INT;
1051     a : ANALOG_DATA;
1052     b AT %I* : ANALOG_DATA;
1053 END_VAR
1054 [...]
1055     a := i; // is an error, if i exceeds the range -4095..4096;
1056     a := (a*2)/3; // no error. a*2 may exceed the subrange, but the assigned value
1057                  // does not
1058     a := (a*20)/30; // is an error, if a*20 exceeds the range -32768..32767 of INT
1059     i := a;        // no error
1060     // externally supplied values of b may exceed the range without leading to an
1061     // error
1062     IF NOT LE (-4096,b,4095) THEN (*error handling*) END_IF;
1063 END_PROGRAM
  
```

1064 6.4.4.4.2 Initialization

1065 The default initial values for data types with subrange shall be the first (lower) limit of the
 1066 subrange.

1067 The user can initialize the data type with a user-defined value from the subrange. This initiali-
 1068 zation has priority.

1069 For instance, as shown in the example in Table 11, the default initial value of elements of type
 1070 ANALOG_DATA is -4095, while with explicit initialization, the default initial value is zero (as
 1071 declared).

1072 6.4.4.5 Array data type

1073 6.4.4.5.1 General

1074 The declaration of an array data type specifies that a sufficient amount of data storage shall
 1075 be allocated for each element of that type to store all the data which can be indexed by the
 1076 specified index subrange(s), as illustrated in Table 11.

1077 An array is a collection of data elements of the same data type. Elementary and user-defined
 1078 data types, function block types and classes can be used as type of an array element. This
 1079 collection of data elements is referenced by one or more subscripts enclosed in brackets and
 1080 separated by commas. It shall be an error if the value of a subscript is outside the range
 1081 specified in the declaration of the array.

1082 NOTE This error can be detected only at runtime for a computed index.

1083 The maximum number of array subscripts, maximum array size and maximum range of sub-
 1084 script values are implementer specific.

1085 The limits of the index subrange(s) shall be literals or constant expressions. Arrays with vari-
 1086 able length are defined in 6.5.3.

1087 In the ST language a subscript shall be an expression yielding a value corresponding to one
 1088 of the sub-types of generic type ANY_INT.

1089 The form of subscripts in the graphical languages defined in Clause 8 is restricted to single-
 1090 element variables or integer literals.

EXAMPLE

a) Declaration of an array

```
VAR myANALOG_16: ARRAY [1..16] OF ANALOG_DATA
    := [8(-4095), 8(4095)];    // user-defined initial values
END_VAR
```

b) Usage of array variables in the ST language could be:

```
OUTARY[6,SYM] := INARY[0] + INARY[7] - INARY[i] * %IW62;
```

1091 6.4.4.5.2 Initialization

1092 The default initial value of each array element is the initial value defined for the data type of
1093 the array elements.

1094 The user can initialize an array type with a user-defined value. This initialization has priority.

1095 The user-defined initial value of an array is assigned in form of a list which may use paren-
1096 theses to express repetitions.

1097 During initialization of the array data types, the rightmost subscript of an array shall vary most
1098 rapidly with respect to filling the array from the list of initialization values.

EXAMPLE Initialization of an array

```
A: ARRAY [0..5] OF INT := [2(1, 2, 3)]
is equivalent to the initialization sequence 1, 2, 3, 1, 2, 3.
```

1099 If the number of initial values given in the initialization list exceeds the number of array en-
1100 tries, the excess (rightmost) initial values shall be ignored. If the number of initial values is
1101 less than the number of array entries, the remaining array entries shall be filled with the de-
1102 fault initial values for the corresponding data type. In either case, the user shall be warned of
1103 this condition during preparation of the program for execution.

1104 The user-defined assignment of the initial value of the data type is a feature in Table 11.

1105 6.4.4.6 Structured data type

1106 6.4.4.6.1 General

1107 The declaration of a structured data type (STRUCT) specifies that this data type shall contain a
1108 collection of sub-elements of the specified types which can be accessed by the specified
1109 names, as illustrated in Table 11.

1110 An element of a structured data type shall be represented by two or more identifiers or array
1111 accesses separated by single periods “.”. The first identifier represents the name of the struc-
1112 tured element, and subsequent identifiers represent the sequence of element names to ac-
1113 cess the particular data element within the data structure. Elementary and user-defined data
1114 types, function block types and classes can be used as type of a structure element.

1115 For instance, an element of data type ANALOG_CHANNEL_CONFIGURATION as declared in
1116 Table 11 will contain a RANGE sub-element of type ANALOG_SIGNAL_RANGE, a MIN_SCALE
1117 sub-element of type ANALOG_DATA, and a MAX_SCALE element of type ANALOG_DATA.

1118 The maximum number of structure elements, the maximum amount of data that can be con-
1119 tained in a structure, and the maximum number of nested levels of structure element address-
1120 ing are implementer specific.

1121 Two structured variables are assignment compatible only if they are of the same data type.

EXAMPLE Declaration and usage of a structured data type and structured variable

a) Declaration of a structured data type

```

TYPE
  ANALOG_SIGNAL_RANGE: (BIPOLAR_10V, UNIPOLAR_10V);
  ANALOG_DATA: INT (-4095 .. 4095);
  ANALOG_CHANNEL_CONFIGURATION:
    STRUCT
      RANGE:      ANALOG_SIGNAL_RANGE;
      MIN_SCALE:  ANALOG_DATA;
      MAX_SCALE:  ANALOG_DATA;
    END_STRUCT;
END_TYPE

```

b) Declaration of a structured variable

```

VAR
  MODULE_CONFIG:  ANALOG_CHANNEL_CONFIGURATION;
  MODULE_8_CONF:  ARRAY [1..8] OF ANALOG_CHANNEL_CONFIGURATION;
END_VAR

```

c) Usage of structured variables in the ST language:

```

MODULE_CONFIG.MIN_SCALE := -2047;
MODULE_8_CONF[5].RANGE := BIPOLAR_10V;

```

6.4.4.6.2 Initialization

The default initial value of a structured data type is constituted of the initial values of its elements.

The default initial value of such an element is the initial value of its type. A user-defined initial value for an element can be defined, which takes precedence over the default initial value as initial value.

A user-defined initial value can be specified for a structured data type. The user-defined initial value is given by specifying values for the elements of the data type, using the syntax defined in Annex A.

A user-defined initial value for a structured data type does not need to define a value for each element of that structure.

If no user-defined initial value is specified for a structured data type, then the initial value of that data type is the default initial value.

If a user-defined initial value does not define a value for an element, the value for that element in the structure's default initial value is used as initial value.

These rules apply recursively if the type of an element of a structured data type is itself a structured data type.

NOTE Informally, the initial value for a structured data type is the default initial value of the data type, unless specifically overridden for some or all of its elements.

Formally, let `Init_D` be the default initial value of a structured data type `S` and `Init_U` the user-defined initial value of `S`. If `C` is a component of `S` that is included in the definition of `Init_U`, the initial value of `C` is defined to be the value of `C` in `Init_U`. Otherwise, the initial value of `C` defined by `Init_D`.

EXAMPLE: Definition of a structure and initial values

```

TYPE
  T : INT := 1;

```

```

1156 S : STRUCT
1157 I : T;          // initial value is 1
1158 J : T := 20;    // initial value is 20
1159 K : T := 30;    // initial value is 30
1160 L : T := 40;    // initial value is 40
1161 END_STRUCT;
1162
1163 U : S := (K := 300); // initial values are (I=1,J=20,K=300,L=40)
1164
1165 V : U := (L := 4000); // initial values are (I=1,J=20,K=300,L=4000)
1166 END_TYPE
1167

```

1168 The initial value 1 of the element I is its default initial value, which is the initial value of T.

1169 The initial value 20 of the element J is its user-defined initial value.

1170 The initial value (I:=1, J:=20, K:=30, L:=40) of the structural type S is its default initial value,
1171 which is constituted by the initial values of its elements.

1172 The initial value (I:=1, J:=20, K:=300, L:=40) of U is composed of its default initial value,
1173 which is the initial value of S and of U's user-defined initial value, which is (K:=300). The lat-
1174 ter takes precedence.

1175 The initial value (I:=1, J:=20, K:=300, L:=4000) of V is composed of its default initial value,
1176 which is the initial value of U and of V's user-defined initial value, which is (L:=4000). The lat-
1177 ter takes precedence.

1178 6.4.4.7 Relative location for elements of structured data types (AT)

1179 6.4.4.7.1 General

1180 The locations (addresses) of the elements of a structured type can be defined relative to the
1181 beginning of the structure.

1182 In this case the name of each component of this structure shall be followed by the keyword AT
1183 and a relative location. The declaration may contain gaps in the memory layout.

1184 The relative location consists of a '%' (percent), the location qualifier and a bit or byte location.
1185 A byte location is an unsigned integer literal denoting the byte offset. A bit location consists of
1186 a byte offset, followed by a '.' (point), and the bit offset as unsigned integer literal out of the
1187 range of 0 to 7. White spaces are not allowed within the relative location.

1188 The components of the structure shall not overlap in their memory layout, except if the key-
1189 word OVERLAP has been given in the declaration.

1190 Overlapping of strings is beyond the scope of this standard.

1191 NOTE Counting of bit offsets starts with 0 at the least significant bit. Counting of byte offsets starts at the begin-
1192 ning of the structure with byte offset 0.

EXAMPLE Relative location and overlapping in a structure

```

TYPE
  Com1_data: STRUCT
    head      AT %B0:      INT;          // at location 0
    length     AT %B2:      USINT := 26;  // at location 2
    flag1      AT %X3.0:    BOOL;          // at location 3.0
    end        AT %B25:     BYTE;          // at 25, leaving a gap
  END_STRUCT;

  Com2_data: STRUCT OVERLAP
    head      AT %B0:      INT;          // at location 0
    length     AT %B2:      USINT;        // at location 2
    flag2      AT %X3.3:    BOOL;          // at location 3.3
    data1      AT %B5:      BYTE;          // at locations 5, overlapped
    data2      AT %B5:      REAL;          // at locations 5 to 8
    end        AT %B19:     BYTE;          // at 19, leaving a gap
  END_STRUCT;

  Com_data: STRUCT OVERLAP // C1 and C2 overlap
    C1 at %B0: Com1_data;
    C2 at %B0: Com2_data;
  END_STRUCT;
END_TYPE

```

1193 6.4.4.7.2 Initialization

1194 Overlapped structures cannot be initialized explicitly.

1195 6.4.4.8 Directly represented components of a structure – partly specified using “* ”

1196 The asterisk notation “*” in Table 11 can be used to denote not yet fully specified locations for
 1197 directly represented components of a structure.

EXAMPLE Assigning of the components of a structure to not yet located inputs and outputs.

```

TYPE
  HW_COMP: STRUCT;
  IN      AT %I*: BOOL;
  VAL     AT %I*: DWORD;
  OUT     AT %Q*: BOOL;
  OUT_VAR AT %Q*: WORD;
  ITNL_VAR: REAL; // not located
  END_STRUCT;
END_TYPE

```

1198 In the case that a directly represented component of a structure is used in a location assign-
 1199 ment in the declaration part of a program, a function block type, or a class, an asterisk “*”
 1200 shall be used in place of the size prefix and the unsigned integer(s) in the concatenation to
 1201 indicate that the direct representation is not yet fully specified.

1202 The use of this feature requires that the location of the structured variable so declared shall
 1203 be fully specified inside the VAR_CONFIG...END_VAR construction of the configuration for
 1204 every instance of the containing type.

1205 Variables of this type shall not be used in a VAR_INPUT, VAR_IN_OUT, or VAR_TEMP sec-
 1206 tion.

1207 It is an error if any of the full specifications in the VAR_CONFIG...END_VAR construction is
 1208 missing for any incomplete address specification expressed by the asterisk “*” in any
 1209 instance of programs or function block types which contain such incomplete specifications.

1210 6.4.4.9 Directly derived data type

1211 6.4.4.9.1 General

1212 A user-defined data type may be directly derived from an elementary data type or a previously
1213 user-defined data type.

1214 The effective datatype of a directly derived datatype T is the type U it is derived from, unless
1215 U is itself a directly derived datatype. In this case the effective datatype of T is the effective
1216 datatype of U.

1217 This may be used to define new type-specific initial values.

EXAMPLE Directly derived data type

```
TYPE
myInt1123:      INT := 123;
myNewArrayType: ANALOG_16_INPUT_DATA := [8(-1023), 8(1023)];
Com3_data:      Com2_data:= (head := 3, length :=40);
R1: REAL := 1.0;
R2: R1; // the effective data type of R2 is REAL
END_TYPE
```

1218 6.4.4.9.2 Initialization

1219 The default initial value of a directly derived datatype is the initial value of the data type it is
1220 derived from.

1221 A user-defined initial value may be defined as initial value and takes precedence over the de-
1222 fault initial value, according to the provisions of the effective datatype.

EXAMPLE 1 User-defined data types - usage

Given the declaration of ANALOG_16_INPUT_DATA in Table 11

and the declaration VAR INS: ANALOG_16_INPUT_DATA; END_VAR

the variables INS[1] through INS[16] can be used anywhere a variable of type INT could be used.

EXAMPLE 2

Similarly, given the definition of Com_data in Table 11

and additionally the declaration VAR telegram: Com_data; END_VAR

the variable telegram.length can be used anywhere a variable of type USINT could be used.

EXAMPLE 3

Given the declarations of ANALOG_16_INPUT_CONFIGURATION, ANALOG_CHANNEL_CONFIGURATION and
ANALOG_DATA in Table 11

and the declaration VAR CONF: ANALOG_16_INPUT_CONFIGURATION; END_VAR

the variable CONF.CHANNEL[2].MIN_SCALE can be used anywhere that a variable of type INT could be
used.

1223 6.4.4.10 References

1224 6.4.4.10.1 Reference declaration

1225 A reference data type is defined using the keyword REF_TO, followed by a type description of
1226 a base type.

1227 Note: The base type may be an elementary data type or a user defined data type.
1228

1230 If the base type is a directly derived datatype or if its definition includes directly derived
1231 datatypes, their effective types (see 6.4.4.9) are regarded instead.

Example:

```

TYPE
  A : INT;
  B : ARRAY [1..2] OF INT;
  C : ARRAY [1..2] OF A;
  D : C;
  E : REF_TO B;
  F : REF_TO D; (* Type F is equal to type E *)
END_TYPE

```

A value of a reference data type is either

- the value NULL or
- a reference to an instance of the base data type or
- a reference to an instance of a type that extends the base type, where the base type is a class or a function block

The value NULL represents a reference to no instance.

A reference type with base type U can be used where a reference type with base type T is expected, if U extends T.

Example:

```

CLASS A ... END_CLASS
CLASS B EXTENDS A ... END_CLASS
CLASS C EXTENDS B ... END_CLASS

TYPE
  RA : REF_TO A;
  RB : REF_TO B; (* Value of type RB can be given, where a value of type
RA is expected, but not where a value of type RC is expected *)
  RC : REF_TO C; (* Value of type RC can be given, where a value of type
RA or RB is expected *)
END_VAR

```

6.4.4.10.2 Initialization

The default initial value for a reference data type is NULL.

A user-defined initial value can be specified for a reference data type.

6.4.4.10.3 Operations

Given an instance X, the reference operation REF in REF(X) yields a reference to that instance. The type of REF(X) is a reference data type with the data type of X as base type.

Given a reference value R, the dereference operation ^ (caret) in R^ yields the instance referenced by R. The type of R^ is the base type of the reference data type of R.

Note: A dereferenced value can be used like a variable of the respective base type.

The expression REF(X)^ is equivalent to X.

Dereferencing shall be done explicitly.

It is an error, if the value NULL is dereferenced.

It is an error, if a constant instance is referenced, e.g. a variable that is defined in a VAR CONSTANT section.

It is an error, if a reference value may be used beyond the lifetime of the referenced instance.

1292 NOTE This may be the case, if an instance is referenced, whose lifetime may end during the execution of the
1293 programmable controller. This includes for example variables that are defined in the context of a function-
1294 definition, in a VAR_TEMP section or in a VAR_IN_OUT section.
1295

1296 NOTE 1 Possible checks of NULL references can be done at compile time, by the runtime system, or by the appli-
1297 cation program.

1298 NOTE 2 Reference arithmetic is not recommended and is beyond the scope of this part of IEC 61131.
1299

EXAMPLE 1

```

TYPE
S1: STRUCT
  SC1: INT;
  SC2: REAL;
END_STRUCT;
A1: ARRAY[1..99] OF INT;
END_TYPE

VAR
  myS1: S1;
  myA1: A1;
  myRefS1: REF_TO S1 := REF(myS1);
  myRefA1: REF_TO A1 := REF(myA1);
  myRefInt: REF_TO INT := REF(myA1[1]);
END_VAR

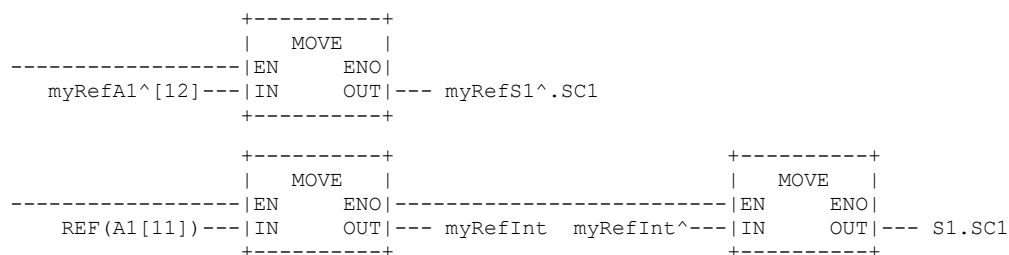
myRefS1^.SC1 := myRefA1^[12];    // in this case, equivalent to S1.SC1 := A1[12];
myRefInt := REF(A1[11]);

S1.SC1 := myRefInt^;              // assigns the value of A1[11] to S1.SC1

```

EXAMPLE 2

Graphical representation of the statements of Example 1



1300

1301 Table 12 defines the features for reference operations.

1302

Table 12 – Reference operations

No	Description	Example
	Declaration	
1	Declaration of a reference type	<pre> TYPE myRefType: REF_TO INT; END_TYPE </pre>
	Assignment and comparison	
2a	Assignment reference to reference	myRefType1 := myRefType2;
2b	Assignment reference to parameter of function, function block and method	myFB (a := myRefS1);
2c	Comparison with NULL	IF myRefS1 = NULL THEN ...