Technical Paper
PLCopen Promotional Committee Training
As part of the
Software Construction Guidelines initiative

Sub Committee
Coding Guidelines

PLCopen Document, Version 1.0, Official Release

DISCLAIMER OF WARANTIES

The name 'PLCopen®' is a registered trade mark and together with the PLCopen logos owned by the association PLCopen.

This document is provided on an 'as is' basis and may be subject to future additions, modifications, or corrections. PLCopen hereby disclaims all warranties of any kind, express or implied, including any warranty of merchantability or fitness for a particular purpose, for this moment. In no event will PLCopen be responsible for any loss or damage arising out or resulting from any defect, error or omission in this document or from anyone's use of or reliance on this document.

Copyright © 2016 by PLCopen. All rights reserved.

Date: April 20, 2016

Total number of pages: 127

The following paper is an official PLCopen document:

Coding Guidelines

It summarises the results of the PLCopen workgroup Software Construction Guidelines Task Force Coding Guidelines, containing contributions of all its members.

The present specification was written thanks to the members of this Task Force:

Person	***Company***
Andreas Weichelt	Phoenix Contact
Barry Butcher	Omron
Bernhard Jany	Siemens
Bernhard Werner	3S / Codesys
Bert van der Linden	ATS International
Boris Waldeck	Phoenix Contact
Carina Schlicker	HS Augsburg
Christoph Berger	HS Augsburg
Denis Chalon	Itris
Edward Nicolson	Yaskawa
Eric Pierrel	Itris
Geert Vanstraelen	Macq
Hans-Peter Otto	privat
Hendrik Simon	RWTH Aachen
Hiroshi Yoshida	Omron
Kevin Hull	Yaskawa
Matthias Kremberg	Phoenix Contact
Peter Erning	ABB
René Heijma	Omron
Rolf Hänisch	Fraunhofer FOKUS
Sebastian Biallas	RWTH Aachen
Wolfgang Zeller	HS Augsburg
Eelco van der Wal	PLCopen

Change Status List:

Version	Date	Change/ comment
V 0.1	March 24, 2015	Initial Document as result of work in the original wiki site
V 0.2	June 5, 2015	As result of the webmeeting on June 1 based on the feedback
V 0.3	July 1, 2015	As a result of the webmeeting and last feedback in
V 0.99	July 23, 2015	Version to be published to the open community as Release for Comments till Oct. 23, 2015
V 0.99A	Jan. 17, 2015	As a result of the Face to face meeting in Frankfurt, including decisions on all feedback items
V 0.99B	Jan. 20, 2016	As result of the final feedback and editorial issues
V 1.0	April 20, 2016	Official published version

Table of Contents

1. INTRODUCTION 9

2. HOW-TO USE THIS DOCUMENT 10

2.1. METHODOLOGY USED TO BUILD THIS DOCUMENT 11

2.2. DOCUMENT STRUCTURE 12

2.3. RULES DESCRIPTION FORMAT 13

2.4. REFERENCES 15

3. NAMING RULES 16

3.1. ADDITIONAL RULES FOR VARIABLES ONLY 16

3.1.1. Avoid physical addresses 16

3.1.2. Define type prefixes for Variables (if used) 17

3.2. TASKS, PROGRAMS, FUNCTIONS BLOCKS, FUNCTIONS, VARIABLES, UDTS AND NAMESPACES 20

3.2.1. Define the names to avoid 20

3.2.2. Define the use of case (capitals) 22

3.2.3. Local names shall not shadow global names 25

3.2.4. Define an acceptable name length 27

3.2.5. Define naming rules for namespaces 29

3.2.6. Define the acceptable character set 31

3.2.7. Different element types should not bear the same name 32

3.2.8. Define name prefixes for user defined types 33

4. COMMENT RULES 35

4.1. COMMENTS SHALL DESCRIBE THE INTENTION OF THE CODE 36

4.2. ALL ELEMENTS SHALL BE COMMENTED 38

4.3. AVOID NESTED COMMENTS 39

4.4. COMMENTS MAY NOT INCLUDE CODE 40

4.5. USE SINGLE LINE COMMENTS 41

4.6. DEFINE COMMENTS LANGUAGE 43

5. CODING PRACTICE 44

5.1. ACCESS TO A MEMBER SHALL BE BY NAME 44

5.2. ALL CODE SHALL BE USED IN THE APPLICATION 45

5.3. ALL VARIABLES SHALL BE INITIALIZED BEFORE BEING USED 47

5.4. DIRECT ADDRESSING SHOULD NOT OVERLAP 51
5.5. APPLICATIONS SHALL BE WELL DESIGNED 53
5.6. AVOID EXTERNAL VARIABLES IN FUNCTIONS, FUNCTION BLOCKS AND CLASSES 54
5.7. ERROR INFORMATION SHALL BE TESTED 56
5.8. FLOATING POINT COMPARISON SHALL NOT BE EQUALITY OR INEQUALITY 58
5.9. TIME AND PHYSICAL MEASURES COMPARISON SHALL NOT BE EQUALITY OR INEQUALITY ..59
5.10. LIMIT THE COMPLEXITY OF POU CODE 60
5.11. AVOID MULTIPLE WRITES FROM MULTIPLE TASKS 63
5.12. MANAGE SYNCHRONIZATION AMONG TASKS 65
5.13. PHYSICAL OUTPUTS SHALL BE WRITTEN ONCE PER PLC CYCLE 68
5.14. POUS SHALL NOT CALL THEMSELVES DIRECTLY OR INDIRECTLY 69
5.15. POUS SHALL HAVE A SINGLE POINT OF EXIT 71
5.16. READ A VARIABLE WRITTEN BY ANOTHER TASK ONLY ONCE PER CYCLE 72
5.17. TASKS SHALL ONLY CALL PROGRAM POUS AND NOT FUNCTION BLOCKS 74
5.18. USAGE OF PARAMETERS SHALL MATCH THEIR DECLARATION MODE 75
5.19. USE OF GLOBAL VARIABLES SHALL BE LIMITED 77
5.20. USAGE OF JUMP AND RETURN SHOULD BE AVOIDED 81
5.21. FUNCTION BLOCK INSTANCES SHOULD BE CALLED ONLY ONCE 84
5.22. USE VAR_TEMP FOR TEMPORARY VARIABLE DECLARATION 86
5.23. SELECT APPROPRIATE DATA TYPE 88
5.24. DEFINE MAXIMUM NUMBER OF INPUT/OUTPUT/IN-OUT VARIABLES OF A POU 91
5.25. DO NOT DECLARE VARIABLES THAT ARE NOT USED 94
5.26. DATA TYPES CONVERSION SHOULD BE EXPLICIT 95
5.27. A GLOBAL VARIABLE MAY BE WRITTEN ONLY BY ONE PROGRAM 97
5.28. AVOID DEPRECATED FEATURES 98
6. LANGUAGES 99
6.1. DEFINE INDENTATION 99
6.2. FUNCTION BLOCK DIAGRAM FBD 100
6.2.1. Avoid assignments of intermediate results within networks 100
6.2.2. Define maximum complexity of single network 101
6.3. LADDER (LD) 102
6.3.1. A coil should not be followed by a contact 102
6.3.2. Define maximum rung complexity 103

6.4. SEQUENTIAL FUNCTION CHART (SFC) 104
6.4.1. Closing divergent paths 104
6.4.2. Do not program an SFC action block in SFC 106
6.4.3. Define maximum complexity 107
6.5. STRUCTURED TEXT (ST) 108
6.5.1. Define General formatting rules 108
6.5.2. Usage of Continue and Exit instruction should be avoided 110
6.5.3. Define the maximum line length 112
6.5.4. Loop variables should not be modified inside a FOR loop 113
6.5.5. FOR loop variable usage should not be used outside the FOR loop 115
6.5.6. Passing parameters should be clear 117
6.5.7. Use parenthesis to explicitly express operation precedence 119
6.5.8. Define the use of tabs 120
6.5.9. Each IF instruction should have an ELSE clause 121
7. VENDOR SPECIFIC IEC 61131-3 EXTENSIONS 122
7.1. DYNAMIC MEMORY ALLOCATION SHALL NOT BE USED 122
7.2. POINTER ARITHMETIC SHALL NOT BE USED 123
7.3. SOME COMPARATOR INSTRUCTIONS SHALL NOT BE USED FOR POINTER OR REFERENCE MANIPULATION 124
8. ANNEX 1 – OVERVIEW OF THE RULES VIA THEIR PRIORITIES 126

Another way of looking to the rules is via their classification and number, as shown hereunder:

Rule #	*Chapter*	*Name*	*Page*
	3.	***Naming Rules***	***16***
	3.1.	*Additional rules for Variables only*	*16*
N1	3.1.1.	Avoid physical addresses	16
N2	3.1.2.	Define type prefixes for Variables (if used)	17
	3.2.	*Tasks, Programs, Functions Blocks, Functions, Variables, UDTs and namespaces*	*20*
N3	3.2.1.	Define the names to avoid	20
N4	3.2.2.	Define the use of case (capitals)	22
N5	3.2.3.	Local names shall not shadow global names	25
N6	3.2.4.	Define an acceptable name length	27
N7	3.2.5.	Define naming rules for namespaces	29
N8	3.2.6.	Define the acceptable character set	31
N9	3.2.7.	Different element types should not bear the same name	32
N10	3.2.8.	Define name prefixes for user defined types	33
	4.	***Comment Rules***	***35***
C1	4.1.	Comments shall describe the intention of the code	36
C2	4.2.	All elements shall be commented	38
C3	4.3.	Avoid nested comments	39
C4	4.4.	Comments may not include code	40
C5	4.5.	Use single line comments	41
C6	4.6.	Define comments language	43
	5	***Coding Practice***	***44***
CP1	5.1.	Access to a member shall be by name	44
CP2	5.2.	All code shall be used in the application	45
CP3	5.3.	All variables shall be initialized before being used	47
CP4	5.4.	Direct addressing should not overlap	51
CP5	5.5.	Applications shall be well designed	53
CP6	5.6.	Avoid external variables in functions, function blocks and classes	54
CP7	5.7.	Error information shall be tested	56
CP8	5.8.	Floating point comparison shall not be equality or inequality	58
CP28	5.9	Time and physical measures comparison shall not be equality or inequality	59
CP9	5.10.	Limit the complexity of POU code	60
CP10	5.11.	Avoid multiple writes from multiple tasks	63
CP11	5.12.	Manage synchronization among tasks	65

CP12	5.13.	Physical outputs shall be written once per PLC cycle	68
CP13	5.14.	POUs shall not call themselves directly or indirectly	69
CP14	5.15.	POUs shall have a single point of exit	71
CP15	5.16.	Read a variable written by another task only once per cycle	72
CP16	5.17	Tasks shall only call program POUs and not Function Blocks	74
CP17	5.18.	Usage of parameters shall match their declaration mode	75
CP18	5.19.	Use of global variables shall be limited	77
CP19	5.20.	Usage of jump and return should be avoided	81
CP20	5.21.	Function block instances should be called only once	84
CP21	5.22.	Use VAR_TEMP for temporary variable declaration	86
CP22	5.23.	Select appropriate data type	88
CP23	5.24.	Define maximum number of input/output/in-out variables of a POU	91
CP24	5.25.	Do not declare variables that are not used	94
CP25	5.26.	Data types conversion should be explicit	95
CP26	5.27.	A global variable may be written only by one PROGRAM	97
CP27	5.28.	Avoid deprecated features	98
	6.	***Languages***	***99***
L1	6.1.	Define indentation	99
	6.2.	*Function Block Diagram FBD*	*100*
L2	6.2.1.	Avoid assignments of intermediate results within networks	100
L3	6.2.2.	Define maximum complexity of single network	101
	6.3.	*Ladder (LD)*	*102*
L5	6.3.1.	A coil should not be followed by a contact	102
L6	6.3.2.	Define maximum rung complexity	103
	6.4.	*Sequential function chart (SFC)*	*104*
L7	6.4.1.	Closing divergent paths	104
L8	6.4.2.	Do not program an SFC action block in SFC	106
L9	6.4.3.	Define maximum complexity	107
	6.5.	***Structured text (ST)***	***108***
L4	6.5.1.	Define general formatting rules	102
L10	6.5.2.	Usage of Continue and Exit instruction should be avoided	110
L11	6.5.3.	Define the maximum line length	112
L22	6.5.4.	Loop variables should not be modified inside a FOR loop	113
L13	6.5.4.	FOR loop variable usage should not be used outside the FOR loop	115
L14	6.5.5.	Passing parameters should be clear	117

L15	6.5.6.	Use parenthesis to explicitly express operation precedence	119
L16	6.5.7.	Define the use of tabs	120
L17	6.5.8.	Each IF instruction should have an ELSE clause	121
	7.	***Vendor Specific IEC 61131-3 Extensions***	***122***
E1	7.1.	Dynamic memory allocation shall not be used	122
E2	7.2.	Pointer arithmetic shall not be used	123
E3	7.3.	Some comparator instructions shall not be used for pointers or reference manipulation	124

1. Introduction

Although there are coding guidelines for many programming languages, these are nearly non-existent for the important area of industrial control, e.g. IEC 61131-3 and its PLCopen extensions.

Nevertheless, the software in the industrial environments becomes more and more important, the software projects become larger, and the costs of errors increase. Software nowadays absorbs half of the initial project costs and between 40 and 80% deals with maintenance over the life cycle costs of the software.

In order to deal with the complexity of larger programs one needs modern software development processes supporting a structured approach. Also, we need to increase the efficiency in coding via re-use of pre-defined functionalities and to help to better understand the program over the life cycle.

With the above message PLCopen invited interested parties to join the working group of Software Construction Guidelines. The kick-off meeting resulted in several working groups for the different areas of interest, including their working packages (targets):

- Coding Guidelines
- Software quality issues and software consistency
- Creating PLCopen compliant Function Blocks
- Structuring and decomposition via SFC (do’s & don’ts)
- Guidance for documentation in software programs
- Library usage
- Software development process

The key topic of the new PLCopen Software Construction Guidelines working group is the definition of Rules, Coding Patterns and Guidance and how to use them in Industrial Automation. These rules will be published as technical documents, as well as possibly on websites and software tools, and marketed by PLCopen.

The results of the working group should be based on the IEC 61131-3 1st and 2nd edition standard but should be easily extensible to the 3rd edition which was released in February 2013.

The aim of the subgroup Coding Guidelines is to define a set of rules and to provide a PLCopen proposal how these rules can be used. Nowadays large automation companies have their own rules but many mid-size companies or IEC 61131-3 beginners are very interested in using PLCopen guidelines. Such guidelines will have a great impact in expanding IEC 61131-3 further in the world.

The rules will be very useful to train users and can be a good basis for universities to help them teach IEC 61131-3 programming more efficiently.

2. How-to use this document

The IEC 61131-3 standardizes programming languages and techniques, a major step forward in the PLC technology. These languages and techniques can contribute to the quality of the controller application. However IEC 61131-3 does not describe how a programmer can increase the quality of the application program. The IEC 61131-8 gives some guidance, but does not extend to the need of software quality. This technical paper fills in this gap.

This document is designed for PLC programmers who want to increase the product quality of their application program. It's not about process quality; this is the subject of other technical papers. This document consists of a set of rules that a programmer can use during coding and code reviews. Therefore, the technical paper contains no design rules. The set is certainly not complete, but significant enough to give the product quality a boost. The programmer or its organization can add or remove rules from the set.

The working group recommends the programmers take the following steps:

1. List the quality needs of the stakeholders and the maturity of the organization. Bring the quality in line with the organization.
2. Analyze the type of application.
3. Add other available rules to the set of rules from the technical paper.
4. Tailor the set of rules for the application based on the information from the previous steps, create a sub-set of rules.
5. Use the sub-set of rules when coding or reviewing the PLC program.

Ad 1.

The purpose of this technical paper is to increase the quality of software. This requires an organization with certain abilities or maturity, see for instance the Capability Maturity Model for software.

Ad 2.

The IEC 61131-3 standard specifies different languages, but if any are not used, then those language’s rules can be ignored.

Ad 3.

Besides this document, there are other sources that describe quality rules. Think of company standards or general standards. Larger companies have already gained experiences and these experiences recorded in their own standard. Research the available rules. There are also overall software quality standards, such as McCall, etc. These standards are generally not directly applicable to PLC programs, hence this technical paper.

Ad 4.

Once the programmer has collected all the rules, then a selection can be made to create a sub-set of rules that will be applicable to the application. Of course, the sub-set may be the full set of rules.

2.1. Methodology used to build this document

The rules listed here are the combination of common rules used in computer science, rules developed by companies that are part of the working group and rules created by the workgroup based on their experience of PLC development. The documents used here are:

- IEC61131-3
- Misra-C
- JSF++
- Codesys on-line help

See 2.4 References for more details.

A first list has been issued in a spreadsheet, each rule on a row. The row number is the current identifier used in this document. Then for each row a page on a working wiki was created on which the workgroup was working during one year and a half.

The content of this document is the concatenation of all those individual rules which have been discussed during the working sessions.

The rules are written with the 3rd edition of the IEC 61131-3 standard in mind. Many rules can be equally applied to other editions or even outside the IEC 61131-3 scope.

The working group wants to get feedbacks from users about this initiative. In the future, the content of this document is subject to evolution both by rewriting some rules, giving more meaningful examples or adding some new rules. So feel free to contact info <at> PLCopen.org for giving your feedback.

2.2. Document structure

The rules defined in this document have been classified in the following categories:

- Naming: how to name the PLC program elements. Some constraints for the naming and proposal for naming schemes;
- Comment rules: rules about the best way to comment your code so it is easy to read, understand and maintain;
- Coding practice: this section contains rules that are related to coding;
- Language: rules specific to one of the IEC61131 languages;
- Vendor Specific IEC61131 Extension: some rules specific to extensions of the IEC 61131 standard which are not available on all PLC vendors.

To enhance clarity, the chapters are initially organized in order of priorities, highest first. Each paragraph starts with 1. Later additions, irrespective of priorities, are done at the end.

Identifiers per rule contain a prefix for easy reference. The abbreviations are CP for Coding Practice; N for Naming, C for Comment, L for Language, and E for Extension. Each chapter starts with rule 1, like CP1. Future additions are done at the end.

An annex at the end of this document presents other ways to view the rules, based on priority.

2.3. *Rules description format*

Each rule in the document uses the same template. This section describes the different fields and gives the different possible values. A rules description looks like the following example:

Identifier: rule XX

Importance: high

Targeted languages: All

References:

- Misra-C_2004 rule 3.5

Description: When referencing user defined…

Guideline: Access to a member by offset …

Reasoning: Referencing by offset is difficult to read, understand …

Exceptions: None

Example:

Don't:

```
STRUCT example
  X : DINT;
…
```

Do:

```
instance.Z[1] := 'E';
```

Comments: none

- **Identifier**: Required - used to refer to a given rule easily. The current identifier has no semantic, it is just a number.
- **Importance**: Required - high, medium, low – this shows the interpretation of the group of the effect it has on the quality of the application software (high, medium or low)
- **Targeted language**: Required - some rules are only valid for some languages of the IEC61131. Either it is all languages, or the languages where the rule can be applied are listed.
- **References**: Required - when rules was found or inspired by a third party document, the reference is given in this field
- **Description**: Required - this is a longer description of the rule to get clear context
- **Guidelines**: Required - this is a help for the developer to explain how to work around the rule violation.
- **Reasoning**: Required - this gives reasons why this rule is a good idea.

- **Exceptions**: Optional - sometimes it is possible not to follow a rule for good reasons in a given case. This field lists the good reasons not to follow the rule.
- **Example**: Optional - the goal of the example is to help understand the reasoning of the rule and the guidelines to work around the violation with an explanatory example.
  - Don't – the don't part of the example uses red color to highlight bad coding practice
  - Do – the Do part of the example uses green color.
- **Comments**: Optional - this field is used to complete the rule description. It may be used to link with other rules or other subjects which may be out of the scope of this document.

Only fields Exceptions, Example and Comments are optional.

2.4. References

The following standards and committees are referenced from this site.

IEC 61131-3
NAME: IEC 6-1131-3 edition 2 and 3
TITLE: Programmable controllers - Part 3: Programming languages
LOCATION: International Electrotechnical Commission
WEBSITE: http://www.iec.ch/index.htm

IEC 61131-8
NAME: IEC 6-1131-8 edition 1 (and in future edition 3)
TITLE: Programmable controllers - Part 8: Guidelines for the application and implementation of programming languages
LOCATION: International Electrotechnical Commission
WEBSITE: http://www.iec.ch/index.htm

JSF++ coding standard
NAME: JSF++ coding standard
TITLE: JSF Air Vehicle - C++ Coding Standards (Revision C)
DOCUMENT: 2RDU00001 Rev C. December 2005
WEBSITE: http://www.jsf.mil/downloads/down_documentation.htm

MISRA-C
NAME: Motor Industry Software Reliability Association C
TITLE: Guidelines for the Use of the C Language in Vehicle Based Software
EDITION: 2005
WEBSITE: http://www.misra.org.uk/

Codesys On-line help
EDITION: 2015
Website: http://store.codesys.com/codesys-static-analysis.html

3. Naming Rules

3.1. Additional rules for Variables only

3.1.1. Avoid physical addresses

Identifier: Rule N1

Importance: High

Targeted languages: Ladder, Structured Text, Sequential Function Chart, Function Block Diagram

References:

- IEC 61131-3 6.5.5
- IEC 61131-8 3.11.2

Description: Use of hardcoded, system dependent physical addresses shall be avoided

Guideline: Using physical addresses in programs is forbidden - always define a Variable.

Reasoning: Using physical addresses makes code less portable. To use a program on a different manufacturer, or even a different instance on the same manufacturer requires it to be edited. It also makes programs less readable as without a symbol table for reference it can be difficult to know what each address is used for. Later system changes are also more error prone as some access to the physical addresses may be skipped or overlooked.

Exceptions: In some communication protocols a physical addressing is needed to assign variables to a physical location for the order in the communication structure.

Example, use case: none

Comments: It is recommended that the VAR_ACCESS method always be used when accessing variables in remote programmable controllers because it is then possible to use meaningful names for variables. There is always the likelihood that I/O physical addresses will be changed if the remote programmable controller program is modified. It may be convenient to fix VAR_ACCESS names for a number of different programmable controller programs.

3.1.2. Define type prefixes for Variables (if used)

Identifier: Rule N2

Importance: Low

Targeted languages: All

References:

- Wikipedia Hungarian Notation

Description: You can define any scheme used to prefix name variables. These prefixes can include variable attributes like:

- Data Type (Boolean, Numeric, String etc)
- Scope (global, local, parameter)
- Control (input, output)
- System variable
- Zone (e.g. infeed, part 1, part 2, outfeed etc)

Guideline: If you use prefixes, document your notation. One option is to use Hungarian Notation for distinguishing data types. Choose one or many notation(s) in the examples below and use it consistently.
Warning, the same prefix may be used in different notations. In this case, you should use compound prefixes.

Reasoning: Using the variable's name to also communicate other attributes can improve readability and help reduce programming mistakes. For example, knowing the size of the datatype or whether signed or unsigned can be important in the selection of instructions. Knowing a variable is Global can suggest external influence, or caution about undesirable side effects when writing. Writing to an "Input" signal by mistake will have no effect, and is not always shown as an error, for example if intermediate variables are mapped.

Also as programs get bigger and bigger, there is a tendency to group related variables (for example InfeedSpeed, InfeedAlarm, InfeedStatus). This grouping fails if groups are used inconsistently (for example Infeed_Speed, In_Feed_Alarm, ifStatus, inFault etc) so the common 'zone' names could be defined. This is especially true for large, multi-developer teams.

However, adding too many prefixes can create complexity and long variable names. The most useful (and most commonly used) is a prefix for the data type.

Exceptions: Attributes can be omitted if the Programming Support Environment clearly shows them by other means, for instance by hovering over it.

Example, use case:

Examples of a Hungarian Notation implementation coupled to the datatypes as defined in the 3rd edition of the IEC 61131-3 are:

Data Type	*Prefix*
BOOL	x
SINT	si
INT	i

DINT	di
LINT	li
USINT	usi
UINT	ui
UDINT	udi
ULINT	uli
REAL	r
LREAL	lr
TIME	tim
LTIME	ltim
DATE	dt
LDATE	ldt
TIME_OF_DAY / TOD	tod
LTIME_OF_DAY / TOD	ltod
DATE_AND_TIME / DT	dt
LDATE_AND_TIME / DT	ldt
STRING	str
WSTRING	wstr
CHAR	c
WCHAR	wc
BYTE	by
WORD	w
DWORD	dw
LWORD	lw

PLCopen Safety (as last character of the prefix)

All SAFE Datatypes prefix end with:	s

Example: SAFEBOOL prefix is: xs

Note: safety related tools will already differentiate between safe and non-safe datatypes, like with a color. So the usage of this character may not be necessary.

Cf. Table 11 of the 3rd edition of IEC 61131-3.

ENUM	e
NAMED	e
SUBRANGE	sb
ARRAY	a
STRUCT	st
Type STRUCT	ts
Reference	ref
FUNCTION BLOCK	fb
PROGRAM	prg
CLASS	cls

Note: ENUM and NAMED are proposed to use the same prefix.

For arrays, the datatypes does not have to be included.

As alternative, all user derived datatypes can be shown with one prefix: udt, no matter if it is an ENUM, NAMED or any other.

Examples of Scope prefix

Scope	*Prefix*
Global scope	g
Local scope	l
POU Parameter	p
Temporary Variable	tmp

Examples of Control prefix

Control	*Prefix*
Input (read only)	i
Output (read/write)	o

Comments: none

3.2. Tasks, Programs, Functions Blocks, Functions, Variables, UDTs and namespaces

3.2.1. Define the names to avoid

Identifier: Rule N3

Importance: High

Targeted languages: All

References:

- IEC 61131-3 6.1.3
- MISRA-C_2004 20.1

Description: You shall define the words to avoid in object names

Guideline:

- IEC data types and standard library object names must be avoided
- IEC Structured Text keywords must be avoided
- Reserved words (for your platform) must be avoided
- Avoid uncommon abbreviations (or specify them clearly)
- Avoid meaningless names like: Info, Data, Temp, Str, Buf.

Reasoning: In many cases the keywords and reserved words must be avoided as they create build / compiler errors. Even in cases where it is permitted by the compiler (e.g. due to scope) it should be avoided as it can cause confusion and poor maintainability. If you intend your code to be portable across different platforms then you should avoid the Reserved Words from ALL platforms. Truncating words and Three Letter Abbreviations can be misleading and confusing. If necessary, define a list of acceptable abbreviations (like Max, Min, Temp, IP, OP etc) and abbreviations common to your industry.

Exceptions:

Example, use case:

Comments: None

Keywords / reserved word list of IEC 61131-3 Ed.3 starting with a letter:

ABS
ABSTRACT
ACOS
ACTION
ADD
AND
ARRAY
ASIN
AT
ATAN
ATAN2
BOOL
BY
BYTE
CASE
CHAR
CLASS
CONCAT
CONFIGURATION
CONSTANT
CONTINUE
COS
CTD
CTU
CTUD
DATE
DATE_AND_TIME
DELETE
DINT
DIV
DO
DT
DWORD
ELSE
ELSIF
END_ACTION
END_CASE
END_CLASS
END_CONFIGURATION
END_FOR
END_FUNCTION
END_FUNCTION_BLOCK

END_IF
END_INTERFACE
END_METHOD
END_NAMESPACE
END_PROGRAM
END_REPEAT
END_RESOURCE
END_STEP
END_STRUCT
END_TRANSITION
END_TYPE
END_VAR
END_WHILE
EQ
EXIT
EXP
EXPT
EXTENDS
F_EDGE
F_TRIG
FALSE
FINAL
FIND
FOR
FROM
FUNCTION
FUNCTION_BLOCK
GE
GT
IF
IMPLEMENTS
INITIAL_STEP
INSERT
INT
INTERFACE
INTERNAL
INTERVAL
LD
LDATE
LDATE_AND_TIME
LDT
LE

LEFT
LEN
LIMIT
LINT
LN
LOG
LREAL
LT
LTIME
LTIME_OF_DAY
LTOD
LWORD
MAX
METHOD
MID
MIN
MOD
MOVE
MUL
MUX
NAMESPACE
NE
NON_RETAIN
NOT
NULL
OF
ON
OR
OVERLAP
OVERRIDE
PRIORITY
PRIVATE
PROGRAM
PROTECTED
PUBLIC
R_EDGE
R_TRIG
READ_ONLY
READ_WRITE
REAL
REF

REF_TO
REPEAT
REPLACE
RESOURCE
RETAIN
RETURN
RIGHT
ROL
ROR
RS
SEL
SHL
SHR
SIN
SINGLE
SINT
SQRT
SR
STEP
STRING
STRING#
STRUCT
SUB
SUPER
T
TAN
TASK
THEN
THIS
THIS
TIME
TIME_OF_DAY
TO
TOD
TOF
TON
TP
TRANSITION
TRUE
TRUNC
TYPE
UDINT

UINT
ULINT
UNTIL
USING
USINT
VAR
VAR_ACCESS
VAR_CONFIG
VAR_EXTERNAL
VAR_GLOBAL
VAR_IN_OUT
VAR_INPUT
VAR_OUTPUT
VAR_TEMP
WCHAR
WHILE
WITH
WORD
WSTRING
XOR

3.2.2. Define the use of case (capitals)

Identifier: Rule N4

Importance: High

Targeted languages: Ladder, Structured Text, Sequential Function Chart, Function Block Diagram

References:

- IEC 6-1131-3 6.1.2
- MISRA-C_2004 1.4
- GNU 5.4
- Java section 9
- Wikipedia CamelCase

Description: The use of capital letters in object names shall be clear and consistent across the project. Options are (not limited to):

- alllowercase
- underscore_separated (also known as lower_snake_case)
- lowerCamelCase
- UpperCamelCase (also known as PascalForm)
- ALLUPPERCASE or CAPITALIZED
- UPPER_SNAKE_CASE
- OTHER_style

Guideline:

Use the same capitalization for every object instance, even if the tool/compiler doesn't mandate it. The following guidelines are proposed:

- Use UPPER_SNAKE_CASE for CONSTANTS and user defined datatypes and keywords (like BOOL, FOR, TYPE and END_TYPE).
- Use UpperCamelCase for all other multi-word items

Reasoning: Case is not always significant to the tool or compiler but for readability purpose it should be consistent. Since in the IEC 61131-3 standard all identifiers are case insensitive it does not affect the portability.
CamelCase keeps long names readable while being shorter and quicker to type than using underscores. Use "UPPER_SNAKE_CASE" format to separate capitalized words with underscores like used for keywords. These guidelines are also similar to external standards likely to be used for third party code or libraries.

Exceptions: When using prefixes for Variables (see Rule 14) the UpperCamelCase changes to lowerCamelCase.

Example, use case:

Don't:

```
STARTMOTOR();    //Don't use all capitals as it confuses with constants.
startmotor();    //Don't use the same case as it is difficult to read
```

Do:

```
StartMotor();    //UpperCamelCase makes it quick to read
```

Don't:

Declaration of a structured data type (IEC 61131-3 6.4.4.6.1 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
```

Do:

```
TYPE
    ANALOG_SIGNAL_RANGE:
         (Bipolar10Volt,
          Unipolar10Volt);
    ANALOG_DATA: INT (-4095 ... 4095);
    ANALOG_CHANNEL_CONFIGURATION:
        STRUCT
            Range: ANALOG_SIGNAL_RANGE;
            MinScale: ANALOG_DATA;
            MaxScale: ANALOG_DATA;
        END_STRUCT;
END_TYPE
```

Do:

Naming example from PLCopen Motion Control
Over the years the suite of Motion Control specifications was defined within PLCopen. Consistency in the naming conventions was realized after 2010.

Function Blocks: Prefix MC_, and capitalize first letter of each word, and no hyphenation between words. Example: MC_MoveAbsolute

Enum elements: prefix to name is mc, and each word starts with a capital letter. So mcNameName2. Example mcPositive, mcBlendingLow.

Data types and Structures: Prefix MC_ (except AXIS_REF), capitalized and underscore between groups of words. Example: AXIS_REF; MC_BUFFER_MODE, MC_TP_REF, MC_INPUT_REF.

Inputs and outputs: No prefix. Each word start with a capital letter. No hyphen between words. Examples: Busy, CommandAborted, Master, BufferMode

Comments: None

3.2.3. Local names shall not shadow global names

Identifier: Rule N5

Importance: High

Targeted languages: All

References:

- IEC 61131-3 6.9.1
- MISRA-C_2004
- JSF++ 135

Description: The elements Tasks, Programs, Functions Blocks, Functions, Variables and UDTs shall not re-use the same global name, especially not within the same scope/namespace. A local element shall not shadow the global elements. (With shadowing, a local element has the same name as a global element; however the local scope cannot access the global element)

Guideline: Do not use identical names for any tasks programs, functions and function blocks, variables, user defined types, and name spaces. Using prefixes to name your programming elements may help (see rule N2 3.1.2 Define type prefixes for Variables (if used)).

Reasoning: Using the same name for different objects would make code difficult to read. Even if compilation is possible, errors in readability can introduce program mistakes.
Also such code cannot be guaranteed portability either. Even within different namespaces such confusion is best avoided.

Exceptions: none

Example:

Don't:

```
VAR_GLOBAL
    MyCalculation: REAL;     // Global Variable Declaration
END_VAR
FUNCTION_BLOCK MyFirstCalculation // Global Type Declaration
VAR
    MyCalculation: REAL;     // Local variable declaration
                              // any access in the FB accesses this variable
END_VAR
MyCalculation := 1.1;         // scope is within this FB
...
END_FUNCTION_BLOCK

FUNCTION_BLOCK MySimilarCalculation // 1) Declared in the global namespace.
  VAR_EXTERNAL
    MyCalculation : REAL;
  END_VAR
MyCalculation := 1.1;         // same line of code but different access:
                              // to the global variable defined at start
END_FUNCTION_BLOCK
```

Do: Use a different name for the global variable

```
VAR_GLOBAL
    GlobalCalculationResult: REAL;    // Global Variable Declaration
END_VAR
.....
```

Comments: none

3.2.4. Define an acceptable name length

Identifier: Rule N6

Importance: Medium

Targeted languages: Ladder, Structured Text, Sequential Function Chart, Function Block Diagram

References:

- IEC61131-3 6.1.2
- JSF++ AV Rule 46
- MISRA Rule 5.1
- GNU 5.4
- Java Section 9

Description: All named elements should have a mnemonics of an acceptable length. This relates to Tasks, POUs, Functions and Function Blocks and Variables.

Guideline:

- Min. length proposal: 8 characters, or 3 characters for local names.
- Max. length proposal: 25 characters
- On the average: keep the maximum to 15 characters
- Don't use abbreviations, unless required to shorten the name and the abbreviations are expected to be well known.
- Avoid names that are very similar or different only in case. For example, avoid using the names like variable and variables.

Reasoning: When all elements have a mnemonic in a given length range, maintenance is easier and it is easier to understand content. The minimum size is used to ensure that mnemonic are meaningful. Maximum length is used to ensure that a reader will be able to differentiate quickly two elements and some Programming Support Environment truncate the longest names.

Exceptions: It is not required for some elements of limited scope (e.g. Internal Variables):

- loop indexes: as their usage is limited to a given loop and it is common usage to use very short name
- structure members: the structure member name is always used with the instance name so it may be meaningful even when being short.

Example, use case

Don't:

```
FUNCTION Go : BOOL; ...
// too short

FUNCTION aaa: INT; ...
// meaningless name

FUNCTION ReadAndScaleTheTemperatureInput: REAL; ... //Too long
```

Do:

```
FUNCTION StartFeeding: BOOL;     // Name explains function

FUNCTION ReadTemperature: REAL; // Name explains function
```

Don't:

```
VAR

   Go : BOOL; // too short
   aaa: INT; // meaningless name

   MaximumTemperatureForTheThermocoupleInput: REAL; //Too long
END_VAR;
```

Do:

```
VAR
     i: INT;    //  FOR loop counters and indexes only
   MaxTCTemperature: REAL; // Only use clear abbreviations (TC -=
Thermocouple)
END_VAR;
```

Comments: None

3.2.5. Define naming rules for namespaces

Identifier: Rule N7

Importance: Medium

Targeted languages: All

References:

- IEC61131-3, 6.9

Description: Naming rule for namespace and declaration rule for new namespace should be clarified.

Guideline:

- Each namespace should be written in casing manner called "UpperCamelCase".
- On declaring POUs and data types, at least one namespace should be declared in the global namespace. No elements except standard types of IEC 61131-3 should be declared directly in the global namespace.
- The first level namespace just below global namespace should be started with the word which represents the organization/company who is responsible for the specification of the content of the namespaces below it.
- Regarding sub-namespaces below the namespace of company/organization, each company/organization should define its own semantic naming rule. Such sub-namespace names may consist of sub-committee name, functional category, product name, etc...

Reasoning:

- Standard functions/function blocks of IEC 61131-3 are in the global namespace whose name is empty. The global namespace is applied for each namespace without USING directive by default. Thus declaring elements directly in the global namespace increases the risk of conflict when a short name used in any namespace shadows the same name in another namespace.
- POU/data type library may be distributed across vendors/organizations. In order to avoid naming conflict, non-standard POUs/data types need to be declared in the namespace whose name is as unique as possible according to the name of vendors/organizations.

Exceptions: This rule can be only followed in Programming Support Environment that supports namespace feature. Applications or library which are to be used only within the specific organization where naming can be fully controlled.

Example, use case:

Don't: Types declared directly in the global namespace

```
FUNCTION_BLOCK MyCalculation // 1) Declared in the global namespace.
  ...
END_FUNCTION_BLOCK

NAMESPACE SomeCompany.XseriesCPU
    FUNCTION_BLOCK MyCalculation // 2) The same name is used also.
        ...
    END_FUNCTION_BLOCK
    PROGRAM MainProgram
        VAR MyCal : MyCalculation;
        // Usage of short type name above becomes ambiguous.
        // Need to write "SomeCompany.XseriesCPU.MyCalcuration" for 2).
        // No means to explicitly refer to 1).
    END_PROGRAM
END_NAMESPACE
```

Do: Nothing is directly declared in the global namespace. All namespaces start with organization name.

```
NAMESPACE SomeCompany.XseriesCPU
 USING PLCopen.Motion;
 USING PLCopen.Safety;

 PROGRAM MainProgram
     USING OMAC.OPW.PackML3;
     ...
 END_PROGRAM

 PROGRAM CommProgram
     USING IEC_61131_5;
     ...
 END_PROGRAM
END_NAMESPACE
```

Comments: None

3.2.6. Define the acceptable character set

Identifier: Rule N8

Importance: Medium

Targeted languages: All

References:

- IEC61131-3 6.1.2
- JSF++ AV Rule 9
- MISRA Rule 3.2

Description: The character set should be explicitly defined.

Guideline: The mnemonics should use a subset from the character set:

- Identifiers should not start with a digit character
- Only alphanumeric and underscore characters should be used in mnemonics. Accents should not be used.
- Identifiers should use characters from ASCII 7 bits.

Reasoning: Using special characters may generate some problems of portability to future Programming Support Environment version and/or to other platforms. It also reduces the readability in an international context. Finally, the restriction to only alphanumerical characters ensures a better distinction between variable names.

Exceptions: Comments can use character sets from the native languages. Also if the scope of usage of the program is known (like China) then the usage of the national character set can be applied to identifiers.

Examples:

Don't:

```
départ := true;  // .... Lines of code
depart := false; //Using the special character 'é', leads to a
confusion between mnemonics.
```

Do:

```
Depart := True;     // No accent in the variable name
        //  Autorise le départ du wagonnet   // Accent in the
comment-
```

Comments: None

3.2.7. Different element types should not bear the same name

Identifier: Rule N9

Importance: Medium

Targeted languages: All

References:

- IEC 6-1131-3 6.9.1
- MISRA-C_2004
- JSF++ 135

Description: The elements types Tasks, Programs, Functions Blocks, Functions, Variables and User Defined Types should not share the same name in the same scope.

Guideline: Do not use identical names for any tasks programs, functions and function blocks, variables, UDTs and name spaces.

Reasoning: Using the same name for different object types would make code difficult to read. Even if compilation is possible, errors in readability can introduce program mistakes.

Exceptions: none

Example:

Don't:

```
VAR_GLOBAL
    MyCalculation: REAL;     // Global Variable Declaration
END_VAR

FUNCTION_BLOCK MyCalculation // Global Type Declaration with same name
VAR_IN_OUT
    MyCalculation : REAL;     // Input variable declaration
                              // any access in the FB accesses this variable
END_VAR

MyCalculation := 1.1;         // scope is also external to this FB
...
END_FUNCTION_BLOCK

PROGRAM MyCalculation        // Declared in the global namespace.
  VAR
    MyCalculation : MyCalculation;
  END_VAR
...                          // Note: it is not possible to access the global variable
                             // from here due to shadowing
END_PROGRAM
```

Comments: none

3.2.8. Define name prefixes for user defined types

Identifier: Rule N10

Importance: Low

Targeted languages: All

References: none

Description: You can define any scheme used to prefix names for user defined types. These prefixes can include attributes like:

- Type (Function, Function Block, Structure, Array, etc.)
- Domain (e.g. MC for Motion Control, SF for Safety, etc)

Guideline: If you use prefixes, document your notation. Choose one or many notation(s) in the examples below and use it consistently.
The proposal is to use capitals for the prefixes for user defined type names.
Warning, the same prefix may be used in different notations. In this case, you should use compound prefixes.

Reasoning: Using the type name to also communicate other attributes can improve readability and help reduce programming mistakes.

Exceptions: Attributes can be omitted if the Programming Support Environment clearly shows them by other means, for instance by hovering over it.

Example, use case:

Cf. Table 11 of the 3rd edition of IEC 61131-3.

ENUM	E
NAMED	E
SUBRANGE	SB
ARRAY	A
STRUCT	ST
Reference	REF
FUNCTION	FU
FUNCTION BLOCK	FB
PROGRAM	PRG
CLASS	CLS
INTERFACE	I

Note: ENUM and NAMED are proposed to use the same prefix.

For arrays, the datatypes does not have to be included.

As alternative, all user derived datatypes can be shown with one prefix: UDT, no matter if it is an ENUM, NAMED or any other.

Comments: none

4. Comment Rules

This chapter is related to commenting the program and the application so that it is easier to read, understand and modify. The following guidelines are good practices about writing good comments.

4.1. Comments shall describe the intention of the code

Identifier: Rule C1

Importance: High

Targeted languages: All

References:

- JSF++ 130

Description: All code shall be explained by a comment, although that doesn't necessarily mean 1 line of comment per code. Comments shall describe the intention of the code.

Guideline: Review code to ensure it is explained by a comment, either:

- At the end of the code line
- Separate comment, immediately prior to the code
- Block comment and the start of a sub section of code (e.g. whole IF THEN ELSE block)
- Function/Function Block comment, if the code is small enough to be wholly explained by the comment
- Program comment, if the code is small enough to be wholly explained by the comment

Reasoning: Good programming practice dictates that well written code with well-chosen variable names should be self-documenting. Commenting every rung or line is therefore not always necessary, unless it is complex or potentially ambiguous.
You can divide comments into five categories (McConnell 1993, p. 463):

1. *Repeat of the code,*
2. *Explanation of the code,*
3. *Marker in the code,*
4. *Summary of the code*
5. *Description of the code's intent.*

What are good comments? Comments that describe the intentions of the programmer. Where do you draw the line? The boundary is at the transition from Intention (what, or function) to Extension (how, program code). The Extension (Program Code) is coded in a formal language, such as IL, ST, LD, FBD or SFC and the Intention is written in an informal language, such as English or Chinese. One of the qualities of an informal language is that it is likely to inconsistent interpretations. Furthermore programmers are not always informal language experts. So one merely describes the intentions in the shape of block comments (Marker or Summary) or routine commentary. In exceptional cases, a programmer may explain a program part.

For example it may be clear to comment several statements in each branch of a conditional IF-THEN statement with a single comment. If the branches are small, and the logic clear enough then it may be clear with a single comment prior to the whole conditional block. You can even argue that code in a small Program/Function/Function Block can be explained by a single comment. All the code should be explained by a comment at some scope. The comments should describe the intention of the code.

However there should be no code that performs tasks other than those that the comments state.

Exceptions: Any use of 'magic numbers' or physical addresses should always be commented

Example:

Don't**:**

Code blocks with no comments:

```
IF IsValid(TCInput) THEN

  Temperature := TCInput *
                 TCScale + TCOffset;

ELSE

  TCBadQuality := TRUE;

END_IF;
```

Comments that add no value over the code:

```
IF IsValid(TCInput) THEN

  // TCInput is valid

  Temperature := TCInput *
                  TCScale + TCOffset;

ELSE
  // TCInput is not valid
  CBadQuality := TRUE;

END_IF;
```

Do**:**

Branch comments should add value:

```
IF IsValid(TCInput) THEN
  // Scaling of the
  //Temperature

    Temperature := TCInput *
                TCScale + TCOffset;

ELSE
  // Some error reading the
  // temperature value

  TCBadQuality := TRUE;

END_IF;
```

Whole block explained well by a single comment:

```
// Check for a new maximum
// temperature reading

MaxReading := nReadings[0];

FOR index:=1 TO 10 DO

  IF nReadings[Index] > MaxReading
  THEN

    MaxReading := nReadings[Index];
  END_IF;

END_FOR;
```

Comments: None

4.2. All elements shall be commented

Identifier: Rule C2

Importance: High

Targeted languages: Ladder, Structured Text, Sequential Function Chart, Function Block Diagram

References:

- JSF++ 132, 134

Description: All POU (Program, Function Block and Function), Task, User-Defined Types, Resource and Variable elements shall be explained by a comment.

Guideline: Include clear comments for all elements, including Tasks, Programs, Function Blocks, Functions, User Defined Types, Variables and Resources.

Reasoning: Clear comments communicate the intention of each element, and aid readability and improve understanding of the program.

Exceptions: None

Example: None

Comments: None

4.3. Avoid nested comments

Identifier: Rule C3

Importance: Low

Targeted languages: All

References:

- MISRA-C_2004 section 2.3

Description: Nesting of multiline comments must be avoided

Guideline: In the released versions of programs there should be no nested comments, even if it's possible during development and debug phases.

Reasoning: Editing of nested comments is a common source of accidently mismatching comment delimiters and unintentionally commenting out code. This editing error can be missed by the compiler if valid code can still be compiled.

As a general principle Code in comment should not be used for version control or configuration management purposes; proper tools should be used for that.

Exceptions: None

Example:

Don't:

```
(* Any comment with the end marker has been forgotten

Critical_section_that_must_be_executed_which_is_found_in_the
comment();

(* <- The start marker comment is commented because the end marker
of the previous comment has been forgotten *)
```

Comments: None

4.4. Comments may not include code

Identifier: Rule C4

Importance: Low

Targeted languages: Structured Text

References:

- MISRA-c_2004 2.4
- JSF++ 127

Description: Valid code statements may not be commented out and left in programs

Guideline: In the released versions of programs there should be no code in comments, even if it's possible during development and debug phases.

Reasoning: Editing multi-line Comments is a common source of accidently commenting out code. This editing error can be missed by the compiler if valid code can still be compiled. Equally, code that is not required and is commented out can easily be accidently reinstated.

While commenting out a few lines of code is commonplace in the debugging and commissioning phase it should not be used for released software: if the code is not used it should be deleted, or if it needs to be disabled another construct should be used e.g. removing Program from a Task or a "IF 0 THEN" construct.

As a general principle Code in comments should not be used for version control or configuration management purposes; proper tools should be used for that.

Exceptions: None

Example: None

Comments: None

4.5. Use single line comments

Identifier: Rule C5

Importance: Low

Targeted languages: Structured Text

References:

- IEC 6-1131-3 6.1.5
- MISRA-c_2004 2.2

Description: Define the type of Comment definition character to be used

Guideline: Use single line comments, i.e. //

Reasoning: Although multiple commenting styles are available, for consistency and portability you should define which style to use. Multi-line comments can be problematic and be the cause of accidental commenting in or out of code depending on the way the workbench manages the nested comment. Also not all Programming Support Environment support IEC 3rd Edition. Commenting out some code containing only '//' style comments avoids the errors of nested comments.

Exceptions: During testing and debugging phases, multi-line comments can be used to exclude portion of codes. Single line comments with // should not be used for this purpose.

Example:

Don't:

```
(* Multiline comments can be dangerous because matching close
comment can be deleted by accident.

b:=a;         // commented out code can be activated by accident

// or active code after this block accidentally commented out.

*)

(* (* NESTED COMMENTS *) *)
```

Do:

```
<statement>;   // End of line comment


// Block Comment
// <the following block...>
// <...>
<statement>;
```

Comments: For existing code with (*, one can use the /* to exclude portions of code to differentiate, if the system allows this.

4.6. Define comments language

Identifier: Rule C6

Importance: Low

Targeted languages: Ladder, Structured Text, Sequential Function Chart, Function Block Diagram

References: None

Description: You may define which spoken language your comments may be written in. This applies to code comments in both ST and Ladder, plus any other comments like Variable comments, Project comments, Task comments, source control comments etc.

Guideline: Use English only for all comments

Reasoning: Understanding and maintaining the comments improves readability, reduces mistakes and therefore development costs. For all developers, current and future, to understand the comments, a common language should be chosen. This includes teams with members of international members and also multi-site teams. Any common language is sufficient, but English is the most widely spoken language internationally.

It may be desirable to have the comments duplicated in more than 1 language, especially where the languages are completely different alphabets e.g. English and Chinese. This can be done manually although some tools already have facilities to do this. You should still consider which language is mandatory, and ensure your code review process checks for untranslated comments.

Exceptions: None

Example: None

Comments: None

5. Coding Practice

5.1. Access to a member shall be by name

Identifier: Rule CP1

Importance: High

Targeted languages: All

References:

- Misra-C_2004 rule 3.5

Description: When referencing user defined types like structures in the code, the references shall be done using the member name and not using an offset between the beginning of the structure and the position of the member in memory.

Guideline: Access to a member by offset should be avoided.

Reasoning: Referencing by offset is difficult to read, understand and maintain. If for some reason one another member is inserted before, then all offsets of the following members have to be recalculated. Moreover it relies on memory implementation (like big endian vs. little endian) and especially, the offset of a given structure depends on the PLC type: the alignment rules are not always the same.

Exceptions: None

Example:

Don't:

```
STRUCT EXAMPLE_STRUCT
  X : DINT
  Y : BOOL;
  Z : STRING[40];
END_STRUCT;

VAR
instance : EXAMPLE_STRUCT AT %MW500;
END_VAR
// Write the first character of Z:
%MW504 := 'E';
```

Do:

```
instance.Z[1] := 'E';
```

Comments: none

5.2. All code shall be used in the application

Identifier: Rule CP2

Importance: High

Targeted languages: All

References:

- Codesys SA0001, SA0031
- Itris Automation Square I3
- Misra C 14.1
- JSF++ Rule 186

Description: Dead code is not allowed

Guideline: All parts of the application code should be reachable under certain conditions so that they are not dead code.

Reasoning: Dead code affects the readability and maintainability of the program and can be a symptom of a functional problem. Different kinds of dead code exists:

- Unreferenced functions :
  - re-use of an already existing application to create a new application slightly different
  - developer mistake forgetting to call part of the application during development
  - part of the code used in a specific environment : testing, simulation of missing hardware
- Code not reachable unconditionally - techniques used by developer to bypass part of the code :
  - due to unconditional go to followed by code without called label
  - condition always false or true

Exceptions: A dead code with a nice comment delimiting and explaining correctly the reason why the code was bypassed may be used. In such case, the code section should be small enough so that a developer sees as evidence that it is dead code.

Example:

Do:

```
...
//
// The following section handles the optional device FOO.
// It is bypassed for application THIS_PARTICULAR_APPLICATION
because
// this device is not loaded for this application

IF FALSE THEN

  // unreachable code
  ...

END_IF;
```

```
// End of bypassed section for THIS_PARTICULAR_APPLICATION
```

Comments: none

5.3. All variables shall be initialized before being used

Identifier: Rule CP3

Importance: High

Targeted languages: All

References:

- IEC 61131-3 3rd Ed. section 6.5.1 and 6.5.6
- IEC 6-1131-8 Section 3.1.1
- JSF++ - rule 142

Description: A variable shall be initialized before being read by another part of the code, whatever the cycle: cold start, warm start, first cycle, normal cycle.

Guideline:

- According to IEC 61131-3 all variables have default initial values. If the programming system does not support this feature (for example for ARRAYs), the user has to initialize the variables explicitly in the user program.
- Using an initialization at the variable declaration is good but the behavior may be different depending if it is a cold start or a warm one
- Using code to explicitly initialize variable is a maintainable and readable way
- When creating User Defined Types specify a good Default Initial Value. Any variables created of this type will not need to be explicitly defined.

Reasoning: Reading uninitialized variables in your code gives undetermined behavior to the code.

Exceptions:

- If the PLC initializes explicitly variables to 0, and if this variable should be initialized to 0, it is not required to explicitly initialize the variables to zero. However this can affect portability of the code.
- Variables that have RETAIN attribute will automatically have their values initialized to their retained upon power reset.
- Variables that are linked to physical inputs do not need to be initialized.

Example:

Do:

```
PROGRAM Initialization
VAR

    NumOfRetries: INT:= 3;          // direct initialisation
    Enable: BOOL:= TRUE;          // direct initialisation
    ConfTimerPreselection : TIME := T#5s;

    StateLastPosition : INT;
    SpeedAverage : REAL;
END_VAR;

    StateLastPosition := 50;
    SpeedAverage := 0.0;
```

```
...
END_PROGRAM
```

IEC 61131-3 also allows the user to specify default initial values for user-defined types. For instance, consider a type declared by:

```
TYPE TempLimit : REAL:= 250.0; END_TYPE
```

Any declared variable of this new type TempLimit is initialized with the default value of 250.0 instead of 0.0 as would be the normal case for all REAL data. Thus, in the following declaration, the variable BoilerMaxTemperature is initialized to 250.0, while the variable PipeMaxTemperature is initialized to 0.0. If the value of zero is not a reasonable maximum temperature for the pipeline, its correct value has to be set before the first usage of the variable. Forgetting this will cause problems. In the present example, the maximum temperature for the boiler is initialized with a proper default initial value. There is no need for a set-up before the first usage, which greatly simplifies a programmable controller program and increases software reliability.

```
VAR_GLOBAL
    BoilerMaxTemperature: TempLimit;
    PipeMaxTemperature: REAL;
END_VAR
```

Comments: More examples of defining default initial values for user-defined types:

- Initialization of enumerated data types, e.g.:

```
TYPE ANALOG_SIGNAL_RANGE :
    (BIPOLAR_10V, (* -10 to +10 VDC *)
      UNIPOLAR_10V, (* 0 to +10 VDC *)
      UNIPOLAR_1_5V, (* + 1 to + 5 VDC *)
      UNIPOLAR_0_5V, (* 0 to + 5 VDC *)
      UNIPOLAR_4_20_MA, (* + 4 to +20 mADC *)
      UNIPOLAR_0_20_MA (* 0 to +20 mADC *)
    ) := UNIPOLAR_1_5V ;
END_TYPE
```

- Initialization of subrange data types, e.g.:

```
TYPE ANALOG_DATA : INT (-4095..4095) := 0 ; END_TYPE
```

- Initialization of array data types, e.g.:

```
TYPE ANALOG_16_INPUT_DATA :
    ARRAY [1..16] OF ANALOG_DATA := [8(-4095), 8(4095)] ;
END_TYPE
```

- Initialization of structured data type elements, e.g.:

```
TYPE ANALOG_CHANNEL_CONFIGURATION :
    STRUCT
        RANGE : ANALOG_SIGNAL_RANGE ;
        MIN_SCALE : ANALOG_DATA := -4095 ;
        MAX_SCALE : ANALOG_DATA := 4095 ;
    END_STRUCT ;
END_TYPE
```

- Initialization of derived structured data types, e.g.:

```
TYPE ANALOG_CHANNEL_CONFIG :
    ANALOG_CHANNEL_CONFIGURATION
        := (MIN_SCALE := 0, MAX_SCALE := 4000);
END_TYPE
```

IEC 61131-3 provides following three layered initialization feature.
User should explicitly specify initial values by these embedded initialization feature, or user's application program.

1) Default initial value for user-defined type

User can specify own default initial values for user-defined types.

If no initial value is specified for the user-defined type, default initial value is succeeded from its base elementary data type. Default initial value for elementary data types have been specified in IEC 61131-3 (See Table 10).

As for examples, see above.

2) Initial value assignment to internal variables (instance) in POU type declaration

If no initial value is specified to the variable, default initial value of the data type is applied.

- Initialization of elementary data type variable
- Initialization of user-defined data type variable
- Initialization of array type variable
- Initialization of structured data type variable
- Initialization of FUNCTION_BLOCK type variable (Only input/output and PUBLIC variables can be initialized)
- Initialization of CLASS type variable (Only PUBLIC variables can be initialized)

As for example, please see feature tables of IEC 61131-3 Ed.3.

All of Table 14, No.2 of Table 41, No.2 of 49

3) Instance specific initial value or location assignment with VAR_CONFIG construct in CONFIGURATION

(See No.11a, 11b of Table 62 of IEC 61131-3)

This initial value declaration overwrites 1) and 2) above.

```
CONFIGURATION Cell_1
    VAR_GLOBAL
        Gvar1 : INT:= 5;
        Gvar2 : INT:= 0;
    END_VAR

    RESOURCE Station1 ON ProcessorType201
        TASK FastPeriodic ( INTERVAL := t#1ms, PRIORITY:=2)
        TASK SlowPeriodic ( INTERVAL := t#15ms, PRIORITY:=4)
        PROGRAM ProgInst1 WITH FastPeriodic
                : MyProgramA (Input1 := GVar1, Output1 => Gvar2)
        PROGRAM ProgInst2 WITH SlowPeriodic
                : MyProgramA (Input1 := GVar2, Output1 => Gvar1)
    END_RESOURCE

    RESOURCE Station2 ON ProcessorType201
        TASK FastPeriodic ( INTERVAL := t#1ms, PRIORITY:=2)
        PROGRAM ProgInst1 WITH FastPeriodic : MyProgramB
    END_RESOURCE

    // Instance specific initialization or location assignment
    VAR_CONFIG
          Station1.ProgInst1.COUNT : INT:= 1;
          Station1.ProgInst2.TIME1 : TON:= (PT:= T#2.5s);
                                   // initialization of function block instance
          Station2.ProgInst1.FbInst1.FbInst2.FbInst3.Count : INT:= 100;
          Station2.ProgInst1.FbInst1.C2 AT %QB25 : BYTE;// I/O assignment
    END_VAR
END_CONFIGURATION
```

5.4. Direct addressing should not overlap

Identifier: Rule CP4

Importance: High

Targeted languages: All

References:

- Itris Coding Standard - S8
- Codesys SA0028
- Misra-C 18.2, 18.3

Description: When assigning a memory location to an object, developer shall take care that the memory is not already assigned for another usage.

Guideline:

Overlap of variables addresses should be avoided

Reasoning: As most of PLC program are designed with a data flow from inputs to outputs, re-using the same memory location for different purpose may not be detected during program testing and commissioning. May be the first variable is written and the read during the first program's part and the second variable at the same location is also written and the read during another part of the program. Unfortunately some sparse condition may occur where one of the two variables won't be written and then the read will access the wrong variable value. It will be a problem very difficult to cope with.

Exceptions: STRUCT OVERLAP types are designed to overlap. On some platforms bit addressable and value addressable types are designed to overlap.

Example:

Don't: in the following example, two variables overlap at %MW451. Most of the time, everything goes well because of program data flow, the memory cell is firstly used by temp, and then by level. By when VeryRareCondition is true, level is not written and then the level value used in the test is in fact part of temp variable.

```
// Wrong example
Temperature : INT AT %MW451;
... else where in the database definition
Level: REAL AT %MW450;

Temperature := %IW56;
IF Temperature > 56 THEN
   StartFans;
END_IF;

// The second part of the code related to the other variable
IF NOT VeryRarecondition THEN
   Level := %IW34;
END_IF;
IF Level < LevelThreshold THEN
   // In the case of VeryRareCondition, the CriticalFunction will be
   // called with a wrong level value as it is a temperature.
   DoCriticalFunction(Level);
END_IF;

// Right example
Temperature : INT AT %MW444;
... else where in the database definition
Level: REAL AT %MW450;
    // the direct addresses do not overlap
Temperature := %IW56;
IF Temperature > 56 THEN
   StartFans;
END_IF;
```

Comments: None

5.5. Applications shall be well designed

Identifier: Rule CP5

Importance: High

Targeted languages: Ladder, Structured Text, Sequential Function Chart, Function Block Diagram

References:

- IEC 61131-8 Section 3.3

Description: All applications shall be well designed, ideally before development phase is started. The design principles shall include object oriented principles like modularization and encapsulation.

Guideline:

- Applications must be well designed
- Make use of Arrays (where supported) to aggregate related variables of the same data type
- Make use of Structures (where supported) to aggregate related variables of different data types
- Make use of Classes where supported or Functions and Function Blocks to reduce complexity of large areas into smaller parts
- Make use of Classes where supported, or Functions and Function Blocks to reuse common code
- Make use of PRIVATE variables in Classes where supported, or Function Block internal variables to encapsulate data
- Design separate programs to use the best language for the job: Ladder, Structured Text, Sequential Function Chart or Function Block Diagram (where supported)

Reasoning: Well-designed applications reduce the total cost of development by making a project objective clearly understandable, and communicating the plan to avoid mistakes, redevelopment, and bugs found at the testing and maintenance stages. Modularizing the design allows each small part to be considered, designed, and even successfully programmed in isolation. The qualities of Classes and Function Blocks are related to object oriented programming (OOP). The function block type is similar to a class, which defines the data structure and computational method within the body of the function block. Individual objects are represented by the private data areas of the individual function block instances. This data can only be modified from outside the function block body in a controlled manner. This enforces the software engineering principles of encapsulation and information hiding, a key element of OOP.

Exceptions: none

Example: none

Comments: none

5.6. *Avoid external variables in functions, function blocks and classes*

Identifier: Rule CP6

Importance: High

Targeted languages: All

References:

- JSF++ AV Rule 207
- IEC61131-3 - 2.5.1

Description: The use of external variables referencing global variables in functions, function blocks and classes shall be avoided. This means do not use VAR_EXTERNAL inside the definition of a function or function block.

Guideline:
Functions, Function Blocks and Classes should not use external variables.

A good alternative to using external references to global variables can be to extend the parameter list, and pass the variables needed for access.

Reasoning:
Good design principles allow Function, Function Blocks and Class POUs to be easily reused. The inner workings are sometimes unknown to the user, and even unexposed. Using any external reference detracts from or prevents reusing the POU. For example if a function accesses 10 external references, then when you copy the function to a new project you must also copy the accompanying global variable definitions including the code affecting these variables.

Function Blocks and Classes also have 'instances' with their own data. Directly using external references can make it impossible to have multiple instances.

Encapsulation of data can minimize integration testing and remove functional testing for pre-tested POUs as the known behavior cannot really change. However using data in external references means the functionality now depends on that external data so requires full retesting to ensure no issues have been introduced.

Also, using external references increases the chance of different POUs performing multiple writes to the same variable. This is related to rule CP26 **"A global variable may be written only by one POU**" and rule CP15 **"Read a variable written by another task only once per cycle"** so avoiding externals can also avoid these accidental uses, which can be very difficult to find and debug.

In addition, the code gets more complex, and the need for using and external variable can be a signal that the program is not coherent and refactoring is necessary.

Finally, such a practice can introduce side-effects, like a function returning different results with the same inputs, which makes it hard to verify correct operation. There should be a very good and well documented reason for any such usage of external data.

Exceptions: Accessing system-defined variables or VAR_GLOBAL CONSTANT may require the use of external references

Example:

Don't:

```
FUNCTION CommandMotor
VAR_INPUT
  Enable : BOOL;
  Default: BOOL;
END_VAR;
VAR_OUTPUT
  Command : BOOL;
END_VAR;
VAR_EXTERNAL
   ModeAuto : BOOL;
END_VAR;

   Command := Enable AND NOT Default AND ModeAuto;
END_FUNCTION;
```

Do:

```
FUNCTION CommandMotor
VAR_INPUT
  Enable : BOOL;
  Default: BOOL;
   ModeAuto : BOOL; //Here the variable is passed as a parameter
END_VAR;
VAR_OUTPUT
  Command : BOOL;
END_VAR;
   Command := Enable AND NOT Default AND ModeAuto;
END_FUNCTION;
```

Comments: This rule also applies to PROGRAM POUs when the programming systems support this. PROGRAM is a reusable POU type which can have input/output/in/out parameters. See Table 47 No.2, Table 62 No.89 of IEC 61131-3.

5.7. Error information shall be tested

Identifier: Rule CP7

Importance: High

Targeted languages: All

References:

- Misra C 16.10
- JSF++ Rule 115

Description: When available, error information returned by a function shall be tested and error condition shall be properly handled.

Guideline: After a call to a function returning error information, the returned error information should be tested and eventually the behavior of the process should be changed in case of error.

Reasoning: an undetected error can have bad consequences on the continuation of the process. A detected and handled error shows as well that the developer has understood the consequences of errors in the called routine. Without such error handling, either developer forgot an error case or consequences of this error have no real impact.

Exceptions: None

Example:

Don't:

```
          +----------+
          | Instance |
    Cond -|En    Eno |-
    Eff1 -|P1    Out |- Result
    Eff2 -|P2 xError |-
         -|P2 iError |-
          |          |
          +----------+

          +---------+
          |   ADD   |
|---------| En  Eno |-
  Result -| i1  Out |- CriticalThreshold
   10000 -| i2      |
          +---------+
```

In this case the result is used directly without checking the error information and then the CriticalThreshold is changed even if there was an error raised by Instance call.

Do:

```
      +----------+
      | Instance |
Cond -|En     Eno |-
Eff1 -|P1     Out |- Result
Eff2 -|P2 xError |- xError
     -|P2 iError |- iError
      |          |
      +----------+
```

```
          +---------+
| xError  |   ADD   |
|---| |---| En  Eno |-
| Result -| i1  Out |- CriticalThreshold
   10000 -| i2      |
          +---------+
```

In this case, the error information generated by the Instance call is effectively used and then the CriticalThreshold is not changed.

Comments: None.

5.8. Floating point comparison shall not be equality or inequality

Identifier: Rule CP8

Importance: High

Targeted languages: All

References:

- Codesys SA0054
- Misra C 13.3
- JSF++ Rule 202

Description: Using equality or inequality operators to detect threshold with floating point variable is prohibited.

Guideline: comparison between floating point variables must use only the following operators: strict less than (<), less than or equal (<=), strict greater than (>), greater than or equal (>=).

Reasoning: The equality operator requires a strict equality between operands. When using floating point number, this equality is almost never and the inequality is almost always.

Exceptions: Compare to 0.0

Example:

Many numbers cannot be represented exactly in floating point notation: - number 0.1 is represented by a binary value that corresponds to decimal 0.100000001490116119384765625 in 24bits single precision.

Moreover, when using floating point numbers, mathematical operations might produce roundiing errors (for example, small differences between large numbers).

Don't:

```
IF TEMP - OLD_Temp = 0.1 THEN
  // execution of this branch is sensitive to rounding errors
   ....
END_IF;
```

Do:

```
IF TEMP - OLD_TEMP < 0.1 THEN
   // executed code
END_IF;
```

or

```
IF REAL_TO_INT((TEMP - OLD_TEMP) * 100) = 10 THEN
   // executed code
END_IF;
```

Comments: None

5.9. Time and physical measures comparison shall not be equality or inequality

Identifier: Rule CP28

Importance: High

Targeted languages: All

References: None

Description: Using equality or inequality operators to detect threshold with time information or physical measure even in Integer format is prohibited.

Guideline: Comparison between time information or physical measures must use only the following operators: strict less than (<), less than or equal (<=), strict greater than (>), greater than or equal (>=).

Reasoning: The equality operator requires a strict equality between operands. When using time information or physical measure, this equality may not be met and the inequality is almost always met.

Exceptions: None

Example:

Don't:

```
IF Distance – InitialPosition = 12 THEN
  // execution of this branch is sensitive to missing measured
  // values
   ....
END_IF;

IF T5.Et = T#10s then
  // Stop the process
END_IF;
```

Do:

```
IF Distance – InitialPosition - 12 < ERROR_MARGIN THEN
   // executed code
END_IF;
```

or

```
IF T5.Et - T#10s < T#100MS THEN
   // executed code
END_IF;
```

Comments: None

5.10. Limit the complexity of POU code

Identifier: Rule CP9

Importance: High

Targeted languages: All

References:

Description: There are different possibilities to measure the complexity of code. It is still very common, just to count the lines of code per POU, or the number of statements per POU. Other metrics like McCabe-metric, Elshof-metric, and Prater-metric can be found in the literature. Every metric has advantages and disadvantages; therefore we do not propose one single metric or a set of metrics.

Some of the metrics are only applicable for textual languages (McCabe, Prater), whereas other metrics are applicable for any kind of code (Elshof).

It is recommended to measure the complexity of code and to set upper limits for the complexity.

Guideline: If the code of a POU exceeds some complexity level, the code should be split up into several POUs.

Reasoning: Complex code is difficult to maintain and a source of errors.

Exceptions: None

Example:

Don't:

The following Function block CHARCURVE has

- Number of Statements 18
- McCabe complexity of 12
- Prater complexity of 3,89
- Halstead complexity of 44,9
- Elshof complexity of 0,14 (the lower the number, the more complex is the function).

```
FUNCTION_BLOCK CHARCURVE
VAR_INPUT
    IN:INT;
    N:BYTE;
END_VAR
VAR_IN_OUT
    P:ARRAY[0..10] OF POINT;
END_VAR
VAR_OUTPUT
    OUT:INT;
    ERR: BYTE;
END_VAR
VAR
```

```
    I:INT;
END_VAR
IF N > 1 AND N < 12 THEN
    ERR:=0;
    IF IN<P[0].X THEN
        ERR:=2;
        OUT:=DINT_TO_INT(P[0].Y);
    ELSIF IN>P[N-1].X THEN
        ERR:=2;
        OUT:=DINT_TO_INT(P[N-1].Y);
    ELSE
        FOR I:=1 TO N-1 DO
            IF P[I-1].X>=P[I].X THEN
                ERR:=1;
                EXIT;
            END_IF;
            IF IN<=P[I].X THEN
                EXIT;
            END_IF
        END_FOR;
        IF ERR=0 THEN
            OUT:=DINT_TO_INT(P[I].Y-(P[I].X-IN)*(P[I].Y-P[I-
1].Y)/(P[I].X-P[I-1].X));
        ELSE
            OUT:=0;
        END_IF;
    END_IF
ELSE
    ERR:=4;
END_IF;
```

Do:

The changed function block with one (private) help method CalculateOut is far less complex:

- Number of Statements 12
- McCabe complexity of 8
- Prater complexity of 2,25
- Halstead complexity of 19,9
- Elshof complexity of 0,32

```
OUT:=0;
ERR:=0;
iIndexFound := -1;
FOR I := 1 TO N-1 DO
    IF iIndexFound < 0 AND ERR = 0 THEN
        IF P[I-1].X >= P[I].X THEN
```

```
            ERR := 1;
        ELSIF IN <= P[I].X THEN
            iIndexFound := i;
        END_IF
    END_IF
END_FOR;
IF ERR = 0 THEN
    OUT := CalculateOut (P[iIndexFound], P[iIndexFound -1]);
ELSE
    OUT := 0;
END_IF;
```

Comments:

Literature references:

- Thomas J. McCabe: "A Complexity Measure" in: IEEE Transactions on Software Engineering, Vol 2, 1976.
- Prater, R. E.: "An axiomatic theory of software complexity metrics", in Computer Journal, Vol. 27, 1984
- Halstead, M.: "Elements of Software Science", Elsevier North-Holland, Amsterdam, 1977
- Elshof, J.: "An Analysis of Commercial PL/I Programs", IEEE Transactions on Software Engineering, Vol. 2, 1976

5.11. *Avoid multiple writes from multiple tasks*

Identifier: Rule CP10

Importance: High

Targeted languages: All

References:

- Itris Automation Square S2
- Codesys SA0006

Description: Any writing to a variable from multiple tasks shall be avoided. A variable shall be written in one task only.

Guideline: When sharing variables among tasks you must avoid writing to a variable from more than one task. When using communication variables between two tasks, it is important to decide which task writes which variables so that a given variable is not written from the two different tasks.

Reasoning: The task exception is triggered by some timing constraints and relative priorities of the running tasks. On systems that implement multitasking in a 'preemptive' nature, or true parallel processing, one task may be interrupted by a higher priority task. In such a case, concurrent writing access to the same variable leads to non-deterministic behavior. The non-deterministic behavior has two consequences:

- Information loss
- Data inconsistency

Exceptions:

- Carefully design and peer review your code with skilled engineers to ensure non-deterministic behavior cannot occur
- With a centralized error monitoring system every task is allowed to set the error flag, but no one should reset the flag.
- Use vendor specific functions, like mutex or semaphore, to manage multiple write operations on the shared resource

Example:

Don't:

Imagine any simple case where a programmer has created a 'LoopCounter' Global Variable for general use throughout their Main program. Now imagine a second developer has copied and pasted a block to a priority program, that happens to run in a different task:

```
VAR_GLOBAL
  LoopCounter;
END_VAR
```

```
Main Program:
...
FOR LoopCounter := 0 TO 10 DO
  IF ValveWarning[LoopCounter] == TRUE THEN
        ProcessWarning := TRUE;
  END_IF;
END FOR;
```

```
Priority Program:
...
MaxReading := Reading[0];
FOR LoopCounter := 1 TO 100 DO
  IF Reading[LoopCounter] > MaxReading THEN
        MaxReading := Reading[LoopCounter];
  END_IF;
END FOR;
```

During execution of the main program, while inside the FOR loop the Priority Program may start to execute at any time (while the LoopCounter may be any value from 0 to 10). At the end of the Priority Program the global variable "LoopCounter" is left with the value of 100 when execution returns to the Main Program but the FOR loop now ends at the next iteration without completing the rest of the iterations. There is also potential for the code to execute now on the wrong element or even worse attempt access data beyond the bounds of the array.

Do:

In the above example, we can solve the problem by avoiding the multiple writes to LoopCounter by creating 2 program variables with local scope. For clarity these should be defined with different names.

Comments: For more information about synchronizing the execution of tasks see *Manage synchronization among tasks* (Rule CP11), page 64.

5.12. Manage synchronization among tasks

Identifier: Rule CP11

Importance: High

Targeted languages: All

References:

Description: Where tasks share data they shall be managed to avoid synchronization issues.

Guideline:

- Avoid interaction between tasks
  or
- Use vendor-specific feature to manage synchronization among tasks

In the case that synchronized data passing is necessary among PROGRAMs which can be assigned with different TASKs, following have to be done.

1) The PROGRAM should access to the global variables via

   A) VAR_EXTERNAL, or

   B) VAR_IN_OUT and assignment declaration to/from global variables in PROGRAM instance declaration in RESOURCE declaration.

2) In addition, some vendor-specific system-defined function or function block should be used which controls system's task scheduling and make specified code block be executed exclusively among other tasks.

Reasoning:

Considering multitasking behavior with synchronization is intrinsically difficult even if the PLC vendors support some feature for it. Pseudo parallel processing by cyclic execution of synchronized process is the key feature of PLC architecture which can overcome undetermined timing-dependent behavior of asynchronous multitasking control system. Thus basically data passing among PROGRAMs which need synchronization is better to be avoided.

If such synchronization is necessary from logic point of view, some code blocks within the body have to be exclusively executed among tasks. As IEC 61131-3 does not specify such a feature as standard, each vendor provides their own specific feature like tryLock(), Lock() of C++ / C# / Java.

Exceptions: Some vendors may provide completely different inter-task synchronization feature.

Example:

In the example below the 2 programs running in different tasks.

Do:

```
// This is an example where:
// 1) A) VAR_EXTERNAL and
// 2) Vendor specific system-defined function "Lock()","Unlock()" are used.


//Acquisition function called with the high frequency task
```

```
PROGRAM Fast_Acquisition
    VAR_EXTERNAL
        ClearSubTotalCount : BOOL;  // This is actually used as input
        SubTotalCount : INT;             // This is actually used as output
    END_VAR
    VAR_INPUT
        SignalInput  : BOOL;   // input signal to count rising-edge
    END_VAR
    VAR
        Old_ClearSubTotalCount : BOOL := FALSE;
    END_VAR

    //---- Beginning of the Body   ----
    Vendor.Lock(LockResId := 0); // Start of atomic process for exclusive
execution
        IF ClearSubTotalCount AND NOT Old_ClearSubTotalCount THEN
            SubTotalCount := 0 ;
        END_IF;
        Old_ClearSubTotalCount := ClearSubTotalCount;
        ClearSubTotalCount := FALSE;

        IF SignalInput AND NOT Old_SignalInput THEN
            SubTotalCount := SubTotalCount + 1 ;
        END_IF;
        Old_SignalInput := SignalInput ;
    Vendor.Unlock(LockResId := 0);  // End of atomic process for exclusive
execution
END_PROGRAM

// Acquisition with main task (100ms)
PROGRAM Main_Acquisition
    VAR_EXTERNAL
        SubTotalCount : INT;  // This is used as input actually.
        ClearSubTotalCount : BOOL; // This is used as output actually.
    END_VAR
    VAR
        GrandTotalCount : INT;
    END_VAR
    //---- Beggingin of the Body   ----
    Vendor.Lock(LockResId := 0); // Start of atomic process for exclusive execution
        GrandTotalCount := GrandTotalCount + SubTotalCount;
        ClearSubTotalCount := TRUE;
    Vendor.Unlock(LockResId := 0); // End of atomic process for exclusive execution
END_PROGRAM

CONFIGURATION CELL_1
    VAR_GLOBAL
        SubTotalCount : INT;
        ClearSubTotalCount: BOOL;
    END_VAR
    RESOURCE STATION_1 ON PROCESSOR_TYPE_1
        TASK SLOW_1(INTERVAL := t#200ms, PRIORITY := 2) ;
        TASK FAST_1(INTERVAL := t#10ms, PRIORITY := 1) ;
        PROGRAM MainAquisitionInst WITH SLOW_1 : Main_Acquisition();
        PROGRAM FastAcquisitionInst WITH FAST_1 : Fast_Acquisition(SignalInput := %I1.1);
    END_RESOURCE
END_CONFIGURATION
```

Comments:

This rule is only for PLC system of "Preemptive scheduling" implementation (See IEC 61131-3, Table 63, No.5b).
Some vendor provides similar system-defined functions.

It may be possible to avoid the need for synchronization. In the case that multiple PROGRAM instances all need to exchange data synchronously from the logic point of view, such PROGRAMs should be changed into FUNCTION_BLOCK and one PROGRAM should be added instead to call the FUNCTION_BLOCK instances synchronously.

As using vendor-specific "lock" feature temporarily blocks switching execution to the task with higher priority. It can make periodic task not to be in time for the specified period.

5.13. *Physical outputs shall be written once per PLC cycle*

Identifier: Rule CP12

Importance: High

Targeted languages: All

References:

- Codesys SA0004
- Itris Automation Square S4

Description: The physical outputs shall be written only once per PLC cycle.

Guideline: The physical output elaboration shall be done in one line of code or one rung.

Reasoning: 1. Undeterministic behavior when outputs are written more than once per PLC cycle. 2. Maintainability – it is difficult when a physical output elaboration is spread along the application. It is a good practice to prepare all variables participating to physical output elaboration and then at the end of the cycle, calculate and write the value for the physical outputs.

Exceptions: Clear and explicit requirements, like specific security architecture may overcome this rule. Sometimes it is necessary to write the output from more than one location, but then it should be done with caution.

Example: None

Comments: None

5.14. POUs shall not call themselves directly or indirectly

Identifier: Rule CP13

Importance: High

Targeted languages: All

References:

- Misra C 16.2
- JSF++ Rule 119
- IEC 61131-8 Section 3.5.4

Description: Recursion shall not be used in an application

Guideline: A recursive algorithm should be replaced by an iterative algorithm.

Reasoning: Recursive algorithms normally consume stack space for each call so deep recursion can cause system failure, even if unplanned. Support for recursive calls of POU is vender-specific so using it also makes the program less portable

Exceptions: None

Example:

Don’t:

The following function is using recursion to calculate the value of factorial (the factorial of N is the product of Nth first

```
FUNCTION Factorial : INT

VAR_INPUT

  X : INT;

END_VAR

  IF X > 1 THEN

    Factoriale := Factorial(X – 1) * X;

  ELSE

    Factorial := X;

  END_IF;

END_FUNCTION;
```

Do:

In this implementation, the recursion was removed and replaced by an iteration using FOR

```
FUNCTION Factorial : INT
VAR_INPUT
  X : INT;
END_VAR
VAR_LOCAL
  Acc : INT;
END_VAR
  FOR I IN 1..X DO
    Acc := Acc * X;
  END_FOR;
  Factorial := Acc;
END_FUNCTION;
```

Comments: None

5.15. POUs shall have a single point of exit

Identifier: Rule CP14

Importance: High

Targeted languages: All

References:

- Codesys SA0090
- Itris Automation Square I6
- Misra C 14.7
- JSF++ Rule 113
- IEC 61508-7 C.2.9

Description: RETURN instruction shall be avoided to exit from POUs. RETURN instruction shall be used only to explicitly return the value of a function.

Guideline: POU structure should be changed to avoid the usage of RETURN instruction before the end of the code. If programming language is Structured Text, conditional instructions should be used for the other languages, use a label at the end of the POU and use JUMP instruction to jump to this last label.

Reasoning: Testability, Readability and maintainability are good reasons to do that. In case of debugging, it is possible to add some code just before returning the POU and we got this information in all cases.

Exceptions: None

Example: None

Comments: None

5.16. Read a variable written by another task only once per cycle

Identifier: Rule CP15

Importance: High

Targeted languages: All

References: None

Description: In one task, multiple reading of a variable written by another task shall be avoided. The other task is able to be executed at any time between the two readings and so the two read values may be different. The program may behave in a non-deterministic way.

Guideline: To avoid this situation, in a task that reads a variable written to in another task, that variable should be copied into a local variable which is then exclusively used. This will ensure that the value won't unexpectedly change and so make program execution more predictable.

This copy could be done automatically by a programming support environment supporting the following features from IEC 61131-3,
Table 47, No.2a "Declaration of inputs of a program"
Table 47, No.2b "Declaration of outputs of a program "
Table 62, No.8b "Connection of GLOBAL variables to PROGRAM inputs"
Table 62, No.9b "Connection of PROGRAM outputs to GLOBAL variables"

When the declaration of program POU input/output variables, the global variables are copied to the input variables of the program before execution, and output variables are copied to the global variables after the execution. The value of input variables is never overwritten during the execution of the program in that cycle. Also output variables are never reflected to the global variables until the cycle execution of the program finishes.

Reasoning: In a program successive reading of a variable declared externally should be consistent. However, in the case of a variable written to by another task, this other task can be executed between two reads due to task switching.

Before running the process, the PLC is sampling the process inputs by copying the input values at a given time into an image memory. This is for the same reason: having constant input during a process execution.

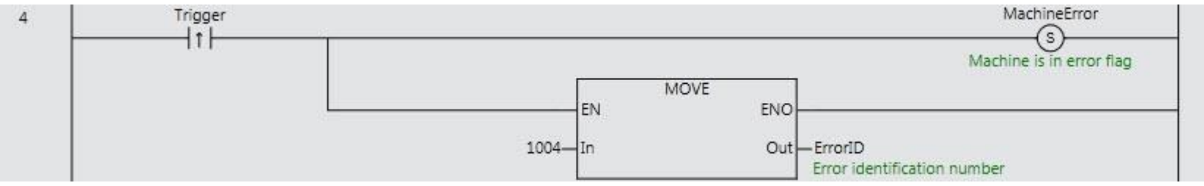
Exceptions: None

Example:

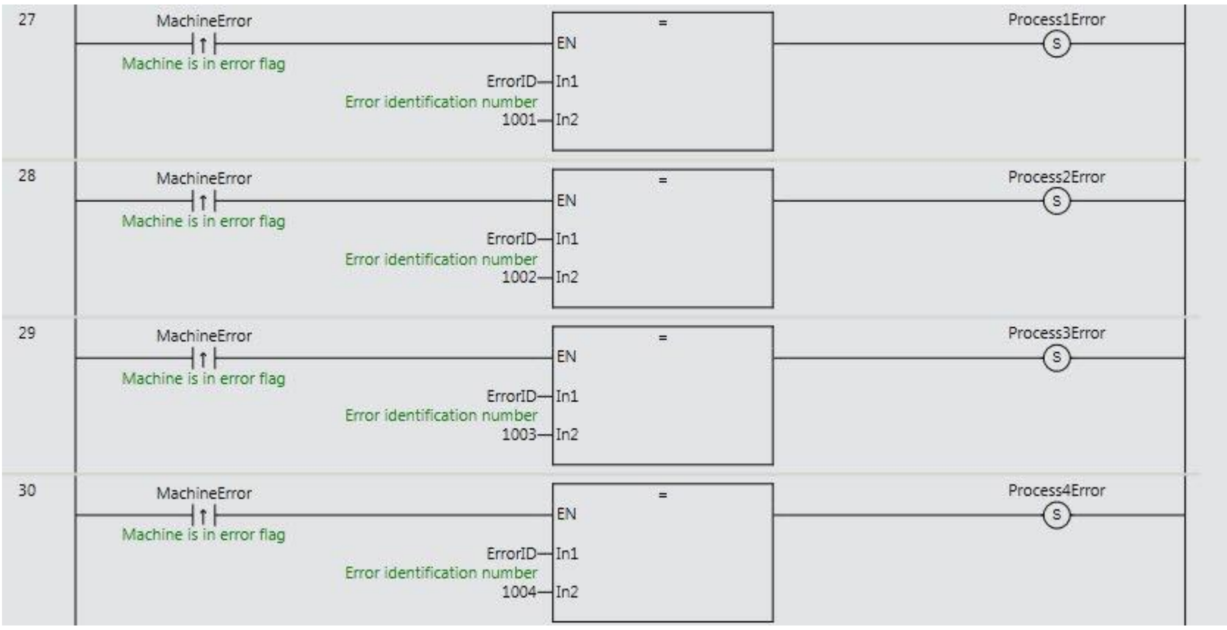
Don't:

In this example every time an error flag is raised, an error code is also raised to identify which part of the process is in error. The MachineError flag is generated from a high priority task that can preempt the main program at every time.

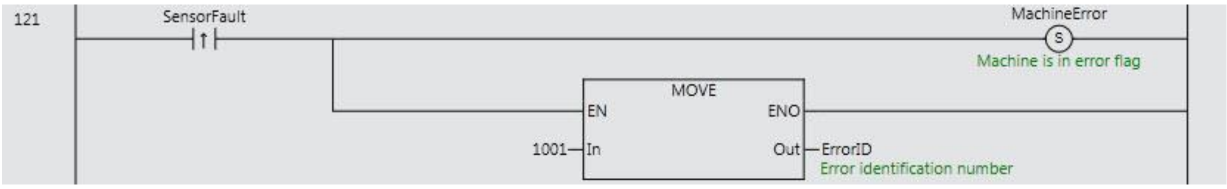
Main Program:


4
Trigger
MachineError
S
Machine is in error flag
MOVE
EN
ENO
1004
In
Out
ErrorID
Error identification number

....


27
MachineError
Machine is in error flag
=
EN
ErrorID
In1
Error identification number
1001
In2
Process1Error
S
28
MachineError
Machine is in error flag
=
EN
ErrorID
In1
Error identification number
1002
In2
Process2Error
S
29
MachineError
Machine is in error flag
=
EN
ErrorID
In1
Error identification number
1003
In2
Process3Error
S
30
MachineError
Machine is in error flag
=
EN
ErrorID
In1
Error identification number
1004
In2
Process4Error
S

Priority Program:


121
SensorFault
MachineError
S
Machine is in error flag
MOVE
EN
ENO
1001
In
Out
ErrorID
Error identification number

If execution of the main program is interrupted after rung 27, but before rung 30, the Priority Program may start to execute overwriting global variable "ErrorID". When execution returns to the Main Program the remaining rungs are evaluated, with the possibility that none of the "ProcessnError" bits are set.
This is a situation the developer of Main Program will not have realistically catered for.

Do:

The solution consists in reading MachineError and ErrorId only once and copying their value in a variable image. Then it ensures that ProcessnError will be consistent. The case where no ProcessnError is flagged when MachineError is flagged is not possible anymore.

If the Programming Support Environment allows it, the input/output of the POU program should be explicitly declared. Then the system takes care of this copy automatically.

Comments: None

5.17. Tasks shall only call program POUs and not Function Blocks

Identifier: Rule CP16

Importance: High

Targeted languages: Ladder, Structured Text, Sequential Function Chart, Function Block Diagram

References:

- IEC 61131-8 Sections 3.5.4 and 3.12.6
- IEC 61131-3 Section 2.7.2

Description: IEC Tasks shall not be configured to call Functions or Function Blocks directly. They shall only be configured to call program POUs.

Guideline: Tasks shall call program POUs only. Direct association of tasks to such function blocks should be avoided.

Reasoning: This is recommended to avoid ambiguity in determining the execution control of indirectly referenced function block instances. The association of tasks with function block instances and its effects on data concurrency are described in 2.7.2 of IEC 61131-3. The programmer should be aware of the fact that use of this feature may produce data consistency errors during program run time. The guidelines provided by the IEC 61131-3 implementer should be consulted to determine the mechanisms provided to assure data consistency. Since these mechanisms are implementation dependent, programs using this feature may not be portable between different IEC 61131-3 compliant systems

Exceptions: None

Example:

Don't: (example from Figure 20 of IEC61131-3. FB1 and FB2 should not be assigned directly to TASKs)

```
RESOURCE STATION_1 ON PROCESSOR_TYPE_1
   VAR_GLOBAL z1: BYTE; END_VAR
   TASK SLOW_1(INTERVAL := t#20ms, PRIORITY := 2) ;
   TASK FAST_1(INTERVAL := t#10ms, PRIORITY := 1) ;
   PROGRAM P1 WITH SLOW_1 :
     F(x1 := %IX1.1) ;
   PROGRAM P2 : G(OUT1 => w,
     FB1 WITH SLOW_1,
     FB2 WITH FAST_1) ;
END_RESOURCE
```

Comments: None

5.18. Usage of parameters shall match their declaration mode

Identifier: Rule CP17

Importance: High

Targeted languages: All

References:

- Codesys SA0009
- Itris Automation square E2
- IEC 61131-8 Section 3.2.2

Description: The parameters declared as input shall be read, the parameters declared as output shall be written and the parameters declared as in/out shall be read and written.

Guideline: The mode of a parameter (input, output or input/output) should reflect the usage of the parameter in the corresponding POU. The following rules apply:

- Each input parameter should be read at least once in the POU code
- Each input parameter should not be written in the POU code
- Each output parameter should be written at least once in the POU code
- Each input/output parameter should be either read or written in the POU code

Reasoning: Not using the parameters declared in the POU interface is a loss of time for the developers and for the machine performance. Using the parameter in the wrong way may lead to side effects when the Programming Support Environment doesn't limit the usage.

Exceptions: None

Example, use case:

Comments:

It is illegal to attempt to pass an output from a Function to an In-Out parameter of a Function Block:

```
                ACC1
     +----+   +-------+
X1---| *  |   | ACCUM |
X2---|    |---|A-----A|---ACC
     +----+   |       |
X3------------|X      |
              +-------+
```

```
ILLEGAL USAGE:
In-out A is not a variable
or function block name
```

It is illegal use to attempt to pass a literal to an In-out parameter:

```
             ACC1
          +--------+
          | ACCUM  |
    2.0---|A------A|---2.0
          |        |
       ---|X       |
          +--------+
```

ILLEGAL USAGE:
In-out A is not a variable
or function block name

5.19. Use of global variables shall be limited

Identifier: Rule CP18

Importance: High

Targeted languages: Ladder, Structured Text, Sequential Function Chart, Function Block Diagram

References:

- Misra C 8.7
- JSF++ AV Rule 207
- Codesys SA0121

Description: A global variable is required to exchange data with an entity outside the program or in another task. It is good practice to declare local variables whenever it is possible.

Guideline:

The use of global variables should be limited in preference for local variables. In the following case, the use of global variable is justified:

- Exchanging data among PROGRAM instances whether in the same TASK or in different TASKS
- Exchanging data with the System: accessing system-defined variable to use execution environment specific feature, such as CurrentTime, ErrorStatus, etc.
- Exchanging data with external devices (physical I/O, communication variables,..)

Reasoning:

Usage of global variables within a POU impairs its ability to be reused. In some development environments the use of an EXTERNAL declaration is not required, which serves to hide the dependency of a POU on external data.This rule improves the testability, maintainability, reusability of applications.

Exceptions: System-defined variables are the only authorized global variables, but there is no need to declare them.

Example:

Don't:

```
RESOURCE Station1 ON ProcesserTypeA

    VAR_GLOBAL

        MainProgDone: BOOL := TRUE;

        InitDone: BOOL := FALSE;

        InitError: BOOL := FALS;

        CurrentTime : TIME;

        CPUErrorStatus : ErrorStats;

    END_VAR
    TASK FastTask(INTERVAL := t#1ms, PRIORITY := 1);
    TASK SlowTask(INTERVAL := t#10ms, PRIORITY := 15);
    PROGRAM ProgInst1 WITH FastTask : Program1;
    PROGRAM ProgInst2 WITH FastTask : Program2;
    PROGRAM ProgInst3 WITH SlowTask : Program3;
    PROGRAM ProgComm  WITH SlowTask : CommProg;
END_RESOURCE
```

Note:

In this "Don't" example, each PROGRAM does not declare input/output variables and accesses global variables via external variables. To know the data flow among PROGRAM instances, the logic of each PROGRAM body need to be examined.

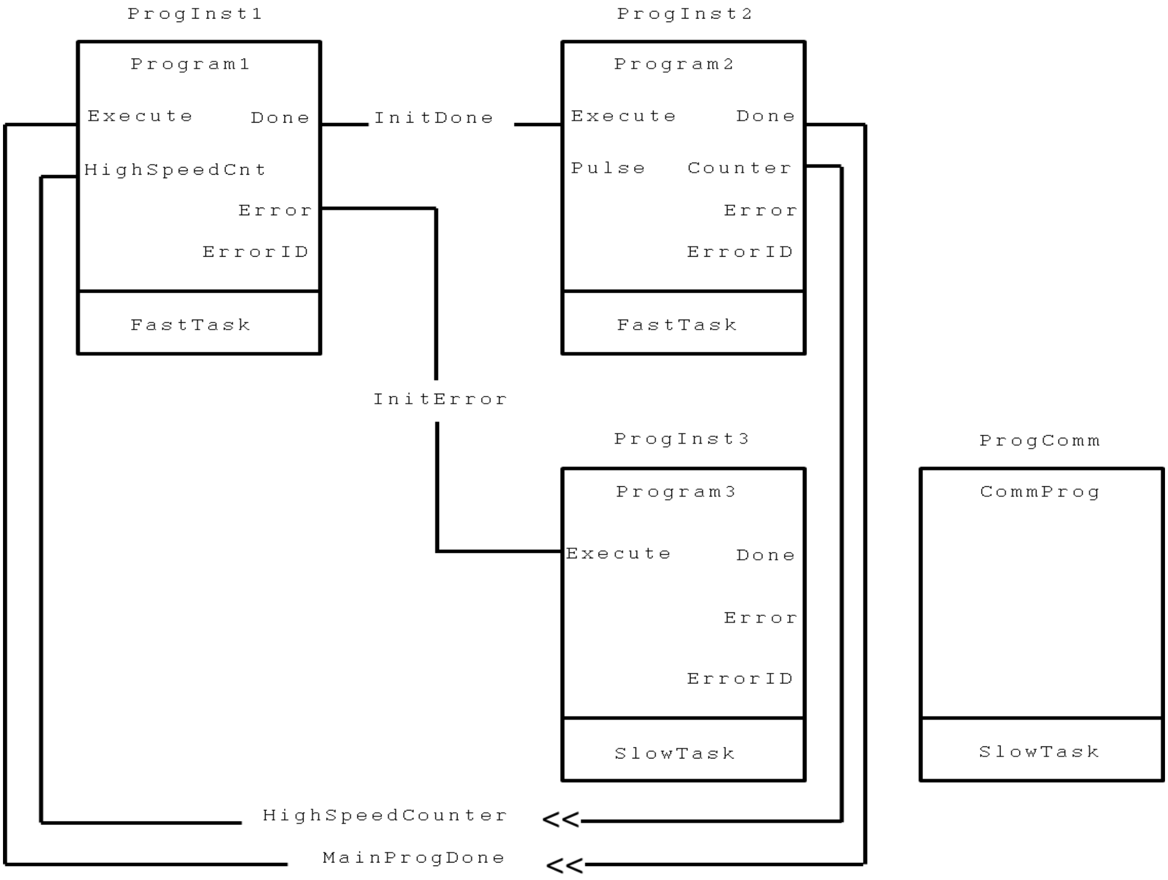
Do:

```
RESOURCE Station1 ON ProcessorTypeA
    VAR_GLOBAL
    //-- 1) Variables for inter-PROGRAM communication
        MainProgDone: BOOL := TRUE;
        InitDone: BOOL := FALSE;
        InitError: BOOL := FALSE;
    //-- 2) System-defined variables to be accessed via VAR_EXTERNAL
        CurrentTime : TIME;
        CPUErrorStatus : ErrorStats;
    //-- 3) Variables exchanged between tasks
        HighSpeedCounter : DINT;
    //-- 4) I/O and communication variables
        HSPulse : BOOL AS %I4.5;
    END_VAR
    TASK FastTask(INTERVAL := t#1ms, PRIORITY := 1);
    TASK SlowTask(INTERVAL := t#10ms, PRIORITY := 15);
    PROGRAM ProgInst1 WITH FastTask
          : Program1(Execute := MainProgDone,
                     HighSpeedCounter := HighSpeedCounter,
                     Done => InitDone,
                    Error => InitError); //Global variable assignment

    PROGRAM ProgInst2 WITH FastTask
          : Program2(Execute := InitDone,
                    Pulse := HSPulse,
                    Counter => HighSpeedCounter,
                    Done => MainProgDone); // Global variables
                                           // assignment
    PROGRAM ProgInst3 WITH SlowTask
          : Program3(Execute := InitError); // Global variables
                                            // assignment
    PROGRAM ProgComm WITH SlowTask : CommProg;
END_RESOURCE
```

Note:

Program instance declaration above explicitly represents data flow among PROGRAM instances shown below. (This diagram is NOT a formal graphical representation of IEC 61131-3)


ProgInst1
Program1
Execute
Done
HighSpeedCnt
Error
ErrorID
FastTask
InitDone
ProgInst2
Program2
Execute
Done
Pulse
Counter
Error
ErrorID
FastTask
InitError
ProgInst3
Program3
Execute
Done
Error
ErrorID
SlowTask
ProgComm
CommProg
SlowTask
HighSpeedCounter
MainProgDone

Comments:

The rules below also relate to the usage of global variables:

1. Rule 5.27 A global variable may be written only by one PROGRAM
2. Rule 5.11 Avoid multiple writes from multiple tasks.

5.20. Usage of Jump and Return should be avoided

Identifier: Rule CP19

Importance: Medium

Targeted languages: Ladder, Function Block Diagram

References:

- Misra C 14.5
- JSF++ Rules 189, 190

Description: The branching instructions Jump and Return should be avoided in Ladder and Function Block Diagram. This also includes any implementation specific branching instructions.

Guideline:

- Unconditional branching instructions Jump and Return shall not be used
- Conditional branching shall not jump backwards

Reasoning: Maintaining and debugging code containing sequence breaking instructions like Jump and Return is harder. Jumping over code to disable it leaves the dead code liable to future execution if the jump is accidentally removed. Jumping backwards can cause irregular scan times, and even cause tight loops that affect the main control function. The construct of EN/ENO can be used for conditional execution in LD and FBD.

Exceptions: Small jumps backwards are sometimes necessary to implement loops or iterations. Extreme caution must be used to ensure that infinite loops cannot occur. It is recommended to switch to Structured Text for these conditional transitions and use IF..THEN..ELSE, CASE, FOR..WHILE.., REPEAT..UNTIL.

Example:

<table>
<tr><th>Don't:</th><th>Do:</th></tr>
<tr><td><pre>
+------>>LABELA
| a   b   c   d |
+---| |---| |---+---( )---( )---+
| | e |
| +---( )---------+
LABELA:
| |
</pre>This code is 'commented out' by skipping over, but still reserves program space and can too easily be accidently reinstated, without compile error, if the rung with the Jump is accidently deleted.</td><td>Release code should have all unneeded code deleted and removed.</td></tr>
<tr><td><pre>
| flg |
</pre></td><td><pre>
| flg   w   x |
+---|/|---| |-------------(S)---+
</pre></td></tr>
</table>

```
+---| |---->>J26                 |
|    w                      x    |
+---| |---------+---------(S)---+
|                                |
J26:
|   flg
+---|/|---->>J27                 |
|    y                      z    |
+---| |---------+---------(S)---+
|                                |
J27:
|                                |
```

```
|                              |
|   flg    y              z    |
+---| |---| |-------------(S)---+
|                              |
```

Conditional code can be controlled with Boolean logic

```
IF NOT flg AND w THEN
    z := TRUE;
ELSIF flg AND y THEN
    x := TRUE;
END_IF;
```

Conditional code can better be expressed in ST, especially more complex and nested structures

```
|          +-----+               |
+----------| MOVE|---------------|
|      0 --|     |-- ctr         |
|          |     |               |
|          +-----+               |
LOOP1:
|          +-----+               |
+----------| MOVE|---------------|
| y[ctr] --|     |-- x[ctr]      |
|          |     |               |
|          +-----+               |
|       +-----+                  |
+-------| Add |------------------+
| ctr --|     |-- ctr            |
|   1  -|     |                  |
|       +-----+                  |
|       +-----+                  |
```

```
FOR ctr := 0 to 10 DO
    x[ctr] := y[ctr];
END_FOR;
```

```
+-------|  <= |------>>LOOP1
| ctr --|     |                 |
|  10  -|     |                 |
|       +-----+                 |
```

Comments: none

5.21. Function block instances should be called only once

Identifier: Rule CP20

Importance: Medium

Targeted languages: All

References:

- IEC 61131-8 Sections 3.3.3 and 3.7
- Itris Automation Square I6

Description: Each instance of a function block should be called only once per PLC cycle.

Guideline: For each variable instantiating a function block, the function block code should be called no more than once per PLC cycle. The call may be conditional.

Reasoning: With a single invocation rule there is typically one place, and only one place, where the input variable of a function block will be assigned a value (unless IEC feature "Separate assignment of input" is employed), which increases the reliability and maintainability of the software.

- Development by copy, paste and modify is error prone, especially with code involving function blocks where it is possible to forget to change the instance, resulting in problems that may be hard to diagnose.
- Moreover depending on function block architecture, calling the same function block twice may change the global behavior. In general, it may be possible to invoke a single instance of a function block several times (“multiple assignment”) within a POU. However, depending on the programmable controller implementation, this possibility may be restricted to a single invocation of each function block instance within a POU. A POU that uses multiple invocations of a single function block instance may be non-portable to such implementations.

A Function Block that does access any physical I/O should be called only one time within a PLC cycle.

A simple guideline would be to call a function block instance only once in the same PLC cycle. Although in practice there can be reasons to call a function blocks more than once in the same cycle (see exceptions hereunder). Check the documentation of the function block if this is possible.

Exceptions: There can be many cases where it is actually wise, efficient or necessary to invoke the same instance multiple times. This can be acceptable for the experienced developer with sufficient code review and program analysis. Examples of some exceptions are:

- A Counter FB that could be counting up many cases during the same PLC cycle.
- As function blocks only have one body, it is possible to simulate different methods using an additional parameter. In this case the same instance of a function block could be called once per simulated 'method' in a PLC cycle
- When using the non-formal invocation it is common to omit any unrequired parameters. It is likely to invoke the instance in multiple places with different parameter lists and these multiple invocations might occur in the same PLC cycle. Care should be taken that the formal invocation of a function block has been used at least once, or that initial parameter values are meaningful

- Motion Function Blocks instances are sometimes called multiple times per PLC scan, for example when Blending between steps or using "BufferMode" to stack up consecutive commands.

Example:

Don't: calling twice a raising edge function block: the second call doesn't see any more the rising edge it-self.

```
FUNCTION_BLOCK Rising_Edge
VAR_INPUT
  S : BOOL;
END_VAR;
VAR
  Old_State : BOOL;
END_VAR;

Old_State := Old_State XOR S;
END_FUNCTION_BLOCK;

VAR
  InputDetectionForDashboardButtonA: Rising_Edge;
  InputDetectionForDashboardButtonB: Rising_Edge;
END VAR;

InputDetectionForDashboardButtonA(InputFilterA);
...

// The second call of the same FB

InputDetectionForDashboardButtonA(InputFilterB);

// Information about the rising edge found previously is lost here
// In fact this is a COPY/PASTE error: the parameters were changed
// but not the instance.
```

Comments: None

5.22. *Use VAR_TEMP for temporary variable declaration*

Identifier: Rule CP21

Importance: Medium

Targeted languages: All languages

References:

- IEC61131-3 (ed3) 6.5.2.1
- IEC61131-3 (ed3) 7.3.3.4.2

Description:

The standard says:

- VAR

  The variables declared in the VAR ... END_VAR section persist from one call of the program or function block instance to another.
  Within functions the variables declared in this section do not persist from one call of the function to another.

- VAR_TEMP

  Within program organization units, variables can be declared in a VAR_TEMP...END_VAR section.
  For functions and methods, the keywords VAR and VAR_TEMP are equivalent.
  These variables are allocated and initialized with a type specific default value at each call, and do not persist between calls.

Guideline: Use the VAR_TEMP construct to declare temporary variables, for example in FOR loop control. For FBs use VAR_TEMP. Temporary variables are created and initialized upon each invocation of the FB, and cannot be accessed outside of the body.

Reasoning: For Functions and Methods VAR and VAR_TEMP are the same but not for Function Blocks. Therefore it is always better to use VAR_TEMP for temporary variables.

Exceptions: none

Example:

Don't:

```
// In the above example
VAR
    values: ARRAY[1..10] OF REAL;
    index: INT;
END_VAR
```

Do:

```
FUNCTION_BLOCK Lifo
VAR_OUTPUT
    currentValue: REAL;
END_VAR
```

```
VAR_INPUT
    newValue: REAL;
END_VAR
VAR
    values: ARRAY[1..10] OF REAL;
END_VAR
VAR_TEMP
    index: INT;
END_VAR

    currentValue := values[ 1 ];
    FOR index := 1 TO 10 DO
        values[ i ] := values[ i + 1 ];
    END_FOR
    values[ 10 ] := newValue;
END_FUNCTION_BLOCK
```

5.23. Select Appropriate Data Type

Identifier: Rule CP22

Importance: Medium

Targeted languages: Ladder, Structured Text, Sequential Function Chart, Function Block Diagram

References:

- IEC 61131-8 Section 3.1

Description: The data type selected for a variable should be appropriate to the data it is intended to store. It should be appropriate to the range of values and operations to be performed on the variable.

Guideline: Choose the correct data type appropriate to the range of values and operations to be performed.

- Use the smallest length necessary to store the range of values
- Do not use signed data types for unsigned data
- Use enumerations where possible
- Use subranges where appropriate
- Group collected data of same type into arrays
- Group collected data of different types into structures
- Do not use the same datatype throughout the program, just to prevent the need for explicit conversions. Use the appropriate ones

Reasoning:

- A correctly data typed variable helps describe its function, making its use somewhat self-explanatory
- "Strongly typed" code, where data type conversions must be explicitly made helps avoid coding mistakes and oversights where some conversion behavior may not be as assumed, and may be missed by commissioning and testing phases
- Compilers are able to use the data type to check assignments and instructions use, to ensure operations are as the developer expects
- Smaller data types typically use less memory, so allow for more variables or larger programs
- Using unsigned data types where appropriate prevents any negative value being assigned accidently, and having to write code, and test the code, to deal with these eventualities.
- The use of enumerated and subrange types make a program even more readable and can contribute to program reliability by helping to avoid the use of unintended values of variables as well as by explicitly expressing the intended semantics of the values of enumerated variables

Exceptions: When sharing data with third party devices the data type may be externally mandated as a sub-optimal data type.

Example:

- If a variable can only hold the values 0 or 1, and is only to be operated on by Boolean operations, then the elementary type BOOL should be chosen;

- If a programmable controller program has to count something and the counts are expected to be in the range from 0 to 1000, a variable of type SINT or USINT cannot be used, since their value ranges only extend from -128 to +127 for SINT and from 0 to 255 for USINT. A reasonable data type for this purpose would be UINT. This has a sufficient value range and the usage of an unsigned integer type also makes it clear that negative values are not expected
- Any code that uses a CASE statement should strongly consider if the expression variable should be an enumerated type, testing against the semantic value rather than numeric value
- An enumerated data type restricts the values of variables of the type to a user-defined set of identifiers. As an example consider

  ```
  TYPE Color : ( Red, Yellow, Green );
  END_TYPE
  ...
  VAR_GLOBAL brickColor : Color; END_VAR
  ```

  Here a new type Color is defined. It may only have three values - Red, Green, or Yellow. IEC 61131-3 does not define numerical values to which these enumerated values may correspond. There also is no conversion function to and from enumerated types to integral types. The values only have to be distinct and reproducible. An assignment of a value to the variable brickColor is possible only if one of the defined color values is used. All other values are flagged as errors

  Another enumerated data types example:

  ```
  TYPE ANALOG_SIGNAL_TYPE : (SINGLE_ENDED, DIFFERENTIAL) ;
  END_TYPE
  ```

  Even though there are only 2 values, using enumeration make the code easier to read and also extend.
- Subrange data types can limit the range of permissible values, e.g.:

  ```
  TYPE ANALOG_DATA : INT (-4095..4095) ; END_TYPE
  ```

- Array data types group data of the same data type, e.g.:

  ```
  TYPE ANALOG_16_INPUT_DATA : ARRAY [1..16] OF ANALOG_DATA ;
  END_TYPE
  ```

- Structured data types group data of the different data types e.g.:

  ```
  TYPE
      ANALOG_CHANNEL_CONFIGURATION :
          STRUCT
              RANGE : ANALOG_SIGNAL_RANGE ;
              MIN_SCALE : ANALOG_DATA ;
              MAX_SCALE : ANALOG_DATA ;
          END_STRUCT ;
      ANALOG_16_INPUT_CONFIGURATION :
          STRUCT
              SIGNAL_TYPE : ANALOG_SIGNAL_TYPE ;
              FILTER_PARAMETER : SINT (0..99) ;
              CHANNEL : ARRAY [1..16] OF
  ANALOG_CHANNEL_CONFIGURATION ;
          END_STRUCT ;
  END_TYPE
  ```

Comments: None

5.24. Define maximum number of input/output/in-out variables of a POU

Identifier: Rule CP23

Importance: Medium

Targeted languages: all

References:

- JSF++ - AV Rule 110

Description: Define a maximum number of input/output/in-out variables of a POU. The number of input variables, output variables, and in-out variables should be within a limited set by your organization.

Guideline: A limit can be around 10.

If more input/output/in-out variables are necessary, consider to reduce the number of pins (input/output/in-out variables) of POUs by the following method.

1. Introduce UDTs (user-defined structured data type) to group some input/output/in-out variables.
2. If some input/in-out variables are a kind of configuration parameter which does not need to be changed on every cyclic call of the FB,

   A) Change them into internal variables, and give them necessary initial values by using VAR_CONFIG struct.

   or

   B) Change them into internal variables with PUBLIC or INTERNAL access specifier, and give them necessary value before calling the FB.

   Note: internal variable with access specifier is a newly added feature of IEC 61131-3 Ed.3.

Reasoning: Using FB with too many pins breaks readability.

Exceptions:

- POUs which are to be used only in ST language.
- POUs which are to be used only in the vendor-specific PSE which supports some feature to hide specified pins of specified POU in graphical language editor.

Example:

Don't: Function block with too many pins

```
(* Graphical representation of FB declaration *)
            +----------------------+
            |     MyFunctionBlock  |
      DINT--|Cfg_Param1--Cfg_Param1|--DINT
     DWORD--|Cfg_Param2--Cfg_Param2|--DWORD
      BOOL--|Cfg_Param3--Cfg_Param3|--BOOL
       INT--|Cfg_Param4--Cfg_Param4|--INT
STRING[32]--|Cfg_Param5--Cfg_Param5|--STRING[32]
      BOOL--|Cfg_Param6     Output1|--BOOL
      DINT--|Cfg_Param7     Output2|--BOOL
      BOOL--|Cfg_Param8     Output3|--BOOL
      BOOL--|Cfg_Param9     Output4|--BOOL
      BOOL--|Enable         Output5|--BOOL
       INT--|Input2         Output6|--DINT
       INT--|Input3         Output7|--BOOL
      BOOL--|Input4         Output8|--BOOL
      BOOL--|Input5         Output9|--BOOL
       INT--|Input6        Output10|--BOOL
      BOOL--|Input7        Output11|--BOOL
      BOOL--|Input8        Output12|--DINT
       INT--|Input9        Output12|--BOOL
      BOOL--|Input10       Output14|--BOOL
      BOOL--|Input11       Output15|--BOOL
       INT--|Input12       Output16|--BOOL
      BOOL--|Input13       Output17|--BOOL
      BOOL--|Input14       Output18|--DINT
       INT--|Input15          Error|--BOOL
      BOOL--|Input16        ErrorID|--DWORD
      BOOL--|Input17               |
       INT--|Input18               |
            +----------------------+
```

Do:

```
(* Graphical representation of FB declaration *)
                                 +-------------------------+
                                 |     MyFunctionBlock     |
                           BOOL--|Enable       CylinderOut1|--DT_CylinderOut
                            INT--|Input2       CylinderOut2|--DT_CylinderOut
                            INT--|Input3       CylinderOut3|--DT_CylinderOut
 ARRAY[1..5] OF DT_EncoderInput--|EncoderInputs       Error|--BOOL
                                 |                  ErrorID|--DWORD
                                 +-------------------------+
```

```
(* Textual declaration of FB *)
FUNCTION_BLOCK MyFunctionBlock
    VAR_INPUT
        Enable : BOOL;
        Input2 : INT;
        Input3 : INT;
        EncoderInputs : ARRAY[1..5] OF DT_EncoderInput
    END_VAR
    VAR_OUTPUT
        CylinderOut1 : DT_CylinderOut;
        CylinderOut2 : DT_CylinderOut;
        CylinderOut3 : DT_CylinderOut;
        Error : BOOL;
        ErrorID : DWORD;
    VAR PUBLIC
        ConfigParams : DT_MyFBConfiguration //Static configuration
parameter.
        // This variable is not shown as pin but can be written
        // separately before calling the FB instance.
    END_VAR
END_FUNCTION_BLOCK
TYPE
    DT_MyFBConfiguration : STRUCT
        Param1 : DINT;
        Param2 : DWORD ;
        Param3 : BOOL;
        Param4 : INT;
        Param5 : STRING[32];
        Param6 : BOOL;
        Param7 : DINT;
        Param8 : BOOL;
        Param9 : BOOL;
    END_STRUCT
    DT_EncoderInput : STRUCT
        IsReverse : BOOL := FALSE;
        Reset : BOOL := FALSE;
        Count : INT := 0;
    END_STRUCT
    DT_CylinderOut : STRUCT
        Solenoid1 : BOOL;
        Solenoid2 : BOOL;
        Solenoid3 : BOOL;
        Solenoid4 : BOOL;
        PilotLamp : BOOL;
        DumpParam : DINT;
    END_STRUCT
END_TYPE
```

Comments: none

5.25. Do not declare variables that are not used

Identifier: Rule CP24

Importance: Medium

Targeted languages: All

References:

- Codesys SA0011, SA0033
- Itris Automation Square S7

Description: Each declared variable should be used in the code

Guideline: All declared variable in an application should be read or written elsewhere in the code.

Reasoning: When development starts from an pre-existing project, removing unused declarations is not always the priority, so unused variables may be part of the final application. Such an application may be more difficult to understand and maintain, and will also lose some performance from a memory point of view.

Exceptions: Variables explicitly identified as spare variables can be defined. In such a case a variable identified as a spare should not be used elsewhere in the code. Those variables are used for keeping spare memory free for future usage and for allowing on line modifications (impossible to declare while the program is running, so variables are declared in advance).

Example: None

Comments: None

5.26. *Data types conversion should be explicit*

Identifier: Rule CP25

Importance: Medium

Targeted languages: All

References:

- Codesys SA0019
- Misra C 10.3
- IEC 61131-3 section 6.6.1.6 and Table 11

Description: Data types conversion should be done explicitly by the developer and not added by the compiler

Guideline: When writing code that converts between types, the developers shall use explicit casts and not rely on the compiler's type inference capabilities.

Reasoning: The data type conversions are responsible for loss of range, loss of precision or loss of signedness. The compiler doesn't care about the loss occurring when doing the conversion between two data types. The conversion shall be explicitly done by the developer so that the correct questions get their answers.
It is good practice to keep as long as possible the initial data type format and convert it to something else before sending the information outside the code: SCADA formatting, other equipment communication.

Exceptions: Implicit datatype conversions can be used when there is no loss of value or precision (see table 11 of IEC 61131-3 3rd edition)

Example:
Don't:

```
VAR
  I : INT := 10;
  J : REAL := 0.55;
END_VAR;

I := I * J;
// If it compiles (compiler not compliant to IEC61131 Ed.3rd)
// Result can be either 0 or 5 depending on the way the compiler
// is interfering types
```

Do either:

```
I := I * REAL_TO_INT(J);
```

or:

```
I := REAL_TO_INT(INT_TO_REAL(I) * J);
```

Note that the result can differ between the 2 cases. Also if there is no explicit typecast the user does not know which solution is chosen by the compiler.

Comments: None

5.27. A global variable may be written only by one PROGRAM

Identifier: Rule CP26

Importance: Low

Targeted languages: All

References:

- Itris Automation Square : S2

Description: A global variable may be written only by one PROGRAM

Guideline: When a global variable is assigned, this assignment should be in one PROGRAM only. Each PROGRAM is in charge of assigning some of the global application variables.

Reasoning: The data flow is easier to understand and read and so the debug and the maintenance are more efficient.

Exceptions: None

Example: None

Comments: None

5.28. Avoid deprecated features

Identifier: Rule CP27

Importance: Low

Targeted languages: Ladder, Structured Text, Sequential Function Chart, Function Block Diagram

References:

- IEC 61131-8 Section 3.12
- IEC 61131-3 3rd edition

Description: Avoid writing code using deprecated or obsolete Data Types, Functions, Function Blocks, System variables etc. Also you may avoid using programming practices which are replaced by improved methods.

Guideline: Avoid all deprecated and obsolete functions and practices, including:

- BCD Data types and instructions
- IEC 61131-3 Second Edition the standard Functions and Function Blocks that have been superseded by IEC 61131-3 Third Edition
- Use of JUMP instruction, where conditional statements are implemented
- Use of multiple Function Blocks processing the same data, where Classes with multiple Methods are provided
- Unnecessary use of global variables. Instead use Local variables and encapsulate data, where possible. In particular, the writing of global variables from more than one program location should be avoided. It is recommended that global variables should be used (if at all) only for supplying values of “global” interest to other program organization units.
- The direct association of tasks with function block instances

Reasoning: The effects of programming technique on software quality should be considered when choosing among the options made available in IEC 61131-3. The latest programming practices to achieve higher software quality are recommended. Also features that are currently deprecated are likely to become obsolete in future versions and not be supported, which can force unplanned rewriting. Excessive use of global variables contradicts the principles of encapsulation and hiding discussed in IEC 61131-8 section 2.4.2.1 and can greatly reduce software reliability, maintainability and reusability.

Exceptions:

- When connecting to legacy devices which *only* support the deprecated features.
- When using third party libraries or code that makes use of deprecated features.

Examples: In IEC61131-3 Ed.3rd, the following features are marked as deprecated and should not be used anymore:

- Octal literals 8#377
- Use of directly represented variables in the body of POUs and methods.
- Standard function "TRUNC"
- Indicator-variable of Action Block of SFC
- Instruction list (IL)

Comments: None

6. Languages

6.1. Define indentation

Identifier: Rule L1

Importance: Low

Targeted languages: Structured Text, Instruction List, textual variable declaration

References:

- JSF++ 44

Description: You may define your use of indentation, and use consistently throughout the project.

Guideline: Use 4 spaces for indentation level.

Reasoning: Indenting code aids readability, particularly for conditional and loop statements. Small indents (e.g. 1-2 spaces) are not always clear. Large indents (e.g. 8 spaces) can mean that nested statements create code too wide for the screen.

Exceptions: none

Example, use case:

Don't: the nested IF instruction should be at the same indentation than Sort call:

```
IF sizeListToSort < 0 THEN
   Sort (numberElements := sizeListToSort,
      direction := 2,
      idEse := idEse,
      Ese := Ese,
      status => statusTemp);

      IF NOT statusTemp THEN  // Indentation is not correct here
             status := statusTemp;
      END_IF;

END_IF;
```

Do:

```
// Initialisation
IF Dem_froid OR Rep_chaud OR Prem_cycle THEN
    Cmd_vanne_remplissage.Temps_ma := 15;
    Cmd_vanne_vidange.Temps_ma := 15;

    Local := true;
ELSE
    // Grafcet
    init_graph := false;
END_IF;
```

Comments: See also 6.6.7 *Define use of tabs,* page 120.

6.2. Function Block Diagram FBD

6.2.1. Avoid assignments of intermediate results within networks

Identifier: Rule L2

Importance: Medium

Targeted languages: Function Block Diagram

References: None

Description: Avoid assignments of intermediate results within networks

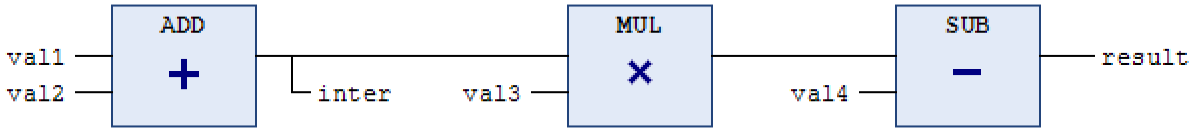
Guideline: In Function Block Diagram networks one should avoid the assignment of variables between blocks.

Reasoning: Side effects in the networks are difficult to see if this construct is used.

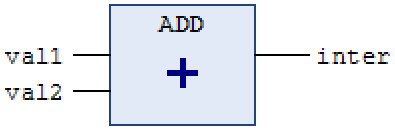
Exceptions: None

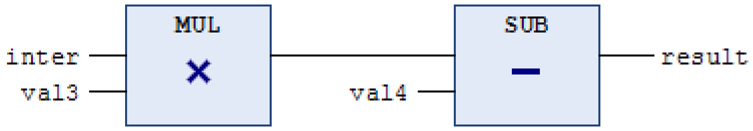
Example:

Don't


ADD
+
MUL
×
SUB
−
val1
val2
inter
val3
val4
result

Do:


ADD
+
val1
val2
inter


MUL
×
SUB
−
inter
val3
val4
result

Comments: None

6.2.2. Define maximum complexity of single network

Identifier: Rule L3

Importance: Medium

Targeted languages: Function Block Diagram

References: None

Description: Define the maximum complexity permissible for a network, for example by limiting the number of elements

Guideline: Function Block Diagram networks should be kept to a maximum of 32 elements (Functions, Function Blocks) per network.

Reasoning: Complex algorithms reduce readability and increase the likelihood of misunderstanding and mistakes. When a diagram grows, it cannot be shown on a single screen (although dependent on screen size and zoom factor). Large diagrams containing too many elements suggest a need to split into more diagrams or sub diagrams, each with a distinct action.

Exceptions: None

Example: None

Comments: None

6.3. Ladder (LD)

6.3.1. A coil should not be followed by a contact

Identifier: Rule L5

Importance: Medium

Targeted languages: Ladder

References:

- MISRA C: 13.4

Description: A coil should not be followed by a contact.

Guideline: Instead of directly using a coil one should insert a second rung. See the ‘Do’ example below.

Reasoning: Readability

Exceptions: None

Example:

Don't: IEC 61131-3 section 8.2.5 shows a contact after a coil: "in the rung shown below, the values of the Boolean output a is always TRUE, while the value of outputs c,d and e upon completion of an evaluation of the rung is equal to the value of the input b."
Problem: the assignment of 'a' by the coil is unexpected here, and can be overseen.

```
|   a    b       c     d   |
+--( )--| |--+--( )---( )--+
|            |      e      |
|            +-----( )-----+
|                          |
```

Do: Use an extra network for the assignment to variable 'a'.

```
| TRUE    a   |
+--| |---( )--+
|             |

|  a   b       c   |
|--| |--| |--+--( )--+
|            |   d   |
|            +--( )--+
|            |   e   |
|            +--( )--+
|                    |
```

Comments: None

6.3.2. Define maximum rung complexity

Identifier: Rule L6

Importance: Medium

Targeted languages: Ladder

References: none

Description: Define the maximum complexity permissible for a rung, for example by limiting the number of ladder elements

Guideline: Ladder rungs should be kept to a maximum of 32 ladder elements (contacts, coils, Functions, Function Blocks) per rung.

Reasoning: Complex algorithms reduce readability and increase the likelihood of mis-understanding and mistakes. When the program grows, the rung cannot be shown on a single screen (although dependent on screen size and zoom factor). Large rungs may contain many sequential elements, or many parallel elements, or often both. When the rung is too large it should be split into more rungs or Functions/Function Blocks, each with a distinct action.

Exceptions: None

Example: None

Comments: None

6.4. Sequential function Chart (SFC)

6.4.1. Closing divergent paths

Identifier: Rule L7

Importance: High

Targeted languages: Sequential Function Chart

References:

- IEC 61131-3 Figure 18a

Description: Simultaneous divergent paths shall be closed correctly.

Guideline: The opening and closing of simultaneous divergent branches should be done correctly. If other divergences are used inside the parallel branches, they should be closed with simultaneous convergence before the end of parallel branches.

Reasoning: It can be unclear if parallel paths start executing steps outside the divergent region and may cause unexpected behavior like token proliferation.

Exceptions: None

Example:

Don't:

Example from IEC61131-3 Figure 18a. Consider Steps B and C are active, then transitions t3, t5 and t7 occur. Step B is still active but should execution also pass through E and F to Step A, which is outside the divergent path? It is unclear and can be implementation dependent.

```
+---------+

|          |
|       +=====+
|       || A ||
|       +=====+
|          |
|          + t1
|          |
|  ===+====+====+===
|     |         |
|  +-----+   +-----+
|  |  B  |   |  C  |
|  +-----+   +-----+
|     |         |
|     |         *--------+
|     |         |        |
|     |         + t2     + t3
|     |         |        |
|     |        +---+    +---+
|     |        | D |    | E |
|     |        +---+    +---+
|     |         |        |
|  ===+====+====+===     |
|          |             |
|          + t4          + t5
|          |             |
|        +---+         +---+
|        | F |         | G |
|        +---+         +---+
|          |             |
|          + t6          + t7
|          |             |
+----------+-------------+
```

Comments: None

6.4.2. Do not program an SFC action block in SFC

Identifier: Rule L8

Importance: Medium

Targeted languages: Sequential Function Chart

References:

- Link to SFC subgroup

Description: When adding actions to an SFC state, it is possible to select the action programming language. Don't choose the SFC language to program such an action.

Guideline: Use only ST, FBD or LD in an action block. If you need to have a state diagram in SFC inside your action block, encapsulate this state machine in a separate FB which is then called in the action block itself.

Reasoning: Programming an action block detail in SFC (sometimes called “nested SFC”) can become complex and difficult to look through. The lower/inner SFC execution behavior is not specified by the IEC61131-3 standard. Then depending on programming software, it may not be executed in every task cycle but only when its related action in the higher/outer SFC is active. Therefore it is difficult or impossible in the lower/inner SFC to react.

Exceptions: none

Example: none

Comments: Most PLC programming software doesn't allow developer to select the SFC language to implement an SFC action. This feature was supported in the 2nd edition of the IEC 61131-3 standard, however it is removed in the 3rd edition.

6.4.3. Define maximum complexity

Identifier: Rule L9

Importance: Medium

Targeted languages: Sequential Function Chart

References: None

Description: Define the maximum complexity permissible for a diagram, for example by limiting the number of elements to 32.

Guideline: Sequential Function Charts should be kept to a user defined maximum steps per chart.

Reasoning: Complex algorithms reduce readability, and increase likelihood of misunderstanding and mistakes. There also comes a time when the whole chart cannot be shown on a single screen (although dependent on screen size and zoom factor). Large charts containing too many elements should be split into 2 or more sub-charts, each with a distinct action.

Exceptions: None

Example: None

Comments: None

6.5. Structured Text (ST)

6.5.1. Define General formatting rules

Identifier: Rule L4

Importance: Low

Targeted languages: Structured Text

References: none

Description: Define general formatting rules and use them coherently in the program.

Guideline:

Section A:

- Put a single space character around infix operators (like + and *) and assignment except when used in formal parameter lists.
- For non-keyword unary operators put no space between operand and operator but put a space on the other side.
- No space after opening '**(**' or before closing parenthesis '**)**'
- One space before '(' and one after ')' in expressions
- No space before '(' and after ')' in calls
- No space before semi-colon '**;**'
- No space before colon '**:**' . One space after
- No white-space characters at the end of a line.
- Always place a space after a comma '**,**' that does not end a line.
- In an array there is no space before and after the brackets []

Section B:

- When multiple parameters are put on multiple lines they should start in the same column.
- When line length is not exceeded put THEN/DO etc. on the same line as IF/WHILE
- When an expression needs to be put on multiple lines start the following lines with the operator.

Section C:

- Use AND and not '**&**'
- Use TRUE and FALSE not 0 and 1 for Boolean values.

Reasoning: Coherent formatting makes it easier to read a program. There is no OR equivalence to “&” for AND. In the first edition of the standard keywords where in upper-case

Exceptions: None

Example:

Don't:

```
if pump.temperature>=90&pump.running
then
    pump.speedmode:= slow ;
```

```
end_if ;
```

Do:

```
IF (Pump.Temperature >= 90) AND Pump.Running THEN
    Pump.SpeedMode := SLOW;
END_IF;
```

Comments: None

6.5.2. Usage of Continue and Exit instruction should be avoided

Identifier: Rule L10

Importance: Medium

Targeted languages: Structured Text

References:

- Misra C 14.5
- JSF++ Rules 189, 190

Description: Execution branching instructions CONTINUE and EXIT should not be used in Structured Text. This also includes any implementation specific instructions like GOTO or JUMP

Guideline: Most execution branching instructions can be replaced with structured instruction like conditional instructions (IF THEN ELSE or CASE) or loop instructions (WHILE, FOR, REPEAT).

Reasoning: Maintaining and debugging code containing sequence breaking instructions like CONTINUE and EXIT is more difficult to understand. Replacing those instructions with more structured instructions makes the code more readable and maintainable.

Exceptions: There are instances where the use of EXIT can enhance clarity and/or performance, though these are rare and so caution should always be exercised in their use. In FOR loops it can be necessary to use additional exit conditions. For CONTINUE this can be applicable to avoid deeply nested IF statements.

Examples:

Rather than using CONTINUE within a loop to skip code execution, use a conditional block to execute it only when needed, for example:

Don't:

```
// Count number of elements in error
Count:= 0;
FOR index:= 1 TO 20 DO
    IF NOT bError[index] THEN
        // Not in error so skip to next element
        CONTINUE;
    END_IF;
     Count:= Count + 1;
END_FOR;
```

Do:

```
// Count number of elements in error
Count:= 0;
FOR index:= 1 TO 20 DO
    IF bError[index] THEN
        // Another element in error
        Count:= Count + 1;
     END_IF;
END_FOR;
```

FOR loops using EXIT can iterate less times than expected. Rewrite the loop to expect an early exit:

Don't:

```
J:= 101;
FOR index := 1 TO 100 DO
    IF WORDS[index] = 'KEY' THEN
        J:= index;
        EXIT;
    END_IF;
END_FOR;
```

Do:

```
index:= 1;
WHILE index <= 100 AND WORDS[index] <> 'KEY' DO
    index:= index+1;
END_WHILE;
J:=index;
```

6.5.3. Define the maximum line length

Identifier: Rule L11

Importance: Medium

Targeted languages: Structured Text

References:

- JSF++ 41

Description: You should define the maximum line length

Guideline: Use a maximum line length for ST of 80 characters

Reasoning: Although some tools and compilers can support very long line lengths, it proves very difficult to read. It either requires lots of panning and scrolling (which loses the context and indentation level) or zooming, which can make the text too small to read. Also, any deep nesting is an indication to break down the design of your code using other POUs.
For maximum portability and readability, define your maximum line length as a value no greater than 80.

Exceptions:

Don't:

```
translatePwr (power := maxEse, powerFactor := 10.0, mode := mode,
const := const, clusterVoltage := clusterVoltage, current =>
maxCurrentEse);
```

Do:

```
translatePwr (power := maxEse,
              powerFactor := 10.0,
              mode := mode,
              const := const,
              clusterVoltage := clusterVoltage,
              current => maxCurrentEse);
```

Example, use case: None

Comments: None

6.5.4. Loop variables should not be modified inside a FOR loop

Identifier: Rule L12

Importance: Medium

Targeted languages: ST

References:

- IEC61131-3 (ed3) 7.3.3.4.2
- JSF++ AV Rule 188
- MISRA Rule 13.6
- CoDeSYS SA0072: Invalid use of counter variable

Description: The standard states "The control variable, initial value, and final value should be expressions of the same integer type and shall not be altered by any of the repeated statements."

Guideline: Modifying loop variables inside a FOR loop is forbidden

Reasoning: The FOR statement is used if the number of iterations can be determined in advance; otherwise, the WHILE or REPEAT constructs are used. Modifying during execution can cause unexpected behavior, including infinite loops, and can be difficult to debug and maintain.

Exceptions: None

Example:

Don't:

```
// This loop is not fine

j:= 11;

FOR i := 0 TO 10 BY 2 DO

    IF WORDS[i] = 'Key' THEN
       j := i ;
       i := 10;  // The loop variable should not be written here
    END_IF;

END_FOR;
```

Do:

```
// This loop is fine

i:= 0;

WHILE i <= 10 AND WORDS[i] <> 'KEY' DO

      i := i + 2;

END_WHILE;

j := i;
```

Comments: After the execution of the loop the value of the control variable is implementation specific.

6.5.5. FOR loop variable usage should not be used outside the FOR loop

Identifier: Rule L13

Importance: Medium

Targeted languages: ST

References:

- IEC61131-3 (ed3) 7.3.3.4.2
- JSF++ Av Rule 136

Description: The IEC 61131-3 standard states: "The value of the control variable after completion of the FOR loop is Implementer specific."

Guideline: Don't use the control variable outside the FOR loop, otherwise, the WHILE or REPEAT constructs can be used.

Reasoning: Since the value of the control variable outside the loop is implementation depended using it would make the code non portable.

Exceptions: None

Example:

Don't:

```
FOR i := 0 TO 100 BY 2 DO

    IF Words[ i] = 'Key' THEN
        EXIT;
    END_IF;

END_FOR;

IF i <= 100 THEN      // value of i is not defined here
    Words[i + 1] := value;
END_IF;
```

Do:

```
// This loop is fine
KeyAtIndex := 101;
FOR i := 0 TO 100 BY 2 DO

    IF Words[i] = 'Key' THEN
        KeyAtIndex := i;
    END_IF;

END_FOR;

IF KeyAtIndex <= 100 THEN
    Words[KeyAtIndex + 1] := Value;
```

```
END_IF;
```

Alternatively, a WHILE loop can be used:

```
i:= 0;
WHILE i <= 100 AND WORDS[i] <> 'KEY' DO
      i := i + 2;
END_WHILE;
IF i <= 100 THEN
      Words[i + 1] := Value;
END_IF;
```

Comments: None

6.5.6. Passing parameters should be clear

Identifier: Rule L14

Importance: Medium

Targeted languages: Structured Text

References:

- IEC 61131-3 Section 2.5.1.1
- IEC 61131-8 Section 3.2.3
- IEC 61508 C.2.9
- JSF++ 58

Description: Passing parameters in ST to Functions and Function Block invocations and method calls should be clear: which parameters are which, and whether they are input or output.

Guideline:

- For "Standard" IEC Functions and Function Blocks use the non-formal invocation type
- For User Defined Functions and Function Blocks use the IEC formal invocation type to name each argument, and use := and => operators to highlight input or output usage and sort into Inputs, Outputs and In-Out parameters
- When parameters have to be omitted or the quantity of parameters is large then use the formal invocation type with each parameter written on a separate line. Each line should have a comment describing the *argument* and its usage (not to be confused with the generic parameter comment in the FB definition).

Reasoning: Using meaningful parameter names makes the code unambiguous. Using the := and => operators also forces developers to consider which variable they expect to get overwritten. However to avoid long lines, and aid readability multiple lines are sometimes needed.

All the Standard IEC Functions (like Numerical and Arithmetic Functions) should be portable and recognizable, so IEC 61131 Section 2.5.1.1 recommends the non-formal invocation type is used.

Exceptions: None

Example:

Don't:

```
// Avoid the non-formal invocation type for custom FBs and long
invocations
A := MyFunction(Bee, 5);
MC_Home(Axis:=var0, Execute:=var1, Done=>var2, Busy=>var3,
CommandAborted=>var4, Error=>var5, ErrorID=>var6);
```

Do:

```
// Use formal argument list: MyFunction will not modify Bee

A := MyFunction(In:=Bee, Max:=5);

MC_Home(Axis:=ShuttleAxis,           // Home the shuttle axis
        Execute:=DoHome,             // at the timing DoHome
        Done=>HomeDone,              // shuttle at home position
        Busy=>ShuttleMoving,          // shuttle is currently homing
        CommandAborted=>HomeAborted, // shuttle homing aborted
        Error=>HomeError,             // error during operation
        ErrorID=>HomeErrorID);        // error code if error flag set
```

Comments: None

6.5.7. Use parenthesis to explicitly express operation precedence

Identifier: Rule L15

Importance: Medium

Targeted languages: Structured Text

References:

- Misra C 12.1
- JSF++ Rule 213

Description: When using operators with a similar precedence, use parenthesis to disambiguate.

Guideline: In a structured text expression using operators AND/OR/= or +/-, use parenthesis to specify explicitly the order of evaluation.

Reasoning: The precedence of operators may change between implementations. When a developer looks at an ambiguous expression, time may be lost due to assumptions about precedence.

Exceptions: None

Example:

Don't:

```
IF A AND B OR C AND D THEN
   ...
END_IF;
```

Do:

```
IF (A AND B) OR (C AND D) THEN
   ...
END_IF;
```

or

```
IF A AND (B OR C) AND D THEN
   ...
END_IF;
```

Comments: None

6.5.8. Define the use of tabs

Identifier: Rule L16

Importance: Low

Targeted languages: Structured Text

References:

- JSF++ 43

Description: You may define your use of tab characters and use consistently throughout the project.

Guideline: Use of tab character (ASCII code 9) should be avoided, and Programming Support Environment set to replace tabs with spaces

Reasoning: The visual interpretation of a tab character varies wildly. It can be interpreted as 8 spaces, or 4 spaces, or undefined spaces until the next tab stop. Therefore for maximum portability, it is recommended to not use tab characters.
Usage of tabs should be done consistently throughout the code and consistently with the Programming Support Environment behavior.

Exceptions: None

Example, use case: None

Comments: Depending on the Programming Support Environment, this may not be the same as using the tab Key on the keyboard. If possible, you should define the Programming Support Environment to replace use spaces for the tab key.

6.5.9. Each IF instruction should have an ELSE clause

Identifier: Rule L17

Importance: Low, High in critical software

Targeted languages: Structured Text

References:

- Codesys SA0075
- Itris Automation Square S12
- Misra C 14.10
- JSF++ Rule 192

Description: For every IF instruction in the code, an ELSE clause should be present.

Guideline: For each IF instruction in the structured text, the developer should add an ELSE clause to ensure that all cases are managed.

Reasoning: It is defensive programming. The developer should always take into account what will happen if the condition is False. This rule should be a requirement for safety critical software.

Exceptions: Development of non-safety critical software

Example, use case:

Don't:

```
IF some_condition THEN
  // some code
  ...
END_IF;
```

Do:

```
IF some_condition THEN
  // some code
  ...
ELSE
  // Nothing needs to happen in this case ! (Deliberate choice)
  ;
END_IF;
```

Comments: None

7. Vendor Specific IEC 61131-3 Extensions

7.1. Dynamic memory allocation shall not be used

Identifier: Rule E1

Importance: High

Targeted languages: All

References:

- =>Misra C 20.4

Description: The application shall not rely on dynamic allocation feature supplied by a PLC nor implement its own memory allocation mechanism.

Guideline: Dynamic allocation is forbidden.

Reasoning: Dynamic memory allocation has undefined, undocumented, implementation defined behavior. It may lead to memory leaks, data inconsistency, memory exhaustion, non-deterministic behavior.

Exceptions: None

Example: None

Comments: None

7.2. Pointer arithmetic shall not be used

Identifier: Rule E2

Importance: High

Targeted languages: ST, IL

References:

- IEC61131-3: not allowed:
  no "pointers" defined in 2nd and 3rd edition of IEC 61131-3,
  in 3rd edition "reference" defined, only with comparison with NULL and assignment operations, see Table 12)
- MISRA C 17,1

Description: The only allowed operators for manipulating pointers are equality and inequality. Developer shall not use pointer arithmetic to calculate a position in a memory to access further data.

Guideline: Should be limited and used only for array access

Reasoning: Pointer arithmetic is implementation dependent and understanding such code requires a thorough understanding of PLC's internals. This practice was only acceptable due to the lack of high level abstractions like structure, function blocks and arrays, but now these are available, they should be used instead of pointer arithmetic. Moreover, as mentioned in rule N1, developers should not use physical addresses for manipulating data.

Exceptions: None

Example: None

Comments: None

7.3. Some comparator instructions shall not be used for pointer or reference manipulation

Identifier: Rule E3

Importance: High

Targeted languages: All

References:

- Codesys SA0061
- Misra-C 17.3

Description: <=, >=, < and > operators shall not be used on pointers or references. Only equality and differences operators are allowed. If the order is required, then developer shall use an explicit array so that the relative positioning of variables is known.

Guideline: Pointer equality and inequality are allowed - all other comparison operators are forbidden

Reasoning: This rule is related to others rules relative to variable mapping in memory. The test relies on knowledge of memory organization of the PLC. It is either undocumented or susceptible to change and it is difficult to maintain.

Exceptions: None

Example:

Don't:

```
// Wrong example
VAR
  x: POINTER;
  y: POINTER;
END_VAR
...
IF X^ < Y^ AND X < Y THEN  -- the comparison X and Y is not allowed
   TEMP := X^:
   X^ := Y^;
   Y^ := TEMP;
END_IF;
```

Do:

```
// Good example - the usage of array is more explicit and relies
// on wide used arrays
VAR
  Values : ARRAY [1..50] OF INT;
  IndexX : INT;
  IndexY : INT;
END_VAR
IF Values[IndexX] < Values[IndexY] AND IndexX < IndexY THEN
  TEMP := Values[IndexX];
  Values[IndexX] := Values[IndexY];
  Values[IndexY] := Temp;
END_IF;
```

Comments: None

8. Annex 1 – overview of the rules via their priorities

Priority	*Rule #*	*Chapter*	*Name*	*Page*
High	N1	3.1.1.	Avoid physical addresses	16
High	N3	3.2.1.	Define the names to avoid	20
High	N4	3.2.2.	Define the use of case (capitals)	22
High	N5	3.2.3.	Local names shall not shadow global names	25
High	C1	4.1.	Comments shall describe the intention of the code	36
High	C2	4.2.	All elements shall be commented	38
High	CP1	5.1.	Access to a member shall be by name	44
High	CP2	5.2.	All code shall be used in the application	45
High	CP3	5.3.	All variables shall be initialized before being used	47
High	CP4	5.4.	Direct addressing should not overlap	51
High	CP5	5.5.	Applications shall be well designed	53
High	CP6	5.6.	Avoid external variables in functions, function blocks and classes	54
High	CP7	5.7.	Error information shall be tested	56
High	CP8	5.8.	Floating point comparison shall not be equality or inequality	58
High	CP28	5.9	Time and physical measures comparison shall not be equality or inequality	59
High	CP9	5.10.	Limit the complexity of POU-code	60
High	CP10	5.11.	Avoid multiple writes from multiple tasks	63
High	CP11	5.12.	Manage synchronization among tasks	65
High	CP12	5.13.	Physical outputs shall be written once per PLC cycle	68
High	CP13	5.14.	POUs shall not call themselves directly or indirectly	69
High	CP14	5.15.	POUs shall have a single point of exit	71
High	CP15	5.16.	Read a variable written by another task only once per cycle	72
High	CP16	5.17	Tasks shall only call program POUs and not Function Blocks	74
High	CP17	5.18.	Usage of parameters shall match their declaration mode	75
High	CP18	5.19.	Use of global variables shall be limited	77
High	L7	6.5.1.	Closing divergent paths	106
High	E1	7.1.	Dynamic memory allocation shall not be used	122
High	E2	7.2.	Pointer arithmetic shall not be used	123
High	E3	7.3.	Some comparator instructions shall not be used for pointers or reference manipulation	124
Medium	N6	3.2.4.	Define an acceptable name length	27
Medium	N7	3.2.5.	Define naming rules for namespaces	29
Medium	N8	3.2.6.	Define the acceptable character set	31
Medium	N9	3.2.7.	Different element types should not bear the same name	32
Medium	CP19	5.20.	Usage of jump and return should be avoided	81
Medium	CP20	5.21.	Function block instances should be called only once	84

Medium	CP21	5.22.	Use VAR_TEMP for temporary variable declaration	86
Medium	CP22	5.23.	Select appropriate data type	88
Medium	CP23	5.24.	Define maximum number of input/output/in-out variables of a POU	91
Medium	CP24	5.25.	Do not declare variables that are not used	94
Medium	CP25	5.26.	Data types conversion shall be explicit	95
Medium	L2	6.2.1.	Avoid assignments of intermediate results within networks	100
Medium	L3	6.2.2.	Define maximum complexity of single network	101
Medium	L5	6.4.1.	A coil should not be followed by a contact	102
Medium	L6	6.4.2.	Define maximum rung complexity	103
Medium	L8	6.5.2.	Do not program an SFC action block in SFC	106
Medium	L9	6.5.3.	Define maximum complexity	107
Medium	L10	6.6.1.	Usage of Continue and Exit instruction should be avoided	110
Medium	L11	6.6.2.	Define the maximum line length	112
Medium	L22	6.6.3.	Loop variables should not be modified inside a FOR loop	113
Medium	L13	6.6.4.	FOR loop variable usage should not be used outside the FOR loop	115
Medium	L14	6.6.5.	Passing parameters should be clear	117
Medium	L15	6.6.6.	Use parenthesis to explicitly express operation precedence	119
Low	N2	3.1.2.	Define type prefixes for Variables (if used)	17
Low	N10	3.2.8.	Define name prefixes for user defined types	33
Low	C3	4.3.	Avoid nested comments	39
Low	C4	4.4.	Comments may not include code	40
Low	C5	4.5.	Use single line comments	41
Low	C6	4.6.	Define comments language	43
Low	CP26	5.27.	A global variable may be written only by one PROGRAM	97
Low	CP27	5.28.	Avoid deprecated features	98
Low	L1	6.1.	Define indentation	99
Low	L4	6.3.	Define general formatting rules	102
Low	L16	6.6.7.	Define the use of tabs	120
Low	L17	6.6.8.	Each IF instruction should have an ELSE clause	121