PLCopen®

for efficiency in automation

Техническая документация PLCopen

рабочая группа

Создание руководств по разработке ПО

представляет

Руководство по кодированию

версия 1.0, официальный релиз

Дата публикации: 20.04.2016

Переводчик:	Е.Кислов
Редактор:	А.Осинский
Версия перевода:	1.1
Дата публикации:	04.06.2018

Оглавление

Список авторов и история версий

Данный документ является официальным документом организации **PLCopen**:

Руководство по кодированию

Он является результатом работы комитета **Разработка руководств по кодированию** рабочей группы **Создание руководств по разработке ПО** и включает вклад каждого из участников:

Участник	**Компания**
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

История версий

Версия	**Дата**	**Описание**
0.1	24.03.2015	Начало работы над документом.
0.2	05.06.2015	Дополнения по результатам видеоконференции, прошедшей 1-го июня.
0.3	01.07.2015	Дополнения по результатам последней видеоконференции.
0.99	23.07.2015	Версия для получения комментариев сообщества (до 23 октября).
0.99A	17.01.2016	Дополнения по результатам встречи в Франкфурте.
0.99B	20.01.2016	Исправление последних замечаний и редакторская правка.
1.0	20.04.2016	Официальный релиз.

Глоссарий

FBD (*Function Block Diagram, язык функциональных блоков*) – графический язык программирования ПЛК;

IL (*Instruction List, список инструкций*) – текстовый язык программирования ПЛК, напоминающий язык Ассемблера;

LD (*Ladder Diagram, язык релейных схем*) – графический язык программирования ПЛК;

SFC (*Sequential Function Chart, язык последовательных схем*) – высокоуровневый графический язык программирования ПЛК, представляющий приложение в виде конечного автомата;

ST (*Structured Text, структурированный текст*) – высокоуровневый текстовый язык программирования ПЛК, напоминающий Паскаль;

POU – обобщенное название пользовательских программных компонентов – функций, функциональных блоков и программ;

ФБ – функциональный блок.

1. Введение

Существует множество книг, посвященных вопросам разработки ПО на различных языках программирования, но лишь единицы из них затрагивают аспекты программирования в области промышленной автоматизации (в частности, программирования на языках стандарта МЭК 61131-3).

При этом роль программного обеспечения в управлении технологическими процессами становится всё более значительной, и вместе с этим растет сложность используемого ПО и критичность последствий его ошибок. В настоящее время затраты на промышленное ПО составляют до 50% стоимости внедрения АСУ и 40-80% стоимости ее сопровождения.

Разработка сложного ПО неразрывно связана с его проектированием. В значительном количестве случаев применяется структурный подход к проектированию ПО. Кроме того, эффективность ПО можно повысить путем повторного использования кода и соблюдением требований к его оформлению.

Организация PLCopen пригласила заинтересованные компании присоединиться к рабочей группе **Создание руководств по разработке ПО**. В результате установочной встречи были сформированы несколько комитетов, каждый из которых стал заниматься созданием руководств по выбранной тематике:

- кодированию;
- проблемам качества и согласованности ПО;
- созданию функциональных блоков, совместимых с моделью поведения **PLCopen**;
- использованию языка SFC;
- документированию ПО;
- использованию библиотек;
- организации процесса разработки ПО.

Основной задачей рабочей группы **Создание руководств по разработке ПО** является определение правил, примеров и советов по созданию программного обеспечения для промышленной автоматизации. Эти правила будут опубликованы в виде отдельных документов и размещены на сайте PLCopen.

Правила базируются на 1-й и 2-й редакциях стандарта **МЭК 61131-3**, но могут быть легко адаптированы под 3-ю редакцию, которая была опубликована в феврале 2013.

Целью комитета **Разработка руководств по кодированию** является создание набора правил программирования и примеров их использования. В настоящее время крупные компании имеют собственные стандарты разработки ПО, но для небольших и средних фирм, а также для новичков в программировании ПЛК, данный документ будет крайне полезен. Подобные документы повысят популярность стандарта **МЭК 61131-3**. Учебные заведения могут использовать их для ознакомления студентов с эффективными способами разработки промышленного ПО.

2. Как использовать этот документ

Стандарт **МЭК 61131-3** определяет языки и методики программирования, что является важным шагом в развитии ПЛК. Его внедрение способствует повышению качества ПО, используемого для программирования контроллеров. Однако стандарт не описывает, как можно повысить качество прикладных программ. Стандарт **МЭК 61131-8** содержит некоторые рекомендации на эту тему, но не предоставляет всей необходимой информации. Целью данного руководства является заполнение этого пробела.

Документ предназначен для программистов ПЛК, которые стремятся повысить качество своего ПО. В нем не рассматриваются вопросы проектирования ПО и управления процессом разработки; эти темы будут рассмотрены в других документах. Данное руководство содержит набор правил, которые программист может использовать при написании и редактировании кода. Этот набор не претендует на полноту, но является достаточным, чтобы повысить качество ПО. Пользователи и организации могут удалять и добавлять правила, адаптируя набор под свои потребности.

Рабочая группа рекомендует разработчикам сделать следующее:

1. Составить список критериев качества разрабатываемого ПО;
2. Проанализировать тип разрабатываемого ПО;
3. Определить дополнительные правила;
4. На основании предыдущих пунктов создать свою версию набора правил;
5. Использовать этот набор при разработке своих приложений.

Примечания:

1. Целью данного документа является повышение качества разрабатываемого ПО. Это требует от организации определенного уровня зрелости управления (см., например, Capability Maturity Model).

2. Стандарт **МЭК 61131-3** определяет несколько языков программирования, и если какие-то из них не используются при разработке – то касающиеся их правила могут быть, соответственно, проигнорированы.

3. Помимо этого документа есть и другие источники информации, рассматривающие вопросы качества ПО. Как правило, крупные организации имеют свои стандарты кодирования. Кроме того, существуют универсальные стандарты для разработки ПО (например, модель McCall). Но универсальные стандарты напрямую не применимы к ПЛК – что и послужило одной из причин создания данного документа.

4. После изучения данного документа программист может составить свою версию набора правил. Конечно же, можно использовать и полный набор.

2.1. История создания

Приведенная в данном документе информация является сочетанием

- общих правил, используемых при разработке ПО;
- правил компаний, сотрудниками которых являются участники рабочей группы;
- правил, созданных участниками рабочей группы на основе их опыта программирования ПЛК.

При создании документа использовались следующие источники:

- МЭК 61131-3
- Misra-C
- JSF++
- Codesys Online Help

Более подробная информация по источникам приведена в п. 2.4.

Первоначально была разработана таблица правил. Номер строки этой таблицы используется в качестве идентификатора правила в данном документе. После этого для каждого правила была создана wiki-страница, и в течение полутора лет рабочая группа занималась их наполнением. Содержание этого документа является сборником всех правил, которые обсуждались во время сессий.

Правила ориентированы на 3-ю редакцию **МЭК 61131-3**, но большинство из них также применимы для других редакций стандарта и даже за его пределами.

Рабочая группа крайне заинтересована в обратной связи от пользователей. В будущем документ будет дорабатываться путем добавления новых правил и уточнения существующих. Поэтому по возможности отправьте свой отзыв на электронный адрес, указанный на сайте plcopen.org.

2.2. Структура документа

Каждая из глав документа посвящена определенному аспекту программирования:

- Имена (**N**) – содержит правила выбора имен переменных и программных модулей;
- Комментарии (**C**) – содержит правила документирования кода;
- Правила кодирования (**CP**) – содержит правила написания кода;
- Языки программирования (**L**) – содержит правила, связанные с использованием конкретных языков программирования ПЛК;
- Правила для расширений стандарта МЭК 61131-3 (**E**) – содержит правила, касающиеся использования функционала, не определенного стандартом МЭК и поддерживаемого не всеми производителями ПЛК.

Главы разбиты на подразделы. В рамках подразделов правила размещены в порядке понижения приоритета. Идентификаторы правил содержат префиксы, характеризующие главу документа (см. выше – **N**, **C** и т.д.). Каждая глава начинается с правила номер 1 (например, N**1**). При выпуске следующих версий документа новые правила будут добавляться в конец соответствующих глав. В приложении 1 приведена классификация правил по приоритету.

2.3. Формат описания правил

Каждое из правил в документе создано по универсальному шаблону. Пример и описание полей шаблона приведены ниже.

Идентификатор: XX

Приоритет: высокий

Языки программирования: все

Ссылки:

- MISRA-C-2004, п. 3.5

Описание: При работе с пользовательскими типами данных…

Применение: Не используйте обращение к элементам через их адреса.

Объяснение: Обращение к вложенному элементу через адрес памяти…

Исключения: нет

Примеры:

Неправильно:

```
%MW504 := 'E';
```

Правильно:

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

Комментарий: нет

- **Идентификатор** (*обязательно*) – короткий буквенно-цифровой код, облегчающий поиск нужного правила;
- **Приоритет** (*обязательно*) – степень важности правила в соответствие с эффектом, оказываемым им на качество ПО (высокий/средний/низкий);
- **Языки** (*обязательно*) – языки программирования стандарта **МЭК 61131-3**, на которые распространяется данное правило (LD/FBD/SFC/ST/все);
- **Ссылки** (*обязательно*) – если правило основано на информации из какого-либо источника, то приводится ссылка на этот источник;

- **Описание** (*обязательно*) – полное и ясное описание правила;
- **Применение** (*обязательно*) – руководство для разработчиков по применению данного правила;
- **Объяснение** (*обязательно*) – обоснование необходимости использования данного правила;
- **Исключения** (*необязательно*) – перечисление ситуаций, в которых не следует использовать данное правило;
- **Примеры** (*необязательно*) – наглядные примеры использования данного правила. Неправильный код выделен красным, правильный – зеленым;
- **Комментарий** (*необязательно*) – дополнительная информация (например, ссылка на другое правило).

Только поля **Исключения**, **Пример** и **Комментарий** являются необязательными к заполнению.

2.4. Ссылки

При создании данного руководства использовались следующие документы и стандарты:

МЭК 61131-3
Название: МЭК 61131-3 (редакции 2 и 3)
Заголовок: Программируемые контроллеры – часть 3: Языки программирования
Разработчик: Международная электротехническая комиссия
Web-сайт: http://www.iec.ch/index.htm

МЭК 61131-8
Название: МЭК 61131-8 (редакция 1, в будущем – также редакция 3)
Заголовок: Программируемые контроллеры – часть 8: Руководство по программированию
Разработчик: Международная электротехническая комиссия
Web-сайт: http://www.iec.ch/index.htm

MISRA-C
Название: Motor Industry Software Reliability Association C
Заголовок: Guidelines for the Use of the C Language in Vehicle Based Software
Редакция: 2005
Web-сайт: http://www.misra.org.uk/

JSF++ coding standard
Название: JSF++ coding standard
Заголовок: JSF Air Vehicle - C++ Coding Standards (Revision C)
Документ: 2RDU00001 Rev C. December 2005
Web-сайт: http://www.jsf.mil/downloads/down_documentation.htm

Codesys Online Help
Редакция: 2015
Web-сайт: http://store.codesys.com/codesys-static-analysis.html

3. Имена

3.1. Переменные

3.1.1. Не используйте физические адреса

Идентификатор: N1

Приоритет: высокий

Языки программирования: LD, ST, SFC, FBD

Ссылки:

- МЭК 61131-3, п. 6.5.5
- МЭК 61131-8, п. 3.11.2

Описание: Не используйте физические адреса (**%QW** и т.п.).

Применение: Не используйте физические адреса для прямого обращения к памяти – вместо этого следует объявить переменную.

Объяснение: Использование физических адресов снижает переносимость программы – потребуется редактировать ее для применения на ПЛК другого производителя (или даже на другом ПЛК того же производителя). Кроме того, это снижает читабельность кода, так как без таблицы соответствий невозможно понять, для чего был использован тот или иной адрес. В свою очередь, снижение читабельности увеличивает шанс появления ошибок при внесении изменений в код. Также стоит отметить, что физическая адресация памяти ввода-вывода может быть изменена при обновлении встроенного ПО (прошивки) ПЛК.

Исключения: При работе с некоторыми протоколами обмена использование физических адресов помогает определить нужный порядок данных – старшим байтом вперед и т.п. (если недоступны другие средства).

Примеры: нет

Комментарий: Рекомендуется всегда применять переменные области **VAR_ACCESS**, поскольку это позволяет обращаться к данным с помощью понятных имен.

3.1.2. Применяйте префиксы

Идентификатор: N2

Приоритет: низкий

Языки программирования: все

Ссылки:

- Википедия Венгерская нотация

Описание: Вы можете использовать любую удобную вам систему префиксов для имен переменных. Эти префиксы позволяют закодировать в имена какую-либо дополнительную информацию, например:

- тип данных;
- область видимости;
- область переменных (входы, выходы);
- зону объекта автоматизации (зона 1, зона 2 и т.д.).

Применение: Если вы используете префиксы – задокументируйте свою систему. Один из вариантов – использовать *венгерскую нотацию*, которая позволяет закодировать в имени переменной ее тип. Выберите один или несколько подходов из приведенных ниже примеров и последовательно применяйте их. Обратите внимание – один и тот же префикс может использоваться в разных обозначениях. В этом случае следует применять составные префиксы.

Объяснение: Кодирование дополнительной информации в именах переменных позволяет повысить читабельность кода и избежать ошибок программирования. Например, кодирование типа данных помогает выбрать нужный оператор конверсии и избежать ошибок, связанных с потерей знака. Кодирование области видимости позволяет при отладке легко отследить глобальные переменные – они могут быть изменены в любом POU, что зачастую приводит к различным проблемам. Кодирование области переменных поможет избежать ошибок, связанных с попытками присвоения значений входным переменным. Также при работе над крупными приложениями (особенно если над ними работает несколько программистов) существует тенденция к группировке связанных переменных с помощью общей части названия (например, *InfeedSpeed, InfeedAlarm, InfeedStatus*). Этот подход терпит неудачу, если заранее не определены допустимые префиксы (в результате можно найти в коде переменные с названиями типа *Infeed_Speed, In_Feed_Alarm, ifStatus, inFault* и т.д.). Однако кодирование с помощью префиксов значительного количества информации делает имена длинными и плохо читаемыми. Наиболее полезным (и широко используемым) является кодирование в префиксах типов данных переменных.

Исключения: Можно отказаться от использования префиксов, если среда разработки предоставляет всю необходимую информацию другими средствами (например, с помощью всплывающих подсказок).

Примеры:

Пример использования венгерской нотации для кодирования с помощью префиксов типов данных переменных в соответствии с 3-й редакцией стандарта **МЭК 61131-3**:

Тип данных	Префикс
BOOL	x
SINT	si
INT	i
DINT	di
LINT	li
USINT	usi
UINT	ui
UDINT	udi
ULINT	uli
REAL	r
LREAL	lr
TIME	t
LTIME	lt
DATE	dt[1]
LDATE	ldt[1]
TIME_OF_DAY/TOD	tod
LIME_OF_DAY/LTOD	ltod
DATE_AND_TIME/DT	dt[1]
LDATE_AND_TIME/LDT	ldt[1]
STRING	s
WSTRING	ws
CHAR	c
WCHAR	wc
BYTE	by
WORD	w
DWORD	dw
LWORD	lw

Пример кодирования безопасных (**SAFE**) типов данных (по спецификации **PLCopen Safety**): ***s*** в качестве последней буквы каждого префикса. Пример: ***xs*** (для SAFE BOOL), ***is*** (для SAFE INT) и т.д. В большинстве случаев ПО с поддержкой безопасных типов данных уже обеспечивает их выделение (например, цветом), так что такое кодирование может не потребоваться.

[1] В исходном тексте руководства префиксы этих типов действительно совпадают (прим. пер.).

Пример кодирования пользовательских типов данных и POU (табл. 11 из 3-й редакции стандарта **МЭК 61131-3**):

Тип данных	Префикс
ENUM	e
NAMED	e
SUBRANGE	sb
ARRAY	a
STRUCT	st
Type STRUCT	ts
REFERENCE	ref
FUNCTION BLOCK	fb
PROGRAM	prg
CLASS	cls

Примечание: ENUM и NAMED используют одинаковый префикс.

В качестве альтернативы для всех пользовательских типов данных может использоваться префикс *udt*.

Пример использования префиксов для кодирования области видимости:

Пространство имен	Префикс
Глобальные переменные	g
Локальные переменные	l
Параметры POU	p
Временные переменные	tmp

Пример использования префиксов для кодирования области переменных:

Область	Префикс
Входы (только чтение)	i
Выходы (чтение/запись)	o

Комментарий: нет

3.2. Задачи, программы, функциональные блоки, функции, переменные, пользовательские типы данных и пространства имен

3.2.1. Определите имена, которые нельзя использовать

Идентификатор: N3

Приоритет: высокий

Языки программирования: все

Ссылки:

- МЭК 61131-3, п. 6.1.3
- MISRA-C-2004, п. 20.1

Описание: Необходимо определить глоссарий названий, которые нельзя использовать в качестве имен переменных и компонентов.

Применение: Запрещается использовать в качестве названий:

- Названия типов данных стандарта **МЭК 61131-3** и модулей библиотеки **Standard**;
- Ключевых слов языка ST;
- Ключевых слов используемой среды разработки.

Не рекомендуется использовать в качестве названий:

- Специфические, не общепринятые аббревиатуры (без их четкого документирования);
- Общие слова, не указывающие на индивидуальные свойства объекта (например, Info, Data, Temp, Str, Buf).

Объяснение: В большинстве случаев ключевые слова зарезервированы и их использование приводит к ошибкам компиляции. Кроме того, даже если ошибок не возникает, некорректное применение ключевых слов снижает читабельность кода и затрудняет сопровождение программы. При создании платформо-независимого приложения следует отказаться от использования ключевых слов всех потенциально возможных платформ. Использование сокращений и трехбуквенных акронимов также плохо отражается на читабельности. При необходимости следует определить список допустимых сокращений (например, Max, Min, Temp, IP) и общепринятых для конкретной отрасли аббревиатур.

Исключения: нет

Примеры:

Список ключевых слов 3-й редакции стандарта **МЭК 61131-3**:

ABS	END_IF		REF_TO	UINT
ABSTRACT	END_INTERFACE	LEFT	REPEAT	ULINT
ACOS	END_METHOD	LEN	REPLACE	UNTIL
ACTION	END_NAMESPACE	LIMIT	RESOURSE	USING
ADD	END_PROGRAM	LINT	RETAIN	USINT
AND	END_REPEAT	LN	RETURN	VAR
ARRAY	END_RESOURSE	LOG	RIGHT	VAR_ACCESS
ASIN	END_STEP	LREAL	ROL	VAR_CONFIG
AT	END_STRUCT	LT	ROR	VAR_EXTERNAL
ATAN	END_TRANSITION	LTIME	RS	VAR_GLOBAL
ATAN2	END_TYPE	LTIME_OF_DAY	SEL	VAR_IN_OUT
BOOL	END_VAR	LTOD	SHL	VAR_INPUT
BY	END_WHILE	LWORD	SHR	VAR_OUTPUT
BYTE	EQ	MAX	SIN	VAR_TEMP
CASE	EXIT	METHOD	SINGLE	WCHAR
CHAR	EXP	MID	SINT	WHILE
CLASS	EXPT	MIN	SQRT	WITH
CONCAT	EXTENDS	MOD	SR	WORD
CONFIGURATION	F_EDGE	MOVE	STEP	WSTRING
CONSTANT	F_TRIG	MUL	STRING	XOR
CONTINUE	FALSE	MUX	STRING#	
COS	FINAL	NAMESPACE	STRUCT	
CTD	FIND	NE	SUB	
CTU	FOR	NON_RETAIN	SUPER	
CTUD	FROM	NOT	T	
DATE	FUNCTION	NULL	TAN	
DATE_AND_TIME	FUNCTION_BLOCK	OF	TASK	
DELETE	GE	ON	THEN	
DINT	GT	OR	THIS	
DIV	IF	OVERLAP	THIS	
DO	IMPLEMENTS	OVERRIDE	TIME	
DT	INITIAL_STEP	PRIORITY	TIME_OF_DAY	
DWORD	INSERT	PRIVATE	TO	
ELSE	INT	PROGRAM	TOD	
ELSIF	INTERFACE	PROTECTED	TOF	
END_ACTION	INTERNAL	PUBLIC	TON	
END_CASE	INTERVAL	R_EDGE	TP	
END_CLASS	LD	R_TRIG	TRANSITION	
END_CONFIGURATION	LDATE	READ_ONLY	TRUE	
END_FOR	LDATE_AND_TIME	READ_WRITE	TRUNC	
END_FUNCTION	LDT	REAL	TYPE	
END_FUNCTION_BLOCK	LE	REF	UDINT	

Комментарий: нет

3.2.2. Определите регистр текста составных названий

Идентификатор: N4

Приоритет: высокий

Языки программирования: LD, ST, SFC, FBD

Ссылки:

- МЭК 61131-3, п. 6.1.2
- MISRA-C-2004, п. 1.4
- GNU 5.4
- Java section 9
- Википедия CamelCase

Описание: Необходимо определить регистр текста составных названий для всех типов объектов проекта. Примерами таких регистров могут служить:

- alllowercase
- lower_snake_case
- lowerCamelCase
- UpperCamelCase
- ALLUPPERCASE
- UPPER_SNAKE_CASE
- OTHER_CASE

Применение: Используйте один и тот же стиль для каждого экземпляра определенного типа объектов, даже если компилятор не делает различий между регистрами. Рекомендуется:

- Использовать UPPER_SNAKE_CASE для констант, пользовательских типов данных и ключевых слов (например, BOOL, END_FOR);
- Использовать UpperCamelCase для всех остальных составных названий.

Объяснение: Регистр текста не всегда учитывается компилятором, но тем не менее может использоваться для повышения читабельности кода. Поскольку в стандарте **МЭК 61131-3** названия нечувствительны к регистру, то стиль написания не влияет на переносимость. *CamelCase* облегчает чтение длинных составных названий и проще в наборе, чем snake_case. *UPPER_SNAKE_CASE* традиционно применяется для написания ключевых слов. Эти принципы характерны и для других стандартов оформления кода.

Исключения: При использовании префиксов (см. правило N2) вместо *UpperCamelCase* следует использовать *lowerCamelCase.*

Примеры:

Неправильно:

```
STARTMOTOR();        // можно спутать с константой
startmotor();        // сложно прочитать
```

Правильно:

```
StartMotor();        // легко прочитать
```

Неправильно:

Объявление структуры (МЭК 61131-3, п. 6.4.4.6.1)

```
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
```

Правильно:

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

Правильно:

В спецификации PLCopen Motion Control с 2010 года принят следующий стиль именования ФБ и типов данных:

Функциональные блоки: префикс MC_ и *CamelCase*
Пример: MC_MoveAbsolute

Перечисления: префикс mc и *CamelCase*
Примеры: mcPositive, mcBlendingLow.

Типы данных и структуры: префикс MC_ (кроме AXIS_REF) и *UPPER_SNAKE_CASE*.
Примеры: AXIS_REF, MC_BUFFER_MODE, MC_TP_REF, MC_INPUT_REF.

Входы и выходы*: CamelCase* без префиксов.
Примеры: Busy, CommandAborted, Master, BufferMode.

Комментарий: нет

3.2.3. Имена глобальных и локальных объектов не должны совпадать

Идентификатор: N5

Приоритет: высокий

Языки программирования: все

Ссылки:

- МЭК 61131-3, п. 6.9.1
- MISRA-C-2004
- JSF++ 135

Описание: Локальные и глобальные объекты не должны иметь одинаковые имена – это приводит к тому, что в пределах POU становится невозможен доступ к таким глобальным объектам.

Применение: Не используйте одинаковые имена для локальных и глобальных объектов. При необходимости применяйте префиксы для кодирования области видимости (см. правило N2).

Объяснение: Использование одного и того же имени для разных объектов снижает читабельность кода. Компилятор может считать это синтаксически корректным, но подобный подход затрудняет понимание программы и является потенциальной причиной ошибок. Кроме того, это негативно влияет на переносимость приложения. Даже при использовании разных пространств имен следует избегать подобных ситуаций.

Исключения: нет

Примеры:

Неправильно:

```
VAR_GLOBAL
        MyCalculation: REAL;            // глобальная переменная
END_VAR

FUNCTION_BLOCK MyFirstCalculation
        VAR
                MyCalculation: REAL; // локальная переменная ФБ
        END_VAR

                MyCalculation := 1.1;  // обращение к локальной переменной
                ...
END_FUNCTION_BLOCK

FUNCTION_BLOCK MySimilarCalculation
        VAR_EXTERNAL
                MyCalculation: REAL; // «импорт» глобальной переменной
         END_VAR

        MyCalculation := 1.1;           // обращение к глобальной переменной
        …
END_FUNCTION_BLOCK
```

Правильно:

```
// Используйте уникальное имя для глобальной переменной

VAR_GLOBAL
         GlobalCalculationResult: REAL;   // глобальная переменная
END_VAR
.....
```

Комментарий: нет

3.2.4. Определите приемлемую длину имен

Идентификатор: N6

Приоритет: средний

Языки программирования: LD, ST, SFC, FBD

Ссылки:

- МЭК 61131-3, п. 6.1.2
- MISRA-C-2004, п. 5.1
- JSF++ 46
- GNU 5.4
- Java Section 9

Описание: Имена любых объектов должны иметь приемлемую длину.

Применение:

- Минимальная длина: 8 символов;
- Максимальная длина: 25 символов;
- Средняя длина: старайтесь ограничиться 15 символами;
- Используйте сокращения только в тех случаях, когда вы уверены, что они будут понятны читателю;
- Избегайте похожих имен, которые легко перепутать (например, *var* и *var**s***) и имен, отличающихся лишь регистром.

Объяснение: Использование имен приблизительно одинаковой длины облегчает чтение кода. Ограничение на минимальную длину позволяет исключить бессмысленные имена, ограничение на максимальную длину – чрезмерно длинные. Кроме того, некоторые среды разработки обрезают слишком длинные имена.

Исключения: Допустимо не использовать данное правило для переменных, имеющих крайне ограниченную область видимости:

- Счетчики циклов и индексы традиционно имеют очень короткие имена (*i, j, k*);
- Элементы структур могут иметь короткие имена, так как используются совместно с пространствами имен.

Примеры:

Неправильно:

```
FUNCTION Go : BOOL; // слишком коротко
…

FUNCTION aaa: INT; // лишено смысла
…

FUNCTION ReadAndScaleTheTemperatureInput: REAL; // слишком длинно
…
```

Правильно:

```
FUNCTION StartFeeding: BOOL;  // имя описывает назначение функции
…

FUNCTION ReadTemperature: REAL;  // имя описывает назначение функции
…
```

Неправильно:

```
VAR
        Go:     BOOL;   // слишком коротко
        aaa:    INT;    // лишено смысла
        MaximumTemperatureForTheThermocoupleInput: REAL; // слишком длинно
END_VAR;
```

Правильно:

```
VAR
         i: INT;  // только для счетчиков и индексов

        MaxTcTemperature: REAL;  // только при использовании очевидных
                                 // сокращений (Tc  - Thermocouple)
END_VAR;
```

Комментарий: нет

3.2.5. Определите правила использования пространств имен

Идентификатор: N7

Приоритет: средний

Языки программирования: все

Ссылки:

- МЭК 61131-3, п. 6.9

Описание: Следует определить правила использования пространств имен.

Применение:

- Названия пространств имен должны записываться в *ВерхнемВерблюжемРегистре*;
- При объявлении POU и пользовательских типов данных хотя бы одно пространство имен должно быть объявлено в глобальном пространстве имен. Никакие объекты, за исключением стандартных типов **МЭК 61131-3**, не должны объявляться напрямую в глобальном пространстве имен;
- Пространство имен первого уровня (по отношению к глобальному) должно являться именем компании, которая является разработчиком объектов, вложенных в это пространство;
- Названия пространств имен второго и более низких уровней определяются компаниями-разработчиками. Они могут содержать название компонента, его категорию и т.д.

Объяснение:

- Функции и ФБ, определенные стандартом **МЭК 61131-3**, расположены в глобальном пространстве имен, которое не имеет названия. Если не используется директива USING, то остальные пространства имен являются вложенными по отношению к глобальному. Это увеличивает шанс конфликта имен объектов;
- Различные компании разрабатывают библиотеки, которые потенциально могут содержать модули с одинаковыми именами. Для предотвращения конфликтов названий при совместном использовании библиотек каждая из них должна содержать свое пространство имен.

Исключения: Данное правило может не соблюдаться при использовании среды разработки, которая не поддерживает пространства имен.

Примеры:

Неправильно:

```
// следующие модули объявлены в глобальном пространстве имен
FUNCTION_BLOCK MyCalculation // 1)
...
END_FUNCTION_BLOCK

NAMESPACE SomeCompany.XseriesCPU
        FUNCTION_BLOCK MyCalculation // 2)
        ...
        END_FUNCTION_BLOCK

        PROGRAM MainProgram
                VAR MyCal : MyCalculation;
            // неочевидно,  какой ФБ будет использован при объявлении
            // для 2) следует использовать "SomeCompany.XseriesCPU.MyCalcuration"
            // невозможно явным образом указать, что используется 1)
                END_VAR
        END_PROGRAM
END_NAMESPACE
```

Правильно:

```
// модули объявляются в своих пространствах имен
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
```

Комментарий: нет

3.2.6. Определите используемый набор символов

Идентификатор: N8

Приоритет: средний

Языки программирования: все

Ссылки:

- МЭК 61131-3, п. 6.1.2
- MISRA-C-2004, п.3.2
- JSF++ 9

Описание: Следует определить набор символов, используемый для написания имен объектов и комментариев.

Применение:

- Имена объектов не должны начинаться с цифр;
- Имена объектов могут содержать только буквы, цифры и символ нижнего подчеркивания. Использование диакритических знаков запрещено;
- Имена объектов должны быть набраны в кодировке ASCII (7 бит).

Объяснение: Использование специальных символов снижает читабельность кода и делает приложение менее переносимым. Кроме того, использование в именах исключительно букв и цифр позволяет лучше различать их.

Исключения: Комментарии могут быть написаны на родном для разработчика языка. Если приложение будет применяться в конкретном государстве (например, Китае), то допускается использовать в именах объектов символы национальной кодировки.

Примеры:

Неправильно:

```
départ := TRUE;
// 100 строк кода…
depart := FALSE; // вы не заметите, что это другая переменная (без 'é')
```

Правильно:

```
depart := TRUE; // имя переменной не содержит спецсимволов
 // Autorise le départ du wagonnet (в комментариях спецсимволы допустимы)
```

Комментарий: нет

3.2.7. Объекты различных типов не должны иметь одинаковые имена

Идентификатор: N9

Приоритет: средний

Языки программирования: все

Ссылки:

- МЭК 61131-3, п. 6.9.1
- MISRA-C-2004
- JSF++ 135

Описание: Задачи, программы, функциональные блоки, функции, переменные и пользовательские типы данных не должны иметь совпадающие названия в пределах одного пространства имен.

Применение: Не используйте одинаковые имена для объектов разных типов.

Объяснение: Использование одинаковых имен для объектов разных типов затрудняет чтение кода. Компилятор может считать это синтаксически корректным, но подобный подход затрудняет понимание программы и является потенциальной причиной ошибок.

Исключения: нет

Примеры:

Неправильно:

```
VAR_GLOBAL
        MyCalculation: REAL; // глобальная переменная
END_VAR

FUNCTION_BLOCK MyCalculation
        VAR_IN_OUT
                MyCalculation : REAL; // локальная переменная ФБ
        END_VAR

        MyCalculation := 1.1; // доступ к локальной переменной ФБ
END_FUNCTION_BLOCK

PROGRAM MyCalculation
        VAR
                MyCalculation : MyCalculation;
        END_VAR
        ... // отсюда невозможен доступ к глобальной переменной MyCalculation
END_PROGRAM
```

Комментарий: нет

3.2.8. Применяйте префиксы для пользовательских типов данных

Идентификатор: N10

Приоритет: низкий

Языки программирования: все

Ссылки: нет

Описание: Вы можете выбрать любую систему префиксов для пользовательских типов данных. С помощью префиксов можно, например, закодировать:

- Тип объекта (структура, перечисление и т.д.);
- Область применения (управление движением и т.д.).

Применение: Если вы используете префиксы – задокументируйте свою систему. Выберите один или несколько подходов из приведенных ниже примеров и последовательно применяйте их. Обратите внимание – один и тот же префикс может использоваться в разных обозначениях. В этом случае следует применять составные префиксы.

Объяснение: Кодирование дополнительной информации в именах переменных позволяет повысить читабельность кода и избежать ошибок программирования.

Исключения: Можно отказаться от использования префиксов, если среда разработки предоставляет всю необходимую информацию другими средствами (например, с помощью всплывающих подсказок).

Примеры:

Пример кодирования пользовательских типов данных и POU (табл. 11 из 3-й редакции стандарта **МЭК 61131-3**):

Тип данных	Префикс
ENUM	e
NAMED	e
SUBRANGE	sb
ARRAY	a
STRUCT	st
Type STRUCT	ts

Примечание: ENUM и NAMED используют одинаковый префикс.

В качестве альтернативы для всех пользовательских типов данных может использоваться префикс *udt*.

Комментарий: нет

4. Комментарии

4.1. Комментарии должны описывать назначение кода

Идентификатор: C1

Приоритет: высокий

Языки программирования: все

Ссылки:

- JSF++ 130

Описание: Код должен сопровождаться комментариями, но это не означает, кто каждая строка должна быть прокомментирована. Комментарии должны описывать назначение кода и намерения программиста.

Применение: Прочитайте ваш код и убедитесь, что он поясняется комментариями, например:

- Комментарием в конце строки;
- Комментарием перед блоком кода;
- Комментариями перед блоком кода и внутри него (например, для конструкции IF THEN ELSE);
- Отдельным комментарием для функции/ФБ/программы, если ее сложность это позволяет.

Объяснение: Правильный подход к программированию подразумевает, что хороший код с ясными именами переменных документирует сам себя. Поэтому не требуется комментировать каждую строку/блок программы, если только она не является сложной или допускает различные толкования.

Вы можете разделить комментарии на 5 категорий (*McConnell 1993*, стр. 463):

1. Повторение кода;
2. Объяснение кода;
3. Позиционные маркеры;
4. Краткое описание кода;
5. Описание предназначения кода.

Какими должны быть правильные комментарии? Описывающими намерения программиста. Чем это отличается от обычного пояснения кода? Следует различать намерение (что делает эта функция?) и реализацию (как она это делает?). Реализация связана с языком программирования (IL, ST, FBD и т.д.), в то время как намерение формулируется на человеческом языке (английском, китайском и т.д.). Такое описание намерений может быть неоднозначным, особенно с учетом того факта, что программисты не всегда являются экспертами в «человеческих» языках и способны ясно выражать свои

мысли. Поэтому обычно программисты ограничиваются кратким описанием кода, маркерами и комментариями, повторяющими код. В особых случаях может быть пояснена часть программы.

Для облегчения понимания программы весь код должен содержать комментарии, которые поясняют его назначение; не должно быть кода, который выполняет задачи, не описанные в комментариях. В ситуации, когда конструкция IF…THEN содержит всего несколько ветвей с простыми условиями – достаточно одного комментария на всю конструкцию. Зачастую небольшие функции\ФБ могут быть прокомментированы одной строкой.

Исключения: Любые «магические числа» и физические адреса должны быть прокомментированы.

Примеры:

Неправильно:	*Правильно:*

Блок кода без комментариев:
 	
IF IsValid(TCInput) THEN
  Temperature := TCInput * TCScale + TCOffset;
ELSE
  TCBadQuality := TRUE;
END_IF;
 	
Комментарии, не добавляющие ясности:
 	
IF IsValid(TCInput) THEN
  // TCInput корректен
  Temperature := TCInput * TCScale + TCOffset;
ELSE
  // TCInput некорректен
  TCBadQuality := TRUE;
END_IF;
	Отдельные комментарии для ветвей, проясняющие код:
	
	IF IsValid(TCInput) THEN
	  // масштабирование температуры
	  Temperature := TCInput * TCScale + TCOffset;
	ELSE
	  // ошибка при чтении значения температуры
	  TCBadQuality := TRUE;
	END_IF;
	
	Комментарий для всего блока:
	
	// Поиск максимальной температуры
	MaxReading := nReadings[0];
	FOR index:=1 TO 10 DO
	  IF nReadings[Index] > MaxReading THEN
	  MaxReading := nReadings[Index];
	  END_IF;
	END_FOR;

Комментарий: нет

4.2. Все объекты должны быть прокомментированы

Идентификатор: C2

Приоритет: высокий

Языки программирования: LD, ST, SFC, FBD

Ссылки:

- JSF++ 132, 134

Описание: Назначение всех программ, функциональных блоков, функций, задач, пользовательских типов данных, ресурсов и переменных должно быть пояснено комментариями.

Применение: Дополните каждый объект проекта комментарием, поясняющим его назначение.

Объяснение: Комментарии повышают читабельность программы и делают ее более понятной.

Исключения: нет

Примеры: нет

Комментарий: нет

4.3. Не используйте вложенные комментарии

Идентификатор: C3

Приоритет: низкий

Языки программирования: все

Ссылки:

- MISRA-C-2004, п. 2.3

Описание: Избегайте использования вложенных многострочных комментариев.

Применение: Релизная версия ПО не должна содержать вложенных комментариев, даже если они использовались на этапе разработки и отладки.

Объяснение: В результате редактирования вложенных комментариев часто возникают проблемы, связанные с потерей маркеров начала/конца комментария, в результате чего часть рабочего кода оказывается закомментирована. При этом компиляция программы может происходить без ошибок.

Запомните следующее правило: закомментированный код не должен использоваться для контроля версий или конфигурации ПО – для этих целей существуют специальные средства.

Исключения: нет

Примеры:

Неправильно:

```
(* (1) маркер конца комментария потерялся – и код стал текстом

Важный_блок_кода_который_никогда_не_будет_вызван();

(* <- это не маркер начала комментария, а просто символы – ведь комментарий начался еще в (1) *)
```

Комментарий: нет

4.4. Комментарии не должны содержать код

Идентификатор: C4

Приоритет: низкий

Языки программирования: ST

Ссылки:

- MISRA-C-2004, п. 2.4
- JSF++ 127

Описание: Комментарии не должны содержать кода.

Применение: Релизная версия ПО не должна содержать закомментированных блоков кода, даже если они использовались на этапе разработки и отладки.

Объяснение: В результате редактирования длинных комментариев часто возникают проблемы, связанные с потерей маркеров начала/конца комментария; если в комментариях присутствовал код – он может быть случайно раскомментирован с непредсказуемыми последствиями.

Закомментированный код является типичным явлением на этапе отладки ПО – но в релизной версии он должен отсутствовать. Следует удалить его или же (в случаях, описанных в правиле CP2) заключить в конструкцию типа «IF FALSE THEN …».

Запомните следующее правило: закомментированный код не должен использоваться для контроля версий или конфигурации ПО – для этих целей существуют специальные средства.

Исключения: нет

Примеры: нет

Комментарий: нет

4.5. Используйте однострочные комментарии

Идентификатор: C5

Приоритет: низкий

Языки программирования: ST

Ссылки:

- МЭК 61131-3, п. 6.1.5
- MISRA-C-2004, п. 2.2

Описание: Определите тип комментариев, которые вы будет использовать в своем проекте.

Применение: Используйте однострочные комментарии (// вот такие).

Объяснение: Хотя обычно среда программирования позволяет применять несколько типов комментариев, для единообразия и переносимости следует выбрать и использовать только один из них. Многострочные комментарии являются частым источником ошибок (см. правила C3 и C4) и, кроме того, могут быть использованы только в средах программирования, соответствующих 3-й редакции стандарта **МЭК 61131-3.** Использование однострочных комментариев позволяет избежать описанных проблем.

Исключения: На этапах тестирования и отладки ПО допускается использовать многострочные комментарии для исключения из работы блоков кода. Использовать однострочные комментарии для этой цели запрещается.

Примеры:

Неправильно:

```
 (* многострочные комментарии опасны, так как
символ начала комментария может быть случайно удален (см. правило C4)

b:=a; // закомментированный код может быть случайно раскомментирован
      // или наоборот – может быть закомментирован рабочий код
*)
(* (* вложенный комментарий *) *)
```

Правильно:

```
<выражение>; // однострочный комментарий

        // блочный комментарий

        // <в этом в блоке кода выполняется...>

        // <...>

<выражение>;
```

Комментарий: Для существующих проектов, где применяются однострочные комментарии типа (* текст *) допускается использовать комментарии типа /* код */ для исключения блоков кода (если среда программирования это позволяет). Это облегчит понимание цели комментария.

4.6. Определите язык комментариев

Идентификатор: C6

Приоритет: низкий

Языки программирования: LD, ST, SFC, FBD

Ссылки: нет

Описание: Определите язык комментариев для своего проекта. Это касается как комментариев кода (для языков ST и LD), так и комментариев при объявлении переменных, информации о проекте и т.д.

Применение: Рекомендуется использовать исключительно английский язык для написания комментариев.

Объяснение: Использование и поддержка в актуальном состоянии комментариев повышает читабельность кода и уменьшает число ошибок, что приводит к снижению стоимости разработки и поддержки ПО. Поскольку над проектом в разные (или даже в один и тот же) периоды времени могут работать различные разработчики, следует использовать для комментариев единственный, понятный всем язык. В большинстве случаев для этой цели используется английский язык – который фактически является универсальным языком международного общения. В некоторых случаях представляется полезным иметь в проекте комментарии на нескольких языках (например, английском и китайском). Если среда разработки не имеет встроенных инструментов локализации, то можно сделать это вручную. В этом случае требуется проводить дополнительные инспекции кода на предмет непереведенных комментариев.

Исключения: нет

Примеры: нет

Комментарий: нет

5. Правила кодирования

5.1. Доступ к вложенному элементу должен производиться по его имени

Идентификатор: CP1

Приоритет: высокий

Языки программирования: все

Ссылки:

- MISRA-C-2004, п. 3.5

Описание: При работе с пользовательскими типами данных (например, структурами) обращение к вложенным элементам должно производиться с помощью имен, а не адресов памяти.

Применение: Не используйте обращение к элементам через их адреса (см. правило N1).

Объяснение: Обращение к вложенному элементу через адрес памяти является трудным для чтения и восприятия. Если структура была изменена, то адреса ее элементов также подвергнутся изменениям. Кроме того, распределение адресов элементов связано с особенностями конкретного ПЛК (порядком байт, выравниванием и т.д.).

Исключения: нет

Примеры:

Неправильно:

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

VAR
        instance : EXAMPLE_STRUCT AT %MW500;
END_VAR

// Запись первого символа строки Z:
%MW504 := 'E';
```

Правильно:

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

Комментарий: нет

5.2. В проекте не должно быть неиспользуемого кода

Идентификатор: CP2

Приоритет: высокий

Языки программирования: все

Ссылки:

- Codesys SA0001, SA0031
- Itris Automation Square I3
- MISRA-C-2004, п. 14.1
- JSF++ 186

Описание: Проект не должен содержать «мертвого кода».

Применение: Любой код, содержащийся в проекте, при определенных условиях должен выполняться.

Объяснение: Мертвый код снижает читабельность программы, затрудняет ее сопровождение и обычно является признаком какой-либо проблемы. Существует несколько разновидностей мертвого кода:

- Исполняемый, но ненужный код:
  - Код, оставшийся при использовании старого приложения в качестве шаблона для нового;
  - Код, появившийся в результате ошибки разработчика;
  - Код, используемый для отладки или симуляции.
- Неисполняемый код:
  - Код, расположенный под меткой, к которой никогда не выполняется переход;
  - Код, расположенный под условием, которое никогда не становится истинным.

Исключения: Небольшие порции мертвого кода допустимы в том случае, если они сопровождаются комментариями, четко описывающими причины, по которым это было сделано.

Примеры:

Правильно:

```
//…
// Следующая секция кода обрабатывает опциональное устройство FOO.
// Оно не используется в приложении THIS_APPLICATION…
// …так как недоступно для ПЛК 1 (модель ПЛК, не поддерживающего функционал)
// …но потребуется при  переносе проекта на ПЛК 2
// (модель ПЛК, поддерживающего функционал)

IF FALSE THEN
        // неисполняемый код
        ...
END_IF;
// Конец неисполняемого блока кода для приложения THIS_APPLICATION
```

Комментарий: нет

5.3. Все переменные должны инициализироваться начальными значениями

Идентификатор: CP3

Приоритет: высокий

Языки программирования: все

Ссылки:

- МЭК 61131-3 (3-я редакция), п. 6.5.1 и 6.5.6
- МЭК 61131-8, п. 3.1.1
- JSF++ 142

Описание: Переменная должна быть инициализирована начальным значением перед любым использованием в программе независимо от состояния ПЛК («холодный» перезапуск, «теплый» перезапуск, первый цикл, обычный цикл).

Применение:

- В соответствии с **МЭК 61131-3** все переменные имеют начальные значения, присваиваемые им по умолчанию (если начальное значение не задано в явном виде). Если конкретное ПО не поддерживает (или только частично поддерживает) этот функционал, то пользователь должен инициализировать переменные в своем коде;
- Присвоение начальных значений в области объявления переменных является хорошим подходом, но в некоторых ситуациях это может не производить никакого эффекта (например, при холодном или теплом перезапуске);
- Инициализация переменных в коде делает программу более читабельной и упрощает ее сопровождение;
- При создании пользовательского типа данных задайте всем его переменным значения по умолчанию – тогда не придется выполнять инициализацию объектов, принадлежащих данному типу.

Объяснение: Использование в коде неинициализированных переменных может привести к непредсказуемым последствиям.

Исключения:

- Если ПЛК по умолчанию инициализирует переменные нулевыми значениями и если требуется именно такое поведение – то явная инициализация не требуется. Однако это может повлиять на переносимость программы.
- Энергонезависимые (RETAIN) переменные автоматически инициализируются значениями, которые они имели до перезагрузки ПЛК;
- Переменные, связанные с физическими входами ПЛК не нуждаются в инициализации.

Примеры:

Правильно:

```
PROGRAM Initialization
VAR
        NumOfRetries:               INT   := 3;       // инициализация при объявлении
        Enable:                     BOOL:= TRUE;
         ConfTimerPreselection:     TIME := T#5s;
         StateLastPosition:         INT;
         SpeedAverage:              REAL;
END_VAR;

 StateLastPosition := 50;           // инициализация в коде
 SpeedAverage := 0.0;
…
END_PROGRAM
```

Стандарт **МЭК 61131-3** позволяет инициализировать переменные пользовательских типов данных при их создании:

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

Любая переменная типа *TempLimit* будет по умолчанию инициализирована со значением **250.0**. В следующем примере переменная *BoilerMaxTemperature* будет инициализирована со значением **250.0**, а переменная *PipeMaxTemperature* – со **значением 0.0**. Очевидно, что в некоторых случаях использование **0.0** в качестве начального значения является неприемлемым – поэтому требуется явная инициализация переменных. В свою очередь, переменная *BoilerMaxTemperature* не требует явной инициализации, так как ее начальное значение было определено при создании пользовательского типа данных – и это упрощает программу, а также делает ее более надежной.

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

Комментарий: Примеры инициализации пользовательских типов данных:

Инициализация перечисления:

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

Инициализация интервального типа данных:

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

Инициализация структуры:

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

Инициализация массива структур:

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

Инициализация структуры:

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

Инициализация псевдонима структуры:

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

Стандарт **МЭК 61131-3** предоставляет несколько способов инициализации переменных:

1) Инициализация пользовательских типов данных

Программист может указать начальные значения для вложенных переменных при создании типа. Если значения не указаны, используются значения по умолчанию для соответствующего элементарного типа данных (см. **МЭК 61131-3**, табл. 10). Примеры инициализации пользовательских типов данных приведены выше.

2) Инициализации переменных при объявлении

Начальные значения могут быть указаны при объявлении переменной или экземпляра POU. Если значение не указано, то используется значение по умолчанию для соответствующего типа данных.

Таким образом можно инициализировать:

- Переменные элементарного типа данных;
- Переменные пользовательских типов данных;
- Массивы;
- Экземпляры структур;
- Функциональные блоки (только входы/выходы и PUBLIC переменные);
- Классы (только PUBLIC переменные).

См. примеры в 3-й редакции стандарта **МЭК 61131-3** (табл. 14, табл. 41 пп. 2, табл. 49 пп. 2).

3) Инициализация через конфигурацию (VAR_CONFIG)

См. примеры в **МЭК 61131-3** (табл. 62, пп. 11a и 11b). Этот тип инициализации перезаписывает значения, присвоенные с помощью способов 1) и 2).

```
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
        // инициализация через конфигурацию
        VAR_CONFIG
                Station1.ProgInst1.COUNT : INT:= 1;
                Station1.ProgInst2.TIME1 : TON:= (PT:= T#2.5s);
                // инициализация экземпляра ФБ
                Station2.ProgInst1.FbInst1.FbInst2.FbInst3.Count : INT:= 100;
                // привязка переменных к физическим адресам входов-выходов
                Station2.ProgInst1.FbInst1.C2 AT %QB25 : BYTE;
        END_VAR
END_CONFIGURATION
```

5.4. Избегайте наложения адресов

Идентификатор: CP4

Приоритет: высокий

Языки программирования: все

Ссылки:

- Itris Coding Standard – S8
- Codesys SA0028
- Misra-C-2004, п. 18.2 и 18.3

Описание: Если переменная привязывается к адресу памяти, следует убедиться, что данный участок памяти еще нигде не используется.

Применение: Избегайте наложения используемых адресов.

Объяснение: Большинство приложений ПЛК работают по общему принципу: *чтение входов – выполнение алгоритмов – запись выходов*. Выявление повторного использования адресов в коде в данном случае достаточно затруднительно. Рассмотрим следующую ситуацию: в фрагменте кода 1 по адресу памяти записывается значение 1, которое потом считывается и обрабатывается в программе, а в выполняемом далее фрагменте кода 2 – записывается и считывается значение 2. Вполне возможно, что это долгое время не будет вызывать никаких негативных эффектов. Но при определенном, сложно прогнозируемом стечении обстоятельств в фрагменте кода 2 запись значение 2 может не произойти – и тогда в алгоритме будет использовано значение 1 из совершенно другого фрагмента кода, что может привести к непредсказуемым последствиям.

Исключения: Тип STRUCT OVERLAP (UNION) предназначен для размещения данных по одним и тем же адресами памяти (например, для реализации побитового доступа к значениям типов данных, которые такой доступ не предусматривают).

Примеры:

Неправильно:

```
Temperature : INT AT %MW451;

// Переменная Level занимает адреса MW450-451 (2 регистра)
// …и поэтому перекрывает переменную Temperature
Level: REAL AT %MW450;

Temperature := %IW56;

IF Temperature > 56 THEN
        StartFans();
END_IF;

// здесь записывается только первый регистр переменной Level
IF NOT VeryRarecondition THEN
        Level := %IW34;
END_IF;

IF Level < LevelThreshold THEN
        // выполнение этого условия зависит от значения переменной Temperature
        // …что совершенно неочевидно и может привести к ошибкам
        DoCriticalFunction(Level);
END_IF;
```

Правильно:

```
// в данном примере адреса не перекрываются
Temperature :   INT AT %MW444;
Level:          REAL AT %MW450;

Temperature := %IW56;

IF Temperature > 56 THEN
        StartFans();
END_IF;
```

Комментарий: нет

5.5. Перед началом кодирования выполняется проектирование

Идентификатор: CP5

Приоритет: высокий

Языки программирования: LD, ST, SFC, FBD

Ссылки:

- МЭК 61131-8, п. 3.3

Описание: Перед началом кодирования должны быть выполнены работы по проектированию приложения. Паттерны проектирования могут соответствовать принципам ООП – модульная архитектура программы, инкапсуляция и т.д.

Применение:

- Проектирование – **необходимый** этап в разработке ПО, предшествующий кодированию;
- Используйте массивы (где это возможно), чтобы повысить связность данных одного типа;
- Используйте структуры (где это возможно), чтобы повысить связность данных разных типов;
- Используйте классы (где это возможно) или функции/ФБ, чтобы уменьшить сложность программы, разбив ее на небольшие изолированные фрагменты;
- Используйте классы (где это возможно) или функции/ФБ, чтобы обеспечить повторное использование кода;
- Используйте PRIVATE переменные классов (где это возможно) или локальные переменные функций/ФБ, чтобы обеспечить инкапсуляцию данных;
- Для каждой программы используйте оптимальный язык программирования в зависимости от выполняемых ею задач (где это возможно).

Объяснение: Хорошо спроектированное приложение обходится дешевле в разработке, облегчает процесс кодирования и уменьшает количество ошибок. Использование модульной архитектуры позволяет независимо разрабатывать и отлаживать различные фрагменты приложения. Понятия «класс» и «функциональный блок» тесно связаны с парадигмой объектно-ориентированного программирования. Функциональный блок (и класс) включают в себя данные и производимые над ними операции. Отдельные объекты программы представляют собой экземпляры соответствующих ФБ. Эти экземпляры обладают интерфейсом (входами и выходами), упрощающим их использование. Внутренние данные блока недоступны для пользователя, что обеспечивает инкапсуляцию.

Исключения: нет

Примеры: нет

Комментарий: нет

5.6. Избегайте использования в POU внешних переменных

Идентификатор: CP6

Приоритет: высокий

Языки программирования: все

Ссылки:

- JSF++ 207
- МЭК 61131-3, п. 2.5.1

Описание: Внешние (VAR_EXTERNAL) переменные не должны использоваться в функциях, функциональных блоках и классах.

Применение: Функции, ФБ и классы не должны содержать внешних переменных. При необходимости следует увеличить количество аргументов соответствующих POU.

Объяснение: Хороший подход к проектированию ПО подразумевает повторное использование функций, ФБ и классов. Их исходный код не всегда доступен пользователю. Применение внешних переменных серьезно затрудняет (или даже делает невозможным) процесс повторного использования POU. Например, если ФБ использует 10 внешних переменных – вы должны скопировать их свой в проект вместе со всем остальным кодом, который связан с этими переменными. Кроме того, ФБ и классы могут иметь несколько экземпляров – но при использовании внешних переменных в большинстве случаев приходится ограничиться только одним. Внешние переменные не позволяют проводить модульное тестирование POU – и, кроме того, процесс тестирования приходится повторять при каждом изменении кода, влияющем на значения внешних переменных. Использование внешних переменных повышает шанс ошибок, связанных с изменением одних и тех же переменных разными POU (см. правила CP26 и CP15); поиск и отладка этих ошибок крайне затруднительны. Зачастую наличие внешних переменных является сигналом о проблемах приложения и потребности в его переработке. Внешние переменные также могут являться причиной побочных эффектов – например, появления в проекте функций, которые возвращают различные значения при одном и том же наборе значений аргументов. Для использования внешних переменных должна быть очень веская причина и любое их применение должно быть тщательно задокументировано.

Исключения: Внешние переменные могут потребоваться для доступа к системным переменным и глобальным константам (VAR_GLOGAL CONSTANT).

Примеры:

Неправильно:

```
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;
```

Правильно:

```
FUNCTION CommandMotor
VAR_INPUT
        Enable:         BOOL;
        Default:        BOOL;
        ModeAuto :      BOOL;  // дополнительный аргумент
END_VAR;
VAR_OUTPUT
        Command:        BOOL;
END_VAR;

 Command := Enable AND NOT Default AND ModeAuto;

END_FUNCTION;
```

Комментарий: Это правило также применяется к программам (PROGRAM), если среда разработки поддерживает использование их экземпляров. Стандарт **МЭК 61131-3** предусматривает такую возможность (см. табл. 47 пп. 2 и табл. 62 пп. 89).

5.7. Осуществляйте обработку ошибок

Идентификатор: CP7

Приоритет: высокий

Языки программирования: все

Ссылки:

- MISRA-C-2004, п. 16.10
- JSF ++ 115

Описание: Если POU возвращает бит и/или код ошибки - эта информация должна обрабатываться в коде.

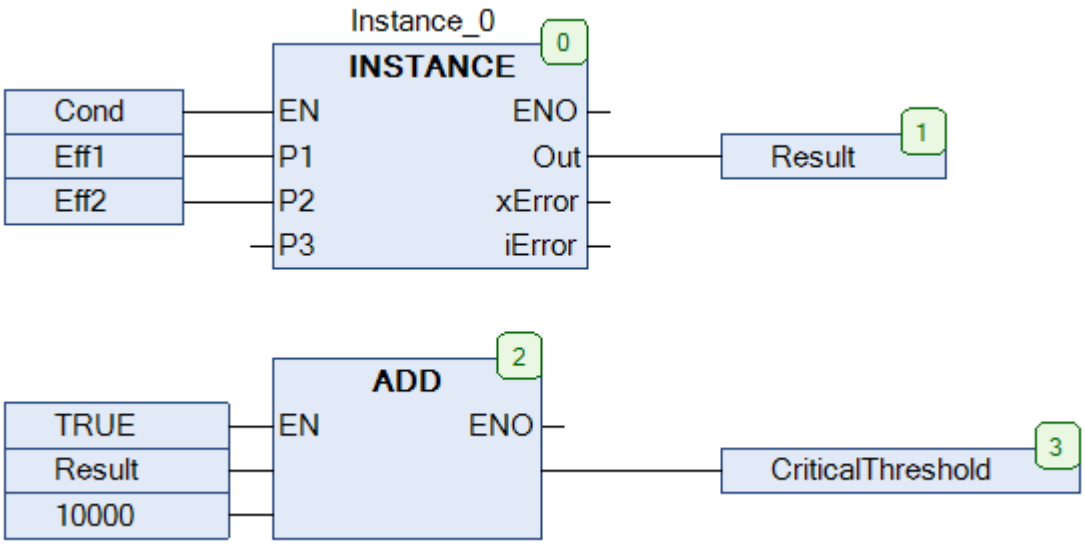
Применение: Обрабатывайте в коде бит/код ошибки POU после его вызова, чтобы при необходимости изменить логику работы программы в случае возникновения аварийных ситуаций.

Объяснение: При возникновении необрабатываемых ошибок ПЛК будет продолжать выполнять заданный алгоритм – что может иметь непредсказуемые последствия для технологического процесса. Обработка ошибка показывает, что разработчик проанализировал все (или, по крайней мере, наиболее вероятные) потенциальные ошибки и предусмотрел реакцию системы в случае из возникновения (например, оповещение оператора).

Исключения: нет

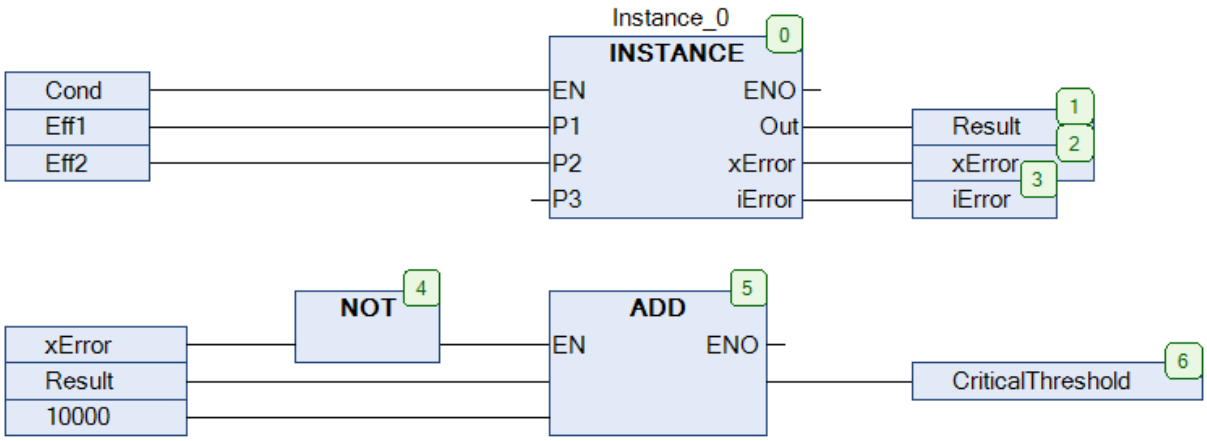
Примеры:

Неправильно:



Даже при возникновении ошибки в экземпляре ФБ *Instance* результат его работы будет использован для вычисления порогового значения *CriticalThreshold*.

Правильно:



При возникновении ошибки в экземпляре ФБ *Instance* результат его работы не будет использован для вычисления порогового значения *CriticalThreshold*.

Комментарий: нет

5.8. Значения с плавающей точкой не должны проверяться на равенство и неравенство

Идентификатор: CP8

Приоритет: высокий

Языки программирования: все

Ссылки:

- Codesys SA0054
- Misra-C-2004, п. 13.3
- JSF++ 202

Описание: Для значений с плавающей точкой запрещается использовать операции проверки на равенство и неравенство.

Применение: Для сравнения значений с плавающей точкой должны использоваться только операторы < (меньше), > (больше), <= (меньше или равно), >= (больше или равно).

Объяснение: Оператор равенства подразумевает возможность строгого соответствия сравниваемых значений. В случае значений с плавающей точкой вероятность этого практически исключена.

Исключения: Допустимо сравнение с нулем (0.0).

Примеры:

Многие числа не могут быть точно записаны в представлении с плавающей точкой. Например, значение 0.1 согласно стандарту IEEE 754-2008 фактически представляет число 0.100000001490116119384765625. Кроме того, в процессе арифметических операций над числами с плавающей точкой практически всегда возникают погрешности округления.

Неправильно:

```
IF TEMP - OLD_Temp = 0.1 THEN
        // этот код может никогда не выполниться из-за ошибок округления
        ....
END_IF;
```

Правильно:

```
IF TEMP - OLD_TEMP < 0.1 THEN
        // исполняемый код
END_IF;
```

или

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

Комментарий: нет

5.9. Значения времени и физические величины не должны проверяться на равенство и неравенство

Идентификатор: CP28

Приоритет: высокий

Языки программирования: все

Ссылки: нет

Описание: Для значений времени и физических величин запрещается использовать операции проверки на равенство и неравенство.

Применение: Для сравнения значений времени и физических величин должны использоваться только операторы < (меньше), > (больше), <= (меньше или равно), >= (больше или равно).

Объяснение: Оператор равенства подразумевает возможность строго соответствия сравниваемых значений. В случае значений времени и физических величин вероятность этого практически исключена.

Исключения: нет

Примеры:

Неправильно:

```
IF Distance – InitialPosition = 12 THEN
        // этот код может никогда не выполниться из-за ошибок округления
        ....
END_IF;

IF T5.Et = T#10s then
        // этот код может никогда не выполниться из-за дискретности таймера
END_IF;
```

Правильно:

```
IF Distance – InitialPosition - 12 < ERROR_MARGIN THEN
        // исполняемый код
END_IF;
```

или

```
IF T5.ET - T#10s < T#100MS THEN
        // исполняемый код
END_IF;
```

Комментарий: нет

5.10. Определите допустимую сложность POU

Идентификатор: CP9

Приоритет: высокий

Языки программирования: все

Ссылки: нет

Описание: Существуют различные способы определения сложность кода. Некоторые из них очевидны – например, подсчет числа строк кода и количества используемых конструкций. Другие методики рассмотрены в соответствующей литературе (McCabe, Elshof, Prater и т.д.). Любой способ имеет преимущества и недостатки – поэтому нельзя рекомендовать какой-то конкретный из них. Например, методики McCabe и Prater подходят только для текстовых языков программирования, а Elshof – для любого. Изучив различные подходы, пользователь может выработать критерии оценки сложности POU и установить допустимые пределы сложности своего кода.

Применение: Если сложность POU превышает заданный предел, то следует разделить его на несколько отдельных POU.

Объяснение: Сложность POU затрудняет его отладку и является потенциальной причиной ошибок программирования.

Исключения: нет

Примеры:

Неправильно:

Рассмотрим ФБ CHARCURVE:

- число конструкций: 16
- сложность по McCabe: 12
- сложность по Prater: 3,89
- сложность по Halstead: 44,9
- сложность по Elshof: 0,14 (чем ниже – тем сложнее)

```
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[I1].Y)/(P[I].X-P[I-1].X));
                ELSE
                        OUT:=0;
                END_IF;
        END_IF
ELSE
        ERR:=4;
END_IF;
```

Правильно:

Уменьшим сложность ФБ, добавив метод CalculateOut:

- Число конструкций: 12
- Сложность по McCabe: 8
- Сложность по Prater: 2,25
- Сложность по Halstead: 19,9
- Сложность по Elshof: 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;
```

Комментарий: Ссылки на тематическую литературу:

- 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. Избегайте записи переменной из разных задач

Идентификатор: CP10

Приоритет: высокий

Языки программирования: все

Ссылки:

- Itris Automation Square S
- Codesys SA0006

Описание: Избегайте записи переменной из разных задач приложения. Каждая переменная должна записываться только в пределах одной задачи.

Применение: Если переменная используется несколькими задачами, то изменение ее значения должно производиться только в одной из них. Это также справедливо и для переменных, применяемых для обмена данных между задачами.

Объяснение: Переключение задач в большинстве случаев зависит от частоты их вызова и назначенных приоритетов. В системах с вытесняющей или параллельной многозадачностью менее приоритетная задача может быть прервана задачей с более высоким приоритетом. Если каждая из них может производить запись в одну и ту же переменную, то алгоритм работы приложения становится непредсказуемым. Это может привести к потере или нарушению целостности данных.

Исключения: При продуманном проектировании и тщательном тестировании можно избежать непредсказуемого поведения приложения с помощью использования флагов разрешения записи и специфических функций конкретной платформы (например, мьютексов и семафоров).

Примеры:

Неправильно:

Предположим, один из программистов, работающих над приложением, создал глобальную переменную LoopCounter для использования в своей задаче. Другой из программистов скопировал и отредактировал часть его кода для использования в другой задаче:

VAR_GLOBAL LoopCounter: INT; 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

Основная программа (Main Program) в любое время может быть прервана более приоритетной (Priority Program), при этом в момент переключения задач значение переменной LoopCounter может быть любым (в диапазоне 0..10). После выполнения приоритетной задачи переменная будет иметь значение 100, и поэтому цикл FOR в основной задаче не будет выполнен. Таким образом, существует вероятность, что код будет работать с неверными (или даже несуществующими) элементами массива.

Правильно:

В приведенном выше примере можно было бы объявить в каждой программе свою локальную переменную, желательно – с разными названиями.

Комментарий: Вопросы синхронизации задач рассмотрены в правиле CP11 *Синхронизируйте выполнение задач*.

5.12. Синхронизируйте выполнение задач

Идентификатор: CP11

Приоритет: высокий

Языки программирования: все

Ссылки: нет

Описание: Если различные задачи используют общие данные, то их выполнение должно быть синхронизировано.

Применение: Избегайте доступа к одним и тем же данным из разных задач или используйте специфические функции конкретной платформы, чтобы синхронизировать их выполнение.

В случае необходимости обмена данными между программами, привязанными к разным задачам, следует:

- Использовать в программе внешние переменные (VAR_EXTERNAL) для доступа к глобальным переменным, или VAR_IN_OUT переменные, связанные с глобальными переменными или переменными, объявленными в ресурсах ПЛК (RESOURCE).
- Использовать специальные функции/ФБ, позволяющие синхронизировать выполнение задач (например, мьютексы и семафоры).

Объяснение: Реализация на ПЛК многозадачности с синхронизацией данных является крайне трудоемкой даже при наличии специальных функций/ФБ, которые может предоставлять производитель конкретных контроллеров. Архитектура ПЛК подразумевает псевдопараллельную обработку данных с помощью циклического выполнения последовательности задач, что позволяет избежать непредсказуемых задержек, характерных для асинхронных многозадачных систем управления.

Поэтому лучше избегать синхронизации программ. Если же такая синхронизация необходима с логической точки зрения, то связанные с ней данные и блоки кода должны выполняться атомарно с эксклюзивным доступом. Так как стандарт **МЭК 61131-3** не рассматривает подобные ситуации, то производители ПЛК сами в некоторых случаях предоставляют средства, аналогичные функциям tryLock(), Lock() из языков C++/C#/Java.

Исключения: Некоторые производители предоставляют принципиально иные средства синхронизации задач.

Примеры:

Правильно:

```
// Приведем два примера
// 1) с использованием VAR_EXTERNAL
// 2) С использованием абстрактных специфических функций "Lock()" и "Unlock()"

// эта программа вызывается задачей с низким временем цикла (т.е. очень часто)
PROGRAM Fast_Acquisition
VAR_EXTERNAL
        ClearSubTotalCount:         BOOL; // используется как вход
        SubTotalCount:              INT;    // используется как выход
END_VAR
 VAR_INPUT
        SignalInput :               BOOL; // детектируемый импульс
 END_VAR
 VAR
        Old_ClearSubTotalCount :     BOOL := FALSE;
 END_VAR

 //---- тело программы ----
 Vendor.Lock(LockResId := 0); // начало атомарного процесса с эксклюзивным доступом

 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); // конец атомарного процесса с эксклюзивным доступом

END_PROGRAM
```

```
// эта задача вызывается основной программой (время цикла = 100 мс)
PROGRAM Main_Acquisition
 VAR_EXTERNAL
          SubTotalCount :            INT;    // используется как вход
           ClearSubTotalCount :    BOOL; // используется как выход
 END_VAR
 VAR
 GrandTotalCount :                  INT;
 END_VAR

//---- тело программы ----
 Vendor.Lock(LockResId := 0); // начало атомарного процесса с эксклюзивным доступом
 GrandTotalCount := GrandTotalCount + SubTotalCount;
 ClearSubTotalCount := TRUE;
 Vendor.Unlock(LockResId := 0); //конец атомарного процесса с эксклюзивным доступом

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
```

Комментарий: Это правило применимо только для ПЛК с вытесняющей многозадачностью (см. **МЭК 61131-3**, табл. 63, п. 5b). Некоторые производители предоставляют схожий функционал для конкретной платформы.

Существует способ избежать необходимости синхронизации программ – для этого следует заменить их на функциональные блоки, которые будут вызываться в конкретной программе, связанной с определенной задачей.

Также следует помнить, что использование в задаче специфических функций типа Lock() блокирует вызов всех остальных задач приложения (даже задач с более высоким приоритетом). Это может нарушить заданную периодичность выполнения тех или иных POU.

5.13. Физические выходы должны записываться только раз за цикл ПЛК

Идентификатор: CP12

Приоритет: высокий

Языки программирования: все

Ссылки:

- Codesys SA0004
- Itris Automatin Square S4

Описание: Физические выходы должны записываться только один раз за цикл ПЛК.

Применение: Производите запись значений физических выходов только в одном конкретном фрагменте приложения.

Объяснение: Многократная запись значений физических выходов в цикле ПЛК приводит к неопределенному поведению программы. Кроме того, это усложняет отладку. Корректнее будет подготовить данные в переменных, а в конце цикла передать значения непосредственно на выходы ПЛК.

Исключения: Многократная запись выходов в пределах цикла допустима в определенных случаях (например, из-за применения специфической безопасной архитектуры). Также это может быть оправданным, когда в зависимости от различных условий запись выходов надо производить из разных фрагментов кода – но делать это следует с осторожностью.

Примеры: нет

Комментарий: нет

5.14. POU не должны вызывать сами себя

Идентификатор: CP13

Приоритет: высокий

Языки программирования: все

Ссылки:

- МЭК 61131-8, п. 3.5.4
- MISRA-C-2004, п. 16.2
- JSF++ 119

Описание: Рекурсивные вызовы объектов запрещены.

Применение: Вместо рекурсивных вычислений следует использовать итеративные.

Объяснение: Рекурсивные вычисления используют системный стек для каждого вызова, поэтому глубокая рекурсия может привести к сбою системы. Кроме того, не все производители ПЛК поддерживают эту возможность, что делает приложение менее переносимым.

Исключения: нет

Примеры:

Неправильно:

```
// использование рекурсии для вычисления факториала
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;
```

Правильно:

```
// в данном случае рекурсивные вычисления заменены на итеративные
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;
```

Комментарий: нет

5.15. POU должен иметь одну точку выхода

Идентификатор: CP14

Приоритет: высокий

Языки программирования: все

Ссылки:

- МЭК 61508-7, п. 2.9
- MISRA-C-2004, п. 14.7
- Codesys SA0090
- Itris Automatin Square I6
- JSF++ 113

Описание: Избегайте использования инструкции RETURN для выхода из POU. Инструкция RETURN должна использоваться только для возвращения значения функции.

Применение: Измените структуру POU, чтобы избежать использования инструкции RETURN. Если POU написан на языке ST, то используйте операторы условного перехода. Если POU написан на графическом языке, то используйте метки (label) и инструкцию JUMP.

Объяснение: Наличие у POU нескольких точек выхода снижает читабельность кода и затрудняет отладку приложения. При необходимости вы можете добавить блок кода в конце POU, чтобы в любой ситуации получить нужную информацию (например, информацию о результате выполнения POU).

Исключения: нет

Примеры: нет

Комментарий: нет

5.16. Считывайте переменную, записываемую другой задачей, только один раз за цикл ПЛК

Идентификатор: CP15

Приоритет: высокий

Языки программирования: все

Ссылки: нет

Описание: Избегайте многократного чтения переменной, записываемой другой задачей, в пределах одного цикла ПЛК. Другая задача может быть выполнена в любое время, и значение переменной между двумя операциями чтения может измениться, что способно привести к непредсказуемым последствиям.

Применение: Для предотвращения описанной проблемы следует копировать значение считываемой переменной в локальную переменную, и далее работать только с локальной переменной. Значение локальной переменной не может внезапно измениться, и это делает выполнение программы более определенным.

Операция копирования может выполняться автоматически, если ПО поддерживает следующий функционал из стандарта **МЭК 61131-3:**

- Табл. 47, п. 2a «Чтение входных переменных программы»;
- Табл. 47, п. 2b «Запись выходных переменных программы»;
- Табл. 62, п. 8b «Привязка входных переменных программы к глобальным переменным»;
- Табл. 62, п. 9b «Привязка выходных переменных программы к глобальным переменным».

При привязке глобальных переменных к входным/выходным переменным программы операции чтения (из глобальных переменных во входные) и записи (из выходных переменных в глобальные) происходят только один раз за цикл ПЛК – соответственно, перед выполнением кода программы и после него.

Объяснение: Данные, с которыми работает программа ПЛК, должны быть согласованными. Однако в случае многократного (в пределах цикла ПЛК) обращения к данным, изменяемым в другой задаче, согласованность может нарушиться. По той же причине при работе с физическими входами ПЛК однократно копирует образ области входов в локальные переменные перед началом рабочего цикла.

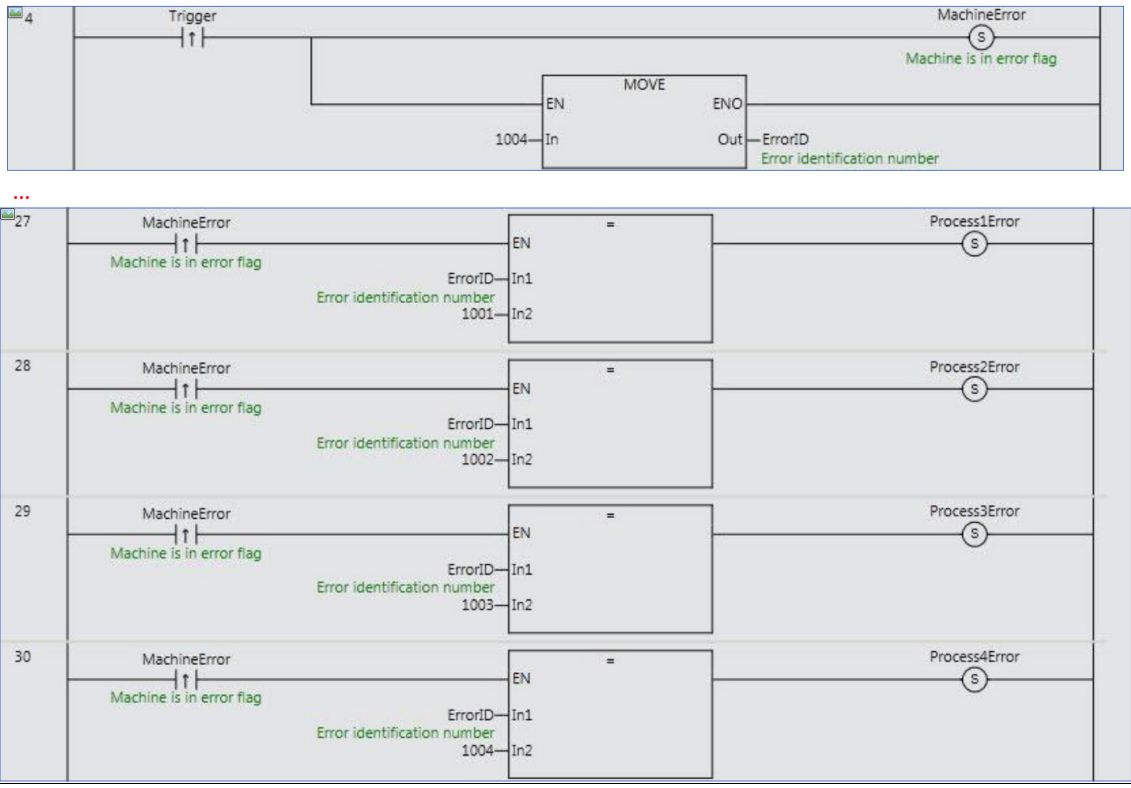
Исключения: нет

Примеры:

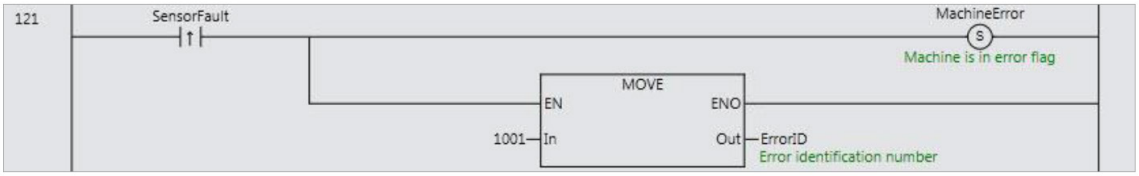
Неправильно:

В этом примере при появлении общего флага ошибки происходит генерация флага конкретной ошибки на основании ее кода. Флаг MachineError генерируется в задаче с более высоким приоритетом, которая может вытеснить основную задачу.

Основная программа:



Программа с более высоким приоритетом:



Если основная программа будет прервана более приоритетной между выполнением цепей 27 и 30, то значение бита MachineError может измениться, и при продолжении работы основной программы бит ошибки не будет сгенерирован.

Правильно:

В приведенном выше примере следует в основной программе копировать бит и код ошибки (MachineError и ErrorID) в локальные переменные – это гарантирует согласованность данных и корректность их обработки. Это может быть сделано автоматически, если ПО поддерживает подобный функционал.

Комментарий: нет

5.17. Функции и функциональные блоки не должны быть привязаны к задачам

Идентификатор: CP16

Приоритет: высокий

Языки программирования: LD, ST, SFC, FBD

Ссылки:

- МЭК 61131-3, п. 2.7.2
- МЭК 61131-8, п. 3.5.4 и п. 3.12.6

Описание: К задачам должны быть привязаны только программы, но не функции и функциональные блоки.

Применение: Не привязывайте к задачам функции и функциональные блоки – вызывайте их в программах, которые, в свою очередь, будут привязаны к задачам.

Объяснение: Если функциональный блок используется в программе и при этом привязан к задаче – могут возникнуть ошибки в согласованности данных. Эта ситуация рассматривается в п. 2.7.2 стандарта **МЭК 61131-3**. Программист должен осознавать возможность подобных последствий и по возможности избегать их.

Даже если конкретное ПО содержит механизмы согласования данных для таких случаев – привязка ФБ к задачам снижает переносимость приложения на другие платформы.

Исключения: нет

Примеры:

Неправильно:

Пример из МЭК 61131-3 (рис. 20). ФБ FB1 и FB2 не должны быть привязаны к задачам.

```
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
```

Комментарий: нет

5.18. Операции над переменными должны соответствовать их области

Идентификатор: CP17

Приоритет: высокий

Языки программирования: все

Ссылки:

- МЭК 61131-8, п. 3.2.2
- Codesys SA0009
- Itris Automation Square E2

Описание: Переменные области VAR_INPUT должны считываться, переменные области VAR_OUTPUT – записываться, переменные области VAR_IN_OUT – считываться и записываться.

Применение: Операции над переменной должны соответствовать ее области:

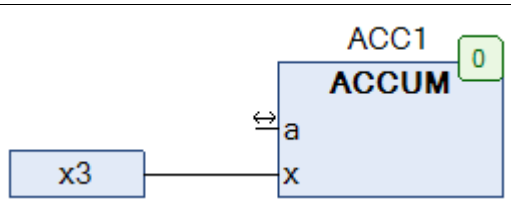
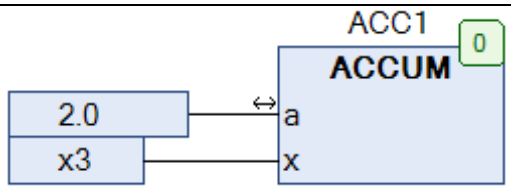
- Любая входная (VAR_INPUT) переменная должна хотя бы раз считываться в коде POU;
- Любая входная (VAR_INPUT) переменная не должна записываться в коде POU;
- Любая выходная (VAR_OUTPUT) переменная должна хотя бы раз записываться в коде POU;
- Любая переменная области «вход-выход» (VAR_IN_OUT) должна хотя бы по разу считываться и записываться в коде POU;

Объяснение: Если переменная не используется в коде – то ее следует удалить (см. правило CP2). Если переменная используется некорректным (не соответствующим ее области) образом, то это может привести к ошибкам и непредсказуемым побочным эффектам.

Исключения: нет

Примеры:

Неправильно:

	Некорректное использование: К переменной A области «вход-выход» не привязана переменная или экземпляр ФБ.
	Некорректное использование: К переменной A области «вход-выход» привязана не переменная или экземпляр ФБ.

Комментарий: нет

5.19. Разумно используйте глобальные переменные

Идентификатор: CP18

Приоритет: высокий

Языки программирования: LD, ST, SFC, FBD

Ссылки:

- MISRA-C-2004, п. 8.7
- JSF++ 207
- Codesys SA0121

Описание: Глобальные переменные необходимы для обмена данными между программами (в т.ч. между программами, связанными с разными задачами). Во всех остальных случаях используйте локальные переменные.

Применение: По возможности следует использовать локальные переменные. Глобальные переменные допустимо использовать в следующих ситуациях:

- При обмене данными между программами (в пределах одной или разных задач);
- При обмене данными с внешними устройствами (опрос физических входов-выходов, опрос подключенных к ПЛК устройств и т.д.);
- Для доступа к системным переменным (системному времени, переменным диагностики и т.д.).

Объяснение: Использование глобальных переменных в POU затрудняет его повторное применение. В некоторых средах разработки можно обращаться к глобальным переменным напрямую, без объявления дополнительных внешних (VAR_EXTERNAL) переменных. Это затрудняет чтение кода и делает неочевидным зависимость POU от внешних данных. Поэтому для обеспечения повторного использования POU и упрощения его отладки следует придерживаться данного правила.

Исключения: Системные переменные являются глобальными – но у пользователя нет необходимости их объявлять.

Примеры:

Неправильно:

```
// в этом примере каждая программа не имеет входных/выходных  переменных и
// обращается к глобальным переменным через внешние переменные.
// для того чтобы определить поток данных, следует изучить код каждой…
// …программы.

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
```

Правильно:

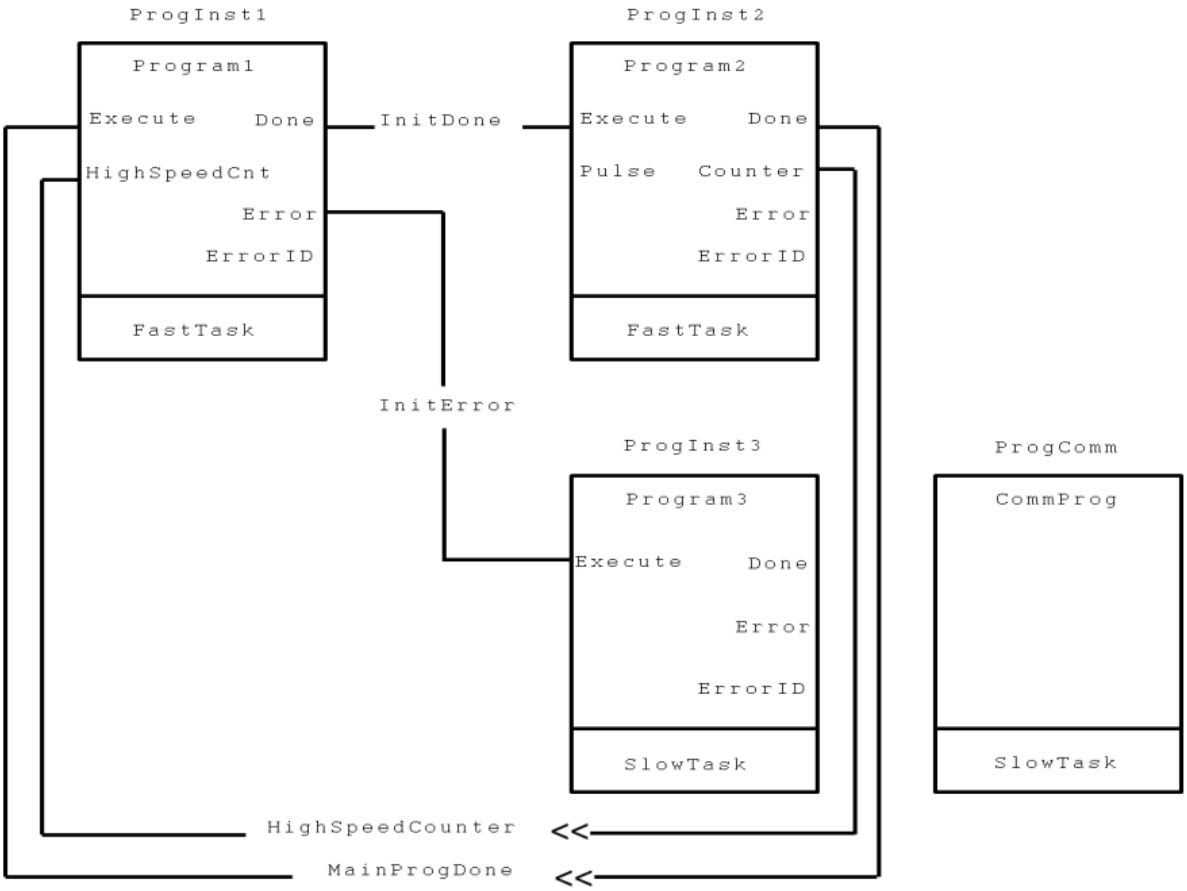
```
RESOURCE Station1 ON ProcessorTypeA
    VAR_GLOBAL
            //-- 1) Переменные для обмена данными между программами
             MainProgDone: BOOL := TRUE;
             InitDone: BOOL := FALSE;
             InitError: BOOL := FALSE;
            //-- 2) Системные переменные с доступом через VAR_EXTERNAL
            CurrentTime : TIME;
            CPUErrorStatus : ErrorStats;
             //-- 3) переменные для обмена данными между задачами
             HighSpeedCounter : DINT;
             //-- 4) переменные для обмена данными с I/O и внешними устройствами
            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); // присвоение переменных аргументам программы

     PROGRAM ProgInst2 WITH FastTask: Program2(Execute := InitDone,
            Pulse := HSPulse, Counter => HighSpeedCounter,
                    Done => MainProgDone); // …

    PROGRAM ProgInst3 WITH SlowTask: Program3(Execute := InitError); // …
    PROGRAM ProgComm WITH SlowTask : CommProg;
END_RESOURCE
```

Примечание: В данном примере поток данных является прозрачным для пользователя. Ниже приведено его графическое представление (эта диаграмма не относится к стандарту МЭК 61131-3).



Комментарий: С использованием глобальных переменных также связаны правила **CP10** *Избегайте записи переменной из разных задач* и **CP26** *Глобальная переменная должна записываться только одной программой*.

5.20. Избегайте использования операторов JUMP и RETURN

Идентификатор: CP19

Приоритет: средний

Языки программирования: LD, FBD

Ссылки:

- Misra-C-2004, п. 14.5
- JSF++ 189, 190

Описание: Следует избегать ветвления с использованием операторов JUMP и RETURN в языках LD и FBD. Это относится и ко всем специфическим операторам ветвления (GOTO и т.д.).

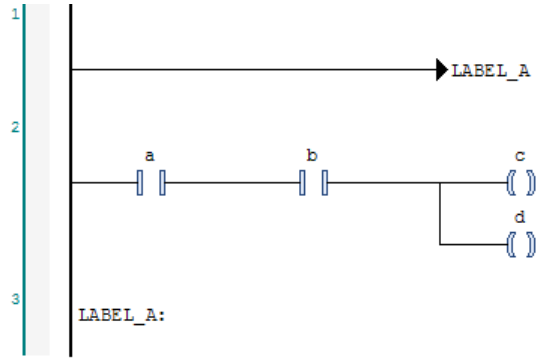
Применение:

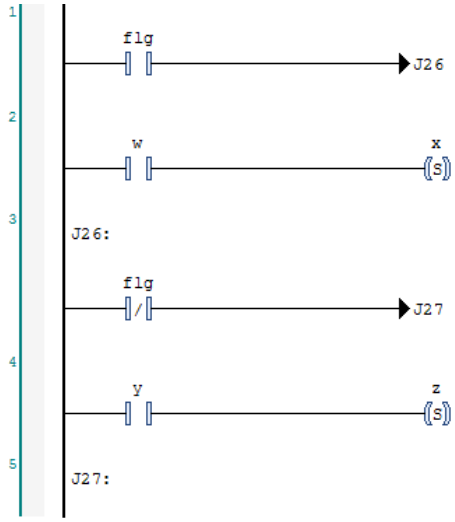
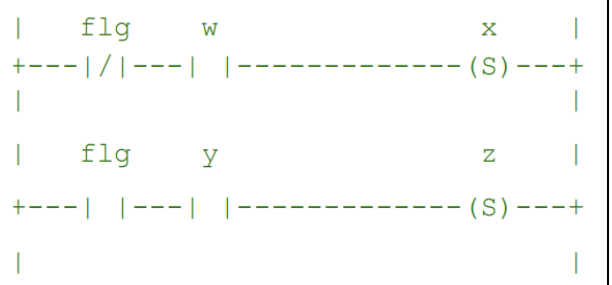
- Операторы JUMP и RETURN не должны применяться для реализации безусловного ветвления;
- При реализации условного ветвления не должно быть возвращения к цепям, расположенным выше.

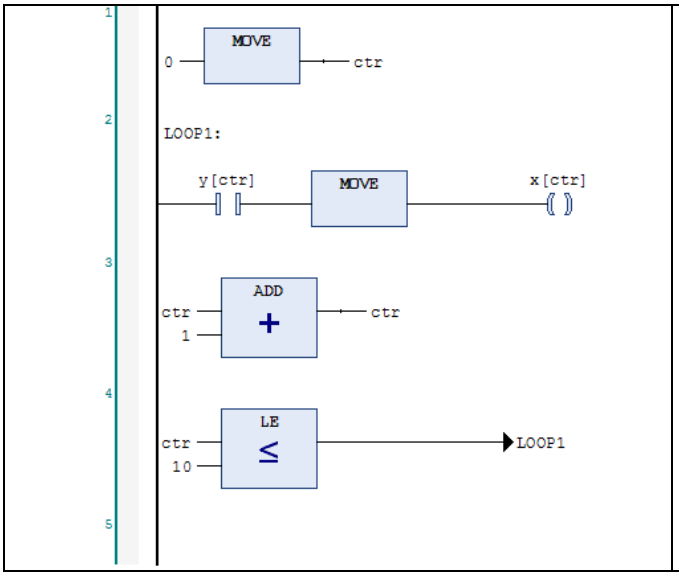
Объяснение: Использование операторов JUMP и RETURN нарушает нормальную последовательность исполнения, что затрудняет отладку приложения, а также приводит к появлению «мертвого» кода, который станет активным, если оператор будет случайно удален. Возвращение к предыдущим цепям делает время выполнения цикла программы непредсказуемым и может привести к логическим петлям. Для реализации условного ветвления в языках LD и FBD могут использоваться специальные входы/выходы EN/ENO.

Исключения: Возвращение к предыдущим цепям может потребоваться для реализации циклов и итеративных процедур. Тем не менее, следует использовать эту возможностью с крайней осторожностью, чтобы избежать возникновения логических петель. В этих случаях рекомендуется использовать язык ST и реализовывать условные переходы с помощью операторов IF…THEN…ELSE, FOR, CASE и т.д.

Примеры:

Неправильно:	*Правильно:*
 Если переход к метке A будет случайно удален – то приведенный код станет выполняться, что может привести к незапланированным последствиям.	Удалите неисполняемый код.

	
	Условные переходы проще реализовать на ST, особенно если они содержат сложные вложенные действия:
	
	IF NOT flg AND w THEN
	z := TRUE;
	ELSIF flg AND y THEN
	x := TRUE;
	END_IF;




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

Комментарий: нет

5.21. Экземпляры ФБ должны вызываться только раз за цикл ПЛК

Идентификатор: CP20

Приоритет: средний

Языки программирования: все

Ссылки:

- МЭК 61131-8, п. 3.3.3 и 3.7
- Itris Automation Square I6

Описание: Каждый экземпляр функционального блока должен вызываться только раз за цикл ПЛК.

Применение: Вызывайте каждый экземпляр ФБ не более одного раза за цикл ПЛК. Вызов может производиться по условию.

Объяснение: Если экземпляр ФБ вызывается только раз за цикл ПЛК, то в большинстве случаев присвоение значений для его входов происходит только в одном фрагменте кода (если не используется привязка отдельных входов, которая допускается стандартом **МЭК 61131-3**). Это повышает надежность ПО и упрощает его отладку.

- При копировании и последующем использовании фрагментов кода с вызовом экземпляра ФБ легко допустить ошибку, забыв внести изменения в скопированный фрагмент.
- В зависимости от архитектуры ФБ, многократный вызов его экземпляра в пределах одного цикла ПЛК может приводить к некорректным результатам. Некоторые среды разработки в принципе не допускают многократный вызов экземпляров ФБ – так что код с подобной реализацией становится непереносимым.

Экземпляр ФБ, который взаимодействует с физическими входами-выходами ПЛК, должен вызываться только один раз в пределах цикла в соответствии с правилом **CP12** *Физические выходы должны записываться только раз за цикл ПЛК*.

Таким образом, наиболее простой рекомендацией является однократный вызов экземпляра ФБ в пределах цикла ПЛК. Тем не менее, можно представить ситуации, в которых многократный вызов также будет допустим (см. ниже). По возможности изучите документацию на ФБ, чтобы принять оптимальное решение.

Исключения: Существуют случаи, в которых многократный вызов ФБ является разумным и эффективным решением. Опытные разработчики, тщательно проектирующие ПО и проводящие инспекции кода, могут позволить себе использовать эту возможность. Рассмотрим несколько таких ситуаций:

- ФБ типа «счетчик» (например, счетчик суммарного числа различных тревог – очевидно, что в пределах цикла ПЛК может возникнуть несколько тревог);
- ФБ с дополнительным аргументом, который определяет, какую операцию должен выполнить ФБ. Таким образом, в течение одного цикла один экземпляр ФБ может использоваться для выполнения различных операций.
- ФБ, связанные с управлением движением (MotionControl) иногда вызываются несколько раз в пределах цикла ПЛК – например, для вычисления смещения между шагами или размещения в буфере последовательности команд.

При вызове экземпляра ФБ могут опускаться любые необрабатываемые параметры. Экземпляр может вызываться в различных фрагментах кода с разными наборами параметров, и эти вызовы могут происходить в пределах одного цикла. Следует проконтролировать, что ФБ хотя бы раз вызывается с присвоением всех аргументов или что их значения по умолчанию являются корректными.

Примеры:

Неправильно:

```
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);
...
// повторный вызов ФБ
InputDetectionForDashboardButtonA(InputFilterB);

// Информация о предыдущем вызове ФБ была потеряна
// Копирование и вставка (копипаст) привели к ошибке:
// программист отредактировал название аргумента, но забыл про имя экземпляра
```

Комментарий: нет

5.22. Используйте VAR_TEMP для объявления временных переменных

Идентификатор: CP21

Приоритет: средний

Языки программирования: все

Ссылки:

- МЭК 61131-3 (3-я редакция), п. 6.5.2.1
- МЭК 61131-3 (3-я редакция), п. 7.3.3.4.2

Описание: в соответствии со стандартом **МЭК 61131-3**:

- **VAR**
  Переменные, объявленные между ключевыми словами VAR…END_VAR, сохраняют свои значения между вызовами функционального блока или программы. При этом переменные функций не сохраняют свои значения между вызовами.
- **VAR_TEMP**
  Переменные, объявленные между ключевыми словами VAR_TEMP…END_VAR, при каждом вызове POU инициализируются начальными значениями. Для функций и методов ключевые слова VAR и VAR_TEMP эквивалентны.

Применение: Используйте ключевые слова VAR_TEMP для объявления временных переменных – например, счетчика цикла FOR. Важно помнить, что временные переменные инициализируются при каждом вызове ФБ и могут быть изменены только из тела ФБ.

Объяснение: Для функций и методов ключевые слова VAR_TEMP и VAR эквивалентны – но при работе с ФБ и программами следует использовать их для объявления временных переменных.

Исключения: нет

Примеры:

Неправильно:

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

Правильно:

```
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
```

Комментарий: нет

5.23. Используйте для переменных наиболее подходящий тип данных

Идентификатор: CP22

Приоритет: средний

Языки программирования: LD, ST, SFC, FBD

Ссылки:

- МЭК 61131-8, п. 3.1

Описание: Тип данных, выбираемый для переменной, должен зависеть от ее назначения. Тип данных определяет диапазон возможных значений переменной и операции, которые можно над ней производить.

Применение: Выберите для переменной тип данных в соответствии с диапазоном ее значений и допустимыми операциями.

- Используйте тип данных, который предоставляет нужный диапазон значений и при этом занимает минимальный объем памяти;
- Не используйте знаковые типы данных для хранения беззнаковых значений;
- Используйте перечисления (где это возможно), чтобы повысить читабельность кода;
- При необходимости ограничивайте диапазоны значений;
- Используйте массивы (где это возможно), чтобы повысить связность данных одного типа;
- Используйте структуры (где это возможно), чтобы повысить связность данных разных типов;
- Не используйте в программе одни и те же типы данных только для того, чтобы избежать конверсии. Выбирайте наиболее подходящий для каждого случая тип.

Объяснение:

- Подходящий тип данных повышает читабельность POU и упрощает его использование;
- Сильно типизированный код с явными преобразованиями типов менее подвержен ошибкам на этапе отладки, когда типы некоторых переменных проекта могут измениться;
- Компиляторы проверяют соответствие типа данных и контекста использования переменных, что упрощает поиск ошибок;
- Использование подходящих типов данных позволяет сэкономить память;
- Использование беззнаковых типов данных для хранения беззнаковых чисел предотвращает возможность случайного присвоения отрицательного числа и позволяет отказаться от дополнительных проверок;
- Использование перечислений позволяет перейти от численных значений к символьным именам, что повышает читабельность кода. Также использование перечислений и ограничения диапазонов переменных позволяет избежать присвоения ошибочных значений.

Исключения: При настройке обмена с другими устройствами используемый тип данных может зависеть от особенностей конкретного устройства.

Примеры:

- Если переменная может принимать только значения 0 и 1, и используется только в логических операциях – то следует выбрать тип BOOL;
- Если переменная используется в качестве счетчика с диапазоном возможных значений 0…1000 – то для нее не могут быть выбраны типы SINT или USINT, так как диапазон их возможных значений составляет -127…128 и 0..255 соответственно. В данном случае следует использовать тип UINT – он обладает подходящим диапазоном (0…65535) и не может быть использован для хранения отрицательных чисел.
- Если переменная определяет шаг оператора CASE, то для нее должен быть выбран тип данных *Перечисление* (ENUM), а не численный тип данных;
- Перечисление позволяет создать переменную, значения которой ограничиваются выбранными разработчиком идентификаторами:

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

Выше приведен пример объявления перечисления Color. Переменная этого типа может принимать только одно из заданных значений – *Red*, *Yellow* или *Green*. Попытка присвоения ей другого значения (например, числа) приведет к ошибкам компиляции. Стандарт **МЭК 61133-3** не определяет численные значения, связанные с идентификаторами перечислений, и функции для их преобразования.
Другой пример объявления перечисления:

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

Хотя этот тип включения всего два возможных значения, его использование повышает читабельность кода.

- Ограничение диапазонов влияет на возможные значения переменной:

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

- Массивы позволяют повысить связность данных одного типа:

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

- Структуры позволяют повысить связность данных разных типов:

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

 TYPE 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
```

Комментарий: нет

5.24. Определите максимальное число входов и выходов POU

Идентификатор: CP23

Приоритет: средний

Языки программирования: все

Ссылки:

- JSF++ 110

Описание: Определите максимально допустимое число входов, выходов и входов-выходов POU.

Применение: Рекомендуется ограничить максимальное число входов-выходов POU десятью. Если требуется большее количество, то можно воспользоваться одним из следующих вариантов:

1. Применять структуры для группировки нескольких входов-выходов POU;

2. Если некоторые входы POU используются для конфигурации и не требуют вызова в каждом цикле ПЛК, то используйте вместо них переменные класса VAR_CONFIG или локальные переменные со спецификаторами PUBLIC или INTERNAL.

Примечание: спецификаторы доступы определены в 3-й редакции стандарта **МЭК 61131-3**.

Объяснение: Большое число входов-выходов POU затрудняет чтение кода.

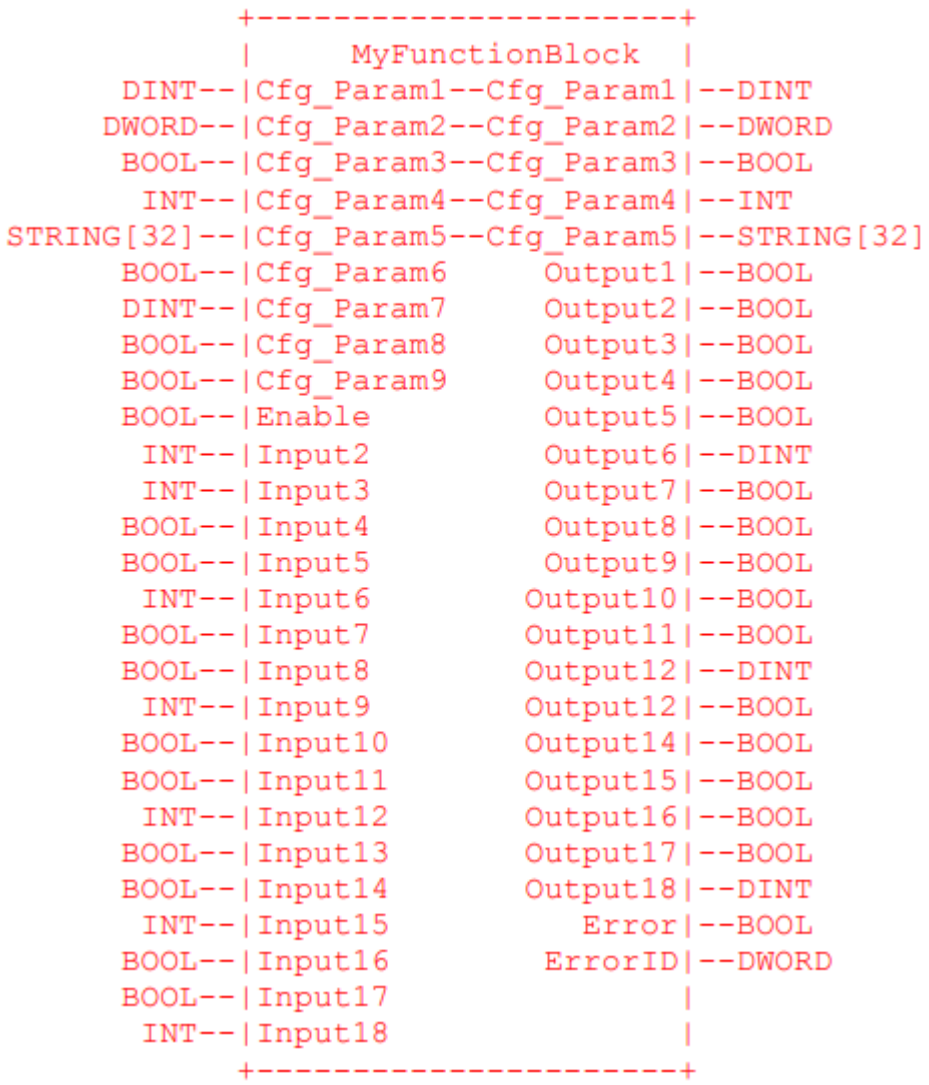
Исключения: Правилом можно пренебречь, если:

- Данный POU вызывается только в POU, написанных на языке ST;
- В применяемой среде разработки имеется возможность скрывать неиспользуемые входы и выходы POU.

Примеры:

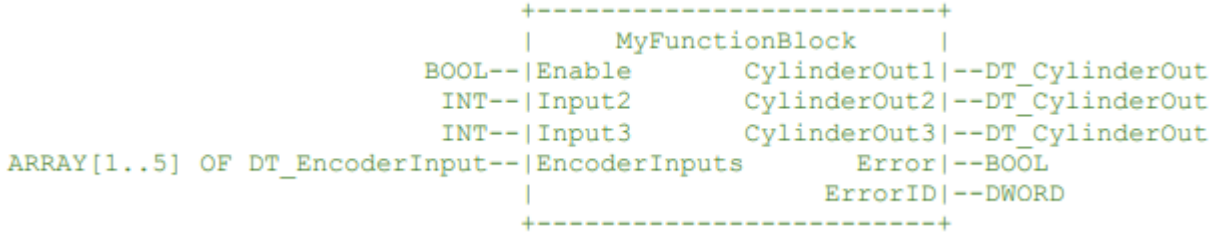
Неправильно:

У этого ФБ слишком много входов и выходов:



Правильно:

Используйте массивы и структуры, чтобы группировать входы и выходы:



```
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 // параметр конфигурации
	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
```

Комментарий: нет

5.25. Все объявленные переменные должны использоваться в коде

Идентификатор: CP24

Приоритет: средний

Языки программирования: все

Ссылки:

- Codesys SA0011, SA0033
- Itris Automation Square S7

Описание: Каждая объявленная переменная должна использоваться в коде.

Применение: Проверьте каждую объявленную переменную и убедитесь, что над ней выполняется хотя бы одна операция в коде.

Объяснение: Если разработка ведется на базе уже существующего приложения, то удаление лишних переменных обычно не является приоритетной задачей, и они могут остаться даже в релизной версии проекта. Это усложняет понимание программы, а также необоснованно увеличивает количество используемой памяти.

Исключения: В некоторых случаях переменные могут быть объявлены «про запас» – например, с целью заранее определить максимально допустимый объем памяти переменных.

Примеры: нет

Комментарий: см. также правило **CP2** *В проекте не должно быть неиспользуемого кода*.

5.26. Используйте явные преобразования типов

Идентификатор: CP25

Приоритет: средний

Языки программирования: все

Ссылки:

- МЭК 61131-3, п. 6.6.1.6 и табл. 11
- Misra-C-2004, п. 10.3
- Codesys SA0019

Описание: Программист должен использовать явные преобразования типов, не оставляя эту задачу компилятору.

Применение: При написании кода, в котором происходит преобразование данных, разработчик должен использовать явные преобразования, не оставляя их выбор на усмотрение компилятора.

Объяснение: При преобразовании типов может произойти нарушение диапазона допустимых значений, потеря точности и потеря знака. Компилятор не обрабатывает эти ситуации (хотя может выдать соответствующие предупреждения), поэтому программист должен добавить в код операторы конверсии. Хорошим подходом является как можно более долгое сохранение исходного типа данных и выполнение преобразований только перед передачей информации (например, в SCADA-систему).

Исключения: Неявные преобразования допустимы, если они не приводят к потере точности и знака (см. табл. 11 из 3-й редакции **МЭК 61131-3**).

Примеры:

Неправильно:

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

I := I * J;

// Если этот код будет скомпилирован…
// …(с помощью компилятора, не соответствующего 3-й редакции МЭК 61131-3)…
// …то результат может быть равен 0 или 5
```

Правильно:

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

или

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

Примечание: Обратите внимание, что для приведенных случаев результат может отличаться. Если пользователь не применяет явных преобразований, то он не будет заранее знать, какое преобразование выполнит компилятор.

Комментарий: нет

5.27. Глобальная переменная должна записываться только одной программой

Идентификатор: CP26

Приоритет: низкий

Языки программирования: все

Ссылки:

- Itris Automatin Square S2

Описание: Глобальная переменная должна записываться только одной программой.

Применение: Каждая глобальная переменная должна записываться только в одной программе.

Объяснение: Описанный подход облегчает чтение и отладку программы.

Исключения: нет

Примеры: нет

Комментарий: нет

5.28. Избегайте использования устаревших возможностей из МЭК 61131-3

Идентификатор: C27

Приоритет: низкий

Языки программирования: LD, ST, SFC, FBD

Ссылки:

- МЭК 61131-3, 3-я редакция
- МЭК 61131-8, п. 3.12

Описание: Избегайте использования устаревших типов данных, функций, ФБ и т.д. Следует предпочесть новые продвинутые методики программирования устаревшим прежним.

Применение: Избегайте всех устаревших возможностей стандарта **МЭК 61131-3**, в т.ч.:

- Двоичного-десятичного типа данных (BCD) и операций над ним;
- Функций и функциональных блоков из 2-й редакции **МЭК 61131-3,** которые были изменены в 3-й редакции стандарта;
- Использования инструкции JUMP (если доступны операторы условного перехода);
- Использования разных функциональных блоков для обработки одних и тех же данных (если доступны классы и методы);
- Некорректного использования глобальных переменных. Используйте локальные переменные для инкапсуляции данных везде, где это возможно. См. также правила CP18 и CP26;
- Привязки экземпляров функциональных блоков к задачам (см. правило CP16).

Объяснение: Использование тех или иных возможностей стандарта **МЭК 61131-3** влияет на качество программного обеспечения. Использование новейших методик разработки позволяет достичь более высокого качества ПО. Устаревшие возможности, с большой вероятностью, будут удалены из следующих версий используемой Вами среды разработки, что приведет к необходимости изменения кода.

Исключения: Устаревшие возможности допустимо использовать в следующих случаях:

- При работе с устройствами, которые не имеют альтернатив этим возможностям;
- При работе с готовыми библиотеками или фрагментами кода, которые используют эти возможности.

Примеры: В 3-й редакции **МЭК 61131-3** следующие возможности описаны как устаревшие и не рекомендуются к использованию:

- Объявление значений в восьмеричной системе счисления (8#377);
- Оператор TRUNC;
- Переменные действий в языке SFC;
- Язык IL.

Комментарий: нет

6. Языки программирования

6.1. Определите размер используемых отступов

Идентификатор: L1

Приоритет: низкий

Языки программирования: ST, IL, область объявления переменных

Ссылки:

- JSF++ 44

Описание: Определите размер отступов, которые вы будете использовать в своем проекте.

Применение: Используйте 4 пробела для каждого отступа.

Объяснение: Использование отступов повышает читабельность кода (особенно в случае использования операторов ветвления и циклов). Небольшие отступы (1-2 пробела) плохо читаются, а слишком большие (8 пробелов) – увеличивают размер строки, что может привести к необходимости ее разбивки на несколько строк.

Исключения: нет

Примеры:

Неправильно:

```
// вложенная конструкция IF…ELSE должна быть размещена на том же уровне…
// …что и вызов ФБ Sort

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

                IF NOT statusTemp THEN // неправильное использование отступов
                        status := statusTemp;
                END_IF;
END_IF;
```

Правильно:

```
// инициализация
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
         // установка флага
         init_graph := false;
END_IF;
```

Комментарий: см. также правило **L16** *Определите правило использования табуляции*.

6.2. Язык функциональных блоков (FBD)

6.2.1. Избегайте получения промежуточных результатов до окончания цепи

Идентификатор: L2

Приоритет: средний

Языки программирования: FBD

Ссылки: нет

Описание: Избегайте получения промежуточных результатов до окончания цепи.

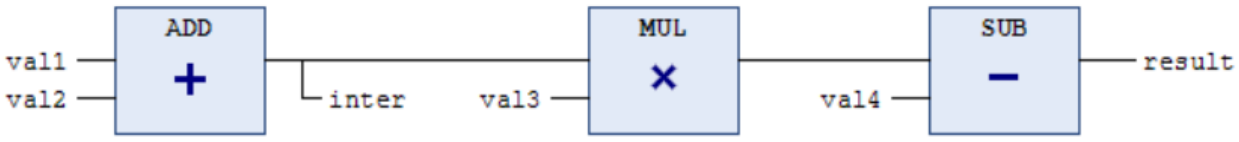
Применение: Присвоение значений переменным должно происходить только в конце цепи.

Объяснение: Получение промежуточных результатов до окончания цепи может привести к непредсказуемым побочным эффектам.

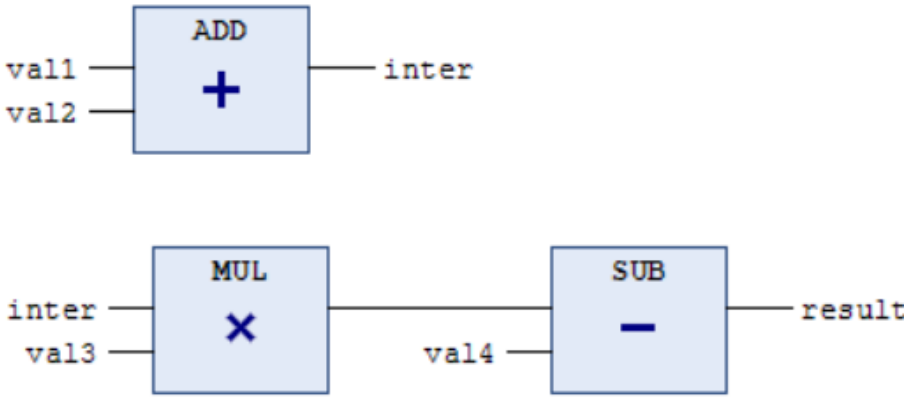
Исключения: нет

Примеры:

Неправильно:



Правильно:



Комментарий: нет

6.2.2. Определите допустимую сложность цепи

Идентификатор: L3

Приоритет: средний

Языки программирования: FBD

Ссылки: нет

Описание: Определите допустимую сложность цепи (например, с помощью ограничения на максимальное число элементов).

Применение: Цепь не должна содержать более 32 элементов.

Объяснение: Схемы с большим числом элементов сложны для восприятия, что затрудняет отладку и повышает вероятность возникновения ошибки. Слишком большие схемы могут не помещаться на дисплее (хотя, конечно, это зависит от размера дисплея и возможности масштабирования в среде разработки). Если очевидно, что цепь является слишком большой – следует разделить ее на несколько отдельных цепей или выделить часть операций в функции/функциональные блоки.

Исключения: нет

Примеры: нет

Комментарий: нет

6.3. Язык релейных диаграмм (LD)

6.3.1. Контакт не должен размещаться после обмотки

Идентификатор: L5

Приоритет: средний

Языки программирования: LD

Ссылки:

- MISRA-C-2004, п. 13.4

Описание: Контакт не должен размещаться после обмотки.

Применение: Вместо размещения обмотки после контакта следует добавить новую цепь.

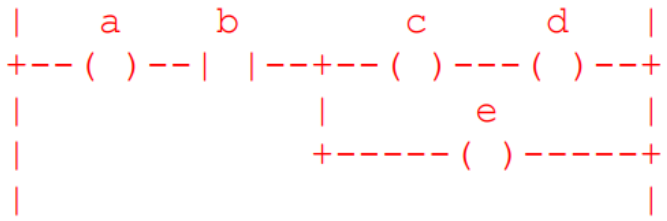
Объяснение: Размещение контактов после обмоток затрудняет чтение схем.

Исключения: нет

Примеры:

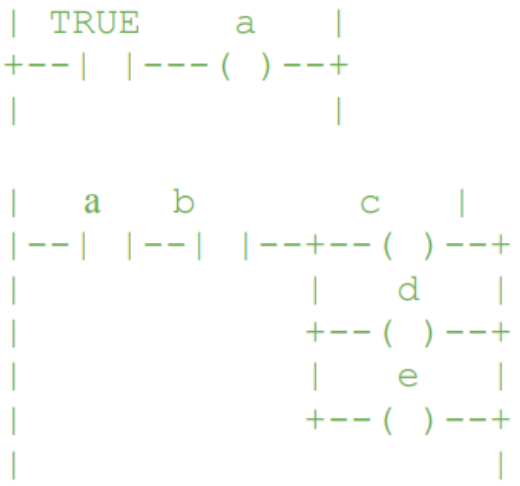
Неправильно:

В стандарте МЭК 61131-3, п. 8.2.5 демонстрируется размещение обмотки после контакта: «в приведенном ниже примере значение переменной *a* всегда TRUE, а значения *c*, *d*, и *e* равны значению *b*».
Проблема: по этой схеме неочевидно, какое значение имеет *a*.



Правильно:

Добавьте дополнительную цепь для присвоения значения:



Комментарий: нет

6.3.2. Определите допустимую сложность цепи

Идентификатор: L6

Приоритет: средний

Языки программирования: LD

Ссылки: нет

Описание: Определите допустимую сложность цепи (например, с помощью ограничения на максимальное число элементов).

Применение: Цепь не должна содержать более 32 элементов.

Объяснение: Схемы с большим числом элементов сложны для восприятия, что затрудняет отладку и повышает вероятность возникновения ошибки. Слишком большие схемы могут не помещаться на дисплее (хотя, конечно, это зависит от размера дисплея и возможности масштабирования в среде разработки). Если очевидно, что цепь является слишком большой – следует разделить ее на несколько отдельных цепей или выделить часть операций в функции/функциональные блоки.

Исключения: нет

Примеры: нет

Комментарий: нет

6.4. Язык последовательных функциональных схем (SFC)

6.4.1. Параллельные ветви должны иметь общие условия начала и окончания

Идентификатор: L7

Приоритет: высокий

Языки программирования: SFC

Ссылки:

- МЭК 61131-3, рис. 18a

Описание: Параллельные ветви должны иметь общие условия начала и окончания.

Применение: Создавайте корректные параллельные ветви, которые имеют общие условия начала и окончания. Если в состав параллельной ветви входит вложенная параллельная ветвь, то она должна быть корректно завершена до завершения родительской ветви.

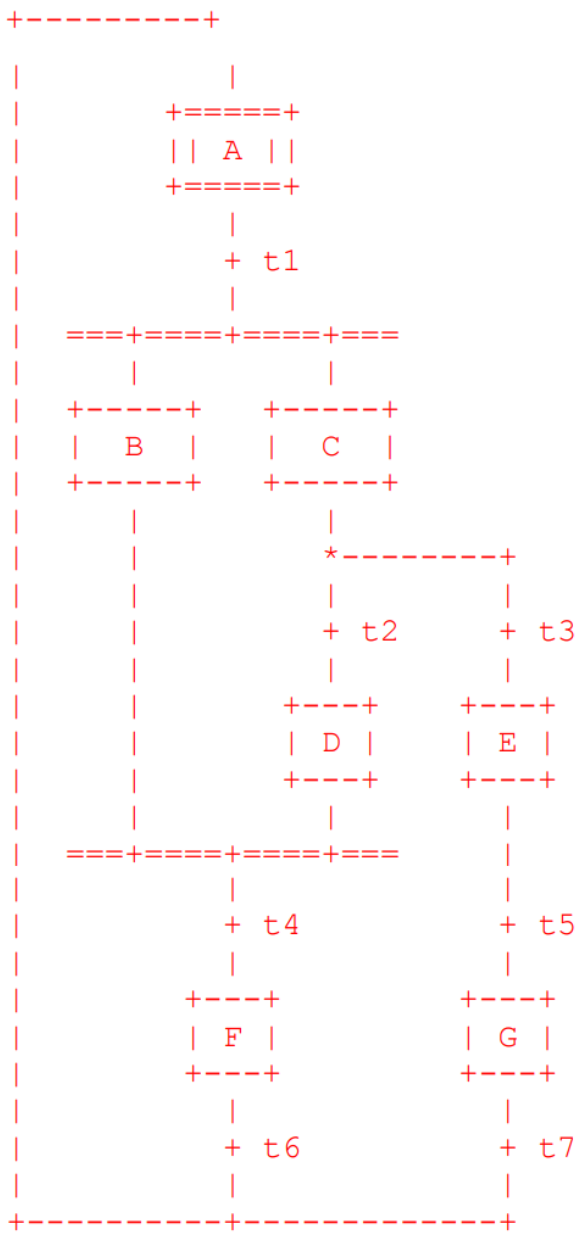
Объяснение: Если параллельные ветви имеют разные условия окончания, то последствия выполнения схемы могут быть непредсказуемыми.

Исключения: нет

Примеры:

Неправильно:

Рассмотрим рис. 18а из стандарта МЭК 61131-3. Предположим, шаги B и C активны, и в этот момент выполняются условия переходов t3, t5 и t7. Произойдет ли выполнение шагов E и G с последующим переходом на шаг A, который расположен до расхождения ветвей? Это совершенно неочевидно и зависит от реализации конкретного ПО.



Комментарий: нет

6.4.2. Действия в SFC-схемах не должны быть написаны на SFC

Идентификатор: L8

Приоритет: средний

Языки программирования: SFC

Ссылки:

- Комитет PLCopen по языку SFC

Описание: Для реализации действий в SFC-схеме может быть выбран любой язык программирования, кроме SFC.

Применение: Используйте язык ST, FBD или LD для кодирования действий. Если в пределах действия нужен конечный автомат – то создайте дополнительный функциональный блок и вызывайте его в этом действии.

Объяснение: Создание действий на языке SFC (иногда называемых «*вложенными SFC-действиями*») усложняет программу и снижает ее читабельность. Стандарт **МЭК 61131-3** не описывает обработку таких действий. В зависимости от реализации конкретного ПО, эти вложенные действия могут выполняться не в цикле ПЛК, а только тогда, когда активен соответствующий шаг – что делает алгоритм выполнения непредсказуемым.

Исключения: нет

Примеры: нет

Комментарий: Большинство сред программирования не позволяют выбрать язык SFC при создании SFC-действия. Такая возможность описана во 2-й редакции стандарта **МЭК 61131-3**, однако в 3-й редакции она была отменена.

6.4.3. Определите допустимую сложность схемы

Идентификатор: L8

Приоритет: средний

Языки программирования: SFC

Ссылки: нет

Описание: Определите допустимую сложность схемы (например, с помощью ограничения на максимальное число шагов).

Применение: Схема не должна содержать более 32 шагов.

Объяснение: Схемы с большим числом элементов сложны для восприятия, что затрудняет отладку и повышает вероятность возникновения ошибки. Слишком большие схемы могут не помещаться на дисплее (хотя, конечно, это зависит от размера дисплея и возможности масштабирования в среде разработки). Если очевидно, что схема является слишком большой – следует разделить ее на несколько отдельных схем.

Исключения: нет

Примеры: нет

Комментарий: нет

6.5. Язык структурированного текста (ST)

6.5.1. Определите правила форматирования кода

Идентификатор: L4

Приоритет: низкий

Языки программирования: ST

Ссылки: нет

Описание: Определите правила форматирования кода и последовательно используйте их в своем приложении.

Применение:

Секция A

Пробелы необходимо ставить:

- до и после каждого оператора;
- после запятой (если она не является последним символом в строке);
- после двоеточия;
- до и после скобок в логических выражениях.

Пробелы не следует ставить:

- между унарными операторами, которые не обозначаются ключевыми словами, и их операндами;
- после открывающей скобки вызова POU и первым параметром, переданным в POU;
- после последнего параметра, переданного в POU и закрывающей скобкой вызова;
- перед открывающей скобкой и после закрывающей скобки в вызовах POU;
- перед точкой с запятой;
- перед двоеточием;
- в конце строки;
- перед открывающей скобкой и после закрывающей скобки массива.

Секция B

- если при вызове POU присвоение его аргументов происходит на отдельных строках – эти строки должны быть выровнены по левому краю;
- операторы IF/THEN (WHILE/DO и т.д.) должны размещаться на одной строке, если это не превышает ее длину (см. правило **L11** *Определите максимально допустимую длину строки кода*);
- если выражение занимает несколько строк, то каждая строка должна начинаться с оператора.

Секция C

- используйте AND вместо ‘&’;
- Используйте TRUE и FALSE вместо ‘1’ и ‘0’ в логических выражениях.

Объяснение: Форматирование кода упрощает его чтение.

Исключения: нет

Примеры:

Неправильно:

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

Правильно:

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

Комментарий: нет

6.5.2. Избегайте использования операторов CONTINUE и EXIT

Идентификатор: L10

Приоритет: средний

Языки программирования: ST

Ссылки:

- Misra-C-2004, п. 14.5
- JSF++ 189, 190

Описание: Следует избегать ветвления с использованием операторов CONTINUE и EXIT в языке ST. Это относится и ко всем специфическим операторам ветвления (GOTO и т.д).

Применение: В подавляющем большинстве случаев можно заменить эти операторы на операторы условного перехода IF…THEN…ELSE или циклы FOR и WHILE.

Объяснение: Использование операторов CONTINUE и EXIT нарушает нормальную последовательность исполнения, что затрудняет отладку приложения. Замена их на конструкции типа IF…THEN…ELSE повысит читабельность кода.

Исключения: В некоторых случаях использование оператора EXIT может повысить ясность и/или производительность кода – например, для выхода из цикла FOR. Использование оператора CONTINUE может помочь избежать множества вложенных операторов IF. Тем не менее, эти случаи встречаются и достаточно редко, и даже тогда упомянутые операторы следует использовать с осторожностью.

Примеры:

Вместо оператора CONTINUE используйте оператор условного перехода IF…THEN.

Вместо оператора EXIT можно изменить оператор цикла на WHILE.

Неправильно:	*Правильно:*

// подсчет числа элементов с ошибкой
Count:= 0;
 	
FOR index:= 1 TO 20 DO
IF NOT bError[index] THEN
// ошибки нет – переходим к след.
// элементу
CONTINUE;
END_IF;
 	
Count:= Count + 1;
 	
END_FOR;
	// подсчет числа элементов с ошибкой
	Count:= 0;
	
	FOR index:= 1 TO 20 DO
	IF bError[index] THEN
	// обнаружен элемент с ошибкой
	Count:= Count + 1;
	END_IF;
	END_FOR;

J:= 101;
 	
FOR index := 1 TO 100 DO
IF WORDS[index] = 'KEY' THEN
J:= index;
EXIT;
END_IF;
END_FOR;
	index:= 1;
	
	WHILE index <= 100 AND WORDS[index] <>
	'KEY' DO
	index:= index+1;
	END_WHILE;
	
	J:=index;

Комментарий: нет

6.5.3. Определите максимально допустимую длину строки кода

Идентификатор: L11

Приоритет: средний

Языки программирования: ST

Ссылки:

- JSF++ 41

Описание: Определите максимально допустимую длину строки кода.

Применение: Ограничьте длину строки кода 80 символами.

Объяснение: Несмотря на то, что некоторые компиляторы и среды разработки поддерживают строки большей длины, их использование снижает читабельность кода. Для просмотра всей строки может потребоваться прокрутка или отдаление изображения – в первом случае теряется эффект от использования отступов, во втором – текст может стать слишком мелким для комфортного чтения (хотя, конечно, это зависит от размера дисплея). Для повышения читабельности и переносимости кода рекомендуется ограничить максимальную длину строки 80 символами.

Исключения:

Неправильно:

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

Правильно:

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

Примеры: нет

Комментарий: нет

6.5.4. Переменная-счетчик не должна изменяться в пределах цикла FOR

Идентификатор: L12

Приоритет: средний

Языки программирования: ST

Ссылки:

- МЭК 61131-3 (3-я редакция), п. 7.3.3.4.2
- MISRA-C-2004, п. 13.6
- JSF++ 188
- Codesys SA007

Описание: В соответствии со стандартом **МЭК 61131-3**: «Начальное значение счетчика, конечное значение счетчика и переменная-счетчик должны принадлежать к одному целочисленному типу данных и не должны изменяться в пределах цикла».

Применение: Запрещается изменение значения переменной-счетчика в пределах цикла FOR.

Объяснение: Цикл FOR используется, если число итераций известно заранее. В противном случае следует использовать циклы WHILE или REPEAT. Изменение значения переменной-счетчика может привести к непредсказуемым последствиям – например, возникновению бесконечного цикла. Отладка приложения с подобными ошибками крайне затруднительна.

Исключения: нет

Примеры:

Неправильно:

```
J := 11;
FOR i := 0 TO 10 BY 2 DO
        IF WORDS[i] = 'Key' THEN
                j := i ;
                i := 10; // изменение значения счетчика цикла
        END_IF;
END_FOR;
```

Правильно:

```
i := 0;
WHILE i <= 10 AND WORDS[i] <> 'KEY' DO
        i := i + 2;
END_WHILE;
j := i;
```

Комментарий: После окончания цикла значение переменной-счетчика зависит от реализации конкретной среды разработки (см. правило **L13** *Переменная-счетчик не должна использоваться за пределами цикла FOR*).

6.5.5. Переменная-счетчик не должна использоваться за пределами цикла FOR

Идентификатор: L13

Приоритет: средний

Языки программирования: ST

Ссылки:

- МЭК 61131-3 (3-я редакция), п. 7.3.3.4.2
- JSF++ 136

Описание: В соответствии со стандартом **МЭК 61131-3**: «Значение переменной-счетчика после окончания цикла не определено и зависит от реализации конкретной среды разработки».

Применение: Не используйте переменную-счетчик за пределами циклами FOR. При необходимости применяйте циклы WHILE и REPEAT.

Объяснение: Использование функционала, реализация которого зависит от конкретного ПО, снижает переносимость приложения.

Исключения: нет

Примеры:

Неправильно:

```
FOR i := 0 TO 100 BY 2 DO
        IF Words[ i] = 'Key' THEN
                EXIT;
        END_IF;
END_FOR;

IF i <= 100 THEN // значение i не определено
        Words[i + 1] := value;
END_IF;
```

Правильно:

```
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;
```

Альтернативный вариант с циклом WHILE:

```
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;
```

Комментарий: нет

6.5.6. Передача аргументов при вызове POU должна быть очевидной

Идентификатор: L14

Приоритет: средний

Языки программирования: ST

Ссылки:

- МЭК 61131-3, п. 2.5.1.1
- МЭК 61131-8, п. 3.2.3
- МЭК 61508, п. 2.9
- JSF++ 58

Описание: При вызове POU должен быть приведен список его параметров с указанием, какие из них являются входами, а какие – выходами.

Применение:

- Для стандартных МЭК-функций и функциональных блоков используйте неформальную передачу аргументов (с перечислением значений без указания названий параметров);
- Для библиотечных и пользовательских функций и функциональных блоков используйте формальную передачу аргументов с поименным присваиванием значений параметрам POU при помощи операторов ‘:=’ для входов и ‘=>’ для выходов. Параметры должны быть сгруппированы в соответствие с их классом: входы, выходы, входы-выходы.
- Если число параметров POU велико или некоторые из них должны быть опущены – используйте формальную передачу аргументов с размещением имени каждого параметра на отдельной строке, сопровождаемой комментарием.

Объяснение: Использование имен параметров делает код понятным и однозначным. Операторы ‘:=’ и ‘=>’ позволяют определить область переменной (вход или выход). Чтобы повысить читабельность кода, присвоение аргументов может производиться на отдельных строках.

Все стандартные МЭК-операторы (например, логические и арифметические) должны быть переносимыми, поэтому при работе с ними следует использовать неформальный вызов (см. **МЭК 61131-3**, п. 2.5.1.1).

Исключения: нет

Примеры:

Неправильно:

```
// избегайте неформального вызова пользовательских ФБ и длинных строк

A := MyFunction(Bee, 5);
MC_Home(Axis:=var0, Execute:=var1, Done=>var2, Busy=>var3,
CommandAborted=>var4, Error=>var5, ErrorID=>var6);
```

Правильно:

```
// Используйте формальный вызов: MyFunction не может изменить параметр Bee

A := MyFunction(In:=Bee, Max:=5);
MC_Home(Axis:=ShuttleAxis,          // устанавливаемая ось
Execute:=DoHome,                    // сигнал установки
Done=>HomeDone,                     // флаг «ось установлена»
Busy=>ShuttleMoving,                // флаг «ФБ в процессе работы»
CommandAborted=>HomeAborted,        // флаг «выполнение ФБ прервано»
Error=>HomeError,                   // флаг ошибки
ErrorID=>HomeErrorID);              // код ошибки
```

Комментарий: нет

6.5.7. Используйте скобки для определения порядка выполнения операций

Идентификатор: L15

Приоритет: средний

Языки программирования: ST

Ссылки:

- MISRA-C-2004, п. 12.1
- JSF++ 213

Описание: При использовании операторов с одинаковым приоритетом используйте скобки, чтобы определить порядок их выполнения.

Применение: При работе с операторами AND/OR и -/+ используйте скобки, чтобы определить порядок выполнения операторов.

Объяснение: Приоритет операторов может зависеть от среды разработки – это будет являться неочевидным и затруднит отладку приложения. Использование скобок позволяет явно задать порядок выполнения операций.

Исключения: нет

Примеры:

Неправильно:

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

Правильно:

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

или

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

Комментарий: нет

6.5.8. Определите правило использования табуляции

Идентификатор: L16

Приоритет: низкий

Языки программирования: ST

Ссылки:

- JSF++ 43

Описание: Определите правило использования табуляции и последовательно используйте его в своем приложении.

Применение: Избегайте использования табуляции (управляющего символа с ASCII-кодом 0x09). Среда разработки должна заменять табуляцию пробелами (если такая возможность поддерживается).

Объяснение: Визуальное представление табуляции может отличаться в зависимости от используемой среды разработки – 4 пробела, 8 пробелов или, например, неопределенное число пробелов до спецсимвола остановки табуляции. Поэтому табуляцию не рекомендуется использовать при разработке переносимого ПО.

Исключения: нет

Примеры: нет

Комментарий: Вы можете настроить автозамену табуляции на пробелы (если ваша среда разработки поддерживает такую возможность).

6.5.9. Каждая конструкция IF должна содержать ветку ELSE

Идентификатор: L17

Приоритет: низкий (высокий при разработке безопасного ПО)

Языки программирования: ST

Ссылки:

- MISRA-C-2004, п. 14.10
- JSF++ 192
- Itris Automation Square S12
- Codesys SA0075

Описание: Каждая конструкция IF должна содержать ветку ELSE.

Применение: Для каждой конструкции IF разработчик должен добавить ветку ELSE и определить действия, которые должны выполняться в этом случае.

Объяснение: Это принцип защитного программирования. Разработчик всегда должен учитывать, что в определенных ситуациях условие не будет выполнено – и эти случаи требуется обрабатывать в коде.

Исключения: Правилом можно пренебречь при разработке ПО, которое не имеет требований к безопасности.

Примеры:

Неправильно:

```
IF some_condition THEN
        // какой-то код
        ...
END_IF;
```

Правильно:

```
IF some_condition THEN
        // какой-то код
        ...
ELSE
        // ничего не происходит (так и задумано)
        ;
END_IF;
```

Комментарий: нет

7. Правила для расширений стандарта МЭК 61131-3

7.1. Избегайте динамического распределения памяти

Идентификатор: E1

Приоритет: высокий

Языки программирования: все

Ссылки:

- MISRA-C-2004, п. 20.4

Описание: Не используйте специфических возможностей ПЛК, которые позволяют реализовать динамическое распределение памяти, а также не создавайте собственный механизм подобного распределения в своем ПО.

Применение: Избегайте динамического распределения памяти.

Объяснение: Динамическое распределение памяти не определено стандартом **МЭК 61131-3**, не документировано и реализация этого механизма зависит от конкретного устройства. Это может привести к утечкам памяти, потере данных и другим непредсказуемым последствиям.

Исключения: нет

Примеры: нет

Комментарий: нет

7.2. Запрещается использовать арифметические операции над указателями

Идентификатор: E2

Приоритет: высокий

Языки программирования: ST, IL

Ссылки:

- МЭК 61131-3 – *понятие «указатель» не используется во 2-й и 3-й редакциях стандарта. В 3-й редакции были добавлены ссылки (REFERENCE), для которых определены исключительно операции присваивания и сравнения с нулем.*
- MISRA-C-2004, п. 17.1

Описание: Единственные арифметические операции, которые можно проводить над указателями – это проверка на равенство и неравенство. Разработчик не должен использовать операции сложения/вычитания над указателями, чтобы получить доступ к какому-либо объекту в памяти ПЛК.

Применение: Арифметические операции над указателями могут быть использованы только в одном случае – для доступа к элементам массива.

Объяснение: Результат арифметических операций над указателями зависит от реализации конкретного ПЛК и требует глубокого понимания принципов работы его внутренних компонентов. Ранее возможность таких операций была приемлема только по причине отсутствия высокоуровневых средств (массивов, структур, функциональных блоков) – но теперь эти средства доступны, и следует использовать именно их. Кроме того, согласно правилу N1, разработчик не должен использовать физические адреса в своем приложении.

Исключения: нет

Примеры: нет

Комментарий: нет

7.3. Операции сравнения не должны выполняться над указателями и ссылками

Идентификатор: E3

Приоритет: высокий

Языки программирования: все

Ссылки:

- Codesys SA0061
- MISRA-C-2004, п. 17.3

Описание: Операции сравнения (<, >, <=, >=) не должны выполняться над указателями и ссылками – разрешены лишь операции проверки на равенство и неравенство. При необходимости следует использовать массивы для явного определения порядка элементов в памяти.

Применение: Запрещается использовать операции сравнения над указателями и ссылками.

Объяснение: Сравнение указателей и ссылок опирается на представление об устройстве памяти конкретного ПЛК. В большинстве случаев оно либо не документировано, либо сложно для понимания, что затрудняет отладку приложения.

Исключения: нет

Примеры:

Неправильно:

```
VAR
        x: POINTER;
        y: POINTER;
END_VAR

…
IF X^ < Y^ AND X < Y THEN        // операции сравнения над указателями запрещены
        TEMP := X^:
        X^ := Y^;
         Y^ := TEMP;
END_IF;
```

Правильно:

```
// используйте массивы вместо сравнения указателей
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;
```

Комментарий: нет

8. Приложение 1 – Список правил с сортировкой по приоритету

Приоритет	ID	Пункт	Название	Страница
Высокий	N1	3.1.1	Не используйте физические адреса	13
Высокий	N3	3.2.1	Определите имена, которые нельзя использовать	17
Высокий	N4	3.2.2	Определите регистр текста составных названий	19
Высокий	N5	3.2.3	Имена глобальных и локальных объектов не должны совпадать	22
Высокий	C1	4.1	Комментарии должны описывать назначение кода	31
Высокий	C2	4.2	Все объекты должны быть прокомментированы	33
Высокий	CP1	5.1	Доступ к вложенному элементу должен производиться по его имени	39
Высокий	CP2	5.2	В проекте не должно быть неиспользуемого кода	40
Высокий	CP3	5.3	Все переменные должны инициализироваться начальными значениями	42
Высокий	CP4	5.4	Избегайте наложения адресов	46
Высокий	CP5	5.5	Перед началом кодирования выполняется проектирование	48
Высокий	CP6	5.6	Избегайте использования в POU внешних переменных	49
Высокий	CP7	5.7	Осуществляйте обработку ошибок	51
Высокий	CP8	5.8	Значения с плавающей точкой не должны проверяться на равенство и неравенство	53
Высокий	CP28	5.9	Значения времени и физические величины не должны проверяться на равенство и неравенство	55
Высокий	CP9	5.10	Определите допустимую сложность POU	57
Высокий	CP10	5.11	Избегайте записи переменной из разных задач	60
Высокий	CP11	5.12	Синхронизируйте выполнение задач	62
Высокий	CP12	5.13	Физические выходы должны записываться только раз за цикл ПЛК	65
Высокий	CP13	5.14	POU не должны вызывать сами себя	66
Высокий	CP14	5.15	POU должен иметь одну точку выхода	68
Высокий	CP15	5.16	Считывайте переменную, записываемую другой задачей, только один раз за цикл ПЛК	69
Высокий	CP16	5.17	Функции и функциональные блоки не должны быть привязаны к задачам	72
Высокий	CP17	5.18	Функции и функциональные блоки не должны быть привязаны к задачам	73
Высокий	CP18	5.19	Разумно используйте глобальные переменные	75
Высокий	L7	6.5.1	Определите правила форматирования кода	108
Высокий	E1	7.1	Избегайте динамического распределения памяти	122
Высокий	E2	7.2	Запрещается использовать арифметические операции над указателями	123
Высокий	E3	7.3	Операции сравнения не должны выполняться над указателями и ссылками	124
Средний	N6	3.2.4	Определите приемлемую длину имен	24
Средний	N7	3.2.5	Определите правила использования пространств имен	26
Средний	N8	3.2.6	Определите используемый набор символов	28
Средний	N9	3.2.7	Объекты различных типов не должны иметь одинаковые имена	29
Средний	CP19	5.20	Избегайте использования операторов JUMP и RETURN	79
Средний	CP20	5.21	Экземпляры ФБ должны вызываться только раз за цикл ПЛК	82
Средний	CP21	5.22	Используйте VAR_TEMP для объявления временных переменных	84

Средний	CP22	5.23	Используйте для переменных наиболее подходящий тип данных	86
Средний	CP23	5.24	Определите максимальное число входов и выходов POU	89
Средний	CP24	5.25	Все объявленные переменные должны использоваться в коде	92
Средний	CP25	5.26	Используйте явные преобразования типов	93
Средний	L2	6.2.1	Избегайте получения промежуточных результатов до окончания цепи	99
Средний	L3	6.2.2	Определите допустимую сложность цепи	100
Средний	L5	6.3.1	Контакт не должен размещаться после обмотки	101
Средний	L6	6.3.2	Определите допустимую сложность цепи	103
Средний	L8	6.4.2	Действия в SFC-схемах не должны быть написаны на SFC	106
Средний	L9	6.4.3	Определите допустимую сложность схемы	107
Средний	L10	6.5.2	Избегайте использования операторов CONTINUE и EXIT	110
Средний	L11	6.5.3	Определите максимально допустимую длину строки кода	112
Средний	L22	6.5.4	Переменная-счетчик не должна изменяться в пределах цикла FOR	113
Средний	L13	6.5.5	Переменная-счетчик не должна использоваться за пределами цикла FOR	115
Средний	L14	6.5.6	Передача аргументов при вызове POU должна быть очевидной	117
Средний	L15	6.5.7	Используйте скобки для определения порядка выполнения операций	119
Низкий	N2	3.1.2	Применяйте префиксы	17
Низкий	N10	3.2.8	Применяйте префиксы для пользовательских типов данных	30
Низкий	C3	4.3	Не используйте вложенные комментарии	34
Низкий	C4	4.4	Комментарии не должны содержать код	35
Низкий	C5	4.5	Используйте однострочные комментарии	36
Низкий	C6	4.6	Определите язык комментариев	38
Низкий	CP26	5.27	Глобальная переменная должна записываться только одной программой	85
Низкий	CP27	5.28	Избегайте использования устаревших возможностей из МЭК 61131-3	96
Низкий	L1	6.1	Определите размер используемых отступов	97
Низкий	L4	6.5.1	Определите правила форматирования кода	108
Низкий	L16	6.5.8	Определите правило использования табуляции	120
Низкий	L17	6.5.9	Каждая конструкция IF должна содержать ветку ELSE	121