Среда исполнения

AXCode Runtime

на базе операционной системы реального времени

Руководство
по программированию для пользователя

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Данный документ описывает основные принципы работы пользователя со средой исполнения AXCode Runtime.

Так как среда исполнения предназначения для работы на программируемых логических контроллерах (далее ПЛК), то при описании использован в качестве примера ПЛК модели Alpha-X CPU.

Далее будут рассмотрены:

- основные термины и определения
- схемы подключения, органы управления и индикации, а также технические характеристики используемого в данном документе оборудования
- процедура установки комплекта необходимого программного обеспечения на персональный компьютер
- процедура компиляции и загрузки программы пользователя на целевое устройство
- принципы и примеры работы с пользовательскими программными блоками
- способы отладки
- работа с интерфейсами связи
- работа с часами реального времени
- работа с энергонезависимой памятью
- описание ошибок и событий

РАЗДЕЛ 1: ОБЩАЯ ИНФОРМАЦИЯ

Терминология

Базовые понятия программирования

Переменная

Ячейка или область в оперативной памяти, которая может хранить данные определенного типа. Переменная может принимать различные значения (в каждый момент времени только одно значение).

Пример создания целочисленной переменной:

```
int variable;
```

Модификатор

Ключевое слово языка программирования, наделяющее переменную теми или иными свойствами. К модификаторам относятся `static`, `extern` и т.п.

Константа

Элемент языка, указывающий на объект данных с фиксированным значением.

Пример создания вещественной константы:

```
const float pi = 3.1416;
```

Перечисление

Список целочисленных констант (0, 1, 2 или 10, 20, 24) одного типа, при этом каждой из констант присвоено имя.

Переменная данного типа может принимать значения только из предварительно заданного, именованного набора возможных значений. Например, перечисление COLOR со значениями RED, GREEN, BLUE для программы будет восприниматься как 0, 1, 2.

Перечисления необходимы для удобства организации программного кода. К примеру, чтобы не держать номера требуемых Modbus-регистров в голове, программист может создать перечисление с картой требуемых регистров:

```
// Карта регистров ПИД-регулятора ECD1
enum ECD2_MAP {
  // Показания измерительного входа
  CURRENT_VALUE = 0,
```

```
    // Уставка логического устройства 1
    SET_VALUE = 1,
    // Гистерезис логического устройства 1
    HYST = 4,
};
```

Структура

Набор переменных и функций, объединённых общим названием типа, используемый для совместного хранения информации.

```
struct Devices
{
    etl::string<128> name;
    uint8_t address;
};
```

Класс

Определяемый пользователем тип, который содержит данные и функции. При написании программы пользователь создает экземпляры (объекты) класса и взаимодействует с ними. В языке C++ структуры и классы являются идентичными понятиями с той лишь разницей, что структуры по умолчанию открыты для доступа, а классы — закрыты.

```
class Devices
{
    etl::string<128> name;
    uint8_t address;
};
```

Понятия платформы AXCode

AXCode

Комплект программного обеспечения для разработки программы пользователя.

Шаблон программы пользователя AXCodeTemplate

Готовый шаблон разработки представляет собой набор папок и файлов, содержащих описание всех функций ядра и включающих все необходимые связи между отдельными файлами с кодом. Шаблон программы пользователя состоит из файлов, предназначенных для написания пользовательской программы.

Компилятор

Это компьютерная программа, которая осуществляет перевод исходного кода на языке C/C++ в эквивалентный ей код на языке машинных команд. В рамках AXCode под компилятором понимается компилятор из языка C/C++ в машинный код. В AXCode имеется несколько компиляторов, нужный выбирается в зависимости от используемой модификации ПЛК.

Модификация	Компилятор
CPU 01-1 00	The xPack GNU RISC-V Embedded GCC
CPU E1-1 00	The xPack GNU Arm Embedded GCC

AXCodeLoader

Компьютерная программа, предназначенная для загрузки ядра с программой пользователя в ПЛК посредством интерфейсов RS-485, USB или Ethernet.

Bootloader

Независимый внутренний программный блок в ПЛК, позволяющий загружать программу пользователя, используя RS-485, Ethernet или USB.

Программа пользователя

Скомпилированные ядро и пользовательский код в формате HEX, предназначенные для загрузки на ПЛК.

Ядро

Совокупность операционной системы и системных задач, обеспечивающая программе пользователя координированный доступ к аппаратным ресурсам ПЛК.

pou (Program Organization Unit, Пользовательская подпрограмма)

Общее название пользовательских подпрограмм. Пользователь может создать несколько pou, при этом все они работают в режиме невытесняющей многозадачности с возможностью задания временного интервала вызова каждой подпрограммы.

Функциональный блок

Класс, поведение которого строго определено и происходит в соответствии с моделью поведения PLCopen. В проекте используются функциональные блока типа Execute и Enable. Подробное описание модели поведения приводится в Функциональные блоки PLCopen.

Автоматический опрос

Программный функционал платформы AXCode, позволяющий работать с внешними входами и выходами модулей расширения без необходимости использования функций работы с Modbus.

Сниппет

Готовые фрагменты кода, которые можно использовать для ускорения процесса написания кода. Как правило, сниппеты используются при объявлении и инициализации функциональных блоков. При наборе названия ФБ, редактор сам предлагает использовать соответствующий сниппет.

Свободный / произвольный протокол

Любой протокол, отличный от Modbus, который для передачи сообщений использует интерфейсы RS-232 или RS-485. Ядро AXCode поддерживает работу с произвольными протоколами с помощью специальных функциональных блоков.

Аппаратные понятия

Встроенные входы и выходы

Дискретные или аналоговые входы и выходы, которые физически находятся непосредственно на ПЛК.

Внешние входы и выходы

Дискретные или аналоговые входы и выходы, которые физически находятся на модулях расширения.

COM-порт

Интерфейс RS-485 или RS-232, предназначенный для связи с внешними устройствами.

Прочие понятия

Компоновщик (линковщик, линкер)

Утилита, которая производит связывание воедино всех объектных (скомпилированных) файлов проекта. Компоновщик управляет распределением памяти.

Система сборки (сборщик)

Это программная утилита, производящая выбор требуемых файлов программы, в зависимости от соответствующих настроек платы/микроконтроллера. Система сборки выбирает только требуемые для текущего микроконтроллера (или платы) файлы еще до компиляции проекта. Используются системы сборки “The xPack Ninja Build” и “The xPack Cmake”.

Терминал (консоль)

Инструмент, который позволяет взаимодействовать с ПЛК через командную строку.

Блокирующий вызов

Блокирующими считаются те операции, при выполнении которых текущая задача вынуждена ждать их окончания, прежде чем начать выполнять следующий код.

Неблокирующий вызов

Операции, при выполнении которых текущая задача не ждет завершения операции, а сразу выполняет следующий код. Неблокирующие операции, как правило, могут выполняться за несколько циклов задачи, в которой они были вызваны. Наглядный пример - функциональный блок Modbus.

Протокол

Набор команд и правил, позволяющий осуществлять обмен данными между двумя и более включенными в сеть устройствами. Устройства, работающие под управлением системы AXCode поддерживают протоколы Modbus RTU и Modbus TCP.

Прямая работа с портом

Обобщенное название процесса работы с интерфейсами RS-485, Ethernet и USB, при котором пользователь может формировать и принимать произвольную текстовую или бинарную посылку, то есть используя произвольный протокол.

Назначение прибора

Целевое является программируемым логическим контроллером (ПЛК). Предназначен для создания автоматизированных комплексов управления технологическими процессами в различных сферах: промышленность, энергетика, транспорт, инженерные системы зданий, системы диспетчеризации и т.п.

ПЛК является свободно программируемым устройством, логика его работы определяется пользователем самостоятельно, посредством написания программы управления на языке C / C++.

Общие принципы работы ПЛК

Процесс работы ПЛК можно условно разделить на 3 основных этапа:

- Чтение собственных входов;
- Вычисление значений выходных и внутренних переменных на основании программы пользователя;
- Установка собственных выходов.

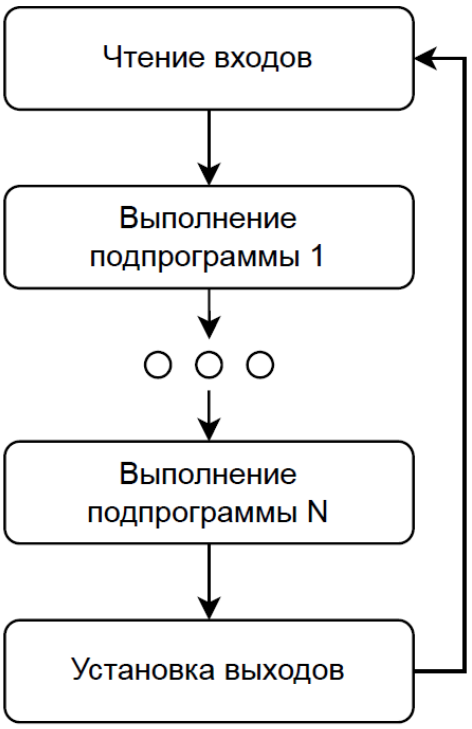


Диаграмма работы ПЛК

В ПЛК, работающих на основе платформы AXCode дополнительно в “фоновом режиме” выполняются несколько системных задач, отвечающих за индикацию, сохранение энергонезависимых переменных, обмен по интерфейсам и прочее. Операционная система реального времени гарантирует, что долгое выполнение либо зависание кода подпрограмм пользователя не влияет на нормальную работу системных задач.

Технические характеристики

Аппаратный состав модулей CPU		
Модификация	Alpha-X CPU 01-1 00	Alpha-X CPU E1-0 00
Количество дискретных входов DI	4	-
Количество дискретных выходов DO	2	-
Количество аналоговых входов AI	-	
Количество аналоговых выходов AO	-	
COM-интерфейсы	3 x RS-485	
USB-интерфейсы	1 x USB 2.0, Device, Type-C	
Ethernet-интерфейсы	-	1, разъем
Часы реального времени	+	
Карта памяти microSD	-	+

Общие параметры		
Модификация	Alpha-X CPU 01-1 00	Alpha-X CPU E1-0 00
Напряжение питания	22...26 В постоянного тока	
Собственное энергопотребление	< 2 Вт	< 4.5 Вт
Максимальное энергопотребление с учетом тока входов	< 3 Вт	< 4.5 Вт
Степень защиты	IP20	
Срок службы	10 лет	

Условия эксплуатации	
Рабочая температура воздуха	-20…50 °С
Относительная влажность воздуха	10…80 % (без образования конденсата)

Интерфейсы RS-485	
Поддерживаемые протоколы	Modbus RTU (Master / Slave), свободный протокол
Максимальная скорость обмена	256 000 б/с
Гальваническая изоляция	1500 В

Интерфейс USB	
Стандарт	USB 2.0
Тип	USB-Device
Гальваническая изоляция	нет

Параметры дискретных входов	
Типы подключаемых датчиков	PNP / NPN
Макс.ток потребления одного входа	7,5 мА
Уровень логического 0 (выключен)	0...4 В
Уровень логической 1 (включен)	11...28 В

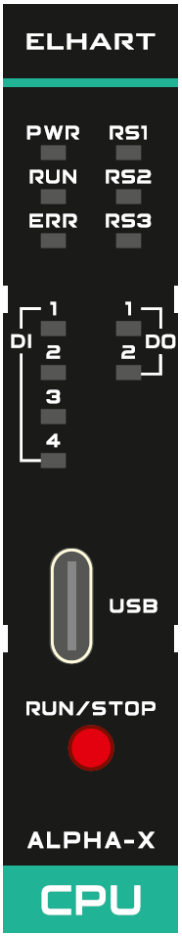
Параметры дискретных выходов	
Тип выходов	Транзисторный, PNP

Падение напряжения	0,7 В
Максимальное допустимое напряжение	28 В постоянного тока
Максимальный ток одного выхода	0,25 А
Максимальный ток группы выходов	2 А

Параметры программы	
Размер пользовательской программы	Зависит от версии пользовательского шаблона AXCodeTemplate, но не менее 64 кБ Flash, 32 кБ RAM
Энергонезависимая память	4 кБ
Минимальное время цикла программы	10 мс
Язык программирования	C / C++
Операционная система	FreeRTOS

Параметры часов реального времени	
Тип батареи RTC	CR2032
Срок службы батареи	10 лет
Погрешность счета времени	2 мин в месяц при +25 °C

Описание органов управления и индикации



Индикаторы

PWR – индикатор наличия питания (зеленый):

- горит - питание в норме;

RSx – индикаторы передачи данных по RS-485 (красный):

- горит / мигает - наличие сигнала в линиях RS-485: COM1, COM2 и COM3 соответственно;

RUN – индикатор выполнения программы пользователя (зеленый):

- горит - режим RUN, модуль работает в соответствии с логикой пользовательской программы;
- мигает - режим BOOT, ядро не запущено, модуль находится в состоянии загрузчика;

ERR – индикатор наличия ошибок (красный):

- горит - наличие записей в журнале ошибок. Модуль корректно выполняет логику программы пользователя только если это возможно при текущих ошибках;
- мигает - режим SERVICE, модуль не выполняет логику программы пользователя;

DIx – индикаторы наличия сигнала на дискретных входах (зелёный):

- горит - есть сигнал на дискретном входе;

DOx – индикаторы включения дискретных выходов (красный):

- горит - дискретный выход замкнут;

Кнопки

RUN/STOP – кнопка переключения режимов работы ПЛК:

- кратковременное нажатие - смена режимов RUN / STOP;
- длительное нажатие более 5 сек - переход в режим BOOT;

Режимы работы

Режим BOOT (ожидание прошивки)

PWR	RUN	ERR
Горит	*Мигает*	*Не влияет*

Режим, при котором не запущено ядро ПЛК. Этот режим нужен только для обновления программы. В данный режим ПЛК переходит автоматически по команде от AXCodeLoader, если соответствующий порт настроен на Slave или Debug. В режиме BOOT ПЛК не отвечает на сообщения от терминала.

Если используемый для загрузки программы порт ПЛК настроен как Modbus Master или произвольный протокол, то для обновления программы потребуется перевести ПЛК в режим BOOT вручную одним из следующих способов:

- нажать и удерживать кнопку RUN/STOP в течении 5 секунд;
- нажать и удерживать кнопку RUN/STOP при подаче питания;

После перевода вручную ПЛК будет находиться в режиме BOOT в течение 60 секунд. Если в этот период не начнется загрузка новой программы пользователя, ПЛК запустит ту программу, которая была загружена ранее. Если в режиме BOOT горит светодиод ERR, значит ПЛК не имеет никакой программы, либо она не может быть запущена.

Режим RUN (работа)

PWR	RUN	ERR
Горит	*Горит*	*Не влияет*

Стандартный режим работы, при котором запущено ядро и происходит выполнение подпрограмм пользователя.

Режим STOP (остановка)

PWR	RUN	ERR
Горит	*Не горит*	*Не влияет*

Режим, при котором запущено ядро и подпрограммы пользователя не выполняются. При этом коммуникационные порты ПЛК будут сконфигурированы

в соответствии с настройками программы пользователя и ПЛК будет отвечать на внешние запросы по интерфейсам.

Перевод в STOP из режима RUN осуществляется однократным нажатием кнопки RUN/STOP или командой kernel stop в отладочном терминале. Также, возможно попадание в этот режим по срабатыванию сторожевого таймера (Watchdog) одной из подпрограмм (Параметры pou и pou_manager).

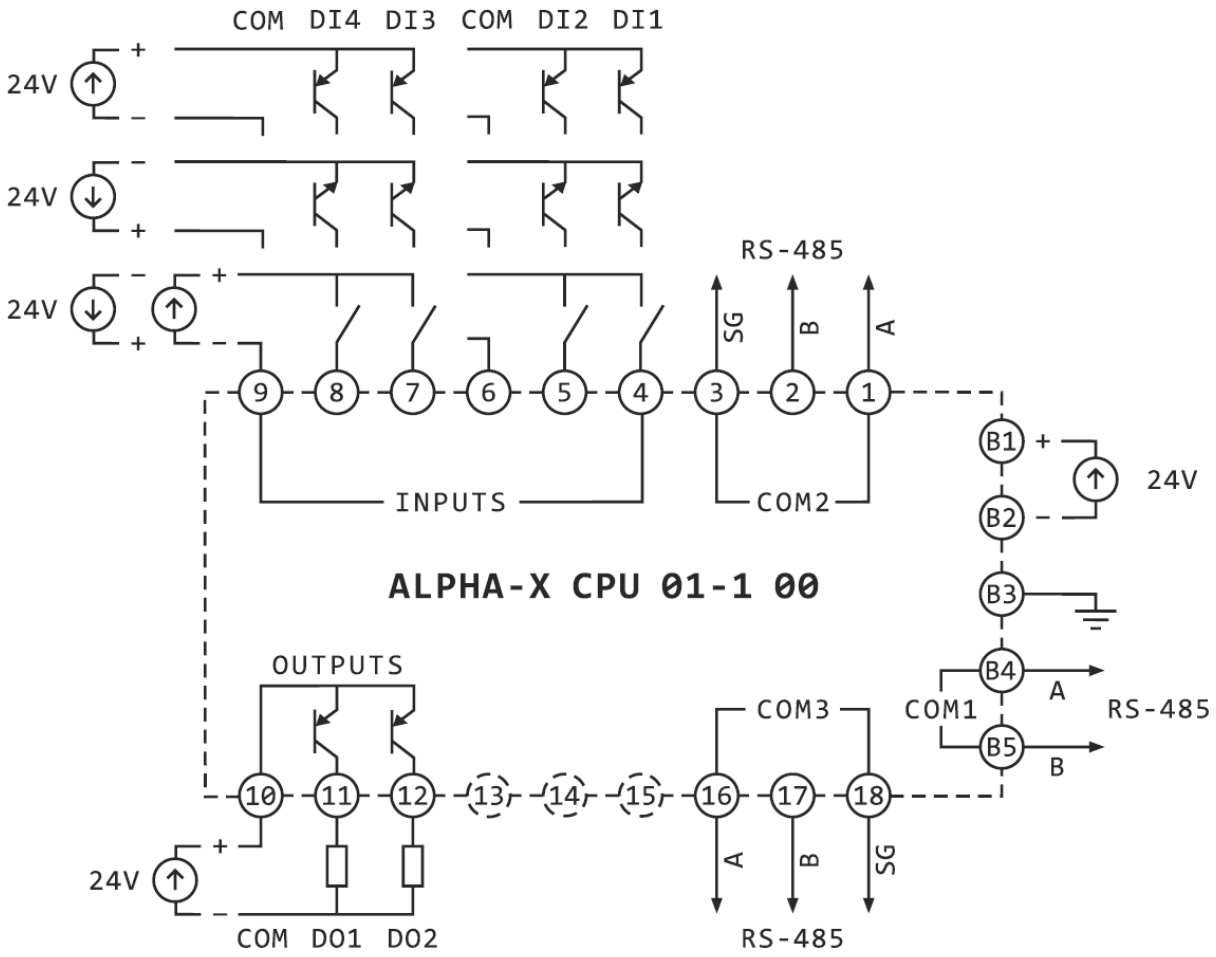
Режим SERVICE (сервисный)

PWR	RUN	ERR
Горит	*Не горит*	*Мигает*

Режим, при котором ядро запущено, однако задача с подпрограммами пользователя не была создана. При этом коммуникационные порты RS-485 будут иметь настройки по умолчанию: скорость 115200 и адрес 1, режим Slave Modbus RTU. Переход в SERVICE из режима RUN осуществляется командой reload service в отладочном терминале или по срабатыванию Watchdog или Hardfault. Кратковременное нажатие кнопки RUN/STOP перезагрузит ПЛК в режим RUN.

Данный режим необходим для сбора отладочной информации при неполадках или для использования ПЛК в режиме повторителя для настройки модулей ввода-вывода через конфигуратор ELHART.

Схема подключения



РАЗДЕЛ 2: ПОДГОТОВКА К РАБОТЕ

Установка комплекта ПО AXCode

1. Для программирования ПЛК используется комплект программного обеспечения **AXCode**, который включает в себя всё необходимое для разработки, загрузки и отладки кода: open-source редактор кода Code-OSS, компилятор, сборщик, загрузчик, расширение для работы с ПЛК и прочее. В зависимости от операционной системы требуется выбрать версию для Win10 или для Win7-8.

Дополнительно на этом же сайте потребуется скачать актуальный шаблон пользовательской программы **AXCodeTemplate**.

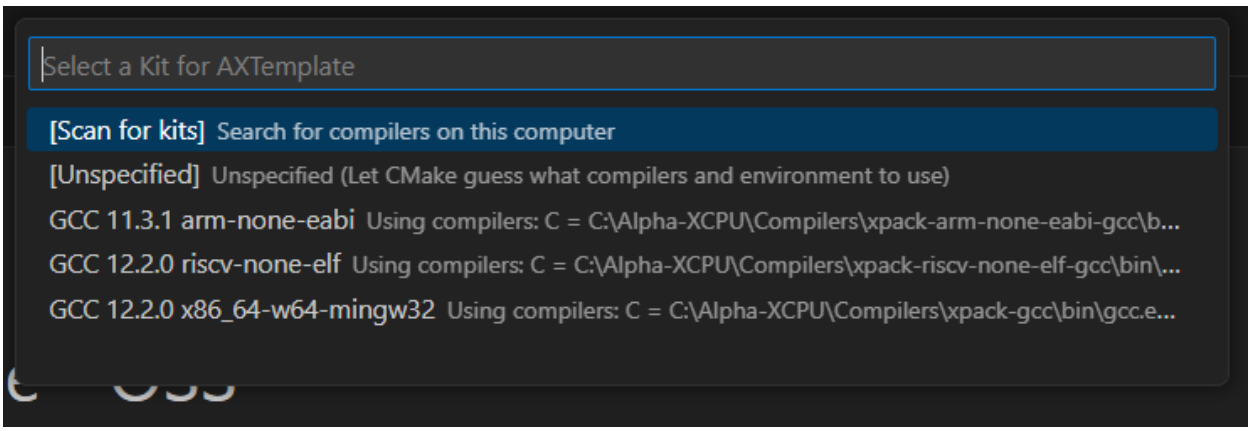
2. Шаблон пользовательской программы **AXCodeTemplate** распаковать. Файлы с кодом пользовательской программы будут находиться в распакованной папке.

	Устанавливать ПО **AXCode** и шаблон пользовательской программы **AXCodeTemplate** следует в директорию, путь к которой не содержит пробелов.

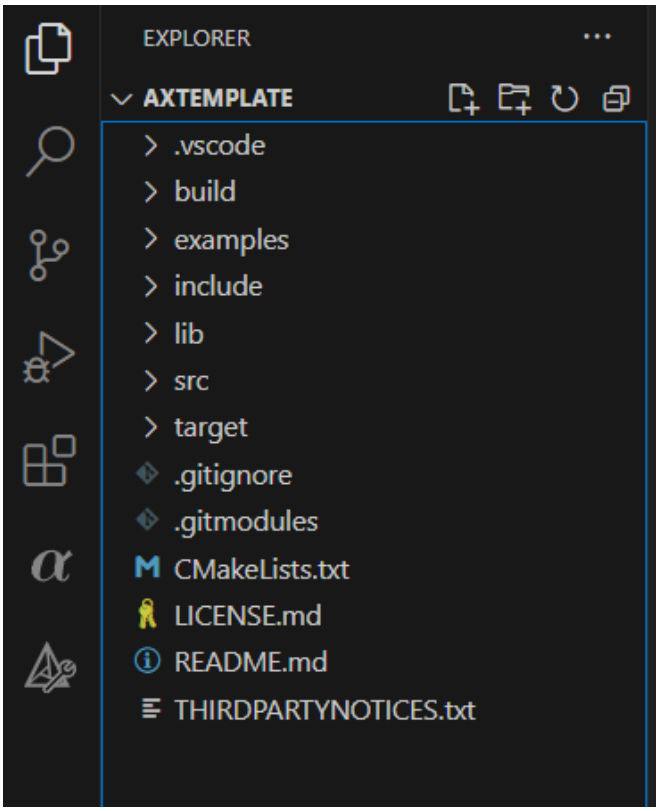
После установки **AXCode**, запустить соответствующий ярлык. Далее, в редакторе открыть папку с распакованным шаблоном. Для этого в редакторе следует выбрать **File** -> **Open Folder** или последовательно нажать сочетание клавиш **Ctrl + K** и **Ctrl + O**.

3. После открытия шаблона по центру сверху появится выпадающее окно, в нем нужно выбрать компилятор, подходящий под архитектуру ПЛК. Архитектура зависит от модификации используемого ПЛК.

Модификация ПЛК	Архитектура	Компилятор
Alpha-X CPU 01-1 00	RISC-V	GCC xx.x.x risc-none-elf
Alpha-X CPU E1-0 00	ARM	GCC xx.x.x arm-none-eabi



Если окно было случайно закрыто или выбран неверный компилятор, чтобы повторно осуществить выбор необходимо нажать **Ctrl + Shift + P** и ввести команду: **CMake: Delete Cache and Reconfigure**.



4. После открытия шаблона в левой верхней части редактора (окно EXPLORER) появится список папок проекта.

.vscode - содержит файл **tasks.json** - настройка задачи загрузчика проекта и файл сниппетов;

build - хранит все файлы, появляющиеся в процессе компиляции (данной директории нет в шаблоне по умолчанию, т.к. создается только после компиляции);

examples - содержит файлы примеров для работы с различными аппаратными сущностями: входы и выходы, интерфейсы, функции времени и т.п.;

include - содержит заголовочные файлы пользовательских подпрограмм;

lib - репозиторий для хранения пользовательских библиотек;

src - содержит файлы пользовательских подпрограмм;

target - содержит набор файлов целевой платформы;

После выбора папки проекта рекомендуется очистить кэш сборщика CMake. Для этого необходимо нажать **Ctrl + Shift + P** и ввести команду: **CMake:**

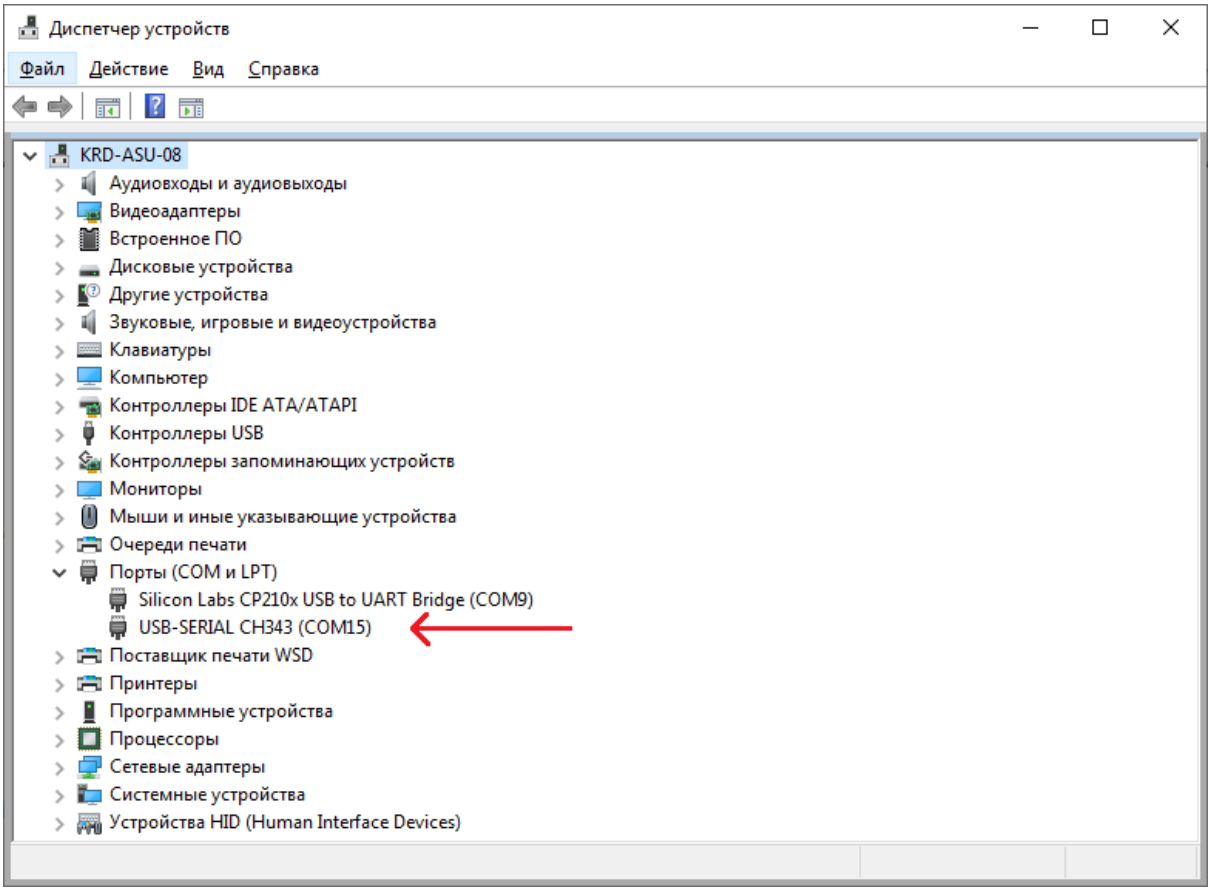
Delete Cache and Reconfigure. Данное действие также требуется выполнять при каждом изменении пути папки с проектом.

Компиляция и загрузка программы

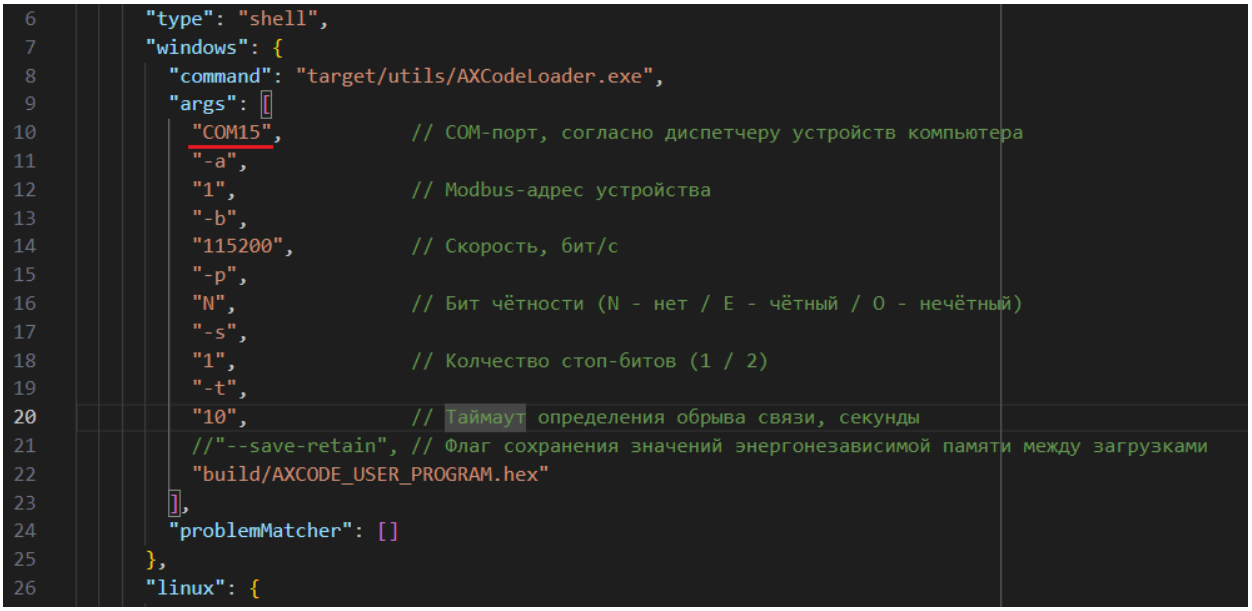
Подключение по USB или RS-485

1. Для загрузки программы в ПЛК посредством интерфейсов USB или RS-485, требуется подключить ПЛК к компьютеру с помощью любого из указанных интерфейсов. Для удобства можно воспользоваться USB, в таком случае для загрузки программы не потребуется дополнительно подавать внешнее питание.

После подключения ПЛК к компьютеру необходимо определить присвоенный ему номер COM-порта в “Диспетчере устройств”. Название устройства в “Диспетчере устройств” может отличаться в зависимости от версии используемой Windows, для Windows 10 устройство будет определяться как «**USB-SERIAL CH343**»

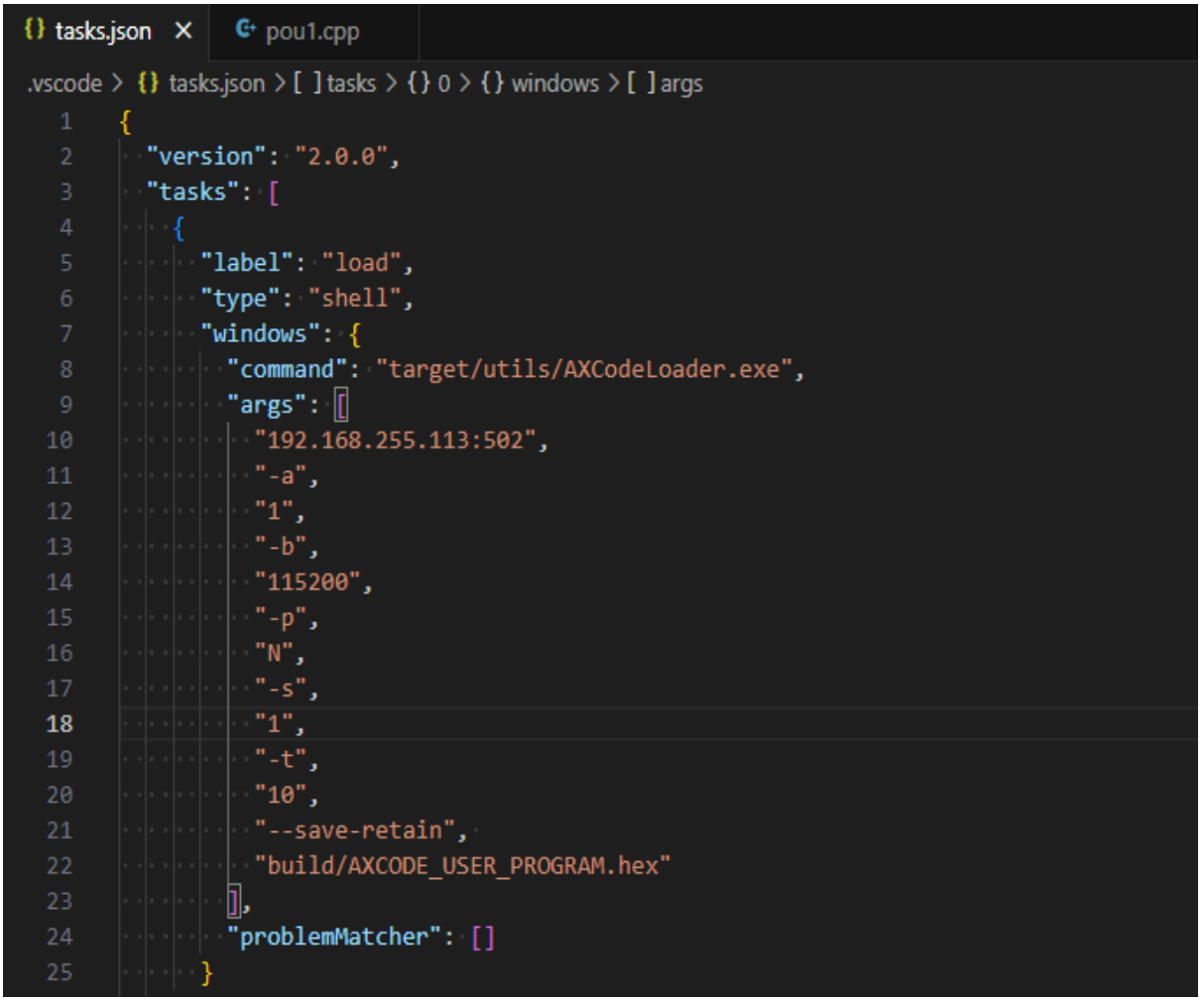


Номер соответствующего COM-порта необходимо вписать в файл **.vscode/tasks.json** в строке 10.



Подключение по протоколу TCP

ПЛК с интерфейсом Ethernet позволяют загружать пользовательскую программы напрямую через сеть. Для этого нужно подключить ПЛК к сети и в файле **.vscode/tasks.json** вместо COM-порта в строке 10 вписать ip адрес и через двоеточие tcp-порт ПЛК.



Для загрузки по Ethernet у ПЛК должен быть статический ip-адрес, который указывается при настройке объекта `EthernetPort` в пользовательской программе. По умолчанию у ПЛК следующий ip-адрес: 192.168.1.100.

```
EthernetPort eth({
  // IP-адрес
  .ipAddr{192, 168, 1, 100},
  // Адрес шлюза
  .gwAddr{192, 168, 1, 0},
  // Маска подсети
  .maskAddr{255, 255, 255, 0},
  // Активация протокола ICMP.PING
  .pingEnable = true,
  // Активация DHCP-клиента
  .dhcpEnable = false
});
```

По умолчанию у ПЛК выбран 502 tcp-порт. Чтобы его изменить необходимо создать экземпляр объекта `ModbusServerTCP`, в последнем параметре которого выбирается tcp порт ПЛК.

```
ModbusServerTCP serverTCP(
    // Таблица регистров группы Holding Registers
    &holding,
    // Таблица регистров группы Input Registers
    &input,
    // Таблица регистров группы Coils
    &coil,
    // Таблица регистров Discrete Inputs
    &discrete,
    // Резервируемое количество сокетов
    1,
    // Порт Modbus TCP
    502
  );
```

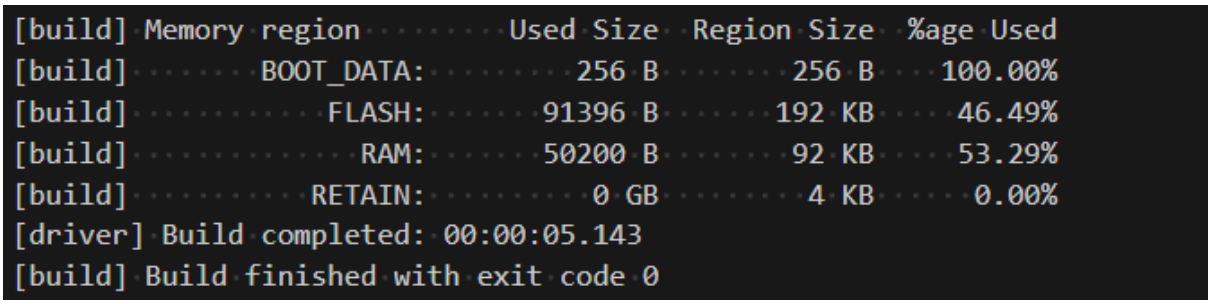
	Загрузка по Ethernet возможна только при наличии свободных сокетов у `ModbusServerTCP` или `HttpServer`. Если у ПЛК отсутствуют свободные сокеты для соответствующих сущностей, то загрузка по сети не будет доступна.

Компиляция и загрузка

После подключения ПЛК к компьютеру по одному из интерфейсов, необходимо произвести компиляцию проекта. Для этого нажать **Ctrl + Shift + P** и ввести команду **CMake: Build** (либо нажать горячую клавишу **F7**).

Если на данном этапе компилятор еще не был выбран, то появится соответствующее выпадающее окно. Для выбора компилятора смотреть п.3 раздела Установка комплекта ПО AXCode.

По окончанию компиляции и сборки в консоль OUTPUT в нижней части программы будет выведена таблица с указанием используемой и свободной памяти ПЛК и прочая служебная информация.



В случае успешной сборки в консоль будет выведена строка:
`Build finished with exit code 0.`

BOOT_DATA – выделенная память загрузочной области, где располагается bootloader. Всегда заполнена на 100%.

FLASH – энергонезависимая память, хранящая код программы ПЛК и не изменяющаяся в процессе работы ПЛК. Включает как код программ пользователя, так и код ядра.

RAM – оперативная память данных. Включает как даныне программ пользователя, так и данные ядра.

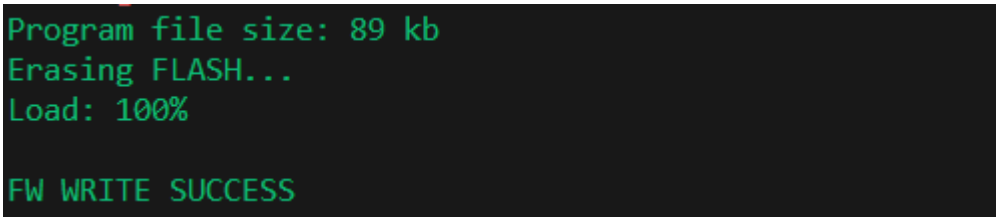
RETAIN – область энергонезависимой памяти, которая может измениться в процессе работы ПЛК. В данную память попадают переменные, объявленные с использованием ключевого слова retain.

В случае неудачной сборки в консоль будет выведена строка:
`Build finished with exit code 1.`

Это сообщение появляется при наличии ошибок в коде проекта. Если в шаблон пользовательской программы не вносились никакие изменения, при этом данное сообщение всё же выводится, то следует очистить кэш сборщика CMake. Для этого необходимо нажать **Ctrl + Shift + P** и ввести команду: **CMake: Delete Cache and Reconfigure**, после чего повторить процесс сборки проекта.

3. После сборки проекта скомпилированный hex-файл можно загрузить на ПЛК. Для этого требуется: нажать **Ctrl + Shift + P** и ввести команду **Tasks: Run Task -> load -> Continue without scanning the task output** (либо нажать горячую клавишу **F8**).

По окончанию успешной загрузки в консоль будет выведена строка:
FW WRITE SUCCESS



Список настроек загрузчика

Файл **tasks.json** содержит настройки задачи загрузки. Для загрузки новой программы пользователя на ПЛК может понадобиться изменить те или иные параметры, перечисленные для свойства “args”.

Настройки modbus-адреса устройства, скорости обмена, паритета и стоп-битов указываются на следующей строчке после специальных идентификаторов -a, -b, -p и -s соответственно. Данные настройки имеют смысл только при загрузке по RS-485 и не влияют на процесс загрузки по USB. Также, загрузка в контроллер по RS-485 возможна только в случае, если соответствующий COM-порт ПЛК работает как Slave-устройство, в противном случае контроллер должен быть переведен в Режим BOOT (ожидание программы пользователя).

Описание параметров задачи загрузки, а также возможных значений аргументов приведено в таблице ниже:

Идентификатор	Описание
COM…	Номер COM-порта, к которому подключен ПЛК.
-a	Modbus-адрес контроллера в режиме Slave. Возможные значения: 1…247.
-b	Скорость обмена последовательного интерфейса. Возможные значения 2400, 4800, 9600, 14400, 19200, 28800, 38400, 57600, 76800, 115200, 128000, 256000 б/с.
-p	Паритет. Возможные значения: N - нет, E - чётный, O - нечётный.
-s	Количество стоп-битов. Возможные значения: 1, 2.
-t	Таймаут определения обрыва связи. Возможные значения: 1…30 секунд.

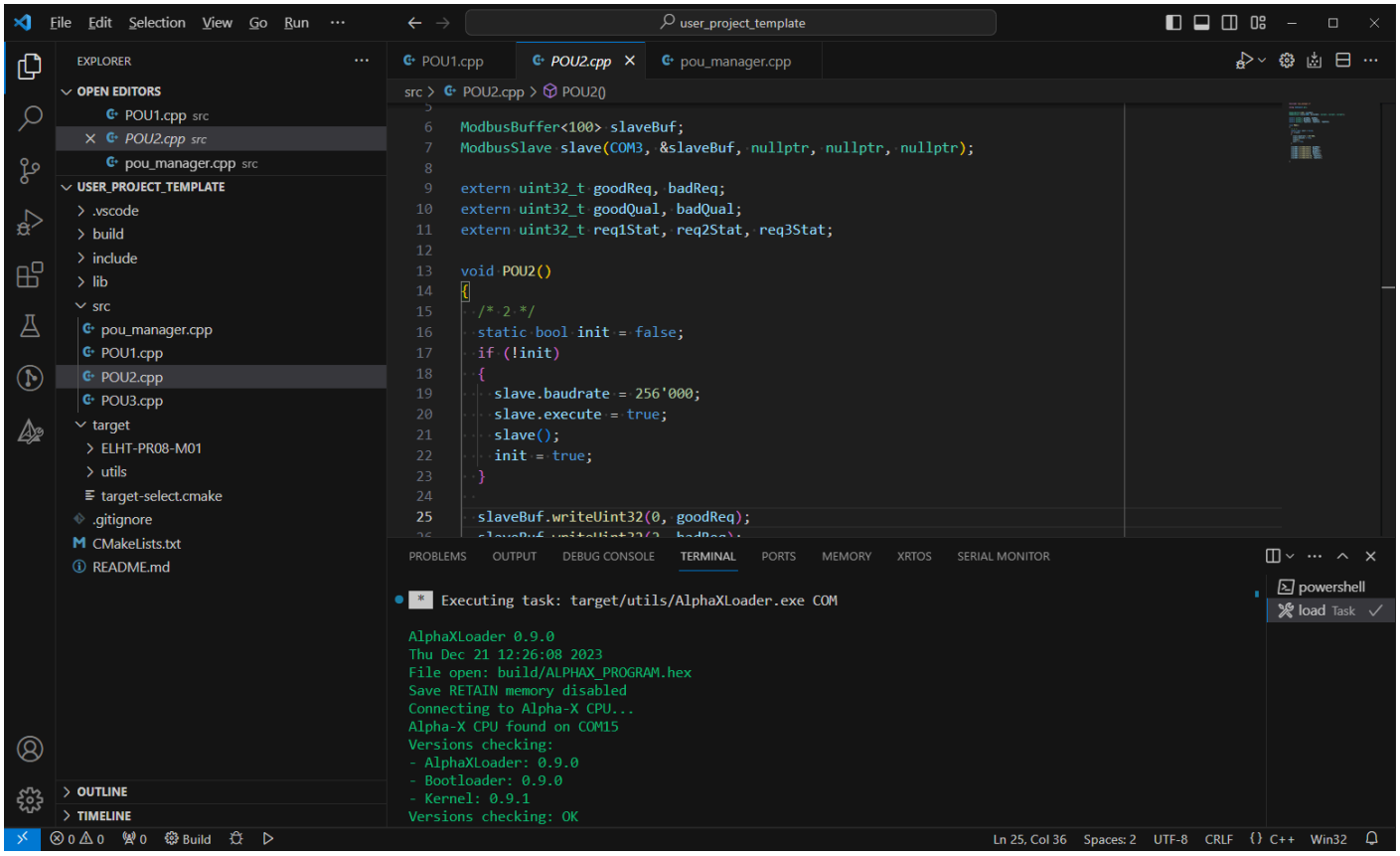
-r	Количество повторных попыток связи при обрыве. Возможные значения: 1…10.
--save-retain	Флаг сохранения переменных типа retain при обновлении программы пользователя. Если данный флаг не установлен, то область памяти retain будет очищена при загрузке новой программы. Более подробно описано в разделе Особенности обновления программы при использовании retain.

Описание инструментов разработки

Code-OSS

В качестве основного редактора кода для программирования ПЛК предлагается использовать «open source» версию редактора кода VS Code, который входит в комплект AXCode.

Стандартное окно редактора показано ниже.

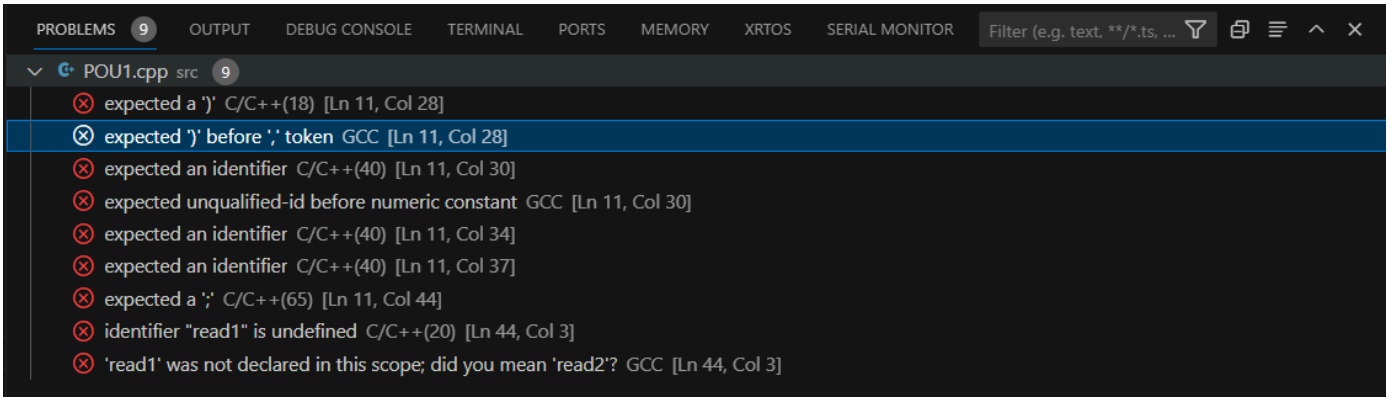


Изменение языка интерфейса среды программирования

Находясь в программе, одновременно нажать **Ctrl + Shift + P** и ввести команду **Configure Display Language**. Далее можно выбрать русский язык.

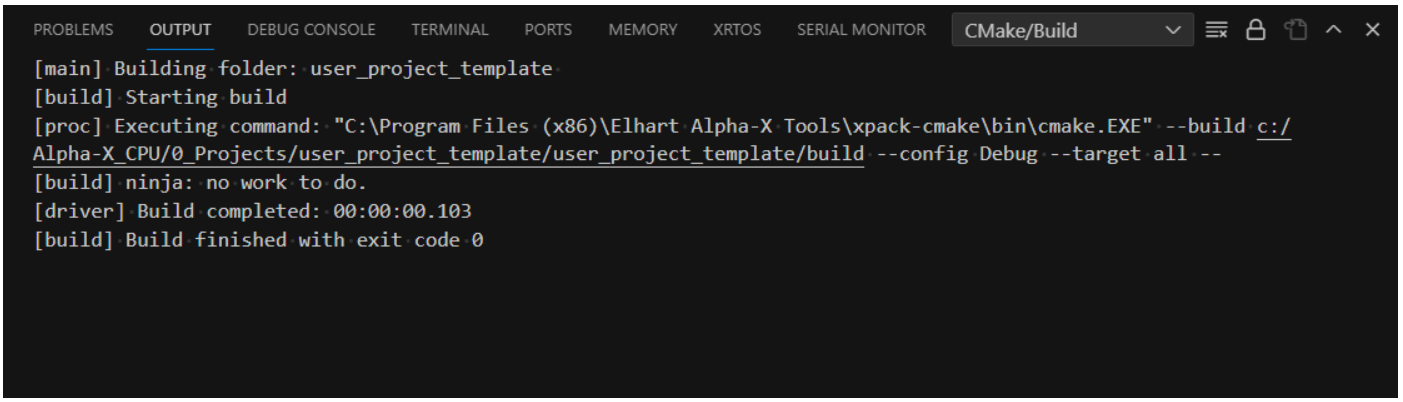
Окно «Problems» (Проблемы)

Окно вывода информации о наличии синтаксических ошибок в коде. Двойной клик по ошибке позволяет открыть место её возникновения в окне редактора.



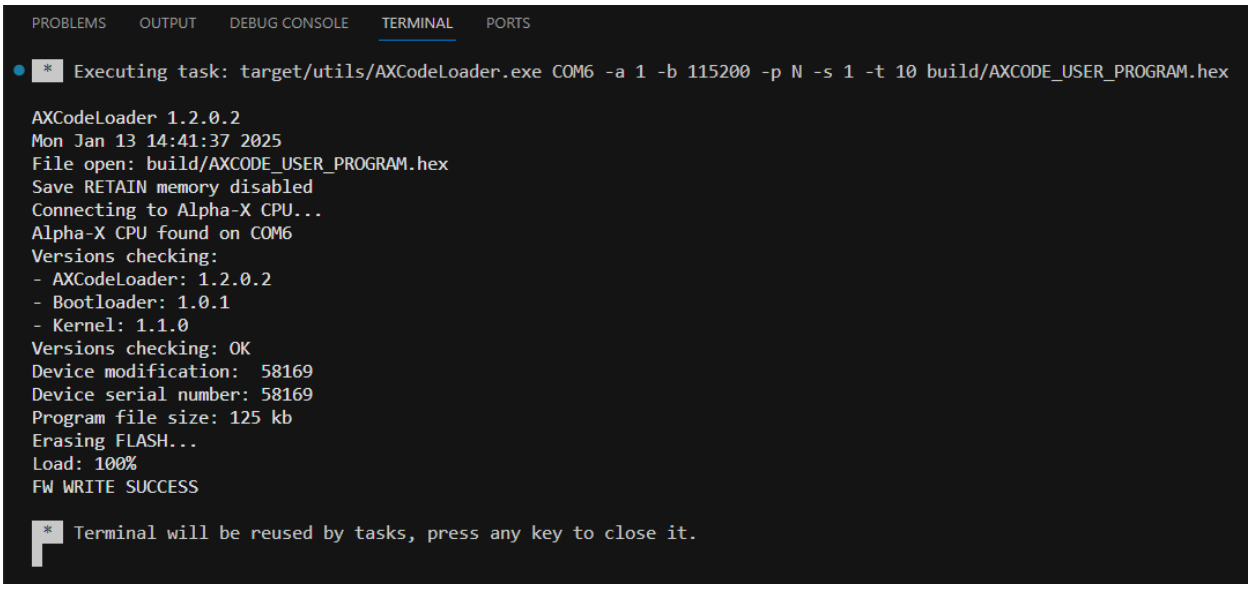
Окно «Output» (Вывод)

Окно вывода информации от расширений. По умолчанию выводит данные расширения-сборщика CMake о результатах процесса сборки и компиляции.



Окно «Terminal» (Терминал)

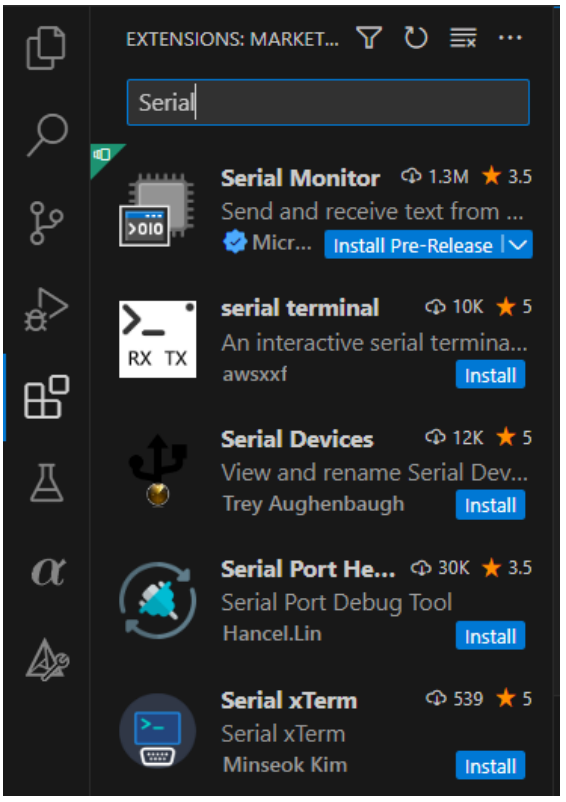
Окно вывода информации о задачах Code-OSS. В данном окне выводится информация о загрузке программы в модуль.



Окно «Extensions» (Расширения)

Вкладка позволяет открыть магазин расширений.

Рекомендуется установить одно из расширений: Serial Monitor. Данное расширение представляет собой терминал, который можно использовать при отладке кода программы. Примеры, связанные с использованием терминала в этом руководстве, будут выполнены с использованием расширения Serial Monitor.



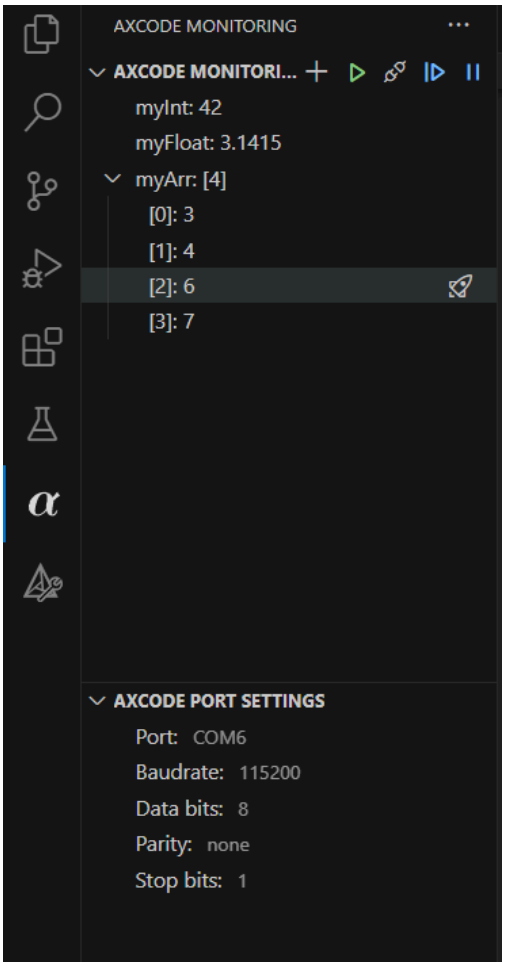
Горячие клавиши

Одним безусловных преимуществ редактора Code-OSS является наличие большого количества горячих клавиш. Полный список представлен тут: https://code.visualstudio.com/shortcuts/keyboard-shortcuts-windows.pdf.

Расширение «AXCODE MONITORING»

Вкладка позволяет следить за значениями переменных в реальном времени, а также форсировать (принудительно задать) их значения. Для этого требуется подключиться к ПЛК по USB или одному из COM-портов (должны быть настроены на режим Debug) и добавить в лист наблюдения названия требуемых переменных.

Следует отметить, что мониторинг возможен только для неоптимизированных переменных со статической длительностью хранения.



РАЗДЕЛ 3: ФУНКЦИИ РАБОТЫ С АППАРАТНЫМИ СРЕДСТВАМИ

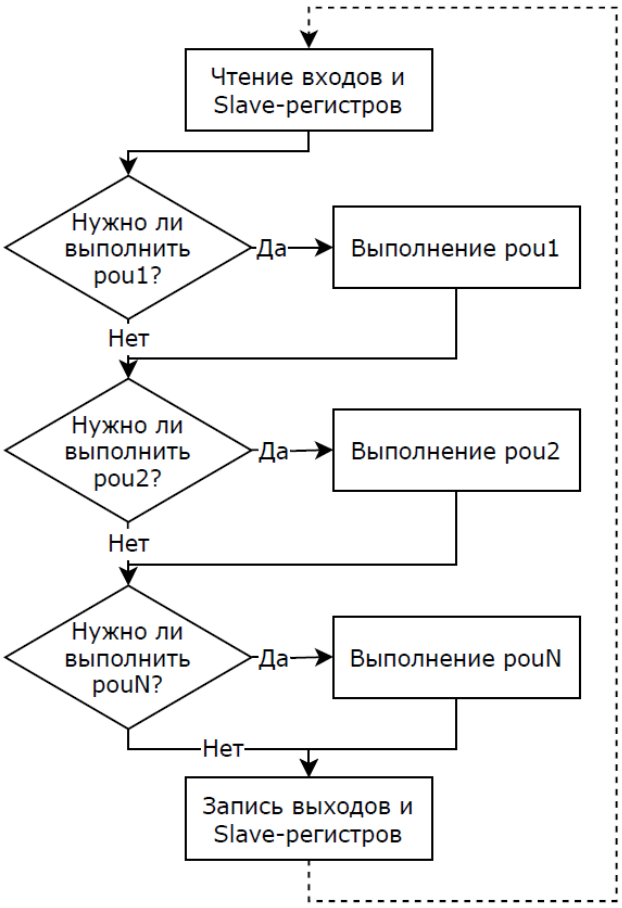
Планировщик задач

Подпрограммы и кооперативная многозадачность

Шаблон программы пользователя содержит папку src, в которой содержатся пользовательские подпрограммы (pou1.cpp, pou2.cpp и т.п.) и файл с их настройками (pou_manager.cpp).

По умолчанию шаблон имеет 3 созданных подпрограммы пользователя, при этом pou2 и pou3 закомментированы и не выполняются. Тем не менее, код, написанный в этих подпрограммах, будет компилироваться, и в случае наличия в них ошибок, компилятор выведет соответствующие сообщения. Полное исключение подпрограммы из процесса компиляции выполняется в соответствии с разделом Добавление и удаление подпрограмм.

Общий принцип работы планировщика указан на диаграмме ниже.



Условие выполнения очередной pou определяется периодом её вызова, который задаётся в pou_manager.cpp. Все подпрограммы работают в режиме кооперативной многозадачности, это означает что выполнение следующей подпрограммы начнётся только после полного завершения выполнения текущей. При этом системные задачи выполняются в режиме вытесняющей многозадачности.

Параметры pou и pou_manager

Для каждой подпрограммы можно задать период вызова и время срабатывания сторожевого таймера. Эти настройки указываются в файле pou_manager.cpp.

```
void PLC_MainSetup()
{
  // Функция POU1, период вызова 100 мс, сторожевой таймер 1 сек
```

```
    AddPOU(POU1, 100, 1000);

    // Раскомментируйте строки ниже для включения pou2.cpp и pou3.cpp
    // AddPOU(POU2, 1000, 1000);
    // AddPOU(POU3, 2000, 1000);
  }
```

Первый аргумент функции `AddPOU` - ***имя добавляемой подпрограммы***.

Второй аргумент функции `AddPOU` - ***период вызова подпрограммы*** - минимальное время между двумя вызовами одной подпрограммы. Задается в миллисекундах. Фактический период вызова может измениться в большую сторону, если в pou содержится множество трудоемких вычислений и ПЛК не успевает выполнить все подпрограммы в назначенное время.

Данная настройка позволяет удобно распределять ресурсы ПЛК. Увеличивая период менее критичных задач можно выделить больше процессорного времени для более частого выполнения самых важных подпрограмм.

Третий аргумент функции `AddPOU` - ***время срабатывания сторожевого таймера (Watchdog)*** - максимально допустимое время выполнения конкретной подпрограммы в миллисекундах. Если подпрограмма исполняется дольше этого времени, ПЛК выполнит одно из следующих действий: перезагрузится, перейдет в режим SERVICE или STOP. Время срабатывания сторожевого таймера позволяет настроить реакцию ПЛК при возникновении ошибок в пользовательском коде, к примеру, если в подпрограмме возник бесконечный цикл.

Значение времени сторожевого таймера можно не указывать, либо задать 0. Тогда сторожевой таймер будет отключен.

Реакция на срабатывание сторожевого таймера настраивается в том же файле. Помимо этого, там же настраивается реакция системы на Hardfault, а также период сохранения retain-переменных:

```
  // НАСТРОЙКИ ПЛАНИРОВЩИКА
  // Реакция системы на срабатывание Watchdog планировщика
  const WATCHDOG_MODE SchedulerConfig::watchdog_mode
{WATCHDOG_MODE::SERVICE};
```

```
    // Реакция системы на HardFault
    const HARDFAULT_MODE SchedulerConfig::hardfault_mode
{HARDFAULT_MODE::SERVICE};
    // Период сохранения retain-переменных
    uint8_t RetainConfig::save_time {10};
```

Режимы обработки сторожевого таймера и обработчика Hardfault приведены в соответствующих перечислениях.

```
    /**
     * \brief Режимы сторожевого таймера
     */
    enum class WATCHDOG_MODE
    {
      SERVICE, //!< Перезагружает ПЛК в сервисный режим
      RELOAD,  //!< Перезагружает ПЛК
      STOP,    //!< Останавливает ПЛК (режим STOP)
      SIZE
    };

    /**
     * \brief Режимы обработчика HardFault
     */
    enum class HARDFAULT_MODE
    {
      SERVICE, //!< Режим остановки ПЛК при возникновении HardFault
      RELOAD,  //!< Режим перезагрузки ПЛК при возникновении HardFault
      SIZE
    };
```

Фактическое время выполнения подпрограмм

Заголовочный файл pou_scheduler.h содержит функции, которые позволяют оценить фактическое время выполнение подпрограмм.

Ниже представлена таблица с доступными функциями. В качестве аргумента в них передается наименование функции pou. Все функции возвращают значение в формате `uint32_t`.

Наименование функции	Возвращаемое значение
`GetCycleTimeAVG()`	Среднее время выполнения pou в микросекундах
`GetCycleTimeMIN()`	Минимальное время выполнения pou в микросекундах
`GetCycleTimeMAX()`	Максимальное время выполнения pou в микросекундах
`GetCycleTimeLAST()`	Время последнего выполнения pou в микросекундах
`GetCyclesCount()`	Количество выполнений pou после запуска ПЛК

Ниже приведён пример использования функции для получения среднего времени выполнения POU1:

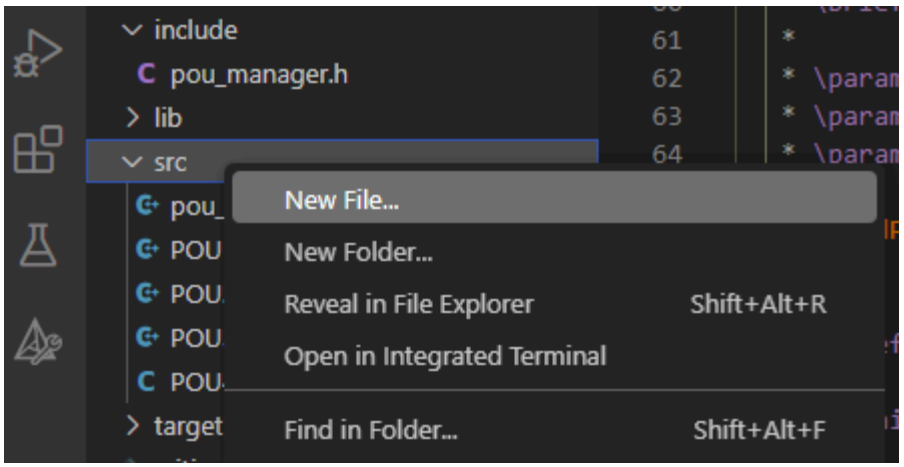
```
void POU1()
{
  uint32_t avg = GetCycleTimeAVG(POU1);
  print_debug("Среднее время выполнения %d\r\n", avg);
}
```

Добавление и удаление подпрограмм

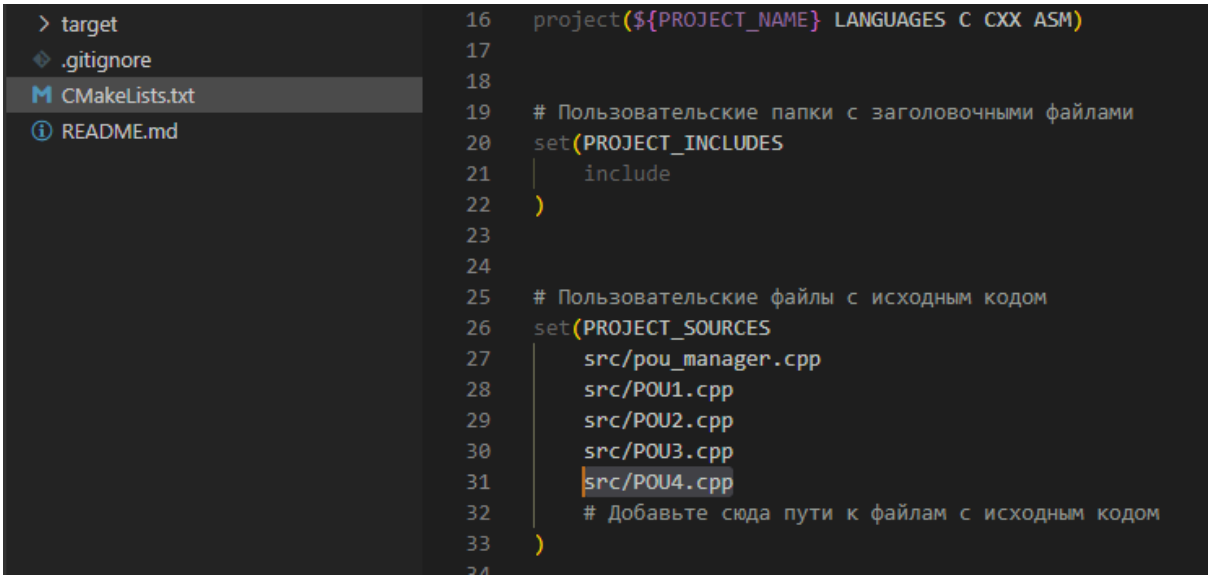
Среда исполнения ПЛК позволяет добавить до 16 подпрограмм пользователя.

Для добавления в проект новой подпрограммы необходимо:

1. В папке src создать новый файл с произвольным именем (например, POU4) и расширением .cpp. Для создания нового файла можно воспользоваться контекстным меню (правая кнопка мыши по папке src).



2. Добавить имя нового pou в шаблон программы в CMakeLists.txt (на примере добавления POU4.cpp):



3. Объявить в заголовочном файле pou_manager.h новую функцию с возвращаемым значением `void`, в примере ниже - `POU4()`.

```
// Объявление функций POU из других файлов
void POU1(); // Функция POU из файла src/pou1.cpp
void POU2(); // Функция POU из файла src/pou2.cpp
void POU3(); // Функция POU из файла src/pou3.cpp
void POU4();
```

4. Добавить в файле pou_manager.cpp вызов объявленной ранее функции POU4. Для этого используется функция AddPOU.

```
void PLC_MainSetup()
{
  AddPOU(POU1, 20, 1000);     // Период 20 мс, вотчдог 1 сек
  AddPOU(POU2, 1000, 1000);   // Период 1 сек, вотчдог 1 сек
  AddPOU(POU3, 2000, 1000);   // Период 2 сек, вотчдог 1 сек
  AddPOU(POU4, 100);
}
```

5. В созданном ранее файле подключить заголовочный файл pou_manager.h и указать использование пространства имен plc.

```
#include "pou_manager.h"
using namespace plc;

void POU4()
{
  // Код, написанный здесь, выполняется циклически
}
```

Для полного удаления подпрограммы из проекта следует проделать действия в обратном порядке, удалив созданный файл pou (на примере добавления - POU4.cpp).

Для временного исключения pou из списка планировщика (например, при разработке) можно в файле pou_manager.cpp закомментировать строку с функцией AddPOU, в которой этот pou упоминается. Таким образом подпрограмма будет скомпилирована, но вызываться на ПЛК не станет.

	Добавленные в CMakeLists.txt файлы всегда влияют на компиляцию программы. Поэтому при наличии в них ошибок проект не будет собран, даже если в pou_manager.h и pou_manager.cpp они не упоминаются.

Видимость переменных в подпрограммах

Без модификаторов

В зависимости от места объявления переменной меняются способы использования и непосредственное поведение данной переменной. В коде ниже

представлены два основных места для объявления пользовательских переменных.

```
#include "pou_manager.h"
using namespace plc;

/* 1 */

void POU1()
{
  /* 2 */
}
```

Переменные без модификаторов, объявленные в области /* 1 */:

1. **сохраняют** свое значение между вызовами pou;
2. значение этих переменных **нельзя** получить в других pou;
3. в других pou **нельзя** использовать переменные с таким же именем.

Переменные без модификаторов, объявленные в области /* 2 */:

1. **не сохраняют** свое значение между вызовами pou, т.е. переменные из этой области при каждом выходе из POU() будут очищаться, а при каждом входе создаваться снова со значениями по умолчанию;
2. значение этих переменных **нельзя** получить в других pou;
3. в других pou **можно** использовать переменные с таким же именем.

Как правило, для большинства задач подходит вариант объявления переменных в области /* 1 */. Однако если необходимо использовать переменные с теми же названиями в других pou, либо передавать значения переменных между pou, то при объявлении переменных необходимо воспользоваться дополнительными модификаторами.

Модификатор static

Модификатор `static` делает переменную статической, то есть значение такой переменной сохраняется между вызовами pou. Область видимости таких переменных ограничена тем pou, в котором переменная была объявлена.

```
#include "pou_manager.h"
using namespace plc;
```

```
/* 1 */
static int16_t myInt1;

void POU1()
{
  /* 2 */
  static int16_t myInt2;
  ...
}
```

Переменные с модификатором `static`, объявленные как в области /* 1 */ так и в области /* 2 */ имеют идентичное поведение:

1. **сохраняют** свое значение между вызовами pou;
2. значение этих переменных **нельзя** получить в других pou;
3. в других pou **можно** использовать переменные с таким же именем.

Модификатор extern

Модификатор `extern` требуется указывать, если переменную, объявленную в одной подпрограмме, требуется использовать в других подпрограммах.

```
// Подпрограмма 1
uint32_t externVar = 42;

void POU1()
{
  ...
}
```

```
// Подпрограмма 2
extern uint32_t externVar;

void POU2()
{
  ...
}
```

Для использования переменной в другой подпрограмме необходимо объявить ее повторно в нужной подпрограмме с использованием ключевого

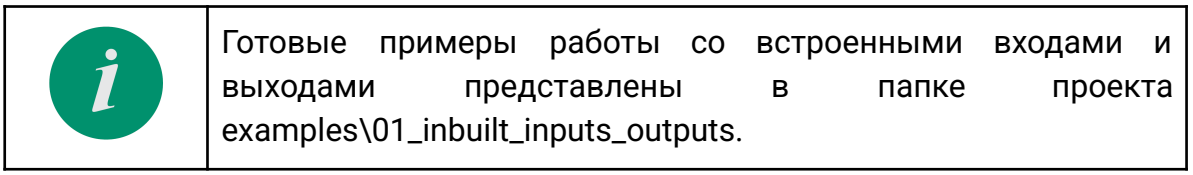
слова `extern`. При этом, использование ключевого слова `extern` подразумевает, что переменная была объявлена и инициализирована в другом месте, поэтому нельзя повторно использовать инициализацию.

Модификатор `extern` можно использовать с объектами классов и retain-переменными.

Встроенные входы и выходы

Дискретные входы, классы DiscreteInput и DiscreteInputArray

Для работы со встроенными дискретными входами используются классы `DiscreteInput` и `DiscreteInputArray`. Как правило, использование класса `DiscreteInputArray` более предпочтительно, так как позволяет работать с набором входов как с массивом.



Готовые примеры работы со встроенными входами и выходами представлены в папке проекта examples\01_inbuilt_inputs_outputs.

Создание отдельного дискретного входа:

```
/**
 * cpu_io - описание доступных входов-выходов для используемой модели CPU
 * CPU_DINPUT1 - номер дискретного входа используемой модели CPU
 */
DiscreteInput di1(cpu_io, CPU_DINPUT1);

void POU1()
{
  ...
}
```

Создание массива дискретных входов:

```
/**
 * cpu_io - описание доступных входов-выходов для используемой модели CPU
 * CPU_DINPUT_SIZE - общее количество дискретных входов используемой модели CPU
 */
DiscreteInputArray<CPU_DINPUT_SIZE> inputs(cpu_io);

void POU1()
{
  ...
}
```

	Не следует создавать экземпляры классов дискретных входов внутри функции POU()!

	Одновременное создание нескольких экземпляров классов `DiscreteInput` или `DiscreteInputArray` для одних и тех же входов недопустимо!

В примерах выше используются значения перечисления `CPU_DINPUT`. В этом перечислении указаны доступные для данной модели ПЛК дискретные входы, при этом значение `CPU_DINPUT_SIZE` используется для определения общего количества доступных входов. Чтобы избежать ошибок, связанных с неверным указанием номеров входов, рекомендуется использование этого перечисления.

	AXCode позволяет быстро перейти к объявлению того или иного объекта, чтобы не искать его вручную. Для этого, на примере перечисления `CPU_DINPUT`, в окне редактора нажмите по нему правой кнопкой мыши и выберите верхний пункт **Go to Definition** или просто щёлкните по объекту левой кнопкой мыши, удерживая клавишу **Ctrl.**

```
//! Нумерация дискретных входов в устройстве \ref cpu_io
enum CPU_DINPUT
{
  CPU_DINPUT1,     //!< Дискретный вход №1
  CPU_DINPUT2,     //!< Дискретный вход №2
  CPU_DINPUT3,     //!< Дискретный вход №3
  CPU_DINPUT4,     //!< Дискретный вход №4
  CPU_DINPUT_SIZE //!< Кол-во дискретных входов
};
```

Вне зависимости от выбранного класса, использование дискретных входов в большинстве случаев ничем не отличается от использования обычных логических переменных типа `bool`.

Использование отдельного дискретного входа:

```
DiscreteInput di1(cpu_io, CPU_DINPUT1);
```

```
void POU1()
{
  // Использование значения входа di1 в операторе if
  if (di1)
  {
    ...
  }

  // Присвоение значения входа di1 другой переменной
  bool myBool = di1;
}
```

Использование дискретного входа в составе массива:

```
DiscreteInputArray<CPU_DINPUT_SIZE> inputs(cpu_io);

void POU1()
{
  // Использование значения входа в операторе if
  if (inputs[0])
  {
    ...
  }

  // Присвоение значения входа другой переменной
  bool myBool = inputs[CPU_DINPUT1];
}
```

Дополнительно, значение дискретного входа можно явно получить с помощью метода `value()` класса `DiscreteInput`.

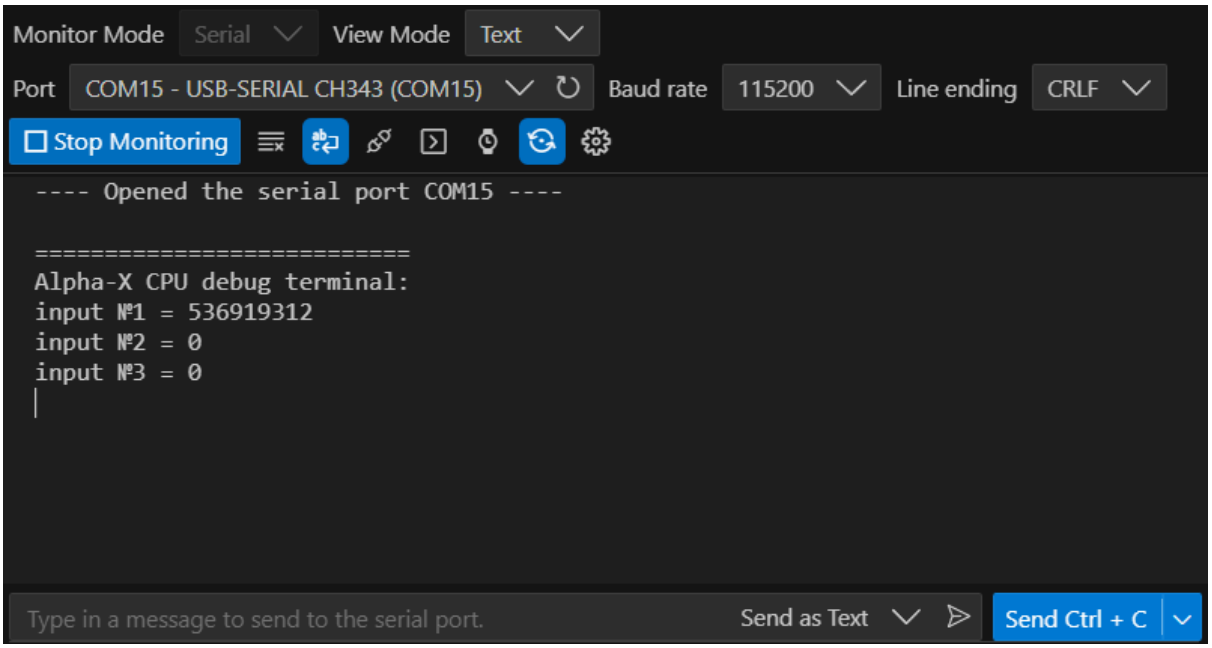
```
DiscreteInputArray<CPU_DINPUT_SIZE> inputs(cpu_io);

void POU1()
{
  // Использование значения входа в операторе if
  if (inputs[0].value())
  {
    ...
  }
```

```
    // Присвоение значения входа другой переменной
    bool myBool = inputs[CPU_DINPUT1].value();
}
```

Способ работы с дискретными входами выбирает пользователь в зависимости от своих предпочтений. Однако стоит отметить, что при передаче значений в функции необходимо использовать метод `value()`, либо явно преобразовывать значение класса в тип `bool`. Ниже приведен пример передачи значения дискретного входа в функцию `print_debug()` тремя способами. Как уже было сказано выше, **первый способ является неверным**.

```
DiscreteInputArray<CPU_DINPUT_SIZE> inputs(cpu_io);

void POU1()
{
  print_debug("input №1 = %d\r\n", inputs[CPU_DINPUT1]); // Неверно!
  print_debug("input №2 = %d\r\n", (bool)inputs[CPU_DINPUT2]);
  print_debug("input №3 = %d\r\n", inputs[CPU_DINPUT2].value());
}
```

Результат выполнения кода:



Чтобы исключить влияние дребезга контактов, обновление состояния встроенных дискретных входов для ПЛК происходит не чаще чем раз в 20 мс. Данный параметр настроить нельзя.

Соответственно, подпрограмма, в которой происходит обработка сигналов дискретных входов, должна вызываться с периодом не меньше, чем время обновления входов. В противном случае существует риск пропустить срабатывание входа (хотя, как правило, нажатие кнопки длится дольше чем 20 мс).

	Контроллер обновляет состояние своих входов единожды перед началом цикла выполнения планировщика pou. При этом значения входов не изменяются на протяжении выполнения всех pou в этом цикле.

	Подпрограмму пользователя (pou), в которой происходит обработка входов рекомендуется вызывать тем чаще, чем короче длительность сигнала, поступающего на дискретный вход. В противном случае существует вероятность “упустить” сигнал соответствующий. Рекомендуемое время вызова подобных pou: 10…50 мс.

Детектирование переднего и заднего фронтов дискретных сигналов

Для отслеживания переднего и заднего фронтов импульса используются методы `front()` и `back()` соответственно. Пример вызова встроенных функций:

```
   DiscreteInputArray<4>    inputs  (cpu_io,  0);  // Массив встроенных
дискретных входов на 4 элемента (нумерация с нуля)

   void POU1()
   {
      bool front = inputs[0].front(); // Передний фронт на входе DI1
      bool back = inputs[1].back();   // Задний фронт на входе DI2
   }
```

Метод `front()` возвращает ИСТИНА, после того как значение дискретного входа изменилось с низкого уровня на высокий. При этом метод будет возвращать ИСТИНА для всех подпрограмм в течение одного цикла

планировщика. Сброс метода произойдет в следующем цикле только в том случае, если он был вызван хотя бы один раз в текущем цикле.

Метод `back()` возвращает ИСТИНА, после того как значение дискретного входа изменилось с высокого уровня на низкий. При этом метод будет возвращать ИСТИНА для всех подпрограмм в течение одного цикла планировщика. Сброс метода произойдет в следующем цикле только в том случае, если он был вызван хотя бы один раз в текущем цикле.

Дополнительный функционал для работы с дискретными и аналоговыми входами и выходами

Для более точного контроля за работой физических каналов ПЛК и модулей, в объектах классов `DiscreteOutput`, `DiscreteInput`, `AnalogOutput` и `AnalogInput` существуют одинаковые функции управления. Например, они позволяют включать или выключать опрос выбранных каналов и получать время с их последней синхронизации с физическим устройством.

Функция	Описание
`time_since_update()`	Возвращает время, прошедшее с обновления состояния выхода
`isEnabled()`	Возвращает состояние опроса дискретного выхода (вкл/выкл)
`enable(bool)`	Включает/выключает опрос дискретного выхода
`disable()`	Выключает опрос дискретного выхода
`scan()`	Считывает состояние входа из привязанного устройства
`update()`	Записывает значение выхода в привязанное устройство

Ниже приведен пример, в котором представлено использование всех этих функций.

```
DiscreteInputArray<CPU_DINPUT_SIZE> dinputs(cpu_io);
AnalogOutputArray<AlphaXModule::AOUTPUT_CHANNEL_SIZE> aoutputs (module4);
```

```
void POU1()
{
  dinputs[0].disable(); // Отключит чтение 1-го встроенного дискретного
                        // входаперед каждым вызовом POU()

  aoutputs[1].enable(false); // Сделает тоже самое, но для 4-ого аналогового
                             // выхода модуля расширения. Данный метод
                             // позволяет управлять состоянием опроса выхода
                             // с помощью переменной типа bool, переданной
                             // вместо значения false

  if(dinputs[2].isEnabled()) { // Возвращает текущее состояние опроса канала
    print_debug("Опрос 2-го встроенного дискретного канала включен\n\r");
  }

  // Выводит время, прошедшее с обновления выхода
  print_debug("Время, прошедшее с обновления состояния выхода: %d\n\r",
              aoutputs[3].time_since_update());

  // Прерывает выполнение POU() и считывает состояние выбранного входа
  dinputs[0].scan();

  // Прерывает выполнение POU() и записывает состояние выбранного выхода
  aoutputs[0].update();
}
```

Дискретные выходы, классы DiscreteOutput и DiscreteOutputArray

Для работы со встроенными выходами ПЛК необходимо создать объект класса `DiscreteOutput` и передать в качестве параметров ссылку на собственные выходы ПЛК `cpu_io` и номер требуемого выхода:

```
// Создание дискретного выхода
DiscreteOutput do1(cpu_io, 0);

void POU1()
{
  // Здесь код выполняется в цикле
```

```
    }
```

	Нельзя создавать экземпляры классов дискретных выходов внутри функции POU()!

	Одновременное создание нескольких экземпляров классов `DiscreteOutput` или `DiscreteOutputArray` для одних и тех же выходов недопустимо!

Как и в случае с дискретными входами, допускается использовать перечисление `CPU_DOUTPUT`:

```
    //! Нумерация дискретных выходов в устройстве \ref cpu_io
    enum CPU_DOUTPUT
    {
      CPU_DOUTPUT1,     //!< Дискретный выход №1
      CPU_DOUTPUT2,     //!< Дискретный выход №2
      CPU_DOUTPUT_SIZE //!< Кол-во дискретных выходов
    };
```

Также с дискретными выходами можно работать как с элементами массива:

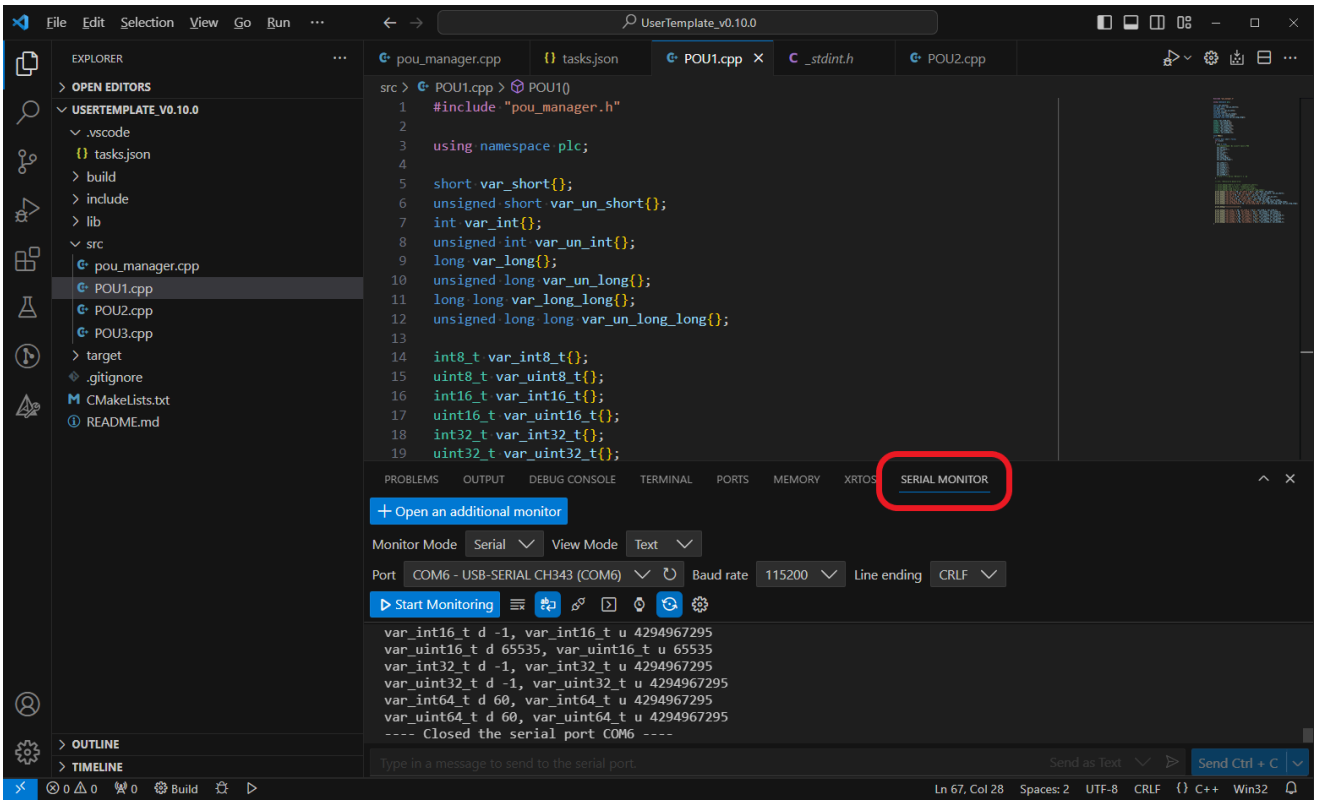
```
    DiscreteInputArray<4>  dInputs (cpu_io); // Массив встроенных
дискретных входов на 4 элемента(нумерация с нуля)
    DiscreteOutputArray<2> dOutputs (cpu_io); // Массив встроенных
дискретных выходов на 2 элемента (нумерация с нуля)

    void POU1()
    {
      // Код, выполняемый циклически
      dOutputs[0] = true; // Включить выход DO1
      dOutputs[1] = dInputs[0]; // DO2 повторяет состояние входа DI1
    }
```

Отладочный терминал Debug

Общая информация

ПЛК поддерживает работу с компьютерными терминалами, что предоставляет возможности для отладки программы. Связанный с этим функционал в проекте имеет общее наименование Debug.

В качестве терминала можно использовать один из бесплатных вариантов: PuTTY, Serial Monitor, MobaXTerm, Serial USB Terminal (для Android) или другой аналогичный. В примерах данного руководства будет показано использование расширения **Serial Monitor**, которое можно скачать во вкладке с расширениями (сочетание **Ctrl + Shift + X**). Окно терминала появится в нижней части редактора (рисунок ниже).



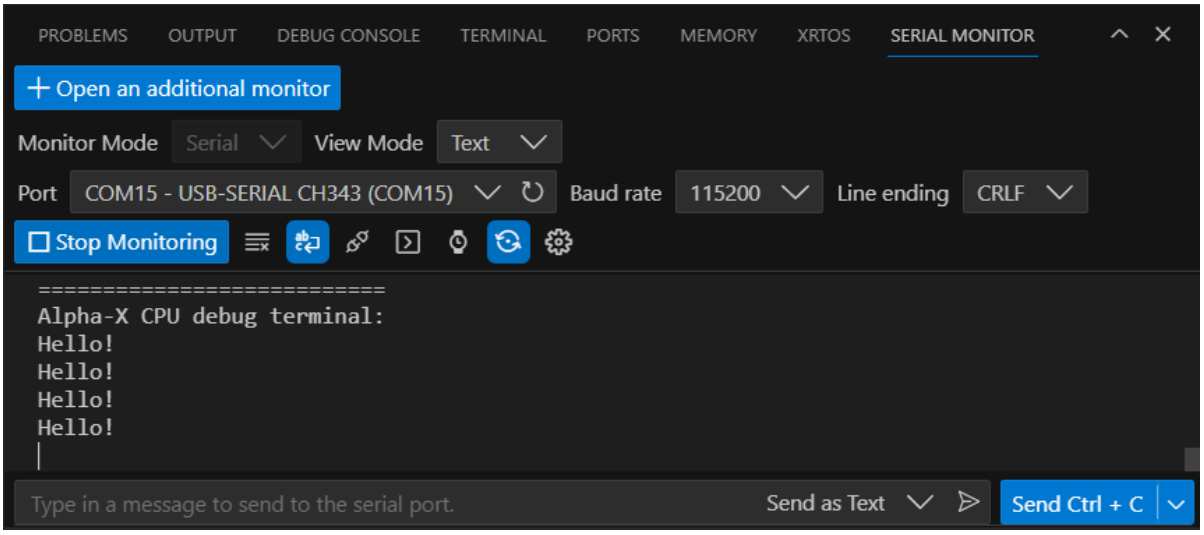
Вывод статической строки в терминал через USB, функция print_debug()

По умолчанию, USB-порт ПЛК настроен на режим работы с терминалом (Debug), поэтому для отправки и приёма сообщений в этом режиме не требуется дополнительных манипуляций.

Отправка сообщений в отладочный порт производится с помощью функций `print_debug()` или `printf()`. Данные функции идентичны. Пример использования:

```
void POU1()
{
  // Код, выполняемый циклически:
  print_debug("Hello!\r\n");

}
```

После подключения к ПЛК с помощью Serial Monitor, в терминал будут выводиться следующие сообщения:



Так как функция вывода в примере выше вызывается каждый цикл программы, то сообщение в терминал будет выводиться периодически, при этом цикл вывода будет зависеть от периода вызова подпрограммы (настройки в файле pou_manager.cpp).

Также, возможен вывод и символов кириллицы.

Символы `\r\n` в конце сообщения являются командой терминалу и необходимы, чтобы каждое следующее сообщение выводилось с новой строки. Некоторые из возможных команд приведены ниже.

Символ	Описание
`\b`	Удаление последнего введенного символа.
`\n`	Перевод строки. Следующий символ будет напечатан с начала новой строки.
`\f`	Перевод строки. Новый символ будет напечатан на новой строке, на позиции, следующей за последним напечатанным символом на предыдущей строке.
`\r`	Возврат на начало строки.
`\t`	Табуляция по горизонтали.
`\v`	Вертикальная табуляция.
`\\`	Вывод обратного слеша.
`\”`	Вывод кавычек.
`%%`	Вывод знака процента.
`\num`	Вывод символа по его коду. Например при выводе print_debug("\123"); будет напечатан символ 'S'.

Пример выше описывает вывод статической строки, содержание которой определено еще на этапе компиляции и не может изменяться от вызова к вызову. Вывод с динамическим формированием строки описан в разделе Вывод динамической строки с подставлением значений переменных.

Функция `print_debug()`, как и остальные функции отладочного терминала, является неблокирующей, то есть не останавливает выполнение подпрограммы на время вывода сообщения. При вызове функций отладочного терминала сообщения добавляются в специально выделенный буфер отладочных сообщений, размером 1 кБ. После отправки сообщений на терминал, они удаляются из буфера.

При высокой частоте добавления сообщений в буфер и низкой скорости отправки в терминал может произойти переполнение буфера, что вызовет появление соответствующей ошибки в логе ошибок ПЛК. Скорость отправки в терминал зависит от используемого интерфейса. При использовании USB

скорость фиксированная и не зависит от настроек. При использовании интерфейсов RS-485 скорость задается пользователем.

При возникновении ошибок или помех, терминал Serial Monitor подставляет символ вместо утерянных символов.

Создание отладочного порта на прочих интерфейсах, класс DebugPort

По умолчанию, другие интерфейсы ПЛК (кроме USB) не настроены на режим работы с терминалом. Для создания отладочного порта в таком случае необходимо создать объект класса `DebugPort`. В качестве параметра передается один из следующих аргументов:

Номер	Перечисление COM_PORT	Фактический интерфейс ПЛК CPU 01-1 00
0	COM1	RS-485, COM1
1	COM2	RS-485, COM2
2	COM3	RS-485, COM3
3	COM4	USB

Пример создания отладочного порта приведен ниже. Для этого кода имеется готовый сниппет.

```
using namespace plc;

DebugPort debug_com1 ({
  .com          = COM1,
  .baudrate     = 115200,
  .parity       = COM_PARITY_NONE,
  .stop_bits    = 1,
  .debug_messages = true,
  .event_messages = true,
  .error_messages = true
```

```
});

void POU1()
{
  // Код, выполняемый циклически:
}
```

Помимо стандартных настроек связи, при создании отладочного порта дополнительно указывается тип сообщений, которые будут выводиться в этот порт.

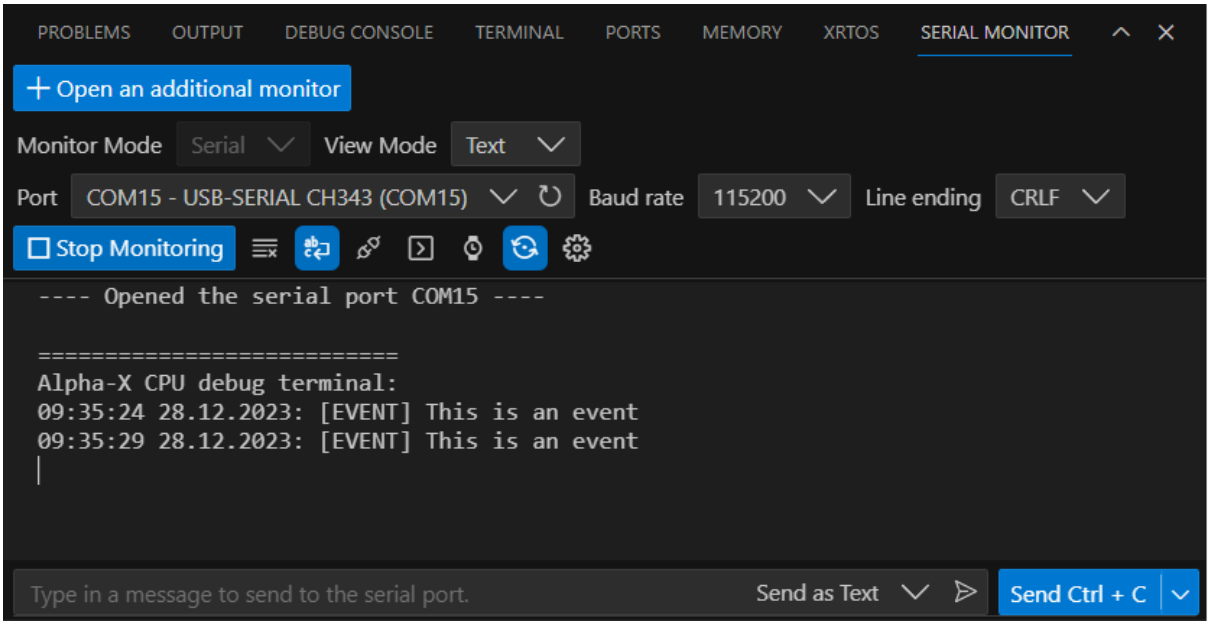
При значении `debug_messages = true`, на указанный отладочный порт будут отправляться сообщения отладки, которые пользователь создает с помощью функции `print_debug()`.

При значении `event_messages = true`, на указанный отладочный порт будут отправляться сообщения типа EVENT (события). События выводятся в терминал в формате: «**Время Дата: [EVENT] Текст события\r\n**».

События могут быть системными и пользовательскими. К системным событиям относятся запуск ПЛК, запись новой прошивки и т.п. Полный список системных событий указан здесь: Список событий (EVENTS). Системные события выводятся в терминал автоматически при появлении.

Пользовательские события создаются с помощью функции `print_event()`. Пользовательские события выводятся в терминал, однако не сохраняются в журнале событий. Пример вывода пользовательского события приведен ниже. Обратите внимание, что добавление отметки времени и перевод строки осуществляются автоматически.

```
void POU1()
{
  // Код, выполняемый циклически:
  print_event("This is an event");
}
```

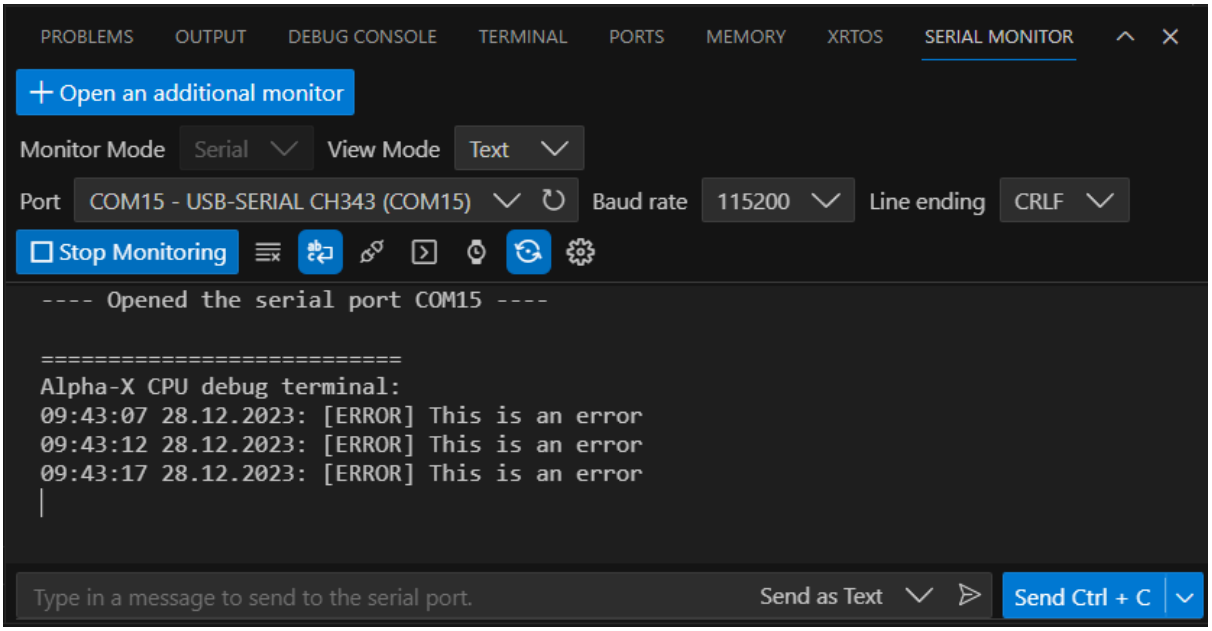


При значении `error_messages = true`, на указанный отладочный порт будут отправляться сообщения типа ERROR (ошибки). Ошибки выводятся в терминал в формате: «**Время Дата: [ERROR] Текст ошибки\r\n**».

Ошибки, как и события, могут быть системными и пользовательскими. К системным ошибкам относятся ошибки памяти, неверной инициализации и т.п. Полный список системных ошибок указан здесь: Список ошибок (ERRORS).

Пользовательские ошибки создаются с помощью функции `print_error()`. Пользовательские ошибки выводятся в терминал, однако не сохраняются в журнале ошибок. Пример вывода пользовательской ошибки приведен ниже. Как и для событий, добавление отметки времени и перевод строки осуществляются автоматически.

```
void POU1()
{
  // Код, выполняемый циклически:
  print_error("This is an error");
}
```



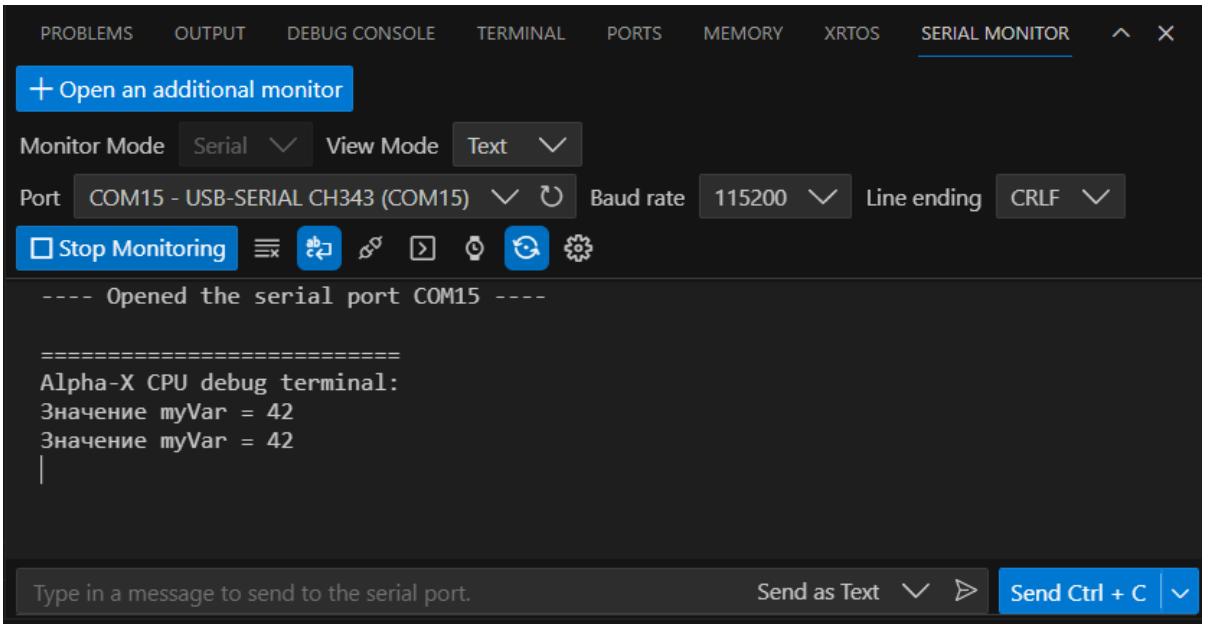
При настройке порта на режим Debug, данный порт нельзя использовать в режимах Modbus, SerialPort и т.п.

При создании нескольких отладочных портов с одинаковыми настройками, отладочные сообщения будут отправляться на все созданные порты. Функции для работы с отладочным терминалом также позволяют формировать строки динамически, подставляя значения переменных в строку перед отправкой. Работа этих функций во многом аналогична известной функции `printf()`.

Вывод динамической строки с подставлением значений переменных

Пример вывода значения переменной в терминал с использованием функции `print_debug()`.

```
int16_t myVar{42};

void POU1()
{
  // Код, выполняемый циклически:
  print_debug("Значение myVar = %d\r\n", myVar);
}
```



Значение переменной подставляется на место спецификатора %d. Данный спецификатор предназначен для вывода значений целочисленных переменных. Возможные спецификаторы вывода для разных типов переменных указаны ниже.

Спецификатор формата	Комментарий
`%hd`	Вывод целого знакового числа в десятичной системе счисления. Типы `short`, `int16_t`.
`%hu`	Вывод целого беззнакового числа в десятичной системе счисления. Типы `unsigned short`, `uint16_t`.
`%d`, `%i`	Вывод целого знакового числа в десятичной системе счисления. Типы `int`, `int32_t`.
`%u`	Вывод целого беззнакового числа в десятичной системе счисления. Типы `unsigned int`, `uint32_t`.
`%x`, `%X`	Вывод целого числа без знака в шестнадцатеричной системе счисления, нижний и верхний регистр соответственно.
`%c`	Вывод символа. Тип `char`.
`%p`	Вывод указателя на переменную, т.е. адреса хранения переменной.

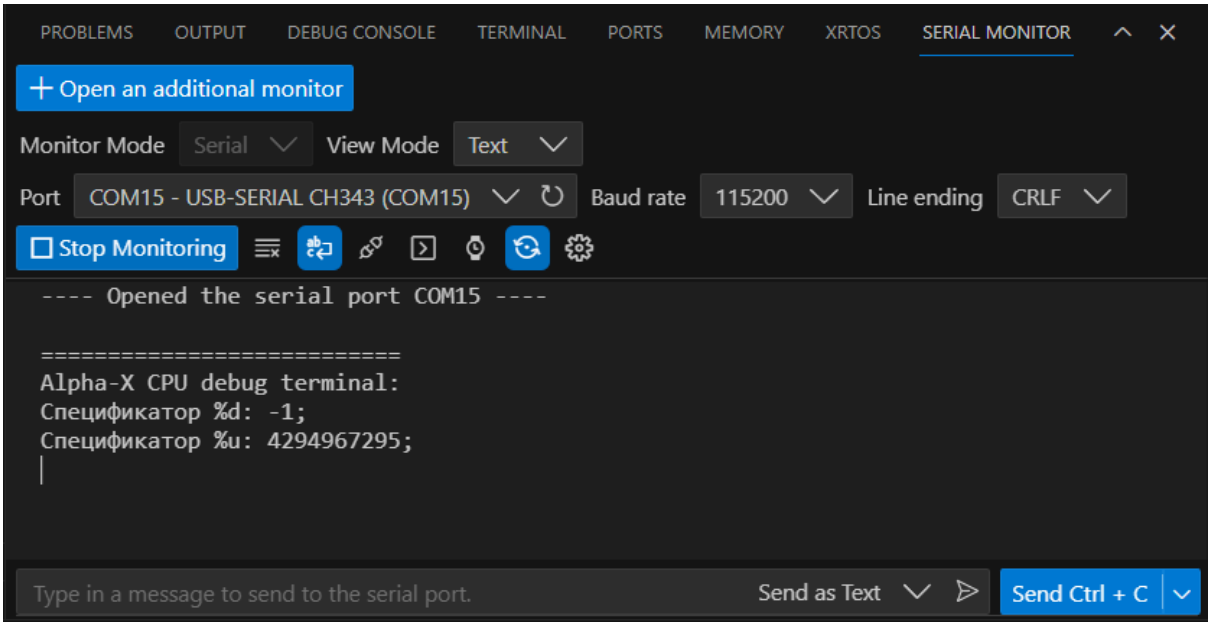
	Вывод переменных с плавающей точкой float, а также прочих типов размером более 4 байт не поддерживается. Для получения значений типа float рекомендуется воспользоваться расширением AXCodeMonitoring.

Спецификаторы вывода `%d` (`%i`) и `%u` можно использовать для вывода целочисленных значений типов переменных размером менее 4 байт. При использовании `%d` (`%i`) со знаковыми и `%u` с беззнаковыми типами, значения будут корректно выводиться в терминал. Однако, если потребуется выводить в терминал переменную размером менее 4 байт одновременно в знаковом и беззнаковом формате, то при использовании спецификаторов `%d` (`%i`) и `%u` число будет приводится к размеру 4 байта и результат будет соответствующим.

Ниже показан один из таких случаев.

```
int16_t myVar{-1};

void POU1()
{
  // Код, выполняемый циклически:
  print_debug("Спецификатор %%d: %d;\r\n", myVar);
  print_debug("Спецификатор %%u: %u;\r\n", myVar);
}
```

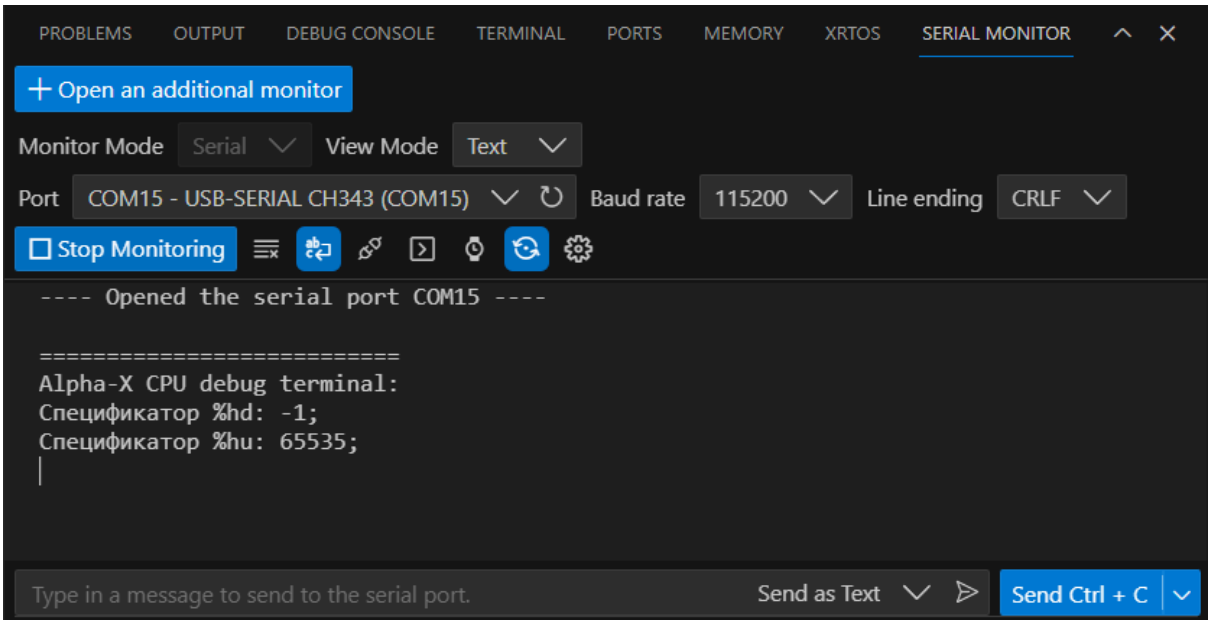
Выводимые сообщения:



Спецификатор `%u` не предназначен для вывода знаковых переменных. Значение, полученное таким образом, не соответствует диапазону переменной `uint16_t`, для которой максимальное значение составляет 65535. Это связано с тем, что 2-байтная переменная была преобразована в 4-байтную, т.к. спецификаторы `%d` (`%i`) и `%u` предназначены для 4-байтовых переменных.

Использование в этом случае спецификаторов 2-байтовых переменных %hd и %hu изменит вывод.

```
int16_t myVar{-1};

void POU1()
{
  // Код, выполняемый циклически:
  print_debug("Спецификатор %%hd: %hd;\r\n", myVar);
  print_debug("Спецификатор %%hu: %hu;\r\n", myVar);
}
```



Количество аргументов, принимаемых функцией `print_debug()`, не ограничено. Однако есть ограничение размера выводимой строки - 128 байт, при этом последний байт отведен под символ конца строки. При превышении размера буфера лишние символы в терминал выводиться не будут. Следует учитывать, что каждый символ кириллицы занимает 2 байта.

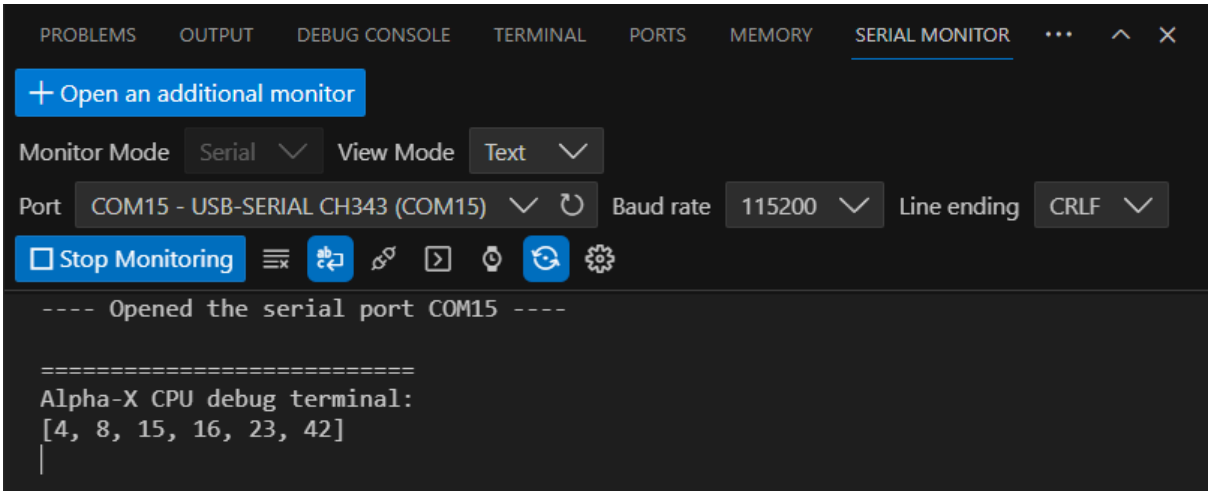
Вывод значений массива

Чтобы вывести данные из массива в терминал недостаточно передать этот массив в качестве аргумента в функцию `print_debug()`, так как в таком случае будет передан только указатель на первый элемент массива.

Для вывода значений массива необходимо вывести значение каждого элемента по отдельности. Для этих целей можно сделать отдельную функцию.

```
// Массив с данными
int16_t myArray[] {4, 8, 15, 16, 23, 42};

// Функция для вывода значений массива в терминал
// arr - ссылка на первый элемент массива
// size - количество элементов массива
void print_array(int16_t* arr, size_t size)
{
  print_debug("[");
  for (size_t i = 0; i < size - 1; i++)
  {
    print_debug("%d, ",arr[i]);
  }
  print_debug("%d]\r\n", arr[size - 1]);
}

void POU1()
{
  // Код, выполняемый циклически:
  print_array(myArray, sizeof(myArray) / sizeof(myArray[0]));
}
```

Результат вызова этой функции указан ниже.


PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
PORTS
MEMORY
SERIAL MONITOR
Open an additional monitor
Monitor Mode
Serial
View Mode
Text
Port
COM15 - USB-SERIAL CH343 (COM15)
Baud rate
115200
Line ending
CRLF
Stop Monitoring
---- Opened the serial port COM15 ----
==========
Alpha-X CPU debug terminal:
[4, 8, 15, 16, 23, 42]

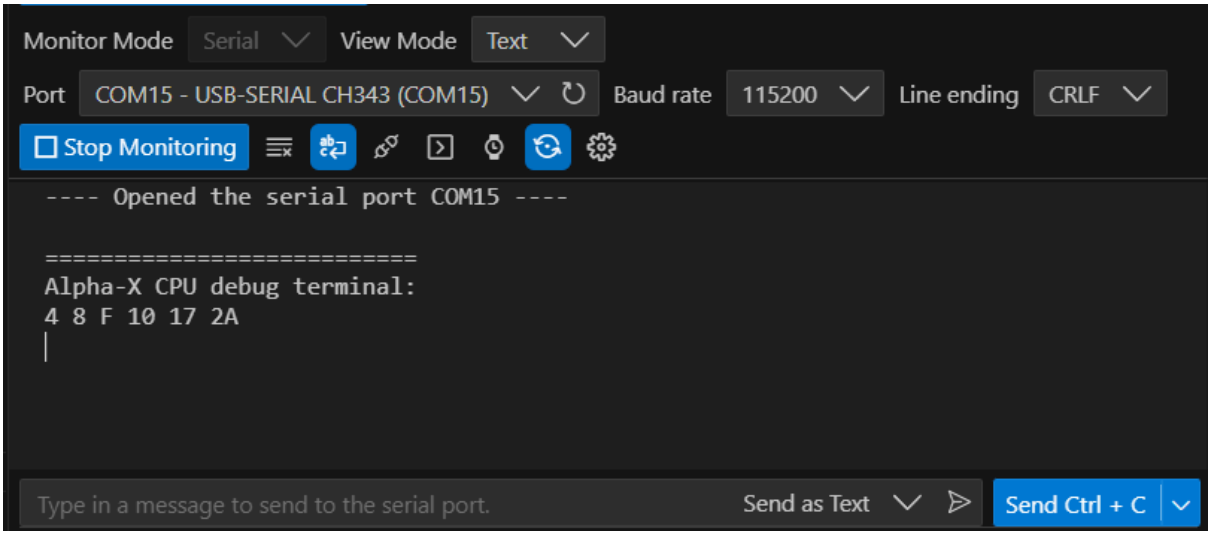
Вывод данных в формате HEX

При работе со сторонними устройствами может возникнуть необходимость выводить необработанные данные в терминал в формате HEX. Одним из вариантов решения данной задачи может быть код ниже.

```
// Массив с HEX-данными
int16_t myArray[] {0x04, 0x08, 0x0F, 0x10, 0x17, 0x2A};

// Функция для вывода значений массива в терминал
void print_array(int16_t* arr, size_t size)
{
  for (size_t i = 0; i < size - 1; i++)
  {
    print_debug("%X ",arr[i]);
  }
  print_debug("%X\r\n", arr[size - 1]);
}

void POU1()
{
  // Код, выполняемый циклически:
  print_array(myArray, sizeof(myArray) / sizeof(myArray[0]));
}
```

Для демонстрационных целей, данные явным образом записываются в массив в HEX-формате, для чего используется префикс `0x`. Для вывода данных в формате HEX используется спецификатор вывода `%X`.

Ниже представлен результат работы функции.



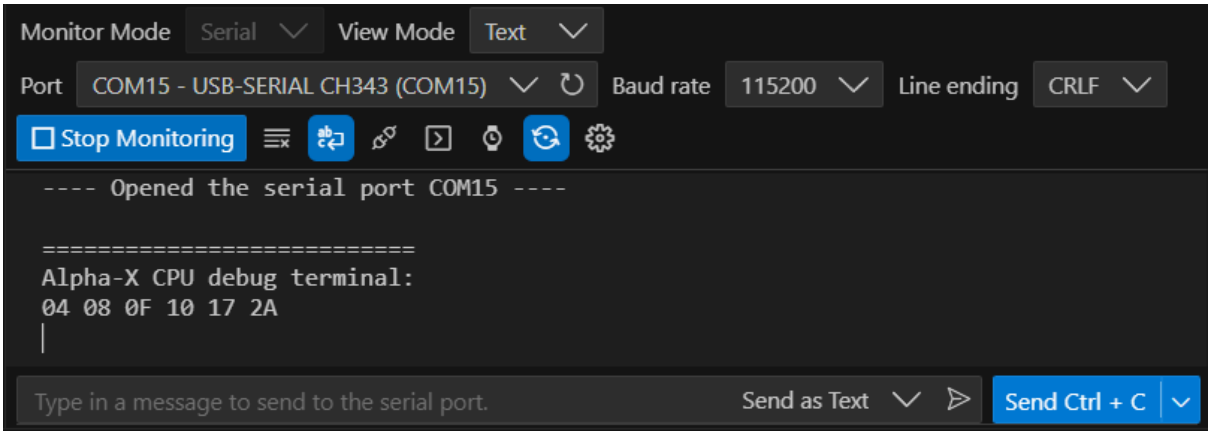
Данные были выведены в терминал в шестнадцатеричном формате, однако при этом были отброшены ведущие нули перед числами. Это может усложнять восприятие данных.

Для форматирования вывода данных можно воспользоваться функционалом библиотеки `<etl>`. Ниже представлен вариант вывода с использованием класса `etl::format_spec`.

```
// Массив с HEX-данными
int16_t myArray[] {0x04, 0x08, 0x0F, 0x10, 0x17, 0x2A};

// Функция для вывода HEX-значений массива в терминал
void print_hex(int16_t* arr, size_t size)
{
  etl::string<128> str;
  etl::format_spec format;
  format.hex().width(2).upper_case(true).fill('0');

  for (size_t i = 0; i < size; i++)
  {
    etl::to_string(arr[i], str, format, true);
    str += ' ';
  }
  str += "\r\n";
  print_debug(str);
}
```

```
void POU1()
{
  // Код, выполняемый циклически:
  print_hex(myArray, sizeof(myArray) / sizeof(myArray[0]));
}
```

Ниже показан результат работы этой функции.



Системные команды отладочного терминала

Вывод сообщений в терминал - не единственный функционал отладочного порта. Также поддерживается обработка команд, отправляемых из терминала.

	Для того, чтобы ПЛК корректно обрабатывал входящие команды от терминала, в настройках терминала необходимо установить автоматическую отправку символов перевода строки и возврата каретки. Если используете расширение SERIAL MONITOR, то в верхней части его окна, в выпадающем списке **Line ending** нужно выбрать **CRLF**.

Есть набор системных команд, которые контроллер умеет обрабатывать по умолчанию. Список этих команд представлен ниже.

Основная команда	Дополнительная команда	Комментарий
`h, help`		Выводит список доступных команд
`kernel`	`run`	Переводит ядро в режим RUN (работа)
	`stop`	Переводит ядро в режим STOP (остановка)
	`state`	Возвращает текущее состояние ядра
`version`		Возвращает модификацию ядра, версию ядра, версию загрузчика, версию платы
`errors`	`show`	Возвращает до 20 последних ошибок из журнала
	`first`	Возвращает до 20 первых ошибок из журнала
	`clear`	Очищает журнал ошибок
	`throw`	Добавляет в журнал отладочную ошибку с кодом cC2
`events`	`show N`	Возвращает до 20 последних событий из журнала, возможно указать начальный номер возвращаемого события N
	`throw`	Добавляет в журнал отладочное событие с кодом 199
`reload`		Перезагрузка контроллера с переходом в режим RUN (работа)

Основная команда	Дополнительная команда	Комментарий
	`service`	Перезагрузка контроллера с переходом в режим SERVICE (сервис)
	`service --converter`	Перезагрузка контроллера с переходом в режим SERVICE (сервис) и переводом порта USB в режим повторителя для работы с конфигуратором
	`bootloader`	Перезагрузка контроллера с переходом в режим BOOT (загрузка)
`u, user`		Префикс пользовательских команд

Прием и обработка пользовательских команд из терминала, функция scan_debug()

Для отправки пользовательской команды из терминала используется префикс `u` или `user`.

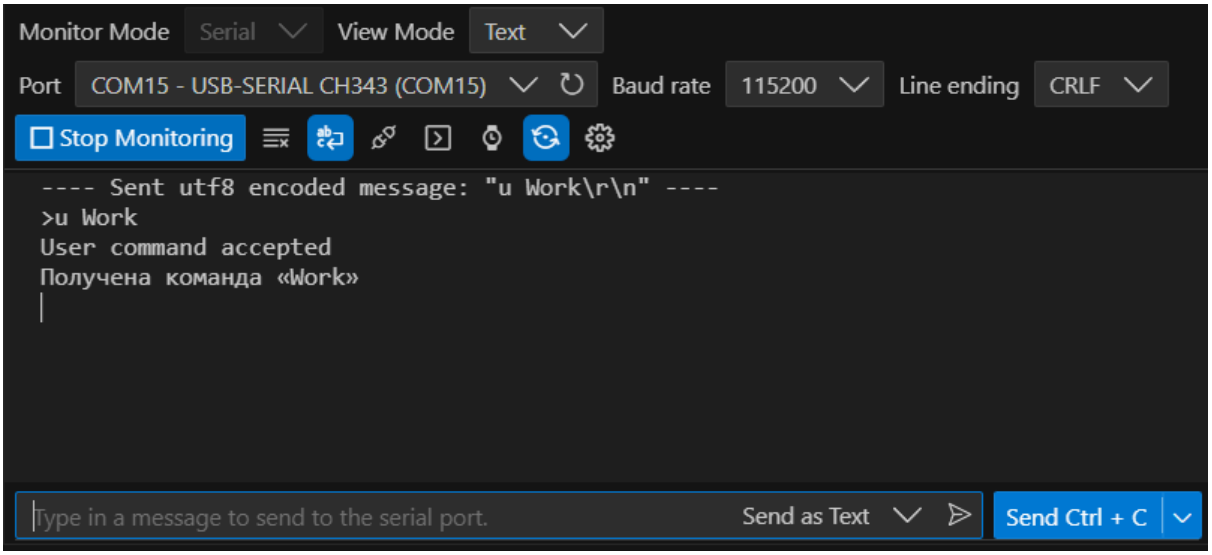
Для получения и обработки таких команд в коде программы используется функция `scan_debug()`. Ниже представлен пример использования данной команды.

```
// Строка размером 50 байт для приема пользовательской команды
etl::string<50> myString;

void POU1()
{
  // Код, выполняемый циклически:
  scan_debug(myString);

  if (!myString.empty()) {
    // Код выполняется, если строка не пустая
    if (myString == "Work")
    {
      print_debug("Получена команда «Work»\r\n");
    } else if (myString == "Stop")
    {
      print_debug("Получена команда «Stop»\r\n");
    } else
    {
      print_debug("Неизвестная команда\r\n");
    }
    // Очистка строки по завершению обработки
    myString.clear();
  }
}
```

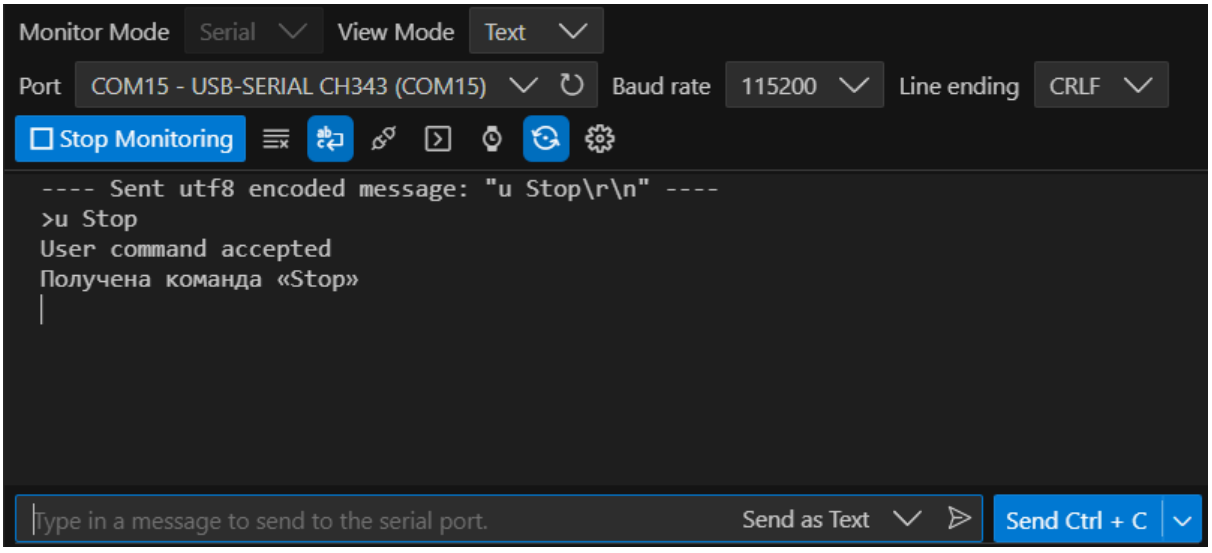
В примере выше программа сравнивает переданное значение с одним из двух вариантов. Если значением строки является «Work», выполняется одно действие. Если значение строки «Stop», выполняется другое действие. Если значение строки не совпадает ни с одним из указанных, выполняется третье

действие. После выполнения одного из действий значение строки очищается. Ниже показан пример работы данной программы:

Отправка команды **u Work**.



Отправка команды **u Stop**.

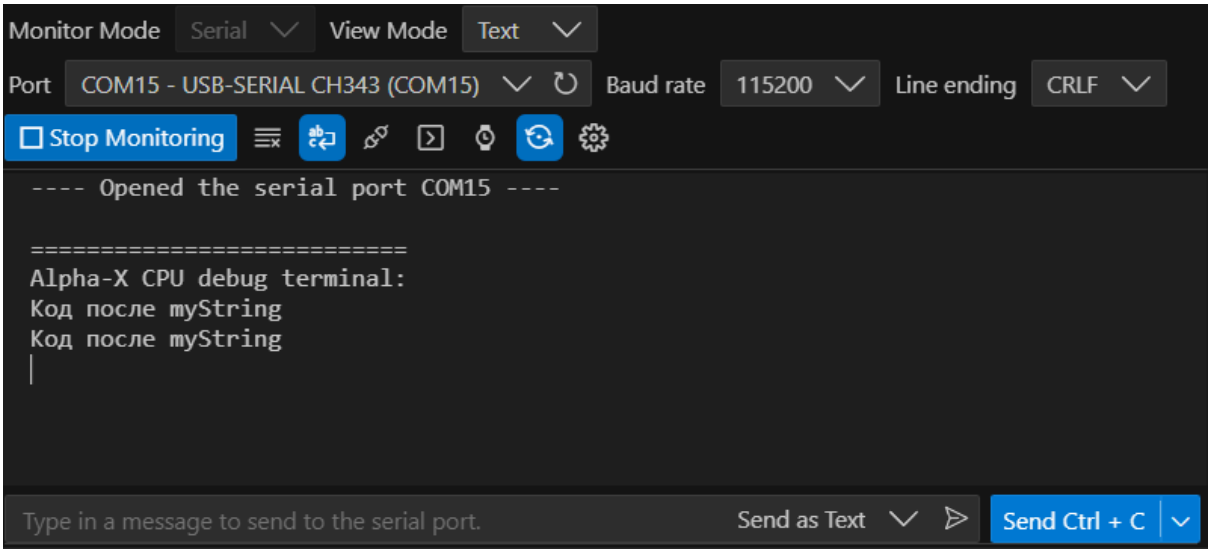


	Функция `scan_debug()` является неблокирующей. Это означает, что независимо от того, передал ли пользователь какую-либо команду в терминал, код после функции scan_debug() будет выполняться так, как если бы этой функции не существовало.

Пример ниже иллюстрирует эту ситуацию.

```
// Строка для хранения пользовательской команды
etl::string<50> myString;

void POU1()
{
  // Код, выполняемый циклически:
  scan_debug(myString);

  print_debug("Код после myString\r\n");
}
```

Результат работы кода:



Функция `print_debug()` выполняется несмотря на то, что пользователь не передает никаких сообщений в терминал. Это связано со способом работы `scan_debug()`: функция не останавливает выполнение дальнейшего кода. Все сообщения из терминала сначала попадают в промежуточный буфер. Функция `scan_debug()` проверяет наличие сообщений в промежуточном буфере. Если сообщения есть, копирует их в переданную пользователем переменную и очищает промежуточный буфер.

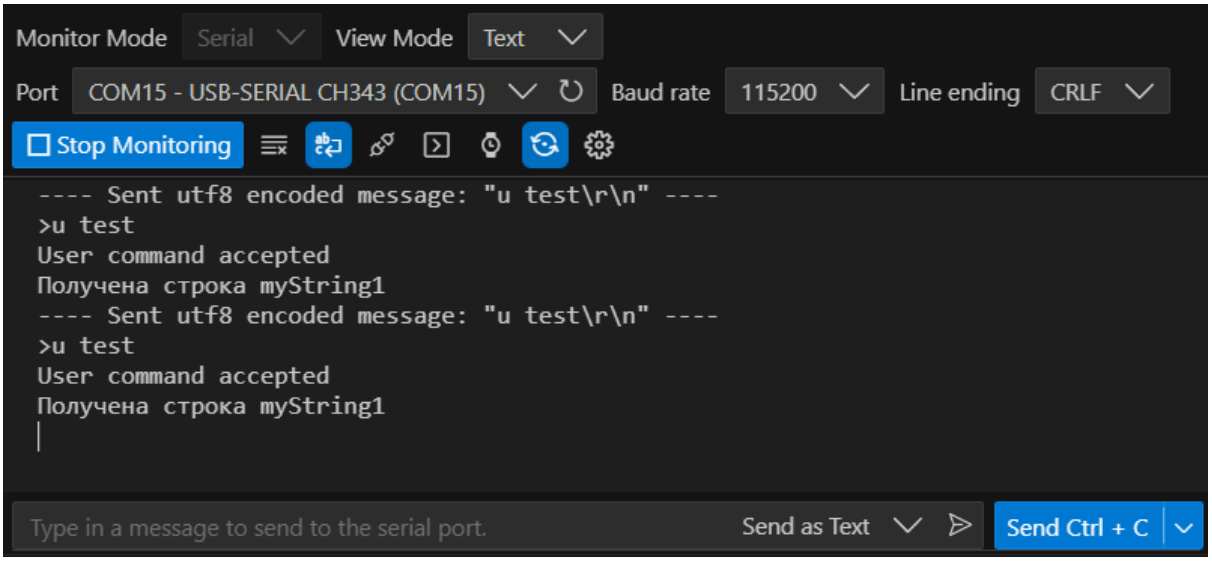
	Следует использовать только один экземпляр функции `scan_debug()` в пределах одного pou. Все последующие функции `scan_debug()` ничего не вернут, так как

	промежуточный буфер будет очищен после выполнения первой по счёту.

Ниже показан пример, иллюстрирующий особенность работы функции. Строка `myString2` никогда не будет заполнена, так как буфер приема будет очищен после выполнения первой функции `scan_debug()`.

```
// Строки для хранения пользовательских команд
etl::string<50> myString1;
etl::string<50> myString2;

void POU1()
{
  // Код, выполняемый циклически:
  scan_debug(myString1);
  scan_debug(myString2);

  if (!myString2.empty())
  {
    print_debug("Получена строка myString2\r\n");
    myString2.clear();
  }
  if (!myString1.empty())
  {
    print_debug("Получена строка myString1\r\n");
    myString1.clear();
  }
}
```

Результат выполнения кода:



Получение числа из строки

Функционал работы с терминалом можно использовать для корректирования уставок и настроек работы ПЛК на объекте. В некоторых случаях это позволяет отказаться от панели оператора.

Пример ниже демонстрирует подобное применение. Для получения целочисленного значения из строки используется функция `sscanf()`.

```
// Строка для получения команды пользователя
etl::string<50> myString;
// Настраиваемые параметры
int16_t value1, value2;

void POU1()
{
  // Код, выполняемый циклически:
  scan_debug(myString);
  if (!myString.empty())
  {
    // Расшифровка строки
    char str[20];
    int16_t newValue;
    sscanf(myString.data(), "%s %hd", str, &newValue);
    myString.assign(str);
```

```
      // Установка параметров
      if (myString == "value1")
      {
        value1 = newValue;
        print_debug("value1 is %hd now", value1);
      }
      if (myString == "value2")
      {
        value2 = newValue;
        print_debug("value2 is %hd now", value2);
      }
      myString.clear();
    }
  }
```

При организации кода как указано выше, пользователь должен отправлять команды из терминала в формате «**u [Параметр] [Значение]**», где:

- u - префикс пользовательской команды,
- [Параметр] ∈ [value1, value2],
- [Значение] ∈ [−32 768…+32 767];

В таком случае в переменным `value1` и `value2` будут присвоены новые значения.

Функции работы со временем

Для создание временных задержек, а также для отслеживания прошедшего времени, в среде AXCode предусмотрено несколько способов:

1. Использовать системный таймер, возвращающий время в миллисекундах, прошедшее с момента запуска ПЛК;
2. Использование функциональных блоков таймеров TON, TOF из библиотеки утилит utils;
3. Использование часов реального времени RTC;

Таймер операционной системы, функция GetSysTicks()

Таймер операционной системы возвращает время после старта контроллера в миллисекундах. Для этого используется следующая функция:

```
// Возвращает время после старта контроллера в миллисекундах
uint32_t GetSysTicks();
```

Данный таймер можно использовать для организации временных задержек. Ниже показан пример использования этой функции: выход do1 активируется через 3 секунды после включения входа di1.

```
// Отметка времени
uint32_t timeStamp;

void POU1()
{
  if (!di1)
  {
    // Отметка времени обновляется каждый раз,
    // до тех пор пока кнопка не нажата
    timeStamp = GetSysTicks();
  }

  do1 = false;
```

```
    if (GetSysTicks() - timeStamp >= 3000)
    {
      do1 = true;
    }
  }
```

Так как функция `GetSysTicks()` возвращает значение в формате uint32_t, использование этого подхода следует избегать, если детектируемый промежуток времени превышает ~4 294 967 секунд (порядка ~49 дней). При необходимости отслеживания промежутков в несколько дней рекомендуется использовать часы реального времени.

Функциональные блоки таймеров библиотеки ulib

Альтернативным вариантом организации временных задержек является использование классов из библиотеки ulib (входит в состав шаблона программы пользователя). Данная библиотека включает в себя следующие типы таймеров:

- TON - таймер задержки включения;
- TOFF - таймер задержки выключения;
- TP - импульс заданной длины;

Ниже представлен пример использования таймера TON. Примеры использования других таймеров можно найти в папке examples шаблона программы пользователя.

```
// Таймер задержки включения с уставкой 3 сек
TON timer({.pt = 3000});

void POU1()
{
  // Запуск таймера
  timer.en = true;
  if (timer.q())
  {
    // Действия по завершению отсчета
    timer.en = false;
```

```
    }
    // Выполнение ФБ таймера
    timer();
}
```

Часы реального времени (RTC)

Для модификаций, имеющих встроенные часы реального времени, доступны соответствующие функции для задания и чтения текущего времени ПЛК.

Для взаимодействия с часами реального времени используется объект `RTC`, уже созданный в заголовочном файле `plc/time_functions.h`. Чтение текущих значений времени, а также настройка часов осуществляется через экземпляр этого класса. Например, чтобы получить текущий год, используется следующий синтаксис:

```
// Получение текущего года
int16_t currentYear = RTC.getYear();
```

Ниже приведены методы объекта RTC, используемые для получения текущего времени. Доступно получение как отдельных значений года, месяца, дня и т.п., так и получение структуры, включающей в себя все эти поля. Методы приведены в заголовочном файле time_functions.h.

```
//! Возвращает текущие время и дату в виде структуры
static TimeStruct getTimeStruct();

//! Возвращает текущие секунды
static int16_t getSeconds();

//! Возвращает текущие минуты
static int16_t getMinutes();

//! Возвращает текущие часы
```

```
    static int16_t getHours();

    //! Возвращает текущий день (число)
    static int16_t getDate();

    //! Возвращает текущий месяц
    static int16_t getMonth();

    //! Возвращает текущий год
    static int16_t getYear();

    //! Возвращает текущий день недели
    static int16_t getDay();
```

Методы, используемые для установки (настройки) часов реального времени.

```
    //! Устанавливает текущие секунды
    static void setSeconds(int16_t seconds);

    //! Устанавливает текущие минуты
    static void setMinutes(int16_t minutes);

    //! Устанавливает текущие часы
    static void setHours(int16_t hours);

    //! Устанавливает текущий день (число)
    static void setDate(int16_t date);

    //! Устанавливает текущий месяц
    static void setMonth(int16_t month);

    //! Устанавливает текущий год
    static void setYear(int16_t year);

    //! Устанавливает текущие время и дату через структуру времени
```

```
static void setTime(const TimeStruct& time_struct);

//! Устанавливает текущие время и дату через переменные времени
static void setTime(
    int16_t seconds,
    int16_t minutes,
    int16_t hours,
    int16_t date,
    int16_t month,
    int16_t year
);
```

Преобразование Unix time

Отметка времени unix — это способ отслеживания времени, принятая в некоторых операционных системах. Этот подсчет начинается в эпоху Unix 1 января 1970 года в UTC. Таким образом, временная метка unix — это всего лишь количество секунд между определенной датой и эпохи Unix. Следует также отметить, что этот момент времени технически не меняется независимо от того, где вы находитесь на земном шаре.

Для конвертирования времени из UNIX-формата в структуру типа `TimeStruct` используется метод `GetTimeStruct()`, который находится в области имен `time_menu`:

```
// Преобразование из Unix в TimeStruct
TimeStruct ts = time_menu::GetTimeStruct(1734951585);
```

Для операции обратного преобразования, из структуры TimeStruct в UNIX-формат, необходимо использовать метод `GetTimeFromStruct()` из области имен `time_menu`:

```
// Преобразование из TimeStruct в Unix
int64_t unixTime = time_menu::GetTimeFromStruct(ts);
```

Функциональные блоки PLCopen

Одним из аспектов работы ПЛК является связь с другими устройствами по интерфейсу (например, опрос внешнего частотного преобразователя). Время выполнения запросов в этом случае является непостоянной величиной и зависит от множества внешних факторов: протокола, количества запрашиваемой информации, скорости работы интерфейса, протяженности линии связи и прочего. При этом время выполнения одного цикла пользовательской программы (pou), как правило, во много раз выше, чем скорость выполнения одного запроса по интерфейсу. Эта особенность требует определенной организации кода, позволяющей избежать ситуаций, когда одна часть обрабатываемых данных уже обновлена, а другая — нет.

Для организации связи с внешними устройствами используются специальные классы, выполненные по модели асинхронных блоков PLCOpen (независимая международная организация, занимающаяся стандартизацией и оптимизацией). Классы такого рода на платформе AXCode имеют общее название **Функциональные блоки**. Работа с такими функциональными блоками будет более привычна для инженеров, имеющих опыт разработки в других популярных текстовых средах разработки для ПЛК.

На диаграмме ниже показан общий принцип работы таких блоков: начало выполнения запускается в первом цикле pou, при этом сама работа происходит «параллельно» циклу основной технологической программы и завершается только после нескольких итераций pou.

С точки зрения ядра это работает следующим образом: запуск функционального блока активирует системную задачу, которая выполняет определенный процесс, например, запрос Modbus RTU.

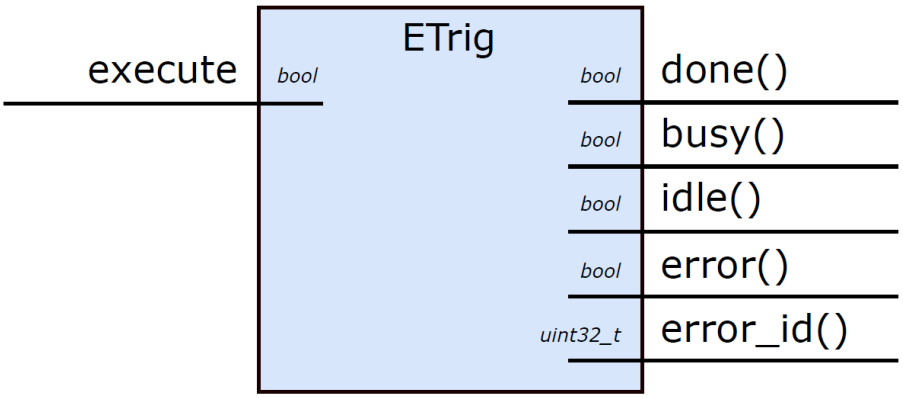
Модель PLCOpen подразумевает две основных вариаций таких функциональных блоков:

- ФБ типа Execute - выполнение ФБ этого типа происходит единожды, при изменении состояния входа `execute` из `false` в `true`. К таким ФБ относится ETrig и его разновидности;
- ФБ типа Enable - выполнение ФБ этого типа происходит непрерывно, пока активен вход `enable`. К таким ФБ относится ETrig и его разновидности;

Разновидности всех возможных блоков можно найти здесь: https://plcopen.org/sites/default/files/downloads/creating_plcopen_compliant_function_block_libraries.pdf

Асинхронный блок связи типа Etrig

Асинхронный блок связи типа Etrig запускается в работу по фронту сигнала на входе `execute`. Ниже представлено графическое представление входов и выходов базового блока .



В реализации для AXCode, все входы блока являются публичными переменными, что позволяет программисту свободно изменять их состояния извне. В свою очередь, все выходы блока являются методами, таким образом их значения можно только прочитать (нельзя изменить).

Обязательные входы блока:

- `bool execute` - вход запуска блока. Блок запускается в работу когда сигнал `execute` переходит с `false` на `true`;

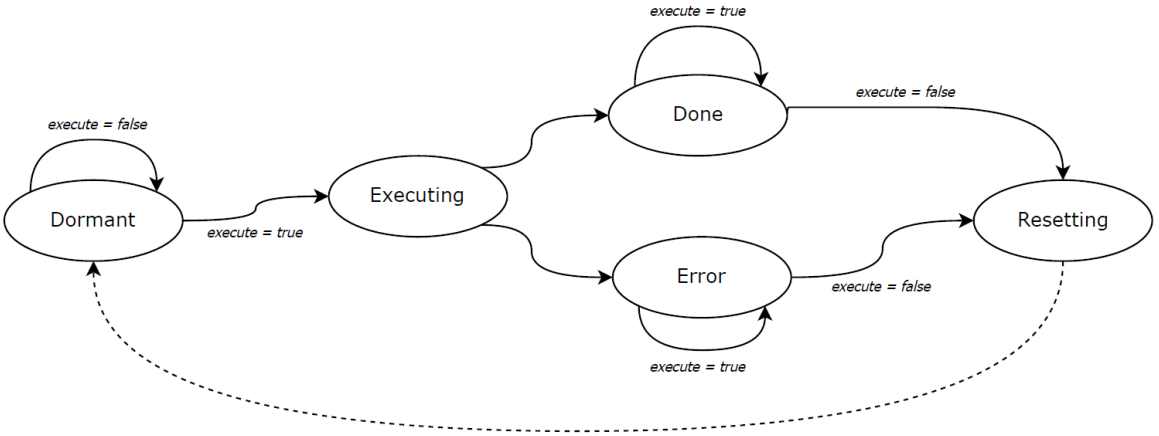
Обязательные выходы блока:

- `bool done()` - метод возвращает значение `true` в результате **успешного** выполнения блока. Данное значение сохраняется до тех пор, пока блок не будет сброшен. Сброс блока осуществляется при переходе `execute` из `true` в `false`;
- `bool busy()` - метод возвращает значение `true` во время выполнения блока. Данное значение сохраняется до тех пор, пока блок не перейдет в состояние **done** или **error**;
- `bool idle()` - метод возвращает значение `true` если блок не выполняется;
- `bool error()` - метод возвращает значение `true` в результате **неудачного** выполнения блока. Данное значение сохраняется до тех пор, пока блок не будет сброшен. Сброс блока осуществляется при переходе `execute` из `true` в `false`;
- `uint32_t error_id()` - метод возвращает номер кода ошибка в случае **неудачного** выполнения блока. Данный код ошибки сохраняется до тех пор, пока блок не будет сброшен. Коды ошибок являются индивидуальными и зависят от конкретной реализации блока;

В один момент времени только один из четырех методов `done()`, `busy()`, `idle()` или `error()` может возвращать значение `true`.

Перечисленные выше входы и выходы являются обязательными для любого блока, выполненного по модели `ETrig`, однако в зависимости от конкретной реализации, могут добавляться дополнительные входы и выходы. К примеру, блок `ModbusMaster` имеет дополнительные входные поля для хранения номера COM-порта, скорости и прочих параметров, свойственных Modbus Master.

Ниже представлена диаграмма состояний и переходов функционального блока типа `ETrig`.



DORMANT (ожидание). В начальный момент времени блок находится в ожидании команды на выполнение. После того, как сигнал `execute` изменился с `false` на `true`, происходит переход в состояние EXECUTING. При этом все входные переменные копируются во внутреннюю память блока. Таким образом, изменение входных параметров в процессе выполнение блока никак не повлияет на результат его первого выполнения.

EXECUTING (выполнение). Блок выполняется, например, происходит связь по интерфейсу. В данном состоянии метод `busy()` возвращает состояние `true`. В случае успешного выполнения блок перейдет в состояние DONE. В случае неудачного выполнения блок перейдет в состояние ERROR. После перехода в следующее состояние метод `busy()` будет возвращать `false`.

DONE (успешно выполнено). Блок успешно выполнил поставленную задачу. Метод `done()` возвращает сигнал `true`. Блок находится в данном состоянии до тех пор, пока сигнал `execute` не примет значение `false`.

ERROR (неудачное выполнение). При выполнении блока возникла ошибка. Метод `error()` возвращает значение `true`. Метод `error_id()` возвращает код ошибки. Блок находится в данном состоянии до тех пор, пока сигнал execute не примет значение `false`.

RESETTING (сброс). Сброс внутренних параметров блока. В зависимости от конкретной реализации блока и выполняемых им задач, в данном состоянии может происходить очистка памяти, закрытие порта связи и прочие действия. После завершения сброса блок возвращается в начальное состояние DORMANT.

Стоит понимать, что каждое из вышеперечисленных состояний может занимать по времени как один так и несколько циклов подпрограммы, при этом

выполнение будет происходить «параллельно» выполнению технологической программы. Тем не менее, обновление выходных переменных блока будет происходить только в момент выполнения pou при вызове самого блока.

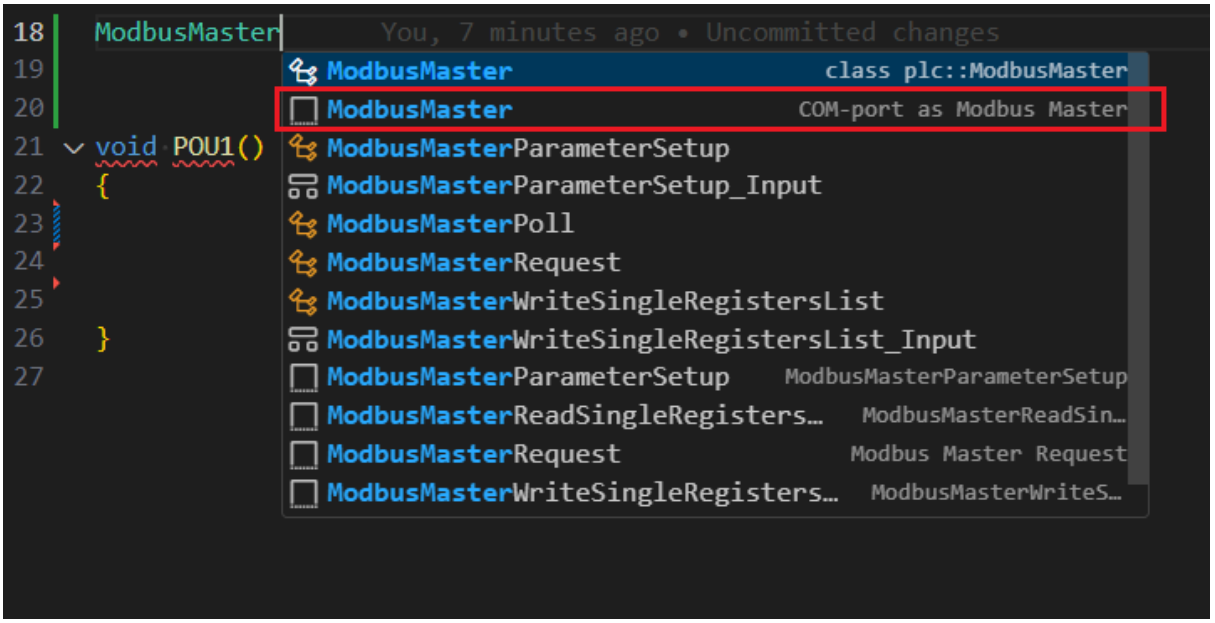
Объявление и инициализация функциональных блоков

Все асинхронные блоки объявляются только за пределами функции POU().

Инициализация происходит передачей данных в формате структуры. Можно указывать не все параметры, тогда оставшиеся будут приняты по умолчанию. Для облегчения работы с функциональными блоками в AXCode добавлены сниппеты.

	Сниппеты - это шаблоны, упрощающие ввод повторяющихся строчек кода. В AXCode сниппеты используются для быстрого объявления функциональных блоков.

Для того, чтобы воспользоваться сниппетом, начните вводить название нужного ФБ после чего выберите соответствующую опцию. Сниппеты в AXCode помечаются иконкой, как на картинке ниже.



После выбора сниппета, редактор автоматически подставит все возможные настроечные параметры для ФБ. Ниже продемонстрировано, как выглядит соответствующий код на примере ФБ `ModbusMaster`.

```
ModbusMaster modbus_master ({
  // Номер COM-порта
  .com              = COM1,
  // Список устройств для автоматического опроса
  .devices          = {},
  // Скорость обмена, бит/с
  .baudrate         = 115200,
  // Четность
  .parity           = COM_PARITY_NONE,
  // Кол-во стоп-битов
  .stop_bits        = 1,
  // Пауза перед отправкой команды опроса (мс)
  .poll_delay       = 3,
  // Пауза после отправки широковещательного запроса (мс)
  .turnaround_delay = 100,
  // Время ожидания ответа (мс)
  .response_timeout = 300,
  // Количество попыток связи
  .failed_attempts  = 3,
  // Время восстановления опроса (мс)
  .restore_timeout  = 3000
});

void POU1()
{
  ...
}
```

Использование функциональных блоков

Ниже представлена типовая схема использования ФБ. В приведенном ниже примере:

- Участок /* 1 */ – условие запуска блока;
- Участок /* 2 */ – обработка результатов успешного выполнения блока;
- Участок /* 3 */ – обработка результатов неудачного выполнения блока;
- Участок /* 4 */ – непосредственный **вызов блока**, при котором происходит обновление выходных переменных;

Подобная структура автоматически создается редактором при выборе сниппета `execute`.

```
void POU1()
{
  /* 1 */
  if (/* условие */)
  {
    blockEtrig.execute = true;
  }
  /* 2 */
  if (blockEtrig.done())
  {
    /* Обработка результатов успешного выполнения */
    blockEtrig.execute = false;
  }
  /* 3 */
  if (blockEtrig.error())
  {
    /* Обработка результатов неудачного выполнения */
    blockEtrig.execute = false;
  }
  /* 4 */
  blockEtrig();
}
```

- В начальный момент времени, если выполняется условие в скобках (участок /* 1 */), то входному полю `execute` присваивается значение `true`. Стоит заметить, что в данный момент выполнение блока еще не началось,

так как переход между состояниями осуществляется на участке /* 4 */, при вызове самого блока. По этой же причине условия на участках /* 2 */ и /* 3 */ выполнены не будут, так как ФБ еще ни разу не вызывался.

- На участке /* 4 */ блок обнаружит изменение сигнала `execute`, копирует все входные переменные во внутреннюю память и перейдет в состояние EXECUTING. В этот момент операционная система запускает соответствующую задачу: например, осуществляет связь по интерфейсу;
- Если время выполнения состояния EXECUTING намного больше, чем период вызова pou, то последующие вызовы pou никаким образом не окажут влияния на работу блока. Блок будет находиться в состоянии EXECUTING до тех пор, пока не получит результат от соответствующей задачи операционной системы;
- Затем, в зависимости от результата выполнения, один из методов `done()` или `error()` вернет значение `true`. Далее пользователь обрабатывает результаты в зависимости от требуемой ему логики. При любом исходе необходимо присвоить полю `execute` значение `false`, чтобы при следующем вызове блока произошел его сброс RESETTING и дальнейший переход в начальное состояние DORMANT.

	Если условие на участке /* 1 */ будет активным всегда, то после выполнения ФБ будет циклически перезапускаться в каждом следующем цикле pou. Это может быть необходимо, если поставленную задачу требуется выполнять постоянно.

Стоит заметить, что в зависимости от того, где располагается вызов блока (участок /* 4 */), будет изменяться и количество циклов pou, требуемых для однократного выполнения одного ФБ. К примеру, если вызов блока будет находиться выше, чем присвоение полю `execute` значения `true`, то запуск выполнения блока будет происходить на один цикл pou позже.

Интерфейсы связи

В зависимости от модификации ПЛК отличается список доступных COM-портов. Актуальный список для конкретного устройства хранится в перечислении `COM_PORT`. в файле system_api.h.

Ниже представлено перечисление для модели Alpha-X CPU 01-1 00.

```
//! Список COM-портов модификации
enum COM_PORT : uint8_t
{
  COM1,
  COM2,
  COM3,
  COM_USB,
  COM_SIZE
};
```

Настройка COM-порта в режиме Modbus Master

Настройка COM-порта в режиме Modbus Master необходима для использования следующего функционала:

- ручной опрос устройств по Modbus RTU;
- автоматический опрос устройств по Modbus RTU;
- автоматический опрос модулей расширения Alpha-X;

Для настройки COM-порта на работу по протоколу Modbus в режиме Master необходимо создать экземпляр объекта `ModbusMaster` за пределами функции POU(). Для автоматической подстановки кода можно воспользоваться соответствующим сниппетом. Параметры инициализации передаются в составе структуры как показано в примере ниже.

```
ModbusMaster modbus_master ({
  // Номер COM-порта
  .com              = COM1,
  // Список устройств для автоматического опроса
  .devices          = {},
  // Скорость обмена, бит/с
  .baudrate         = 115200,
  // Четность
  .parity           = COM_PARITY_NONE,
  // Кол-во стоп-битов
  .stop_bits        = 1,
  // Пауза перед отправкой команды опроса (мс)
  .poll_delay       = 3,
  // Пауза после отправки широковещательного запроса (мс)
  .turnaround_delay = 100,
  // Время ожидания ответа (мс)
  .response_timeout = 300,
  // Количество попыток связи
  .failed_attempts  = 3,
  // Время восстановления опроса (мс)
  .restore_timeout  = 3000
});

void POU1()
{
  ...
}
```

Параметры ModbusMaster	
Название	**Описание**
.com	***COM-порт*** Один из доступных вариантов перечисления `COM_PORT`.
.devices	***Список устройств*** Данное поле необходимо для реализации автоматического опроса (подробнее см. Автоматический опрос). Если функционал автоматического опроса не используется, указать `nullptr` или `{}`.
.baudrate	***Скорость обмена, бит/с*** Список возможных скоростей работы: 256000, 128000, 115200, 76800, 57600, 38400, 28800, 19200, 14400, 9600, 4800, 2400.
.parity	***Четность*** Один из доступных вариантов перечисления `COM_PARITY`.
.stop_bits	***Количество стоп-битов*** Доступные варианты: 1, 2.
.poll_delay	***Пауза перед отправкой команды*** Время выдержки перед отправкой новой команды. Требуется увеличить, если устройства в сети не успевают переключиться в режим приема.
.turnaround_delay	***Пауза после отправки широковещательного запроса***
.response_timeout	***Время ожидания ответа*** Максимальный период ожидания между окончанием отправки запроса и окончанием получения ответного сообщения. В случае, если данное время будет превышено, запрос завершится с ошибкой.
.failed_attempts	***Количество попыток связи*** В случае, если текущий запрос окажется неудачным, контроллер автоматически будет повторять его указанное количество раз, прежде чем вернуть ошибку и перейти к следующим запросам.
.restore_timeout	***Время восстановления опроса***

Параметры ModbusMaster	
Название	**Описание**
	Минимально необходимое время, которое должно пройти с момента неудачного автоматического опроса Slave-устройства, перед тем как контроллер попробует возобновить опрос этого устройства.

Буферы хранения данных ModbusBuffer и ModbusCoilBuffer

Для хранения и передачи данных, полученных по Modbus, используется класс `ModbusBuffer`.

Создавать объект `ModbusBuffer` требуется за пределами функции POU(), при создании указать в угловых скобках размер буфера, как показано в примере ниже.

```
ModbusBuffer<10> request_buffer; // Буфер на 10 Modbus-регистров

void POU1()
{
  ...
}
```

Размер одной ячейки буфера соответствует размеру одного стандартного регистра Modbus - 2 байта. Соответственно, буфер размером 10 предназначен для хранения 10 регистров Modbus.

Класс `ModbusBuffer` имеет методы, позволяющие считать или записать значения с учетом требуемого типа данных. Например, чтобы считать значение типа `float`, которое содержится в 4-м регистре посылки, используется следующий синтаксис:

```
// Считывание типа float из 4-го регистра посылки
float myFloat = request_buffer.readFloat(4);
```

Синтаксис записи значения типа `int` в 7 регистр посылки:

```
// Запись типа int в 7-ой регистр посылки
int16_t myInt = 42;
request_buffer.writeInt16(7, myInt);
```

Полный список доступных методов можно найти в файле plc_modbus_buffer.h.

Класс `ModbusBuffer` предназначен для работы с 16-битными регистрами (Input Registers / Holding Registers). Для работы с дискретными регистрами (Input Contacts / Output Coils) используется класс `ModbusCoilBuffer`.

Для удобства работы с регистрами можно создать перечисление enum, в котором указать карту регистров опрашиваемого устройства. Пример такого подхода можно найти в разделе Modbus Master встроенных примеров examples\05_modbus_rtu_master\04_modbus_ecd2.cpp.

Ручной запрос ModbusMasterRequest

Для работы с Modbus-запросами в ручном режиме используется функциональный блок `ModbusMasterRequest`. Данный блок сделан на базе функционального блока `ETrig`.

Блок `ModbusMasterRequest` предназначен для отправки посылок по протоколу Modbus RTU, когда соответствующий порт ПЛК настроен в режиме Master. Задачей блока является формирование Modbus RTU посылки на основании данных полей блока, приём ответа от Slave-устройства, проверку целостности данных, обработку ошибок и предоставление полученных данных.

Создать экземпляр объекта `ModbusMasterRequest` требуется за пределами функции POU(). Для создания можно воспользоваться соответствующим сниппетом. Параметры инициализации передаются в составе структуры, как показано в примере ниже.

```
ModbusMasterRequest request ({
  // Modbus Master для отправки запроса
  .master               = &modbus_master,
  // Адрес опрашиваемого устройства
  .device_address       = 1,
  // Номер функции Modbus
  .function             = 3,
  // Начальный адрес регистров Modbus в запросе
  .starting_address     = 0,
  // Указатель на таблицу регистров
  .buffer               = &request_buffer,
  // Общее кол-во регистров Modbus
  .registers_count      = 0,
  // Максимальное кол-во регистров Modbus в одном запросе к устройству
  .device_max_registers = 0,
  // Время ожидания ответа в запросе, мс (0 - настройки из Modbus Master)
  .response_timeout     = 0
});


void POU1()
{
  ...
}
```

Параметры ModbusMasterRequest	
Название	**Описание**
.master	***Указатель на объект ModbusMaster***
.device_address	***Modbus-адрес устройства в запросе***
.function	***Modbus-функция*** Поддерживаемые номера функций: 01, 02, 03, 04, 05, 06, 15, 16
.starting_address	***Адрес первого регистра***
.buffer	***Указатель на объект ModbusBuffer***
.registers_count	***Количество регистров в запросе***
.device_max_registers	***Максимальное кол-во регистров Modbus в одном запросе к устройству***
.response_timeout	***Время ожидания ответа в запросе, мс***

Использовать функциональный блок внутри функции POU() можно так, как это было описано в разделе Использование функциональных блоков.

После того, как запрос будет выполнен, метод функционального блока `done()` возвращает значение `true`. Таким образом можно создать условие, при котором будет происходить обработка полученных данных, например, присвоение значений технологическим переменным.

Ниже представлен один из возможных вариантов.

```
void POU1()
 {
   // Код, выполняемый циклически:
   request.execute = true;
   if (request.done())
   {
     // Обработка результатов запроса
     bool myBool = request_buffer.readBool(0);
     int16_t myInt16 = request_buffer.readInt16(1);
```

```
        float myFloat = request_buffer.readFloat(2);
        request.execute = false;
      } else if (request.error())
      {
        // Вывод ошибки в терминал
        print_debug("error!: %d\r\n", request.error_id());
        request.execute = false;
      }
      request();
    }
```

Полный список готовых примеров с реализацией опроса по Modbus представлен в папке examples\05_modbus_rtu_master.

Ошибки ФБ ModbusMasterRequest

Список ошибок представлен в перечислении `MODBUS_ERROR_CODE` в заголовочном файле fb_modbus.h.

```
enum MODBUS_ERROR_CODE
{
  //! Неподдерживаемая функция, ошибка Modbus №1
  MODBUS_ERROR_CODE_ILLEGAL_FUNCTION      = 101,
  //! Неверный адрес регистров, ошибка Modbus №2
  MODBUS_ERROR_CODE_ILLEGAL_DATA_ADDRESS  = 102,
  //! Недопустымие значения регистров в запросе, ошибка Modbus №3
  MODBUS_ERROR_CODE_ILLEGAL_DATA_VALUE    = 103,
  //! Ошибка обраб.данных на конечном устройстве, ошибка Modbus №4
  MODBUS_ERROR_CODE_SERVER_DEVICE_FAILURE = 104,
  //! Ошибка подготовки запроса Modbus Master
  MODBUS_ERROR_CODE_MODBUS_REQUEST_ERROR  = 197,
  //! Получен неизвестный ответ
  MODBUS_ERROR_CODE_UNKNOWN_ANSWER        = 198,
  //! Устройство не отвечает
  MODBUS_ERROR_CODE_DEVICE_NOT_RESPONDING = 199
```

```
};
```

Настройка COM-порта в режиме Modbus RTU Slave

Для настройки COM-порта на работу по протоколу Modbus RTU в режиме Slave необходимо создать экземпляр объекта `ModbusSlave` за пределами функции POU(). Для автоматической подстановки кода можно воспользоваться соответствующим сниппетом. Параметры инициализации передаются в составе структуры как показано в примере ниже.

```
ModbusBuffer<100> request_buffer;

ModbusSlave modbus_slave ({
  // Номер COM-порта
  .com              = COM2,
  // Таблица Holding Registers
  .holding_registers = &request_buffer,
  // Таблица Input Registers
  .input_registers   = &request_buffer,
  // Таблица Coils
  .coils             = {},
  // Таблица Discrete Inputs
  .discrete_inputs   = {},
  // Адрес Slave в сети Modbus
  .address           = 1,
  // Скорость обмена, бит/с
  .baudrate          = 115200,
  // Четность
  .parity            = COM_PARITY_NONE,
  // Кол-во стоп-битов
  .stop_bits         = 1,
  // Пауза перед отправкой ответа (мс)
  .poll_delay        = 0
});

void POU1()
{
```

```
    ...
}
```

Параметры ModbusSlave	
Название	**Описание**
.com	***COM-порт*** Один из доступных вариантов перечисления `COM_PORT`.
.holding_registers	***Указатель на экземпляр ModbusBuffer*** Данные регистры будут доступны внешнему Modbus Master: ● для чтения функцией 3 (0x03); ● для записи функциями 6 (0x06), 16 (0x10); Если регистры типа Holding не используются, при инициализации передать нулевой указатель `nullptr` или `{}`. Можно использовать один экземпляр `ModbusBuffer` как для регистров типа Holding, так и для регистров типа Input.
.input_registers	***Указатель на экземпляр ModbusBuffer*** Данные регистры будут доступны внешнему Modbus Master: ● для чтения функцией 4 (0x04); Если регистры типа Input не используются, при инициализации передать нулевой указатель `nullptr` или `{}`. Можно использовать один экземпляр `ModbusBuffer` как для регистров типа Input, так и для регистров типа Holding.
.coils	***Указатель на экземпляр ModbusCoilBuffer*** Данные регистры будут доступны внешнему Modbus Master: ● для чтения функцией 1 (0x01); ● для записи функциями 5 (0x05), 15 (0x0F); Если регистры типа Coils не используются, при инициализации передать нулевой указатель `nullptr` или `{}`. Можно использовать один экземпляр `ModbusBuffer` как для регистров типа Coils, так и для регистров типа Discrete Input.
.discrete_inputs	***Указатель на экземпляр ModbusCoilBuffer*** Данные регистры будут доступны внешнему Modbus Master: ● для чтения функцией 2 (0x02);

Параметры ModbusSlave	
Название	**Описание**
	Если регистры типа Discrete Inputs не используются, при инициализации передать нулевой указатель `nullptr` или `{}`. Можно использовать один экземпляр `ModbusBuffer` как для регистров типа Discrete Inputs, так и для регистров типа Coils.
.address	***Адрес ПЛК*** Задает Slave-адрес для ПЛК в сети Modbus RTU для выбранного COM-порта.
.baudrate	***Скорость обмена, бит/с*** Список возможных скоростей работы: 256000, 128000, 115200, 76800, 57600, 38400, 28800, 19200, 14400, 9600, 4800, 2400.
.parity	***Четность*** Один из доступных вариантов перечисления `COM_PARITY`.
.stop_bits	***Количество стоп-битов*** Доступные варианты: 1, 2.
.poll_delay	***Пауза перед отправкой ответа*** Время выдержки перед отправкой ответа. Требуется увеличить, если устройство Master в сети не успевает переключиться в режим приема.

Автоматический опрос модулей расширения

Модули расширения Alpha-X имеют интерфейс RS-485, протокол Modbus RTU. По умолчанию, с ними можно работать точно так, как и с любыми другими ведомыми Modbus-устройствами. Дополнительно, имеется функционал, который позволяет опрашивать входы и устанавливать выходы модулей расширения Alpha-X в автоматическом режиме. Это позволяет сократить количество строк кода, связанных с опросом модулей.

Также, для модулей из линейки Alpha-X имеются функциональные блоки настройки, считывания, сброса и сохранения параметров, позволяющие упростить конфигурацию оборудования из программы пользователя.

Для использования автоматического опроса необходимо:

1. Создать объекты модулей за пределами функции POU(). При создании можно использовать обобщенный либо специализированный класс модуля в соответствии с таблицей ниже. При создании экземпляра в конструктор передается Modbus-адрес модуля.

Класс	Тип модуля
`AlphaXModule`	Модуль ввода-вывода комбинированный (DAIO) или любой другой модуль
`AlphaX_DI`	Модуль ввода дискретный
`AlphaX_DO`	Модуль вывода дискретный
`AlphaX_DIO`	Модуль ввода-вывода дискретный
`AlphaX_AI`	Модуль ввода аналоговый

Ниже представлен пример создания автоматического опроса для 4 модулей расширения с Modbus-адресами: 1, 2, 3, 4.

```
AlphaX_DI module1(1);
AlphaX_DO module2(2);
AlphaX_AI module3(3);
AlphaXModule module4(4);

void POU1()
{
  ...
}
```

	Класс `AlphaXModule` может использоваться для опроса любых модулей расширения Alpha-X (DI, DO, AI, DAIO). Этот класс, в отличии от остальных, считывает абсолютно все оперативные параметры модулей Alpha-X. Как следствие, использование этого класса расходует больше оперативной памяти, а также немного увеличивает общий период опроса по Modbus.

2. Далее требуется создать объект класса `ModbusMasterPoll`. Этот класс отвечает за создание автоматических запросов, поэтому в конструктор необходимо передать объекты созданных модулей в качестве аргументов.

```
...

ModbusMasterPoll poll (
  module1,
  module2,
  module3,
  module4
);

void POU1()
{
  ...
}
```

3. Следующим шагом нужно создать и настроить объект `ModbusMaster`. Одним из его аргументов является список устройств `devices`. Здесь следует указать созданный на предыдущем шаге экземпляр класса `ModbusMasterPoll`.

```
...

ModbusMaster modbus_master ({
  .com        = COM1,          // Номер COM-порта
  .devices    = poll,       // Список устройств для авт.опроса
```

```
    .baudrate    = 115200,       // Скорость обмена, бит/с
    .parity      = COM_PARITY_NONE,  // Четность
    .stop_bits   = 1,            // Кол-во стоп-битов
    .poll_delay = 0,             // Пауза перед отправкой команды (мс)
    .response_timeout = 300,     // Время ожидания ответа (мс)
    .failed_attempts  = 3,       // Количество попыток связи
    .restore_timeout  = 3000     // Время восстановления опроса (мс)
});

void POU1()
{
  ...
}
```

На этом этапе автоматический опрос создан. Специальная системная задача ПЛК будет вести опрос модулей расширения параллельно выполнению технологической программы пользователя. Теперь остается только определить объекты, через которые будет происходить взаимодействие с дискретными и аналоговыми входами и выходами.

Для работы с дискретными входами и выходами на модулях расширения используются те же самые классы, что и для работы с встроенными DI и DO (Встроенные входы и выходы).

Пример создания массивов на 8 дискретных входов и 8 дискретных выходов для четвертого модуля:

```
DiscreteInputArray<8> di_mod4(module4);
DiscreteOutputArray<8> do_mod4(module4);

void POU1()
{
  ...
}
```

Аналоговые входы, классы AnalogInput и AnalogInputArray

Для работы с аналоговыми входами используются классы `AnalogInput` и `AnalogInputArray`. Как правило, использование класса `AnalogInputArray` более предпочтительно, так как позволяет работать с набором входов как с массивом.

В конструктор `AnalogInput` передается объект модуля (см. Автоматический опрос модулей расширения) и номер канала, нумерация начинается с 0. Пример создания отдельного аналогового входа:

```
AnalogInput ai(module3, 0);

void POU1()
{
  ...
}
```

Пример создания массива из 8 аналоговых входов показан ниже:

```
AnalogInputArray<8> ai(module3);

void POU1()
{
  ...
}
```

	Создание экземпляров классов дискретных входов внутри функции POU() допустимо только при использовании ключевого слова `static`!

	Одновременное создание нескольких экземпляров классов `AnalogInput` или `AnalogInputArray` для одних и тех же входов недопустимо!

Вне зависимости от выбранного класса, использование аналоговых входов в большинстве случаев ничем не отличается от использования стандартных переменных типа `float`.

Использование отдельного аналогового входа:

```
AnanlogInput ai(module3, 0);

void POU1()
{
  // Использование значения входа ai в операторе if
  if (ai > 15.5)
  {
    ...
  }

  // Присвоение значения ai другой переменной
  float myFloat = ai;
}
```

Использование аналогового входа в составе массива:

```
AnalogInputArray<8> ai(module3);

void POU1()
{
  // Использование значения входа в операторе if
  if (ai[0] > 15.5)
  {
    ...
  }

  // Присвоение значения входа другой переменной
  float myFloat = ai[0];
}
```

Альтернативно, значение аналогового входа можно явно получить с помощью метода `value()`.

```
AnalogInputArray<8> ai(module3);

void POU1()
{
  // Использование значения входа ai в операторе if
  if (ai[0].value() > 15.5)
  {
    ...
  }

  // Присвоение значения входа ai другой переменной
  float myFloat = ai[0].value();
}
```

Способ работы с аналоговыми входами выбирает пользователь в зависимости от своих предпочтений.

	Контроллер обновляет состояние входов единожды перед началом цикла выполнения планировщика pou. При этом значения входов не изменяются на протяжении выполнения всех pou в этом цикле.

Энергонезависимая память

Для сохранения значений в энергонезависимой памяти используется ключевое слово `retain`. В зависимости от модификации ПЛК, может быть разница в объёме доступной энергонезависимой памяти и способе её работы. Таблица соответствия приведена ниже.

Модификация ПЛК	Объём retain-памяти	Периодическое сохранение по времени	Сохранение по обрыву питания
Alpha-X CPU 01-1 00	4 кБ	+	-
Alpha-X CPU E1-0 00	4 кБ	+	+

Для Alpha-X CPU 01-1 00 сохранение в энергонезависимую память происходит периодически, период сохранения настраивается переменной `RetainConfig::save_time` в файле pou_manager.cpp. Период по умолчанию составляет 10 секунд. Если данные энергонезависимых переменных изменились до того, как произошло их сохранение, при этом ПЛК был обесточен, последние изменения будут утеряны.

Период сохранения можно изменить, при этом необходимо понимать, что это повлияет на общий срок службы памяти EEPROM. Для периода 10 секунд срок службы EEPROM памяти составляет не менее 5 лет. Количество циклов перезаписи EEPROM составляет ~15 миллионов. Количество циклов перезаписи, произошедших с момента производства ПЛК, можно узнать с помощью функции `GetNumberOfEepromWriteCycles()`.

Для Alpha-X CPU E1-0 00, помимо периодического сохранения, используется дополнительная аппаратная реализация, гарантирующая сохранение retain-памяти в любой момент при обрыве питания.

Ключевое слово retain

Для сохранения значений в энергонезависимой памяти используется ключевое слово `retain`. Это слово можно применять только с базовыми типами данных. Переменные, отмеченные данным ключевым словом, будут сохранять свои значения при отключении питания.

```
// Значение переменной сохраняется в энергонезависимой памяти
retain uint16_t value;
// Значение массива сохраняется в энергонезависимой памяти
retain float arr[10];
```

	Ключевое слово `retain` применимо только для базовых типов: целые числа, числа с плавающей точкой, символы, логические значения (включая массивы). Ключевое слово `retain` нельзя использовать для полей классов и структур.

Особенности обновления программы с retain-переменными

Энергонезависимые переменные хранятся во внешней EEPROM-памяти, поэтому при обновлении программы пользователя возможны два сценария:

1. Данные в EEPROM перезаписываются значениями, указанными переменным при инициализации. Этот вариант используется при отсутствии соответствующего аргумента `--save-retain` во время вызова загрузчика AXCodeLoader (по умолчанию);
2. При обновлении программы пользователя данные в EEPROM не изменяются, то есть вновь загруженная программа будет использовать данные, ранее сохраненные в энергонезависимой памяти. Этот вариант используется при наличии аргумента `--save-retain` во время вызова загрузчика AXCodeLoader;

	Использование аргумента `--save-retain` всегда связано с риском того, что в процессе компиляции физическая адресация переменных изменится. То есть при двух компиляциях одной и той же программы переменные могут ссылаться на разные физические адреса хранения. Это может привести к непредсказуемому поведению программы, поэтому по умолчанию рекомендуется использовать загрузчик без аргумента `--save-retain`.

	Аргументы AXCodeLoader настраиваются в файле task.json.

РАЗДЕЛ 4. ОПИСАНИЕ ОШИБОК, СОБЫТИЙ И СПЕЦИАЛЬНЫХ РЕГИСТРОВ

Карта адресов системных регистров Modbus RTU

Адрес		Описание	Права доступа
HEX	DEC		
FA00	64000	Номер модификации процессорного модуля	R
FA01	64001	Серийный номер процессорного модуля	R
FA02	64002	Major версия ядра (1.x.x)	R
FA03	64003	Minor версия ядра (x.1.x)	R
FA04	64004	Patch версия ядра (x.x.1)	R
FA05	64005	Состояние встроенных дискретных входов	R
FA06	64006	Состояние встроенных дискретных выходов	R
FA07	64007	Количество ошибок в логе ошибок	R
FA08	64008	Состояние ядра	R
		0 — ядро не запущено	
		1 — инициализация задач	
		2 — рабочий режим (RUN)	
		4 — остановка задач (STOP)	
		8 — Сервисный режим (SERVICE)	
		16 — Ошибка определения состояния ядра	

Адрес		Описание	Права доступа
HEX	DEC		
FA09	64009	Переключение состояния ядра	R/W
		0 - Нет ожидаемого действия	
		1 - Переключить в режим RUN	
		2 - Переключить в режим STOP	
		3 - Перезагрузить модуль	
		4 - Перезагрузиться в режим SERVICE	
		5 - Перезагрузиться в режим BOOT	
FA0A	64010	Инициализация retain-переменных значениями по умолчанию	R/W
FA0B	64011	Параметр "Секунды"	R/W
FA0C	64012	Параметр "Минуты"	R/W
FA0D	64013	Параметр "Часы"	R/W
FA0E	64014	Параметр "День месяца"	R/W
FA0F	64015	Параметр "Месяц"	R/W
FA10	64016	Параметр "Год"	R/W
FA11	64017	Параметр "День недели"	R
FBF4	64500	Код события номер 0	R
FBF5	64501	Номер строки в коде ядра	R
FBF6	64502	MSB от числа времени в секундах	R
FBF7	64503	LSB от числа времени в секундах	R
FBF8	64504	Код события номер 1	R
FBF9	64505	Номер строки в коде ядра	R

Адрес		Описание	Права доступа
HEX	DEC		
FBF6	64506	Unix-time времени события 1, младшее слово	R
FBFB	64507	Unix-time времени события 1, старшее слово	R
…	…	…	…
FD80	64896	Код события номер 99	R
FD81	64897	Номер строки в коде ядра	R
FD82	64898	Unix-time времени события 99, младшее слово	R
FD83	64899	Unix-time времени события 99, старшее слово	R

Список событий (EVENTS)

Модуль выводит в терминал события в формате <код> - <значение>, где код описывает общую информацию по событию, а значение - уточняющая информация.

10 последних во времени событий хранятся в батарейной памяти. Таким образом они не очищаются при сбросе питания (при условии наличия заряда в батарее).

Код	Описание	Комментарии
100	Событие запуска ядра	
101	Событие загрузки новой прошивки	
199	Отладочное событие	
200	События переключения состояния ядра	Значение после тире:
		0 — ядро не запущено
		1 — инициализация задач
		2 — рабочий режим
		4 — остановка задач
		8 — сервисный режим
		16 — ошибка определения состояния ядра
1012	Выход за размер COM-портов	
1013	COM-порт уже используется	
1014	Ошибка инициализации часов реального времени	
1020	Ошибка загрузчика	
1021	Ошибка считывания параметров связи при переходе в загрузчик	
1022	В загрузчике был зафиксирован Hard Fault	
1030	Общая ошибка задачи индикации ПЛК	
1031	Ошибка инициализации задачи индикации ПЛК	
1032	Ошибка создания задачи индикации ПЛК	
1040	Ошибка вывода сообщения об ошибке	
1041	Критическая ошибка (HardFault)	
1042	Необработанное прерывание	

Код	Описание	Комментарии
1050	Общая ошибка команд ПЛК	
1051	Ошибка команды перезагрузки ядра	
1052	Отладочная ошибка	
1200	Общая ошибка библиотеки Modbus	
1210	Общая ошибка Modbus Master	
1211	Неправильные входные параметры запроса Modbus Master	
1212	Неподдерживаемая функция Modbus Master	
1213	Вызов функции обработки запроса без активного запроса Modbus Master	
1214	Ошибка обработки запросов Modbus Master	
1215	Не удалось выбрать регистры для запроса Modbus Master	
1216	Не удалось записать посылку в буфер отправки	
1217	Неверный тип таблицы параметров в функции работы с Coils/Discrete Inputs	
1220	Общая ошибка Modbus Slave	
1230	Общая ошибка внутренних методов библиотеки Modbus	
1231	Не удалось заблокировать таблицу параметров	
1232	Неверный тип таблицы параметров для работы с Coils/Discrete Inputs	
1233	Буфер отправки посылки полон	
1234	Буфер приема посылки пуст	
1240	Общая ошибка функций Modbus	
1241	Не добавлен пользовательский обработчик функций	
1242	Ошибка пользовательского обработчика функций	Должна вызываться пользователем в реализации пользовательских функций в случае ошибок
1250	Общая ошибка опрашиваемых устройств	
1251	Ошибка инициализации опрашиваемых устройств	

Код	Описание	Комментарии
1252	Ошибка обработки входов/выходов устройств	
1253	Ошибка несовпадения модификации устройства и управляющего класса	
1300	Исключение библиотеки устройств	
1310	Общая ошибка канала опроса устройств	
1320	Общая ошибка дискретных входов/выходов библиотеки устройств	
1330	Общая ошибка аналоговых входов/выходов библиотеки устройств	
1400	Общая ошибка библиотеки общих утилит	
1410	Общая ошибка мьютекса	
1411	Разблокировка не заблокированного мьютекса	
1420	Общая ошибка журнала событий	
1421	Ошибка инициализации журнала событий	
1422	Выход за размер журнала событий	
1500	Ошибка библиотеки терминального сервера	
1501	Попытка обработки пустой посылки	
1600	Исключение библиотеки времени	
1601	Тип таблицы не совпадает с таблицей параметров времени	
1700	Общая ошибка библиотеки меню	
1701	Выход за размер меню	
1702	Нулевой указатель в функция работы по указателю	
1703	Не найдены совпадающие переменные в таблицах параметров	
1704	Попытка копировать себя	
1705	Не совпадает тип копируемой таблицы параметров	
1706	Таблица параметров отсутствует в дереве меню	
1707	Выход за размер дерева меню	
1708	Ошибка расчета адресов дерева меню	

Код	Описание	Комментарии
1709	Пересечение адресов в дереве меню	
1800	Общая ошибка библиотеки filesystem	
2000	Общая ошибка библиотеки kernel	
2001	Общая ошибка создания задачи	Может временно использоваться для задач, в которых еще не задана ошибка создания задачи
2010	Общая ошибка операционной системы	
2011	Ошибка по причине Stack Overflow в ОС	Должна передавать имя файла названия задачи, вызвавшей Stack Overflow
2020	Общая ошибка таймеров операционной системы	
2021	Ошибка создания таймера	
2022	Не задан callback для таймера	
2030	Общая ошибка контейнера ресурсов	
2031	Выход за размеры контейнера ресурсов	
2032	Ресурс не найден в контейнере ресурсов	
2040	Общая ошибка диспетчера задач	
2041	Нет места для создания новой задачи в диспетчере задач	
2042	Ошибка выделения памяти под задачу в диспетчере задач	
2043	Ошибка создания задачи в диспетчере задач	
2044	Параметры задачи не найдены в контейнере диспетчера задач	
2045	Установка флага инициализации задачи без неинициализированных задач	
2050	Общая ошибка состояния ядра	
2051	Попытка перехода в неправильное состояние ядра	
2060	Общая ошибка ресурса часов	
2061	Ошибка создания ресурса часов реального времени	
2070	Общая ошибка ресурса ресурсе SerialInterface	
2071	Ресурс SerialInterface не создан	

Код	Описание	Комментарии
2072	Не задана функция приема или отправки в ресурсе SerialInterface	
2073	Ресурс SerialInterface не найден в контейнере	
2074	Ошибка чтения настроек драйвера в ресурсе SerialInterface	
2075	Ошибка получения доступа к DMA в ресурсе SerialInterface	
2080	Общая ошибка задачи отладки	
2081	Неправильный тип сообщения отладки задачи отладки	
2082	Переполнение буфера отправки отладочных сообщений	
2083	Ошибка инициализации задачи отладки	
2084	Ошибка отправки посылки задачи отладки	
2085	Ошибка приема посылки задачи отладки	
2086	Неизвестное состояние задачи отладки	
2087	Ошибка создания задачи отладки	
2090	Общая ошибка задачи дискретных входов/выходов	
2091	Ошибка создания задачи дискретных входов/выходов	
2100	Общая ошибка задачи EEPROM	
2101	Ошибка инициализации задачи EEPROM	
2102	Ошибка чтения переменной задачи EEPROM	
2103	Ошибка записи переменной задачи EEPROM	
2104	Ошибка параметров запроса задачи EEPROM	
2105	Ошибка отправки ответа на запрос задачи EEPROM	
2106	Ошибка создания задачи EEPROM	
2110	Общая ошибка задачи Modbus	
2111	Ошибка инициализации задачи Modbus	
2112	Ошибка отправки посылки Modbus	

Код	Описание	Комментарии
2113	Ошибка приема посылки Modbus	
2114	Неизвестное состояние задачи Modbus	
2115	Ошибка обработки запроса Modbus Master	
2116	Ошибка отправки ответа на запрос Modbus Master	
2117	Ошибка создания задачи Modbus	
2120	Общая ошибка задачи прямого доступа к порту	
2121	Ошибка отправки посылки задачи прямого доступа к порту	
2122	Ошибка запуска посылки задачи прямого доступа к порту	
2123	Ошибка инициализации задачи прямого доступа к порту	
2124	Ошибка обработки запроса прямого доступа к порту	
2125	Ошибка отправки ответа на запрос задачи прямого доступа к порту	
2126	Ошибка создания задачи прямого доступа к порту	
2130	Общая ошибка API часов реального времени	
2140	Общая ошибка событий ядра	
2141	Ошибка инициализации событий ядра	
2142	События ядра не созданы	
2143	Недопустимое значение битовой маски события	Возможные события: Битовая маска не равна нулю. Битовая маска выходит за размер. Битовая маска содержит более одного флага для класса
2150	Общая ошибка очереди ядра	
2151	Ошибка инициализации очереди ядра	
2152	Очередь не создана в операционной системе	
2153	Попытка чтения/записи очереди с нулевым указателем на переменную	
2160	Общая ошибка запроса ядра	
2161	Ошибка инициализации запроса ядра	

Код	Описание	Комментарии
2162	Объекты запроса ядра не созданы	
2163	Неправильное состояние метода реализации обработки запроса	
2164	Вызвана обработка завершенного запроса	
3000	Исключение библиотеки обобщенного МК	
3001	Исключение отсутствия реализации функции	
3010	Ошибка шаблона типового драйвера интерфейса связи	
3011	Нет реализации действия с заданными аргументами в драйвере интерфейса связи	Означает, что выбранное действие не может быть выполнено(например, в реализации нет нужного прерывания)
3012	Чтение пустого буфера передачи в шаблоне типового драйвера интерфейса связи	
3013	Чтение буфера приема без полученной посылки в шаблоне типового драйвера интерфейса связи	
3014	Ошибка отправки посылки в шаблоне типового драйвера интерфейса связи	
3015	Ошибка приема посылки в шаблоне типового драйвера интерфейса связи	
3020	Общая ошибка драйвера BKP	
3021	Ошибка инициализации драйвера BKP	
3022	Нет реализации интерфейса драйвера BKP	
3023	Нулевой размер буфера для чтения/записи	
3024	Выход за пределы размера BKP-памяти	
3030	Ошибка драйвера DMA	
3031	Нет реализации в драйвере DMA	
3032	Необработанное прерывание DMA	
3033	Отсутствие Callback для DMA	
3040	Ошибка драйвера EEPROM	
3041	Нет реализации в драйвере EEPROM	

Код	Описание	Комментарии
3042	Неверный адрес EEPROM	
3043	Ошибка записи в EEPROM	
3044	Ошибка чтения из EEPROM	
3050	Ошибка драйвера GPIO	
3051	Нет реализации в драйвере GPIO	
3052	Неверный адрес вывода GPIO	
3060	Ошибка драйвера I2C	
3061	Нет реализации в драйвере I2C	
3062	Ошибка отправки сообщения в драйвере I2C	
3063	Ошибка приема сообщения в драйвере I2C	
3064	Ошибка отправки START и STOP	
3065	Устройство не отправило ACK	
3070	Ошибка драйвера RCC	
3071	Нет реализации в драйвере RCC	
3072	Отказ основного источника тактирования	
3073	Неправильный адрес периферии	
3080	Ошибка драйвера RTC	
3081	Ошибка инициализации RTC	
3082	Нет реализации в драйвера RTC	
3083	Переполнение счетчика времени RTC	
3090	Ошибка драйвера SPI	
3091	Нет реализации в драйвере SPI	
3092	Ошибка отправки посылки по SPI	
3093	Ошибка приема посылки по SPI	
3100	Ошибка драйвера TIM	
3101	Нет реализации в драйвере TIM	
3102	Необработанное прерывание драйвера TIM	
3110	Общая ошибка драйвера UART	
3111	Нет реализации в драйвере UART	
3112	Ошибка отправки посылки в драйвере UART	

Код	Описание	Комментарии
3113	Ошибка приема посылка в драйвере UART	
3120	Ошибка драйвера USB CDC	
3121	Нет реализации в драйвере USB CDC	
3130	Общая ошибка драйвера IWDG	
3131	Нет реализации в драйвере IWDG	
3132	Произошло срабатывание аппаратного WatchDog	
3140	Общая ошибка драйвера SysTick	
3141	Нет реализации в драйвере SysTick	
3150	Общая ошибка драйвера NVIC	
3151	Нет реализации в драйвере NVIC	
4000	Общая ошибка библиотеки plc	
4010	Общая ошибка задачи планировщика POU	
4011	Ошибка инициализации задачи планировщика POU	
4012	Ошибка создания задачи планировщика POU	
4013	Нет места для создания нового POU в задаче планировщика POU	
4014	POU не найдено в задаче планировщика POU	
4015	Ошибка Watchdog задачи планировщика POU	
4016	Ошибка пользовательского Callback Watchdog задачи планировщика POU	
4017	Срабатывание WatchDog планировщика POU	
4018	Планировщик POU принудительно остановлен	
4020	Общая ошибка журнала событий	
4021	Неверный код пользовательского события	
4022	Выход за размеры журнала событий	
4030	Общая ошибка функциональных блоков	

Код	Описание	Комментарии
4031	Недопустимый переход в конечном автомате функционального блока	
4032	Неизвестное состояние функционального блока	
4040	Общая ошибка функционального блока отладки	
4041	Неверные входные параметры функционального блока отладки	
4042	Ошибка создания ФБ отладки	ФБ был создан локально (например, как static)
4050	Общая ошибка функциональных блоков Modbus	
4051	Неверные входные параметры функционального блока Modbus	
4052	Отправлено слишком много запросов функциональных блоков	
4053	Выход за размер таблицы регистров ModbusBuffer	
4054	Неверные параметры чтения/записи битовой маски Coils/Discrete Inputs	
4055	Ошибка создания ФБ Modbus	ФБ был создан локально (например, как static)
4056	Буфер Modbus заблокирован слишком долго	Одна из системных задач ядра, использующих буфер, выполняется слишком долго
4060	Общая ошибка функционального блока прямого доступа к порту	
4061	Неверные входные параметры функционального блока прямого доступа к порту	
4062	Слишком много запросов функциональных блоков прямого доступа к порту	
4063	Ошибка создания ФБ прямого доступа к порту	ФБ был создан локально (например, как static)
4070	Общая ошибка retain	
4071	Ошибка инициализации retain	
4072	Неправильный период сохранения retain	
4080	Общая ошибка дискретных входов/выходов	

Код	Описание	Комментарии
4081	Ошибка инициализации дискретных входов/выходов	Возможные причины: Вход/выход создается после запуска ядра. Вход/выход с указанным устройством и номером уже создан
4082	Дискретный вход/выход привязан к несуществующему каналу входа/выхода в устройстве	
4090	Общая ошибка аналоговых входов/выходов	
4091	Ошибка инициализации аналоговых входов/выходов	
4092	Аналоговый вход/выход привязан к несуществующему каналу входа/выхода в устройстве	
4093	Ошибка определения типа аналогового входа/выхода	
4100	Общая ошибка ФБ работы с устройствами	
4101	Ошибка инициализации ФБ работы с устройствами	
4102	Неправильные входные параметры ФБ работы с устройствами	
4110	Общая ошибка ФБ файловой системы	
4111	Ошибка инициализации ФБ файловой системы	
4112	Неправильные входные параметры ФБ файловой системы	
4113	Слишком много запросов к задаче ФБ файловой системы	Возникает, если не удалось отправить запрос в файловую систему
4114	Строка пути к файлу/папке не оканчивается терминатором строки	

Список ошибок (ERRORS)

Модуль выводит в терминал ошибки в формате <код> - <значение>, где код описывают ошибку, а значение является отладочной информацией для разработчиков. Для обычного пользователя достаточно расшифровать код.

Код ошибки	Описание	Комментарии
p0	Общая ошибка библиотеки plc	
pS0	Общая ошибка задачи планировщика POU	
pS1	Ошибка инициализации задачи планировщика POU	
pS2	Ошибка создания задачи планировщика POU	
pS3	Нет места для создания нового POU в задаче планировщика POU	
pS4	POU не найдено в задаче планировщика POU	
pS5	Ошибка Watchdog задачи планировщика POU	
pS6	Ошибка пользовательского Callback Watchdog задачи планировщика POU	
pS7	Срабатывание WatchDog планировщика POU	
pS8	Планировщик POU принудительно остановлен	
pE0	Общая ошибка журнала событий	
pE1	Неверный код пользовательского события	
pE2	Выход за размеры журнала событий	
pF0	Общая ошибка функциональных блоков	
pF1	Недопустимый переход в конечном автомате функционального блока	
pF2	Неизвестное состояние функционального блока	
pFD0	Общая ошибка функционального блока отладки	
pFD1	Неверные входные параметры функционального блока отладки	
pFD2	Ошибка создания ФБ отладки	ФБ был создан локально (например, как static)
pFM0	Общая ошибка функциональных блоков Modbus	
pFM1	Неверные входные параметры функционального блока Modbus	

Код ошибки	Описание	Комментарии
pFM2	Отправлено слишком много запросов функциональных блоков	
pFM3	Выход за размер таблицы регистров ModbusBuffer	
pFM4	Неверные параметры чтения/записи битовой маски Coils/Discrete Inputs	
pFM5	Ошибка создания ФБ Modbus	ФБ был создан локально (например, как static)
pFM6	Буфер Modbus заблокирован слишком долго	Одна из системных задач ядра, использующих буфер, выполняется слишком долго
pFM7	Не удалось преобразовать номер ошибки Modbus в ошибку функционального блока	
pFS0	Общая ошибка функционального блока прямого доступа к порту	
pFS1	Неверные входные параметры функционального блока прямого доступа к порту	
pFS2	Слишком много запросов функциональных блоков прямого доступа к порту	
pFS3	Ошибка создания ФБ прямого доступа к порту	ФБ был создан локально (например, как static)
pR0	Общая ошибка retain	
pR1	Ошибка инициализации retain	
pR2	Неправильный период сохранения retain	
pDIO0	Общая ошибка дискретных входов/выходов	
pDIO1	Ошибка инициализации дискретных входов/выходов	Возможные причины: Вход/выход создается после запуска ядра. Вход/выход с указанным устройством и номером уже создан
pDIO2	Дискретный вход/выход привязан к несуществующему каналу входа/выхода в устройстве	
pAIO0	Общая ошибка аналоговых входов/выходов	

Код ошибки	Описание	Комментарии
pAIO1	Ошибка инициализации аналоговых входов/выходов	
pAIO2	Аналоговый вход/выход привязан к несуществующему каналу входа/выхода в устройстве	
pAIO3	Ошибка определения типа аналогового входа/выхода	
pFDV0	Общая ошибка ФБ работы с устройствами	
pFDV1	Ошибка инициализации ФБ работы с устройствами	
pFDV2	Неправильные входные параметры ФБ работы с устройствами	
pFFS0	Общая ошибка ФБ файловой системы	
pFFS1	Ошибка инициализации ФБ файловой системы	
pFFS2	Неправильные входные параметры ФБ файловой системы	
pFFS3	Слишком много запросов к задаче ФБ файловой системы	Возникает, если не удалось отправить запрос в файловую систему
pFFS4	Строка пути к файлу/папке не оканчивается терминатором строки	
e0	Общая ошибка библиотеки общих утилит	
eA0	Общая ошибка мьютекса	
eA1	Разблокировка не заблокированного мьютекса	
eL0	Общая ошибка журнала событий	
eL1	Ошибка инициализации журнала событий	
eL2	Выход за размер журнала событий	
d0	Исключение библиотеки устройств	
dP0	Общая ошибка канала опроса устройств	
dD0	Общая ошибка дискретных входов/выходов библиотеки устройств	
dA0	Общая ошибка аналоговых входов/выходов библиотеки устройств	
mbo	Общая ошибка библиотеки Modbus	

Код ошибки	Описание	Комментарии
mbM0	Общая ошибка Modbus Master	
mbM1	Неправильные входные параметры запроса Modbus Master	
mbM2	Неподдерживаемая функция Modbus Master	
mbM3	Вызов функции обработки запроса без активного запроса Modbus Master	
mbM4	Ошибка обработки запросов Modbus Master	
mbM5	Не удалось выбрать регистры для запроса Modbus Master	
mbM6	Не удалось записать посылку в буфер отправки	
mbM7	Неверный тип таблицы параметров в функции работы с Coils/Discrete Inputs	
mbS0	Общая ошибка Modbus Slave	
mbP0	Общая ошибка внутренних методов библиотеки Modbus	
mbP1	Не удалось заблокировать таблицу параметров	
mbP2	Неверный тип таблицы параметров для работы с Coils/Discrete Inputs	
mbP3	Буфер отправки посылки полон	
mbP4	Буфер приема посылки пуст	
mbE0	Общая ошибка функций Modbus	
mbE1	Не добавлен пользовательский обработчик функций	
mbE2	Ошибка пользовательского обработчика функций	Должна вызываться пользователем в реализации пользовательских функций в случае ошибок
mbD0	Общая ошибка опрашиваемых устройств	
mbD1	Ошибка инициализации опрашиваемых устройств	
mbD2	Ошибка обработки входов/выходов устройств	
mbD3	Ошибка несовпадения модификации устройства и управляющего класса	

Код ошибки	Описание	Комментарии
ts0	Ошибка библиотеки терминального сервера	
ts1	Попытка обработки пустой посылки	
c0	Общая ошибка ПЛК	
c1	Ошибка инициализации ПЛК	
cS0	Общая ошибка системной задачи ПЛК	
cS1	Ошибка инициализации системной задачи ПЛК	
cS2	Выход за размер COM-портов	
cS3	COM-порт уже используется	
cS4	Ошибка инициализации часов реального времени	
cB0	Ошибка загрузчика	
cB1	Ошибка считывания параметров связи при переходе в загрузчик	
cB2	В загрузчике был зафиксирован Hard Fault	
cI0	Общая ошибка задачи индикации ПЛК	
cI1	Ошибка инициализации задачи индикации ПЛК	
cI2	Ошибка создания задачи индикации ПЛК	
cE0	Ошибка вывода сообщения об ошибке	
cE1	Критическая ошибка (HardFault)	
cE2	Необработанное прерывание	
cC0	Общая ошибка команд ПЛК	
cC1	Ошибка команды перезагрузки ядра	
cC2	Отладочная ошибка	
tm0	Исключение библиотеки времени	
tm1	Тип таблицы не совпадает с таблицей параметров времени	
k0	Общая ошибка библиотеки kernel	
k1	Общая ошибка создания задачи	Может временно использоваться для задач, в которых еще не задана ошибка создания задачи
kO0	Общая ошибка операционной системы	

Код ошибки	Описание	Комментарии
kO1	Ошибка по причине Stack Overflow в ОС	Должна передавать как имя файла названия задачи, вызвавшей Stack Overflow
kOT0	Общая ошибка таймеров операционной системы	
kOT1	Ошибка создания таймера	
kOT2	Не задан callback для таймера	
kR0	Общая ошибка контейнера ресурсов	
kR1	Выход за размеры контейнера ресурсов	
kR2	Ресурс не найден в контейнере ресурсов	
kT0	Общая ошибка диспетчера задач	
kT1	Нет места для создания новой задачи в диспетчере задач	
kT2	Ошибка выделения памяти под задачу в диспетчере задач	
kT3	Ошибка создания задачи в диспетчере задач	
kT4	Параметры задачи не найдены в контейнере диспетчера задач	
kT5	Установка флага инициализации задачи без неинициализированных задач	
kS0	Общая ошибка состояния ядра	
kS1	Попытка перехода в неправильное состояние ядра	
kRR0	Общая ошибка ресурса часов	
kRR1	Ошибка создания ресурса часов реального времени	
kRS0	Общая ошибка ресурса ресурсе SerialInterface	
kRS1	Ресурс SerialInterface не создан	
kRS2	Не задана функция приема или отправки в ресурсе SerialInterface	
kRS3	Ресурс SerialInterface не найден в контейнере	
kRS4	Ошибка чтения настроек драйвера в ресурсе SerialInterface	

Код ошибки	Описание	Комментарии
kRS5	Ошибка получения доступа к DMA в ресурсе SerialInterface	
kDTB0	Общая ошибка задачи отладки	
kDTB1	Неправильный тип сообщения отладки задачи отладки	
kDTB2	Переполнение буфера отправки отладочных сообщений	
kDTB3	Ошибка инициализации задачи отладки	
kDTB4	Ошибка отправки посылки задачи отладки	
kDTB5	Ошибка приема посылки задачи отладки	
kDTB6	Неизвестное состояние задачи отладки	
kDTB7	Ошибка создания задачи отладки	
kTDI0	Общая ошибка задачи дискретных входов/выходов	
kTDI1	Ошибка создания задачи дискретных входов/выходов	
kTE0	Общая ошибка задачи EEPROM	
kTE1	Ошибка инициализации задачи EEPROM	
kTE2	Ошибка чтения переменной задачи EEPROM	
kTE3	Ошибка записи переменной задачи EEPROM	
kTE4	Ошибка параметров запроса задачи EEPROM	
kTE5	Ошибка отправки ответа на запрос задачи EEPROM	
kTE6	Ошибка создания задачи EEPROM	
kTMB0	Общая ошибка задачи Modbus	
kTMB1	Ошибка инициализации задачи Modbus	
kTMB2	Ошибка отправки посылки Modbus	
kTMB3	Ошибка приема посылки Modbus	
kTMB4	Неизвестное состояние задачи Modbus	
kTMB5	Ошибка обработки запроса Modbus Master	
kTMB6	Ошибка отправки ответа на запрос Modbus Master	

Код ошибки	Описание	Комментарии
kTMB7	Ошибка создания задачи Modbus	
kTS0	Общая ошибка задачи прямого доступа к порту	
kTS1	Ошибка отправки посылки задачи прямого доступа к порту	
kTS2	Ошибка запуска посылки задачи прямого доступа к порту	
kTS3	Ошибка инициализации задачи прямого доступа к порту	
kTS4	Ошибка обработки запроса прямого доступа к порту	
kTS5	Ошибка отправки ответа на запрос задачи прямого доступа к порту	
kTS6	Ошибка создания задачи прямого доступа к порту	
kAR0	Общая ошибка API часов реального времени	
kOE0	Общая ошибка событий ядра	
kOE1	Ошибка инициализации событий ядра	
kOE2	События ядра не созданы	
kOE3	Недопустимое значение битовой маски события	Возможные события: Битовая маска не равна нулю. Битовая маска выходит за размер. Битовая маска содержит более одного флага для класса
kOQ0	Общая ошибка очереди ядра	
kOQ1	Ошибка инициализации очереди ядра	
kOQ2	Очередь не создана в операционной системе	
kOQ3	Попытка чтения/записи очереди с нулевым указателем на переменную	
kOR0	Общая ошибка запроса ядра	
kOR1	Ошибка инициализации запроса ядра	
kOR2	Объекты запроса ядра не созданы	
kOR3	Неправильное состояние метода реализации обработки запроса	

Код ошибки	Описание	Комментарии
kOR4	Вызвана обработка завершенного запроса	
mco	Исключение библиотеки обобщенного МК	
mc1	Исключение отсутствия реализации функции	
mcC0	Ошибка шаблона типового драйвера интерфейса связи	
mcC1	Нет реализации действия с заданными аргументами в драйвере интерфейса связи	Означает, что заполненное действие не может быть выполнено (например, в реализации нет нужного прерывания)
mcC2	Чтение пустого буфера передачи в шаблоне типового драйвера интерфейса связи	
mcC3	Чтение буфера приема без полученной посылки в шаблоне типового драйвера интерфейса связи	
mcC4	Ошибка отправки посылки в шаблоне типового драйвера интерфейса связи	
mcC5	Ошибка приема посылки в шаблоне типового драйвера интерфейса связи	
mcB0	Общая ошибка драйвера BKP	
mcB1	Ошибка инициализации драйвера BKP	
mcB2	Нет реализации интерфейса драйвера BKP	
mcB3	Нулевой размер буфера для чтения/записи	
mcB4	Выход за пределы размера BKP-памяти	
mcD0	Ошибка драйвера DMA	
mcD1	Нет реализации в драйвере DMA	
mcD2	Необработанное прерывание DMA	
mcD3	Отсутствие Callback для DMA	
mcE0	Ошибка драйвера EEPROM	
mcE1	Нет реализации в драйвере EEPROM	
mcE2	Неверный адрес EEPROM	
mcE3	Ошибка записи в EEPROM	

Код ошибки	Описание	Комментарии
mcE4	Ошибка чтения из EEPROM	
mcG0	Ошибка драйвера GPIO	
mcG1	Нет реализации в драйвере GPIO	
mcG2	Неверный адрес вывода GPIO	
mcIC0	Ошибка драйвера I2C	
mcIC1	Нет реализации в драйвере I2C	
mcIC2	Ошибка отправки сообщения в драйвере I2C	
mcIC3	Ошибка приема сообщения в драйвере I2C	
mcIC4	Ошибка отправки START и STOP	
mcIC5	Устройство не отправило ACK	
mcRC0	Ошибка драйвера RCC	
mcRC1	Нет реализации в драйвере RCC	
mcRC2	Отказ основного источника тактирования	
mcRC3	Неправильный адрес периферии	
mcRT0	Ошибка драйвера RTC	
mcRT1	Ошибка инициализации RTC	
mcRT2	Нет реализации в драйвера RTC	
mcRT3	Переполнение счетчика времени RTC	
mcS0	Ошибка драйвера SPI	
mcS1	Нет реализации в драйвере SPI	
mcS2	Ошибка отправки посылки по SPI	
mcS3	Ошибка приема посылки по SPI	
mcT0	Ошибка драйвера TIM	
mcT1	Нет реализации в драйвере TIM	
mcT2	Необработанное прерывание драйвера TIM	
mcUA0	Общая ошибка драйвера UART	
mcUA1	Нет реализации в драйвере UART	
mcUA2	Ошибка отправки посылки в драйвере UART	
mcUA3	Ошибка приема посылка в драйвере UART	

Код ошибки	Описание	Комментарии
mcUS0	Ошибка драйвера USB CDC	
mcUS1	Нет реализации в драйвере USB CDC	
mcIW0	Общая ошибка драйвера IWDG	
mcIW1	Нет реализации в драйвере IWDG	
mcIW2	Произошло срабатывание аппаратного WatchDog	
mcST0	Общая ошибка драйвера SysTick	
mcST1	Нет реализации в драйвере SysTick	
mcFL0	Ошибка драйвера FLASH	
mcFL1	Нет реализации в драйвере FLASH	
mcFL2	Неверный адрес FLASH	
mcFL3	Ошибка записи в FLASH	
mcFL4	Ошибка чтения из FLASH	
mcFL5	Ошибка стирания FLASH	
mcN0	Общая ошибка драйвера NVIC	
mcN1	Нет реализации в драйвере NVIC	
fs0	Общая ошибка библиотеки filesystem	
mn0	Общая ошибка библиотеки меню	
mn1	Выход за размер меню	
mn2	Нулевой указатель в функция работы по указателю	
mn3	Не найдены совпадающие переменные в таблицах параметров	
mn4	Попытка копировать себя	
mn5	Не совпадает тип копируемой таблицы параметров	
mn6	Таблица параметров отсутствует в дереве меню	
mn7	Выход за размер дерева меню	
mn8	Ошибка расчета адресов дерева меню	
mn9	Пересечение адресов в дереве меню	