CODESYS V3.5

Архивация

Руководство пользователя

06.06.2022
версия 3.0

Оглавление

Глоссарий

ПЛК – программируемый логический контроллер.
ФБ – функциональный блок.

1 Цель и структура документа

Одной из типичных задач автоматизированных систем управления является архивирование данных о технологическом процессе для последующей обработки и анализа (например, для анализа причин аварийных ситуаций и оптимизации режима работы оборудования). В крупных распределенных системах управления эта задача обычно решается на верхнем уровне АСУ – с помощью SCADA-системы, интегрированной с базой данных. В то же время, в локальных системах управления верхний уровень может попросту отсутствовать – поэтому задача архивации ложится на устройства среднего уровня, в большинстве случаев – на программируемые контроллеры.

Контроллеры ОВЕН, программируемые в среде **CODESYS V3.5**, способны архивировать данные во внутреннюю память или на внешний носитель (USB- или SD-накопитель) и считывать данные (например, файлы рецептов, технологические карты и т. д.). Для этого могут использоваться компонент **OwenArchiver** и библиотека **CAA File**, описанные в настоящем руководстве.

Особенности компонента **OwenArchiver**:

- рассчитан на начинающих пользователей, не требует навыков программирования;
- настройка через дерево проекта в несколько кликов;
- жестко заданная структура архива и условия архивации;
- поддерживает только запись архивов (чтение не поддерживается).

Особенности библиотеки **CAA File**:

- рассчитана на продвинутых пользователей;
- требует хороших навыков программирования;
- предоставляет доступ к низкоуровневым ФБ работы с файлами и каталогами, позволяя решить практически любые задачи;
- поддерживает запись, чтение, переименование, копирование и другие операции с файлами;
- поддерживает создание, удаление, копирование и другие операции с каталогами.

В п. 2 приведена основная информация о работе с файлами.
В п. 3 приведено описание компонента **OwenArchiver.**
В п. 4 приведено описание библиотеки **CAA File.**
В п. 5 рассмотрен пример использования библиотеки.
В п. 6 рассмотрены дополнительные вопросы, оставшиеся за пределами примера.

ПРИМЕЧАНИЕ
Разработка ПО для работы с файлами подразумевает высокую квалификацию программиста, а также хорошее знание среды **CODESYS V3.5** и языка ST. Реализация блоков архивации на графических языках (например, CFC) является крайне затруднительной из-за сложности алгоритмов. В программах, написанных на графических языках, можно вызывать готовые блоки, реализованные на языке ST. Документ рекомендуется читать последовательно.

2 Основные сведения о работе с файлами

2.1 Общие сведения о памяти контроллеров

Файл – это именованная область памяти, используемая для хранения данных. Для упрощения работы с файлами используются каталоги, которые позволяют разделять файлы по группам.

Способ организации, хранения и именования файлов на конкретном устройстве зависит от его файловой системы. Пользовательская[1] файловая система контроллеров ОВЕН ПЛК2xx – UBIFS, контроллеров ОВЕН СПК1xx – ext4.

У контроллеров ОВЕН имеется три физически разных области памяти:

- энергонезависимая память (Flash);
- оперативная память (RAM);
- retain-память (MRAM).

Работа с файлами в большинстве случаев подразумевает работу с flash-памятью. Flash-память имеет значительный, но, тем не менее, ограниченный ресурс перезаписи – поэтому для архивации данных в большинстве случаев рекомендуется использовать внешние накопители (USB, SD). Ресурс перезаписи внешних накопителей также ограничен, но их выход из строя не повлияет на работоспособность контроллера и, кроме того, накопители можно оперативно заменить. Информация об общем доступном объеме памяти приведена в руководстве по эксплуатации на соответствующий контроллер. Информация о количестве свободной/занятой памяти доступна в web-конфигураторе и проекте CODESYS (узел Drives в дереве проекта).

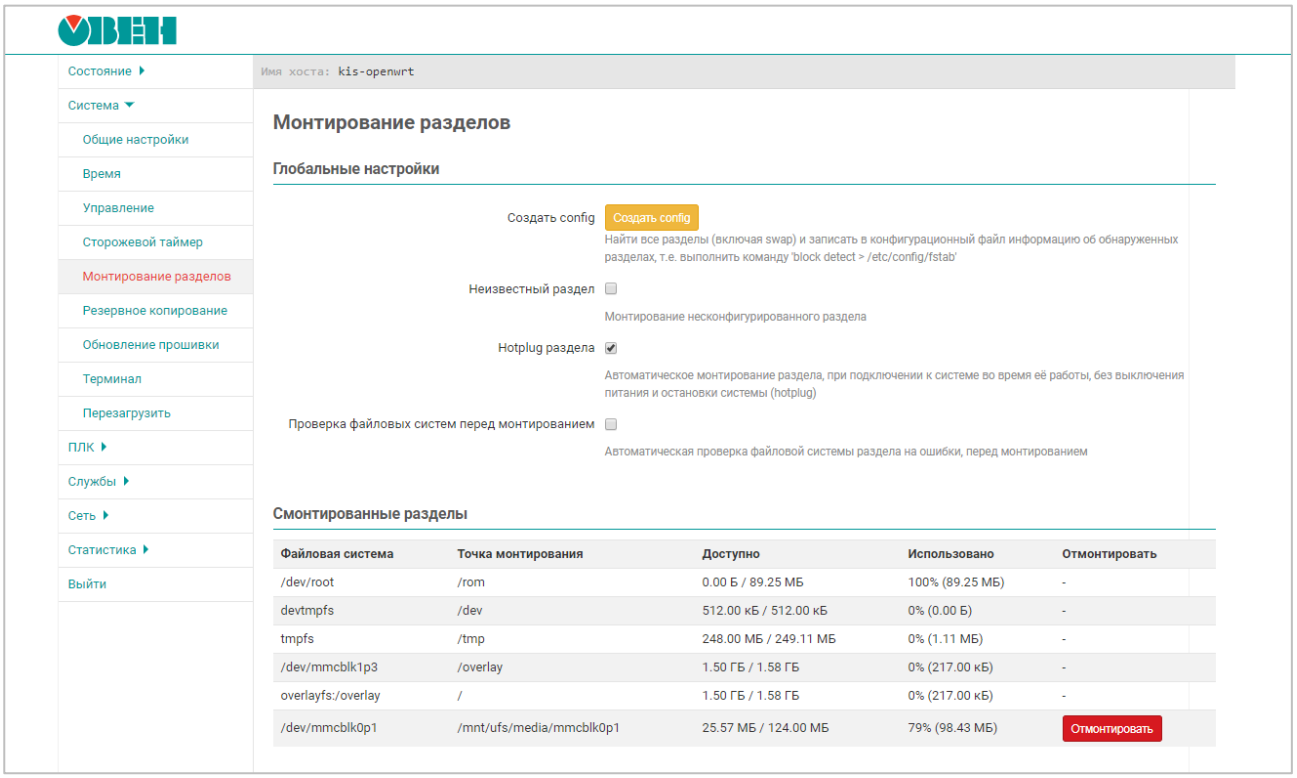


Рисунок 2.1.1 – Информация о памяти контроллера и накопителей в web-конфигураторе

[1] Вообще, память контроллера представлена несколькими файловыми системами (например, для хранения бэкапа прошивки используется squashfs), но с точки зрения разработчика прикладного ПО – вся работа происходит исключительно с пользовательской файловой системой.

2.2 Операции с файлами

Во время работы с файлами используются четыре основные операции:

- открытие файла (если файла не существует – то эта операция создает его);
- чтение из файла;
- запись в файл;
- закрытие файла.

В случае успешного открытия файла создается дескриптор (**handle**), который является идентификатором конкретного файла и используется для всех остальных операций с ним.

Таким образом, схема работы с файлами в упрощенном виде выглядит следующим образом:

Рисунок 2.2.1 – Упрощенная схема работы с файлами

В подавляющем большинстве случаев работа с файлами производится с помощью единичных операций – т. е. файл открывается только на то время, которое нужно, чтобы считать/записать в него требуемые в текущий момент данные. Постоянно держать файл открытым не рекомендуется – в частности, из-за ограничения на максимальное число одновременно открытых файлов и возможном повреждении открытого файла в случае перезагрузки контроллера по питанию.

Библиотека **CAA File** реализует асинхронный доступ к файлам – в связи с этим выполнение блоков может занять несколько циклов задачи контроллера, но остальные задачи (визуализация, обмен и т. д.) в течение этого времени будут продолжать выполняться в штатном режиме. В большинстве случаев удобно реализовать каждую отдельную операцию с файлом (открытие, чтение, запись, закрытие и т. д.) в отдельном шаге оператора **CASE**.

2.3 Требования к подключаемым накопителям (USB/SD)

1. Поддерживаемый стиль разделов – MBR (GPT не поддерживается). Методика определения стиля разделов доступна по ссылке.
2. Рекомендуется использовать накопители с одним разделом – тогда гарантируется монтирование по путям, указанным в п. 2.4.
3. Поддерживаемые файловые системы накопителей – FAT16/FAT32 и ext4. Обновление прошивки/проекта возможно только при использовании накопителя с файловой системой FAT16/FAT32.
4. Перед началом работы рекомендуется отформатировать накопитель с помощью утилиты **HP USB Disk Storage Format Tool**.

2.4 Пути к файлам и накопителям. Монтирование и демонтирование. Узел Drives

Во время работы с файлами необходимо знать пути, по которым они расположены.

Контроллеры ОВЕН, программируемые в **CODESYS V3.5**, работают под управлением ОС Linux.

Пути к каталогам CODESYS и пути монтирования накопителей выглядят следующим образом:

Таблица 2.4.1 – Пути к каталогам CODESYS и пути монтирования накопителей

Директория	СПК1xx [M01]	ПЛК2xx	Заместитель
Рабочая директория	/home/root/CODESYS_WRK/PlcLogic	/root/CODESYS/PlcLogic	$$PlcLogic$$[2]
USB-накопитель	/mnt/ufs/media/sda1		$$USB$$
SD-накопитель	/mnt/ufs/media/mmcblk0p1		$$SD$$
Директория FTP-сервера	/mnt/ufs/home/ftp/in		$$FTP$$
Директория файлов визуализации и web-сервера визуализации	рабочая директория/visu		$$visu$$
Директория файлов трендов	рабочая директория/trend[3]		$$trend$$
Директория файлов тревог	рабочая директория/alarms[3]		$$alarms$$

Заместители могут использоваться в функциях и ФБ работы с файлами (библиотеки **CAA File**, **SysFile** и т. д.), а также путях, указываемых в действии элементов визуализации **Передача файла**.

В ОС Windows (например, в случае работы с виртуальным контроллером **CODESYS Control Win V3**) пути выглядят очевидным образом: **D:\MyFolder\MyFile.txt**

Рабочая директория виртуального контроллера:

C:\ProgramData\CODESYS\CODESYSControlWinV3\<идентификатор_контроллера>\PlcLogic

При работе с накопителями следует соблюдать два правила:

1. Перед работой с накопителем следует проверить, смонтирован (подключен) ли он к файловой системе контроллера.
2. Перед извлечением накопителя из контроллера следует завершить все операции с файлами и демонтировать (отключить) накопитель.

Таргет-файлы контроллеров ОВЕН содержат узел **Drives**, с помощью которого можно получить информацию о том, смонтирован ли накопитель, сколько его памяти свободно и занято, а также демонтировать накопитель. Для работы с узлом следует привязать переменные к его каналам. Список каналов приведен ниже.

[2] Для доступа к рабочей директории заполнитель необязателен, потому что она используется по умолчанию в тех случаях, когда путь не задан. Например, если с помощью ФБ FILE.Open создать файл ‘test.bin’ (без указания пути) – то он будет создан в рабочей директории.

[3] Начиная с версии прошивки **2.4.xxxx.xxxx** директория хранения файлов трендов и тревог может быть изменена в web-конфигураторе ПЛК на вкладке **ПЛК/Настройки**.

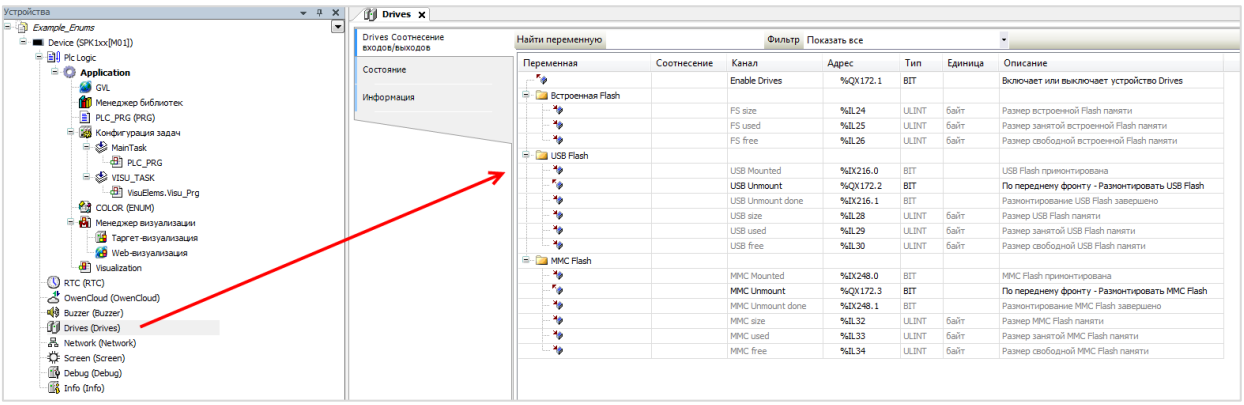


Рисунок 2.4.1 – Каналы узла Drives

Таблица 2.4.2 – Описание каналов узла Drives

Канал	Тип	Описание
Enable Drives	BOOL	Бит управления сбором информации о памяти контроллера и подключенных носителей. Если переменная имеет значение **TRUE**, то в остальных каналах каждые 5 секунд обновляется информация. При значении **FALSE** каналы не содержат информации
Встроенная Flash		
FS wear	USINT	Использованный ресурс перезаписей встроенной flash-памяти (**0…100%**)
FS size	ULINT	Объем Flash-памяти контроллера в байтах[4]
FS used	ULINT	Количество занятой Flash-памяти контроллера в байтах[4]
FS free	ULINT	Количество свободной Flash-памяти контроллера в байтах[4]
USB Flash		
USB Mounted	BOOL	Принимает значение **TRUE** после монтирования USB Flash накопителя, **FALSE** – при демонтировании
USB Unmount	BOOL	**TRUE** – демонтирование USB накопителя. Процедура демонтирования завершается в момент появления значения **TRUE** в канале **USB Unmount done**. До этого момента в канале **USB Unmount** должно сохраняться значение **TRUE**
USB Unmount done	BOOL	Принимает значение **TRUE** после демонтирования USB накопителя. Принимает значение **FALSE** по заднему фронту в канале **USB Unmount**
USB size	ULINT	Объем памяти USB накопителя в байтах
USB used	ULINT	Количество занятой памяти USB накопителя в байтах
USB free	ULINT	Количество свободной памяти USB накопителя в байтах
MMC Flash		
MMC Mounted	BOOL	Принимает значение **TRUE** после монтирования MMC накопителя, **FALSE** – при демонтировании
MMC Unmount	BOOL	**TRUE** – демонтирование SD накопителя. Процедура демонтирования завершается в момент появления значения **TRUE** в канале **SD Unmount done**. До этого момента в канале **SD Unmount** должно сохраняться значение **TRUE**
MMC Unmount done	BOOL	Принимает значение **TRUE** после демонтирования MMC накопителя. Принимает значение **FALSE** по заднему фронту в канале **MMC Unmount**
MMC size	ULINT	Объем памяти MMC накопителя в байтах
MMC used	ULINT	Количество занятой памяти MMC накопителя в байтах
MMC free	ULINT	Количество свободной памяти MMC накопителя в байтах

[4] Здесь отображается не объем физической памяти, а объем области, выделенный системе исполнения CODESYS

2.5 Ограничения на имена файлов и каталогов в ОС Linux

1. Максимальная длина – 255 символов.
2. Символы кириллицы и символ ‘/’ не поддерживаются.
3. Не рекомендуется использовать в названиях следующие символы:
   ! @ # $ & ~ % * ( ) [ ] { } ' " \ : ; > < ` пробел
4. Регистр имеет принципиальное значение. **Test.txt** и **test.txt** – это два разных файла.

2.6 Бинарные и текстовые файлы

С точки зрения формата хранения данных файлы можно разделить на три категории:

- **Бинарные (двоичные)** – информация хранится в двоичном виде. Преимуществом этого формата является фиксированная длина каждой записи (определяемая типами записываемых переменных), что позволяет легко организовать чтение архива;
- **Текстовые (строковые)** – информация хранится в символьном виде. Преимуществом этого формата является простота работы с ним – пользователь может открыть файл в текстовом редакторе или офисном пакете ПО (например, **Microsoft Excel**);
- **Смешанные** – часть информации хранится в символьном виде, часть – в бинарном (например, символьный заголовок и бинарные данные).

Во время работы с текстовыми файлами следует помнить об их кодировке. Среда **CODESYS V3.5** включает два типа переменных, используемых для работы с символами (строками):

- **STRING** – использует 8-битную ASCII-based кодировку, зависящую от конкретного устройства, каждый символ занимает 1 байт;
- **WSTRING** – использует кодировку Unicode (UCS2), каждый символ занимает 2 байта.

В **CODESYS** строки являются нуль-терминированными – т.е. заканчиваются одним (для **STRING**) или двумя (для **WSTRING**) NULL-байтами. NULL-байты формируются средой программирования автоматически. Иными словами:

- переменная **STRING(80)** займет **81** байт (80 однобайтовых символов + 1 байт на NULL);
- переменная **WSTRING(80)** займет **162** байта (80 двухбайтовых символов + два байта на NULL).

Для обработки строк могут использоваться готовые функции следующих библиотек:

- **Standard** (базовые функции для работы со **STRING**);
- **Standard64** (базовые функции для работы с **WSTRING**);
- **StringUtils** (функции для работы со строками, длина которых превышает 255 символов);
- **OwenStringUtils** (дополнительные функции работы со строками, в т. ч. конвертация кодировки строки из ASCII в Unicode и обратно);
- **OSCAT** (дополнительные функции работы со строками).

Следует отметить, что контроллеры ОВЕН не поддерживают отображение в визуализации строк в кодировке **Win1251** (и других подобных ASCII-кодировках) – таким образом, переменные и константы типа **STRING** не могут использоваться для отображения в визуализации кириллических символов. В этом случае следует использовать переменные и константы типа **WSTRING**.

В случае архивирования строк типа **WSTRING** для корректного отображения архива в текстовом редакторе (или другом ПО) следует использовать маркер последовательности байт.

Для форматирования текста строковых переменных (например, для перехода на новую строку, табуляции и т. д.) применяются спецсимволы, которые называются **управляющими последовательностями**. Их список приведен ниже:

Таблица 2.6.1 – Управляющие последовательности для строковых переменных

Символ	Результат использования/Отображаемое значение
$$	$ (символ доллара)
$’	‘ (апостроф)
$L	Перевод строки
$N	Новая строка
$R	Возврат каретки
$P	Новая страница
$T	Табуляция
$xx (xx – код символа в **HEX**)	Символ таблицы ASCII (только для **STRING**)

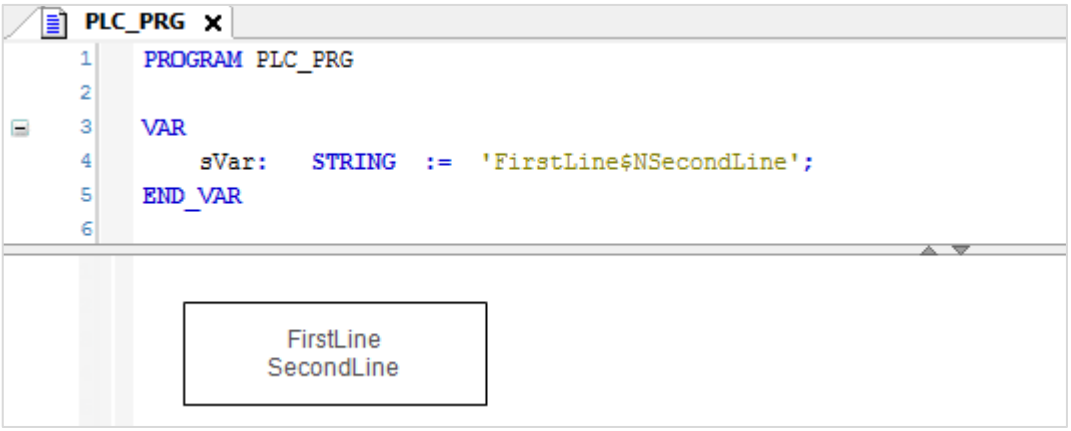


Рисунок 2.6.1 – Использование управляющих последовательностей

2.7 Обработка ошибок и некорректных ситуаций

Во время работы с файлами рекомендуется обратить внимание и реализовать обработку следующих ситуаций:

1. Обработку ошибок ФБ библиотеки **CAA File** (выходы **xError** и **eError**).
2. Проверку монтирования накопителя перед работой с ним.
3. Проверку демонтирования накопителя перед извлечением.
4. Наличие свободного места для архива на накопителе или в памяти контроллера.

2.8 Подключение к файловой системе контроллера

Для упрощения отладки программ, работающих с файлами, можно организовать подключение к файловой системе контроллера, чтобы иметь возможность просматривать и загружать файлы. Для этих целей рекомендуется использовать утилиту **WinSCP**. Утилита распространяется бесплатно и может быть загружена с сайта https://winscp.net/eng/download.php.

После запуска утилиты следует настроить соединение по протоколу **SCP**, указав **IP-адрес** контроллера, имя пользователя – **root** и пароль (по умолчанию – **owen**, может быть изменен в web-конфигураторе). Чтобы подключиться к контроллеру, следует нажать **Войти**.

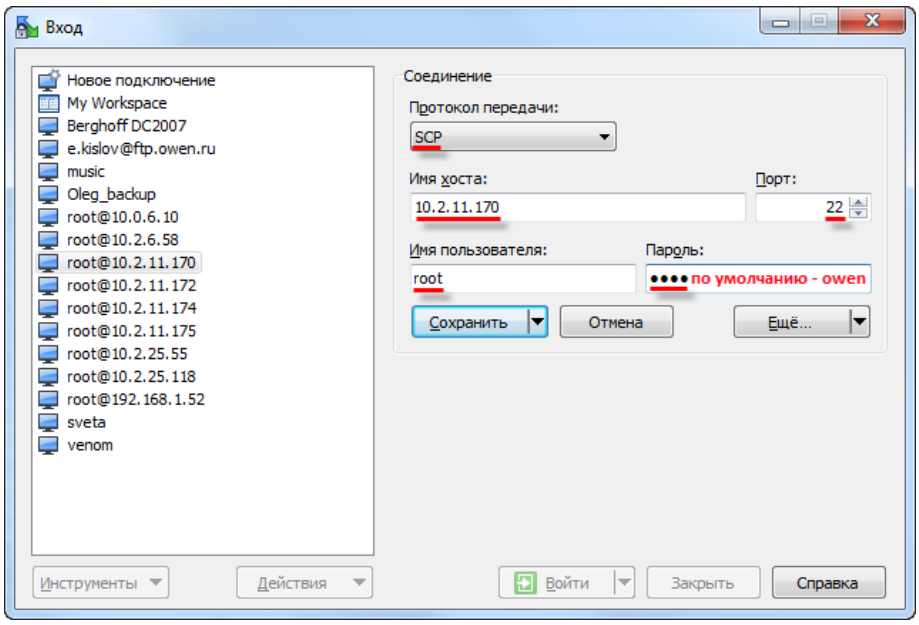


Рисунок 2.8.1 – Настройки соединения в WinSCP

В случае возникновения сообщений типа «**Не могу получить имя каталога на сервере**» следует нажать кнопку **ОК**.

В результате будет открыто окно файлового менеджера с интуитивно понятным интерфейсом.

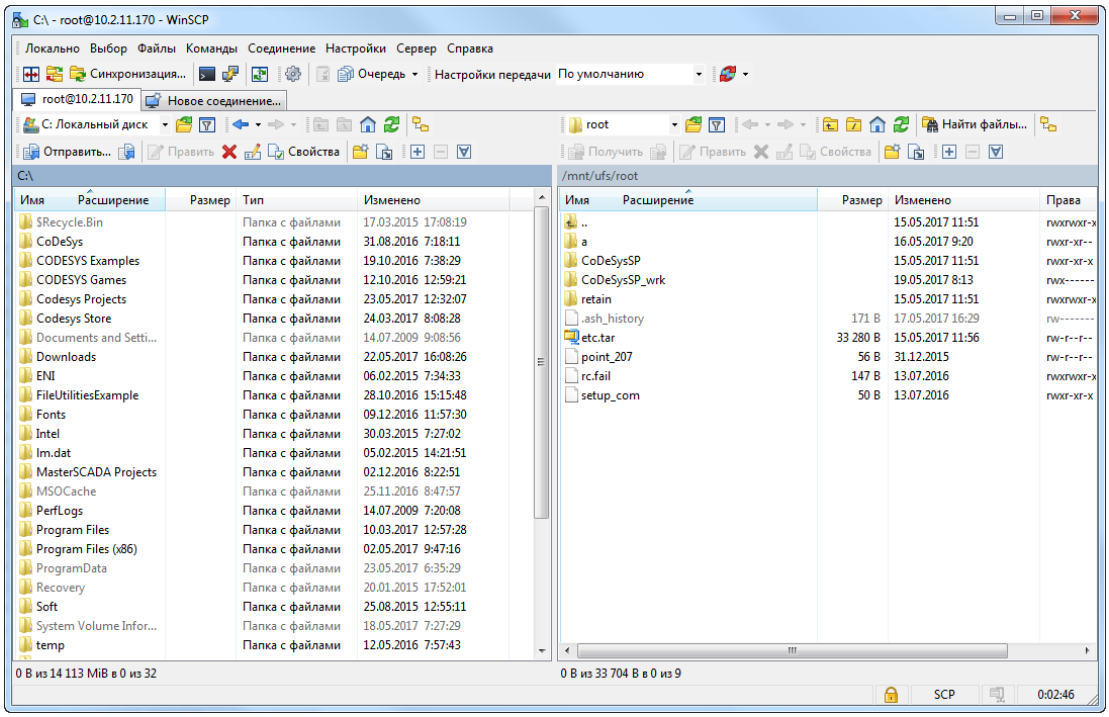


Рисунок 2.8.2 – Окно файлового менеджера WinSCP

2.9 Работа с FTP

Контроллер может использоваться в режиме FTP-сервера. По умолчанию FTP-сервер контроллера запущен. Логин для доступа: **ftp**, пароль по умолчанию: **ftp** (может быть изменен в web-конфигураторе). См. более подробную информацию в руководстве **Краткое описание основных функций Web-интерфейса управления контроллеров ПЛК2xx и СПК1xx**.

Рабочая директория FTP-сервера по умолчанию (может быть изменена в web-конфигураторе контроллера): **/mnt/ufs/home/ftp/in**

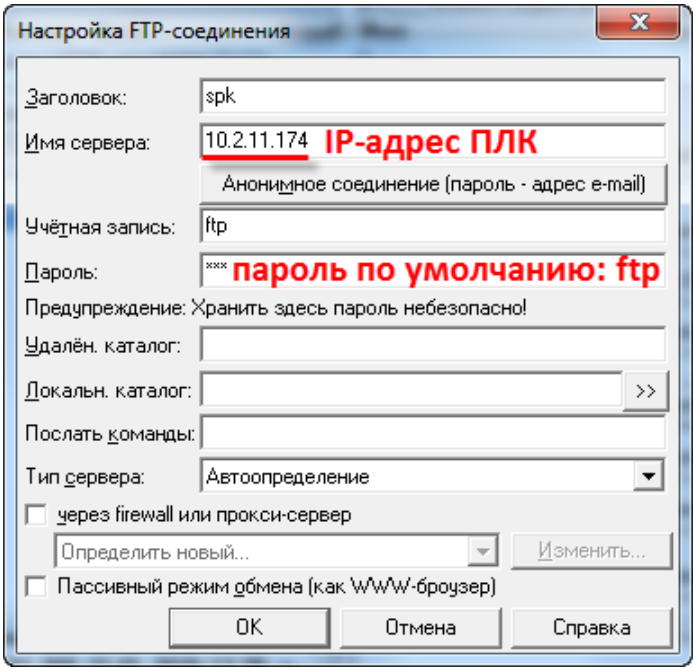


Рисунок 2.9.1 – Настройки FTP-соединения в файловом менеджере Total Commander

Для работы в режиме FTP-клиента следует использовать утилиту **cURL**. Пример использования доступен на сайте компании ОВЕН в разделе **CODESYS V3/Примеры**.

3 Компонент OwenArchiver

3.1 Установка компонента в CODESYS

Компонент **OwenArchiver** представляет собой архиватор, настраиваемый через дерево проекта. Создаваемый архив представляет собой файл формата .csv.

Для работы с компонентом следует установить в CODESYS пакет **OwenArchiver_3.5.4.x.** В настоящем руководстве описывается работа с компонентом версии **3.5.4.9**.

Архиватор распространяется в виде пакета формата **.package**. Пакет доступен на сайте компании ОВЕН в разделе **CODESYS V3/Библиотеки**.

Для установки пакета в **CODESYS** в меню **Инструменты** следует выбрать пункт **Менеджер пакетов**, после чего указать путь к файлу пакета и нажать кнопку **Установить**.

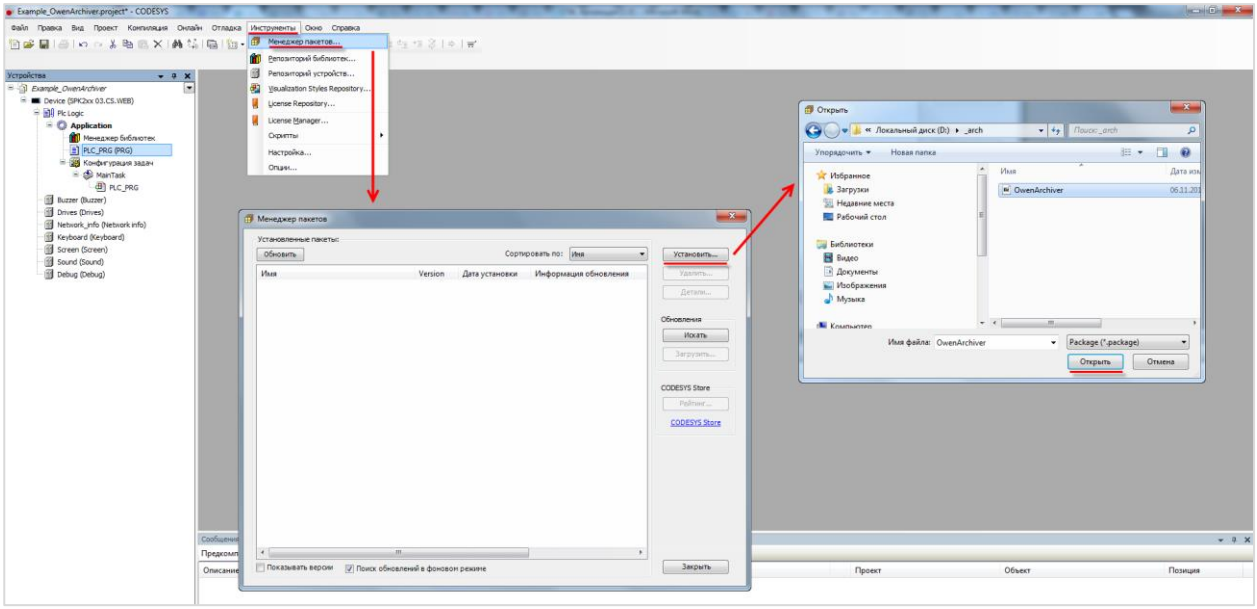


Рисунок 3.1.1 – Установка пакета OwenArchiver в среду CODESYS

В появившемся диалоговом окне следует выбрать пункт **Полная установка**, после чего нажать кнопку **Next**:

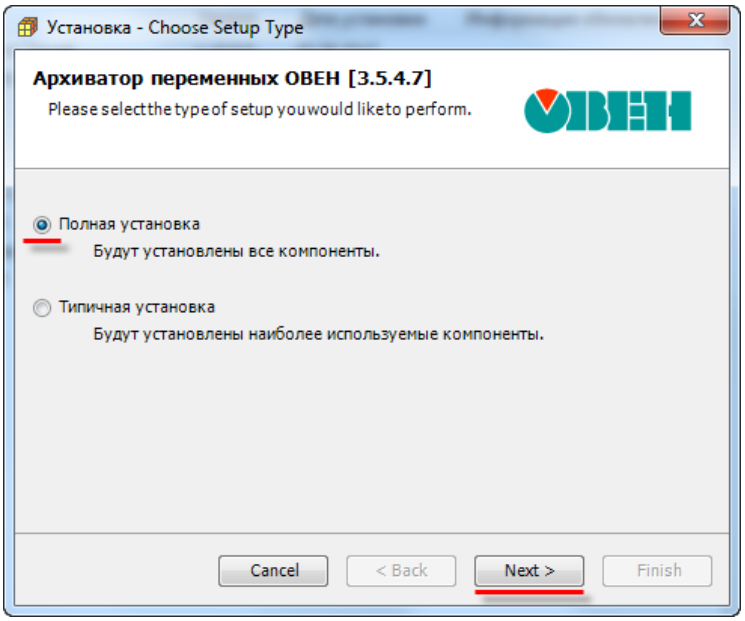


Рисунок 3.1.2 – Начало установки архиватора

После завершения установки следует закрыть диалоговое окно с помощью кнопки **Finish**:

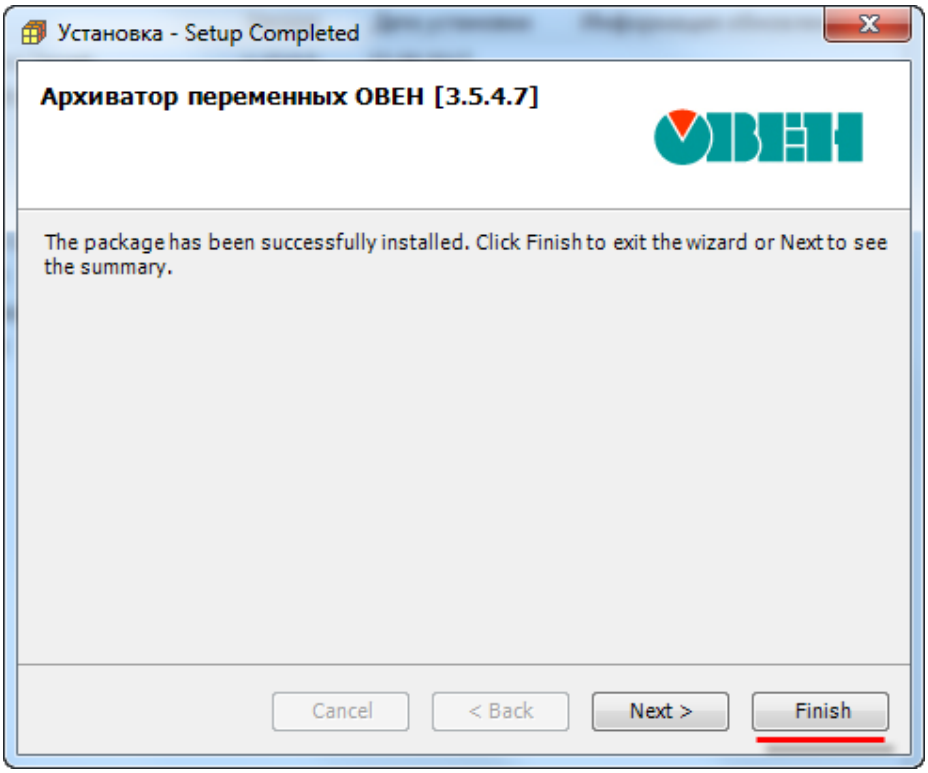


Рисунок 3.1.3 – Завершение установки архиватора

3.2 Добавление архиватора в проект

Чтобы добавить архиватор в проект **CODESYS** следует:

1. Нажать **ПКМ** на узел **Device** и добавить компонент **OwenArchiver**, расположенный во вкладке **Разн.** (**Miscellaneous**):

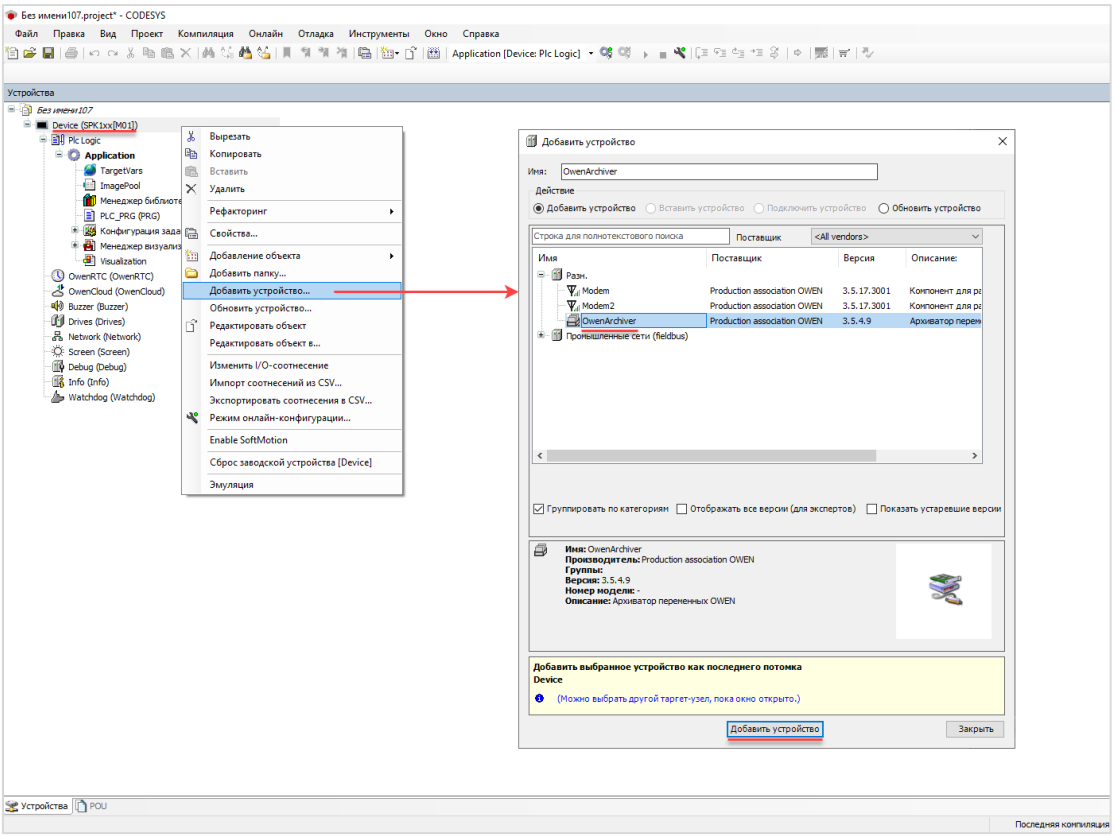


Рисунок 3.2.1 – Добавление архиватора в проект CODESYS

После добавления архиватора в проекте будет автоматически создана задача **OwenArchiver**. Ее не следует удалять или перенастраивать.

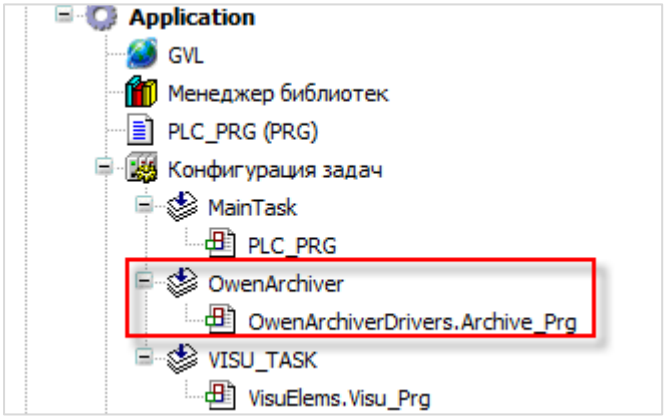


Рисунок 3.2.2 – Внешний вид дерева проекта после добавления архиватора

2. В настройках компонента **OwenArchiver** на вкладке **PCI-шина Конфигурация** указать настройки архива:

Имя архива – должно быть уникальным в рамках проекта;
Режим архивирования – условие добавления записи в архив:

- **Периодически** – записи будет добавляться циклически с периодом, определяемым параметром **Период архивации**;
- **По команде** – записи будут добавляться по переднему фронту заданной логической переменной (см. пп. 3), но не чаще **раза в секунду**;
- **По изменению** – записи будут добавляться при изменении значения любой из переменных архива, но не чаще периода архивации.

Период архивации, сек – время между двумя операциями записи в архив, минимальное значение – **5 секунд**;
Максимальный размер архива, Мб – суммарный объем **всех файлов** архива, максимальное значение – **2047 Мб** (для режима архивации **Непрерывный архив** (см. пп. 4) фактический занимаемый объем в два раза превышает данное значение);
Десятичный разделитель для типов с плавающей точкой – запятая или точка;
Устройство для ведения архива:

- **Директория CODESYS** – архив будет сохраняться во внутренней памяти контроллера (в рабочий каталог по пути **PlcLogic/Archives/<имя_архива>**);
- **USB-flash** – архив будет сохраняться на USB-накопитель (в каталог **/Archives/<имя_архива>**[5]);
- **SD-карта** – архив будет сохраняться на SD-накопитель (в каталог **/Archives/<имя_архива>**[5]);
- **Директория FTP** – архив будет сохраняться во внутренней памяти контроллера (в каталог FTP-сервера);
- **Использовать переменную** – место ведения архива определяется переменной (см. пп. 3).

Имя архива – переменная? – если параметр имеет значение **TRUE,** то имя архива определяется переменной (см. пп. 3).

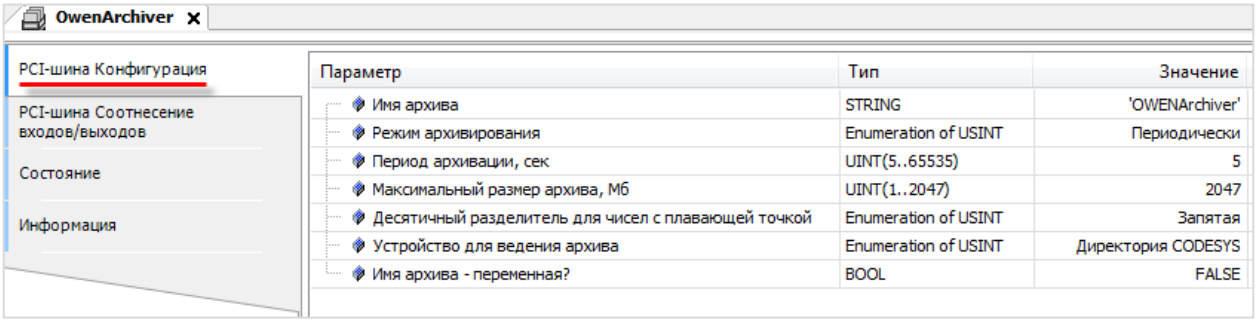


Рисунок 3.2.3 – Настройки архиватора, вкладка PCI-шина Конфигурация

[5] Для режима архивации Непрерывный архив (см. пп. 4) архив будет сохраняться в корне выбранного устройства

3. В настройках компонента **OwenArchiver** на вкладке **PCI-шина Соотнесение входов/выходов** привязать к нужным каналам переменные:

Таблица 3.2.1 – Описание каналов архиватора

Название канала	Тип	Описание
Управление архиватором		
Запустить архиватор	BOOL	Бит управления архиватором. Пока он имеет значение **TRUE** – архиватор запущен. Если бит принимает значение **FALSE** – процесс архивации прекращается
Команда записи	BOOL	По переднему фронту данной переменной в архив добавляется новая запись (только для режима **По команде**, см. пп. 2). Переменная должна изменяться в задаче архиватора (см. рисунок 3.2.2)
Запись лога отладки	BOOL	Если параметр имеет значение **TRUE**, то в памяти контроллера будет сохраняться лог архивации (в рабочий каталог, имя файла будет совпадать с именем архива). Лог содержит список всех операций, производимых архиватором
Конфигурация архиватора		
Путь архивации	USINT/ENUM	Выбор устройства архивации (если в конфигурации выбрано устройство архивации **Использовать переменную**). Изменения вступают в силу по переднему фронту канала **Запустить архиватор**. См. перечисление **WHERE_TO_ARCHIVE** в библиотеке **OwenArchiveDrivers**
Имя архива	STRING(80)	Имя архива (если в конфигурации параметр **Имя архива – переменная?** имеет значение **TRUE**). Указание формата не требуется (всегда используется **.csv**). Изменения вступают в силу по переднему фронту канала **Запустить архиватор**
Статус архиватора		
Код последней ошибки	USINT	Код последней ошибки архиватора. См. библиотеку **OwenArchiveErrors,** содержащую перечисление с кодами ошибок
Статус архиватора	BOOL	Статус архиватора. **TRUE** – архиватор запущен, **FALSE** – остановлен
Использование буфера записи	USINT	Параметр характеризует степень заполнения буфера записи (в %). Эта информация может потребоваться для отладки в сложных проектах с высокой частотой архивации большого количества данных
Размер архива	REAL	Суммарный размер всех файлов архива в мегабайтах

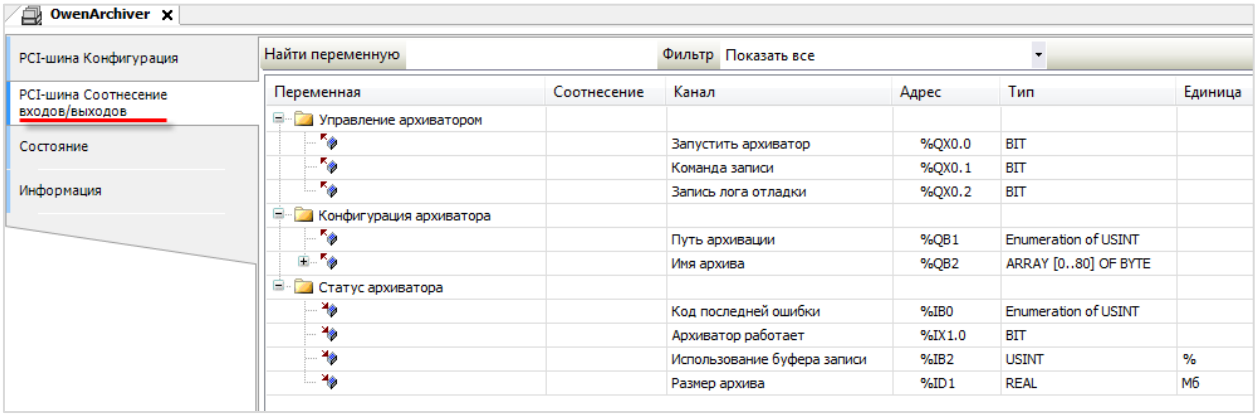


Рисунок 3.2.4 – Настройки архиватора, вкладка PCI-шина Соотнесение входов/выходов

4. В настройках компонента **CSVFormat** на вкладке **CSVFormat Конфигурация** выбрать структуру архива:

   Непрерывный архив – все данные будут записываться в один файл. По достижению его максимального размера (см. пп. 3) будет создан новый файл, а по достижению максимального размера нового файла – первый файл будет удален. Таким образом, фактически архив состоит из двух файлов – текущего (в который записываются данные) и предыдущего;
   Год/Месяц/День – архив за каждые сутки будет записываться в отдельный файл (название – *номер дня*), файлы за каждый месяц будут сохранены в каталоге (название – *номер месяца*), каталоги за каждый год будут сохранены в каталоге (название – *номер года*). По достижению максимального размера архива (см. пп. 3) самые старые файлы будут последовательно удаляться. Если в результате удаления файлов каталог какого-либо месяца окажется пустым, то будет удален;
   Год/Месяц_День – архив за каждые сутки будет записываться в отдельный файл (название – *номер месяца_номер дня*), файлы за каждый год будут сохранены в каталоге (название – *номер года*).

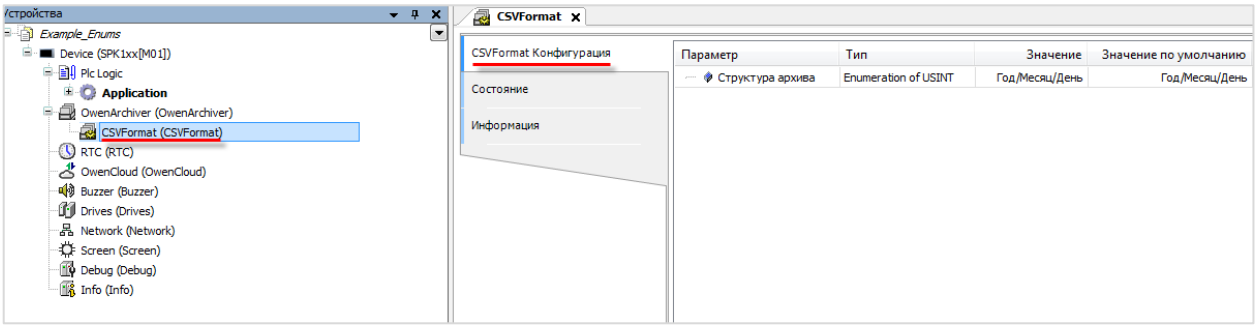


Рисунок 3.2.5 – Настройки архиватора, вкладка CSVFormat Конфигурация

5. Нажать **ПКМ** на компонент **OwenArchiver** и добавить каналы архивируемых переменных нужных типов. Всего архиватор может содержать до **64** каналов.

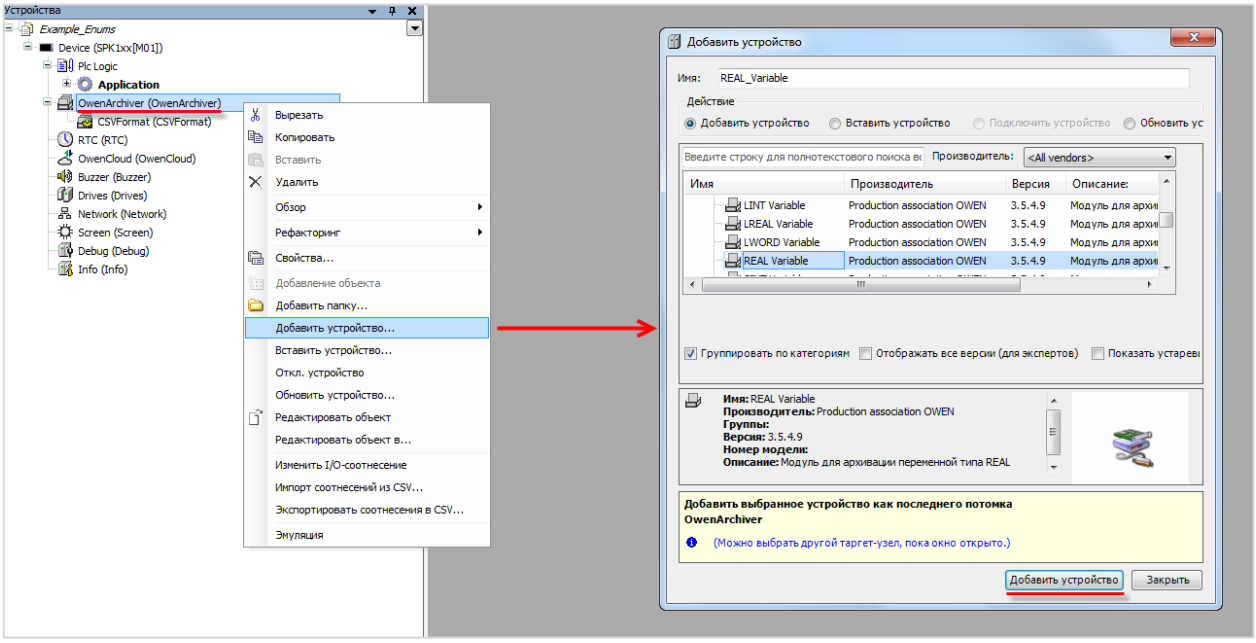


Рисунок 3.2.6 – Добавление каналов архивируемых переменных

В настройках модуля на вкладке **ArchiverVariable Конфигурация** следует указать:

Описание переменной – название, используемое при формировании заголовка архива;

Кол-во знаков после десятичного разделителя – количество знаков после запятой для переменных типа **REAL/LREAL**.

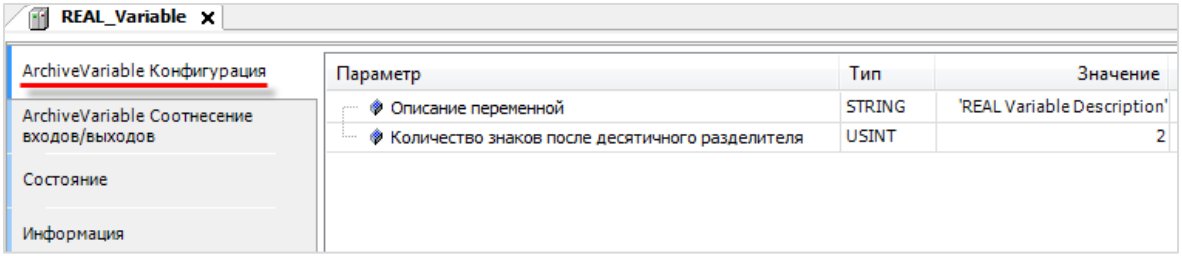


Рисунок 3.2.7 – Настройки канала архивируемой переменной

На вкладке **ArchiverVariable Соотнесение входов-выходов** следует привязать архивируемую переменную соответствующего типа.

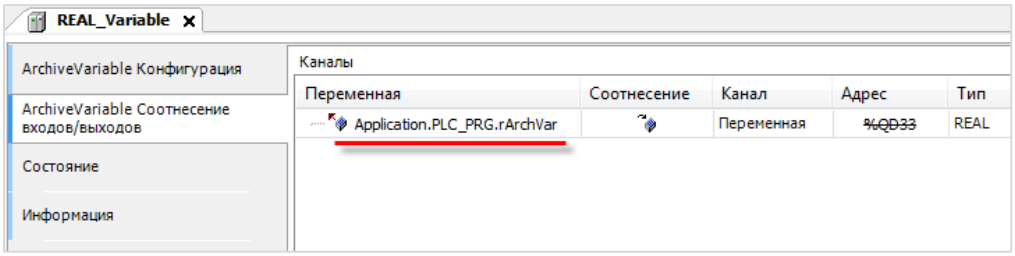


Рисунок 3.2.8 – Привязка архивируемой переменной к каналу

3.3 Ограничения, связанные с использованием архиватора

1. Архиватор может использоваться только на контроллерах ОВЕН и виртуальном контроллере **CODESYS Control Win V3**.
2. Максимальное количество переменных для одного архиватора – **64**.
3. В проекте может использоваться несколько архиваторов, но они должны работать с разными файлами. Максимально допустимое число одновременно работающих архиваторов – **2**. Использование большего количества одновременно запущенных архиваторов может привести к значительной нагрузке на процессор и значительному потреблению оперативной памяти – корректная работа контроллера в данном случае не гарантируется.
4. Архиватор использует **память ввода-вывода CODESYS**. Ее количество ограничено и зависит от модели контроллера (см. конкретные значения в документе **CODESYS V3.5. FAQ**). Эта область также используется компонентами Modbus, системными узлами таргет-файла (например, **Buzzer**) и средой CODESYS. В случае превышения доступного объема памяти во время компиляции проекта возникнут соответствующие ошибки.
5. Архиватор не контролирует объем доступной памяти контроллера и подключенных накопителей. Пользователь может реализовать данный функционал самостоятельно (например, остановку архиватора в случае исчерпания памяти) с помощью каналов системного узла Drives.
6. Для архивируемых строковых (STRING) переменных максимальная длина составляет **80** символов.
7. Использование архиватора в некоторых случаях может привести к периодическому кратковременному сбою при считывании системного времени – системное время будет считано без учета часового пояса. См. более подробное описание проблемы и способ ее решения в этой статье.

3.4 Пример работы с архиватором

Пример создан в среде **CODESYS V3.5 SP17 Patch 3** и подразумевает запуск на **СПК1xx [M01]** с таргет-файлом **3.5.17.31.** В случае необходимости запуска проекта на другом устройстве следует изменить таргет-файл в проекте (**ПКМ** на узел **Device** – **Обновить устройство).**

Пример доступен для скачивания: Example_OwenArchiver.projectarchive

Расширенная версия примера: Example_OwenArchiverExtended.projectarchive

Для работы с архиватором следует:

1. Объявить в программе **PLC_PRG** следующие переменные:

PLC_PRG

```
PROGRAM PLC_PRG
VAR
    (*архивируемые переменные*)
    wArchVar:       WORD;
    rArchVar:       REAL;
    sArchVar:       STRING;

    (*переменные архиватора*)
    xArchEnable:    BOOL;                           // бит управления архиватором
    eArchError:     OwenArchiverErrors.ARCHIVE_ERROR;   // код ошибки
    wsArchError:    WSTRING;                        // сообщение об ошибке
    xArchStatus:    BOOL;                           // статус архиватора (TRUE - запущен, FALSE - остановлен)
END_VAR
```

```
// конвертируем код ошибки архиватора в строку

wsArchError:=OwenArchiverErrors.ARCHIVE_ERROR_TO_WSTRING(eArchError);
```

Рисунок 3.4.1 – Объявление переменных и код программы PLC_PRG

Код программы содержит только вызов функции конвертации кода ошибки архиватора в строку, содержащую описание ошибки.

2. Нажать **ПКМ** на компонент **Device** и добавить компонент **OwenArchiver**:

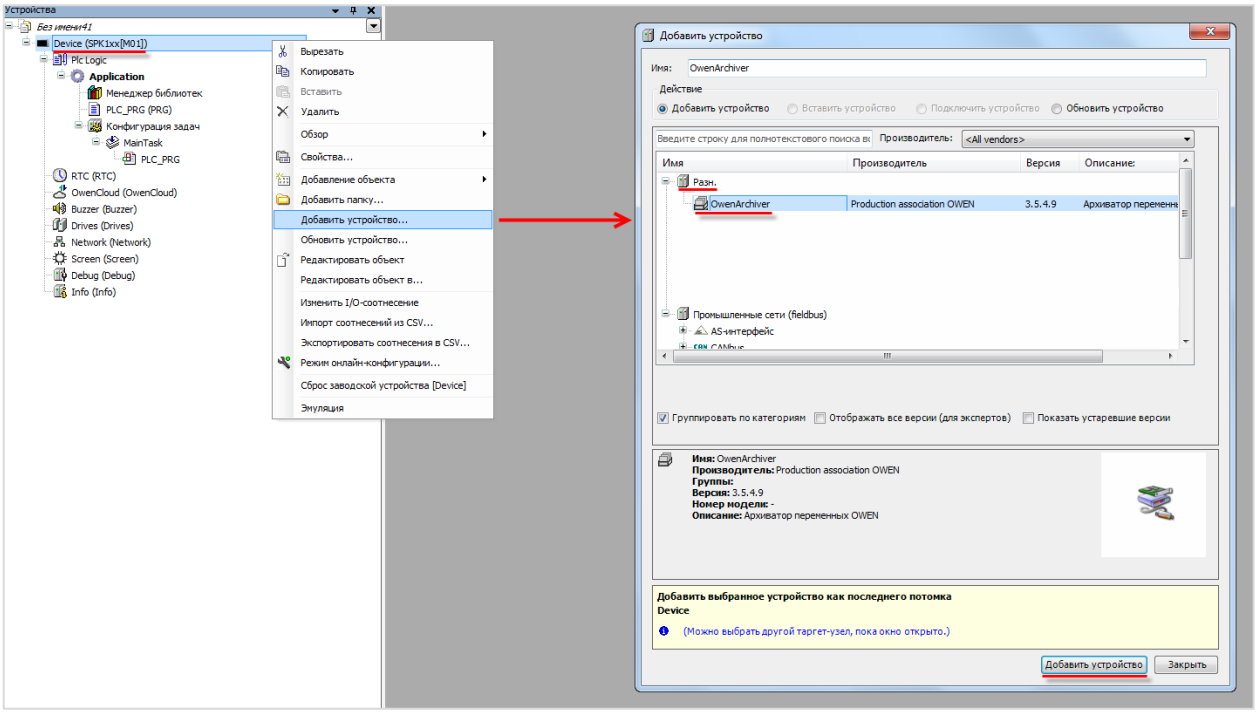


Рисунок 3.4.2 – Добавление компонента OwenArchiver

После добавления в проект компонента **OwenArchiver** будет автоматически добавлена задача **OwenArchiver**. Ее не следует удалять или перенастраивать.

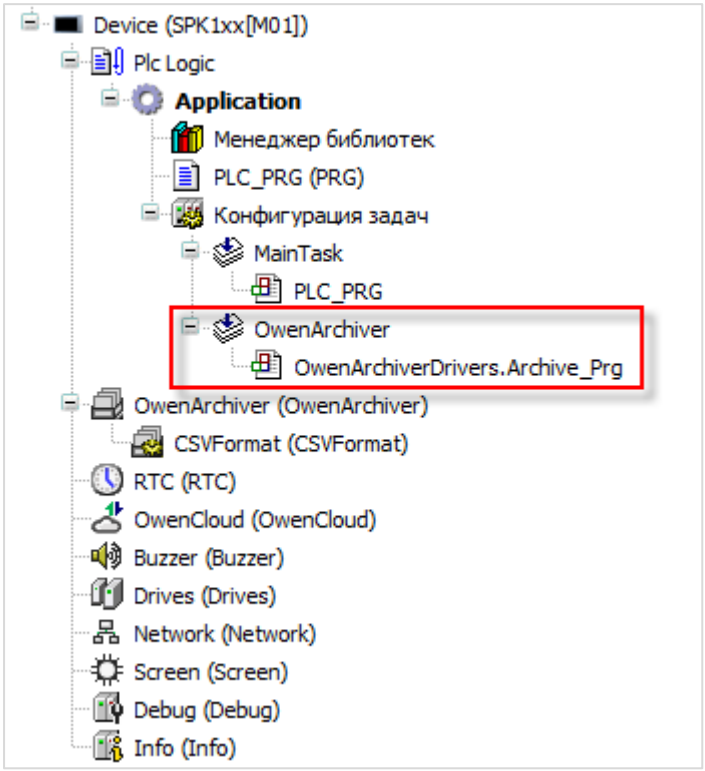


Рисунок 3.4.3 – Внешний вид дерева проекта после добавления архиватора

3. В настройках компонента **OwenArchiver** на вкладке **PCI-шина Конфигурация** следует указать параметры архивации. В данном примере архивация будет производиться на **USB-накопитель** в файл **MyArchive** с периодичностью **5 секунд**.

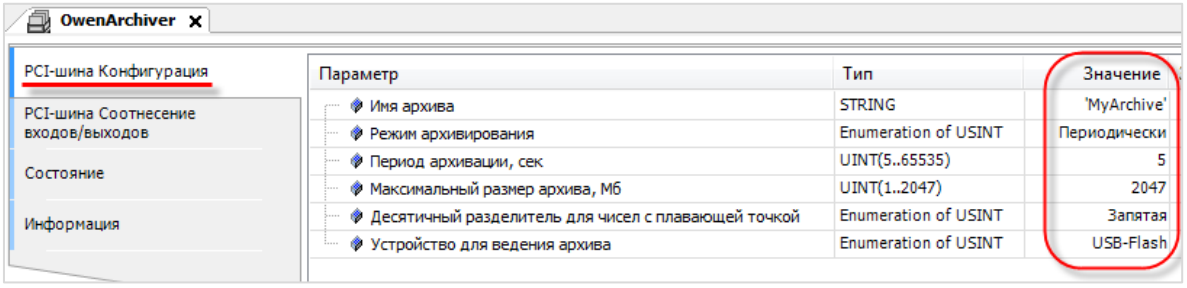


Рисунок 3.4.4 – Настройка параметров архивации

На вкладке **PCI-шина Соотнесение входов-выходов** следует привязать к каналам переменные программы.

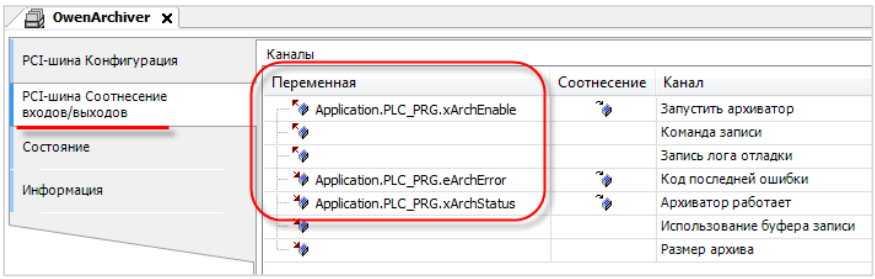


Рисунок 3.4.5 – Привязка переменных контроля архиватора

4. В настройках компонента **CSVFormat** на вкладке **CSVFormat Конфигурация** следует указать нужную структуру архива. В данном примере архивация будет производиться в режиме **непрерывного архива**.

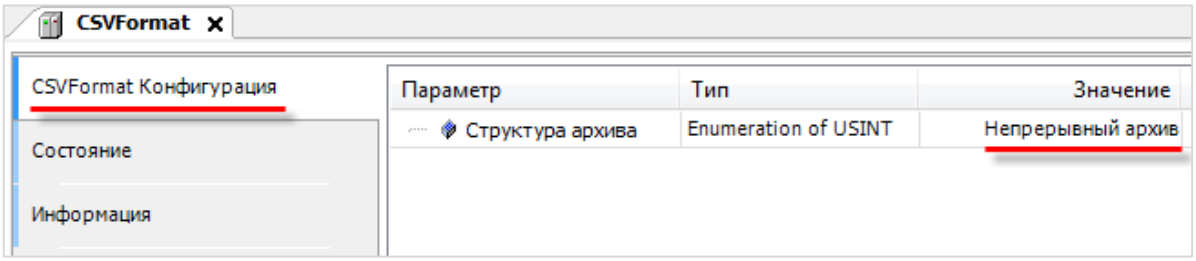


Рисунок 3.4.6 – Выбор режима архивации

В режиме **Непрерывный архив** данные записываются в файл до тех пор, пока не будет достигнут его максимальный размер. После этого файл будет переименован в **<имя_архива_old>**, и будет создан новый файл с названием **<имя_архива>**, в который будут записываться данные. В случае достижения максимального размера этого файла – файл **<имя_архива_old>** будет удален, текущий файл (**<имя_архива>**) будет переименован в **<имя_архива_old>** и будет создан новый файл (**<имя_архива>**), в который продолжится архивация. Таким образом, в каждый момент времени будет существовать два файла архива – текущий и предыдущий.

5. Нажать **ПКМ** на компонент **OwenArchiver** и добавить каналы архивации нужных типов. Максимальное число каналов – **64**. В данном примере будут использоваться каналы типа **WORD**, **REAL** и **STRING** (по одному каналу каждого типа).

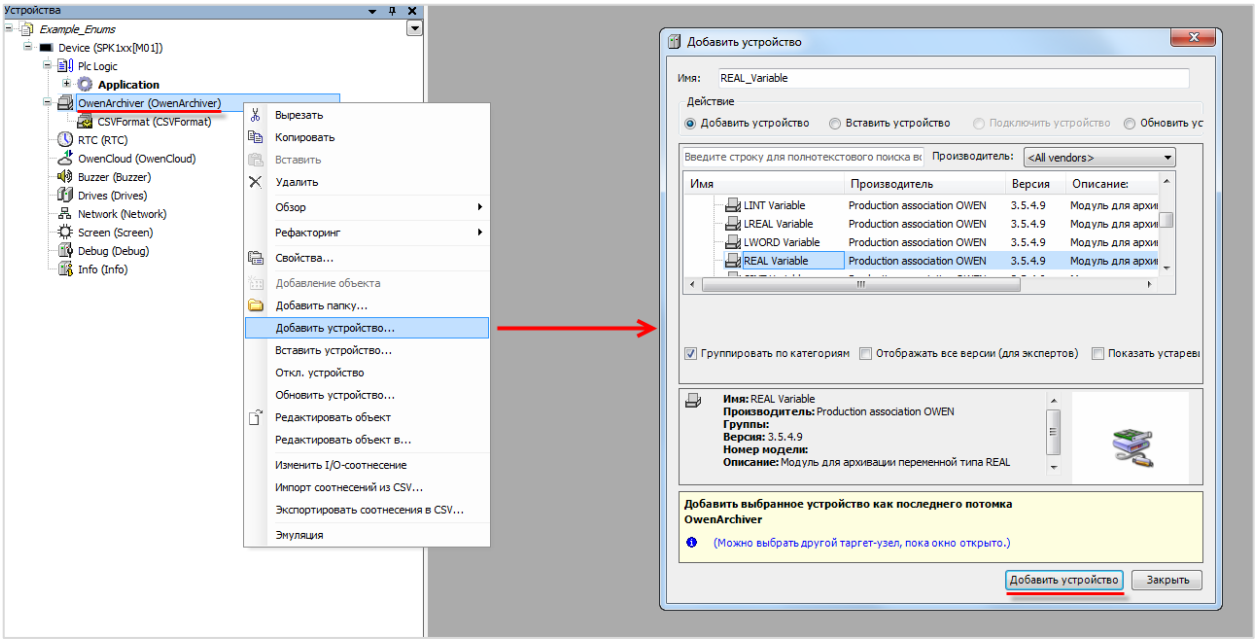


Рисунок 3.4.7 – Добавление каналов архивации

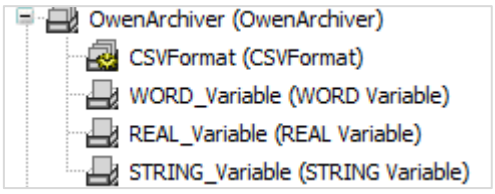


Рисунок 3.4.8 – Внешний вид дерева проекта после добавления каналов архивации

В настройках каждого из каналов на вкладке **ArchiveVariable Конфигурация** следует задать название переменной (оно будет использовать в качестве названия столбца в строке заголовка архива). Для каналов типа **REAL/LREAL** также следует указать используемое количество знаков после запятой.

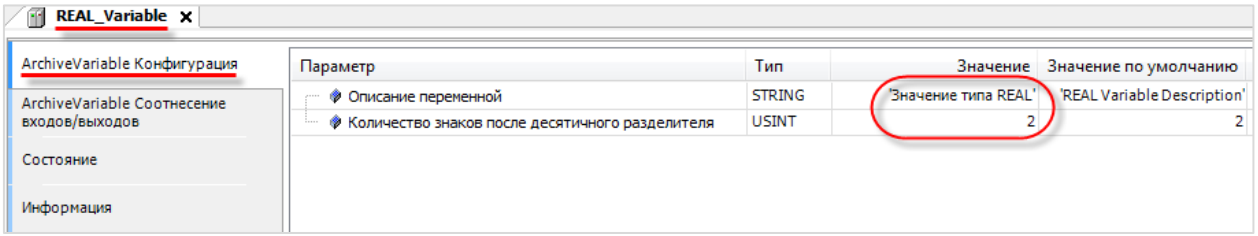


Рисунок 3.4.9 – Настройка канала архивации

В рамках примера используются названия **Значение типа WORD/Значение типа REAL/Значение типа STRING**.

На вкладке **ArchiveVariable Соотнесение входов-выходов** каждого из каналов следует привязать соответствующую переменную:

- к каналу типа **WORD** – переменную **wArchVar**;
- к каналу типа **REAL** – переменную **rArchVar**;
- к каналу типа **STRING** – переменную **sArchVar**.

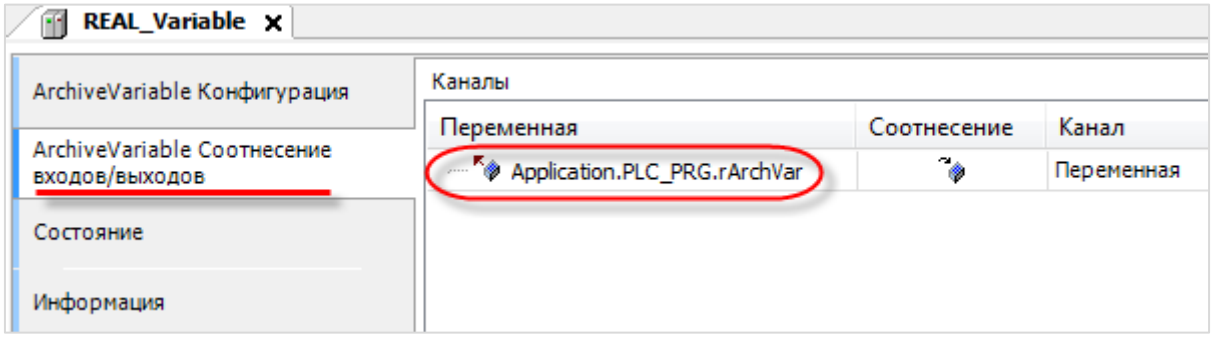


Рисунок 3.4.10 – Привязка переменных к каналам архивации

6. Создать интерфейс оператора.
   В настоящем руководстве не рассматривается подробно процесс разработки визуализации (вся необходимая информация приведена в документе **CODESYS V3.5 Визуализация**).

   На рисунке 3.4.11 приведен внешний вид экрана **Visualization**, который включает в себя:
   - 3 прямоугольника для отображения и ввода значений архивируемых переменных (с привязанными переменными **wArchVar/rArchVar/sArchVar** соответственно);
   - клавишный переключатель **Управление архиватором** с привязанной переменной **xArchEnable**;
   - прямоугольник для отображения сообщений об ошибках с привязанной переменной **wsArchError**;
   - прямоугольник для отображения статуса архиватора с привязанной к параметру **Переключить цвет** переменной **xArchStatus**.

Рисунок 3.4.11 – Внешний вид экрана визуализации в редакторе визуализации

7. Загрузить проект в контроллер и запустить его. Нажать переключатель **Управление архиватором**, чтобы запустить архивацию. Изменить значения архивируемых переменных.

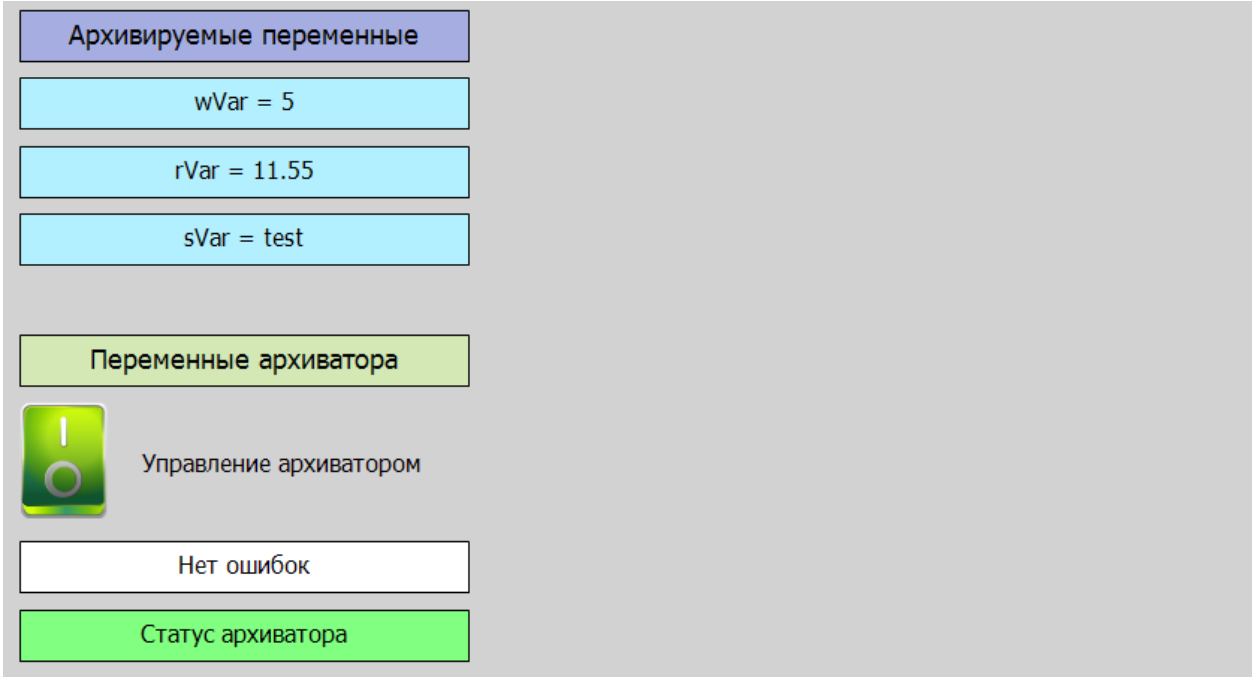
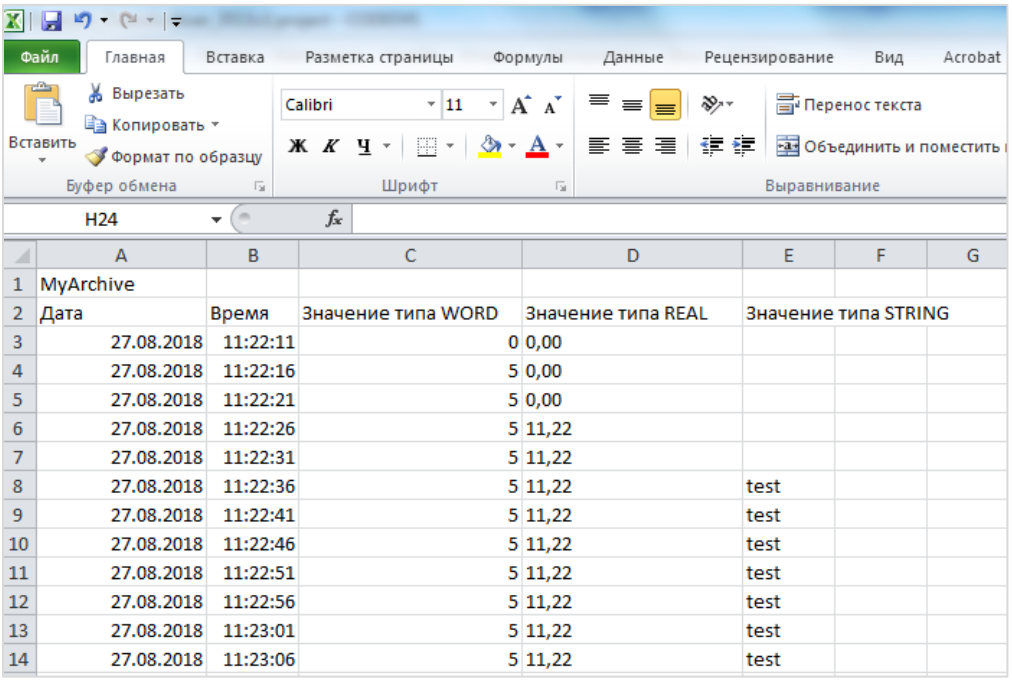


Рисунок 3.4.12 – Внешний вид экрана визуализации после запуска проекта

8. В корневой директории USB-накопителя будет создан файл **MyArchive.csv**.



	A	B	C	D	E	F	G
1	MyArchive						
2	Дата	Время	Значение типа WORD	Значение типа REAL	Значение типа STRING		
3	27.08.2018	11:22:11	0	0,00			
4	27.08.2018	11:22:16	5	0,00			
5	27.08.2018	11:22:21	5	0,00			
6	27.08.2018	11:22:26	5	11,22			
7	27.08.2018	11:22:31	5	11,22			
8	27.08.2018	11:22:36	5	11,22	test		
9	27.08.2018	11:22:41	5	11,22	test		
10	27.08.2018	11:22:46	5	11,22	test		
11	27.08.2018	11:22:51	5	11,22	test		
12	27.08.2018	11:22:56	5	11,22	test		
13	27.08.2018	11:23:01	5	11,22	test		
14	27.08.2018	11:23:06	5	11,22	test		

Рисунок 3.4.13 – Фрагмент архива

9. Рекомендуется ознакомиться с примером получения информации о накопителях – это поможет определять свободный/занятый объем (и в случае необходимости останавливать архивацию), определять статус накопителя (смонтирован/демонтирован), демонтировать его и др.

4 Библиотека CAA File

4.1 Добавление библиотеки в проект CODESYS

Библиотека **CAA File** используется для работы с файлами и каталогами.

Библиотека реализует асинхронный доступ к файлам и каталогам – поэтому выполнение блоков может занять несколько циклов задачи ПЛК, но остальные задачи (визуализация, обмен и т. д.) в течение этого времени будут продолжать выполняться в штатном режиме.

Для добавления библиотеки в проект CODESYS в **Менеджере библиотек** следует нажать кнопку **Добавить** и выбрать библиотеку **CAA File**.

ПРИМЕЧАНИЕ

Версия библиотеки не должна превышать версию таргет-файла контроллера. В противном случае корректная работа контроллера не гарантируется.

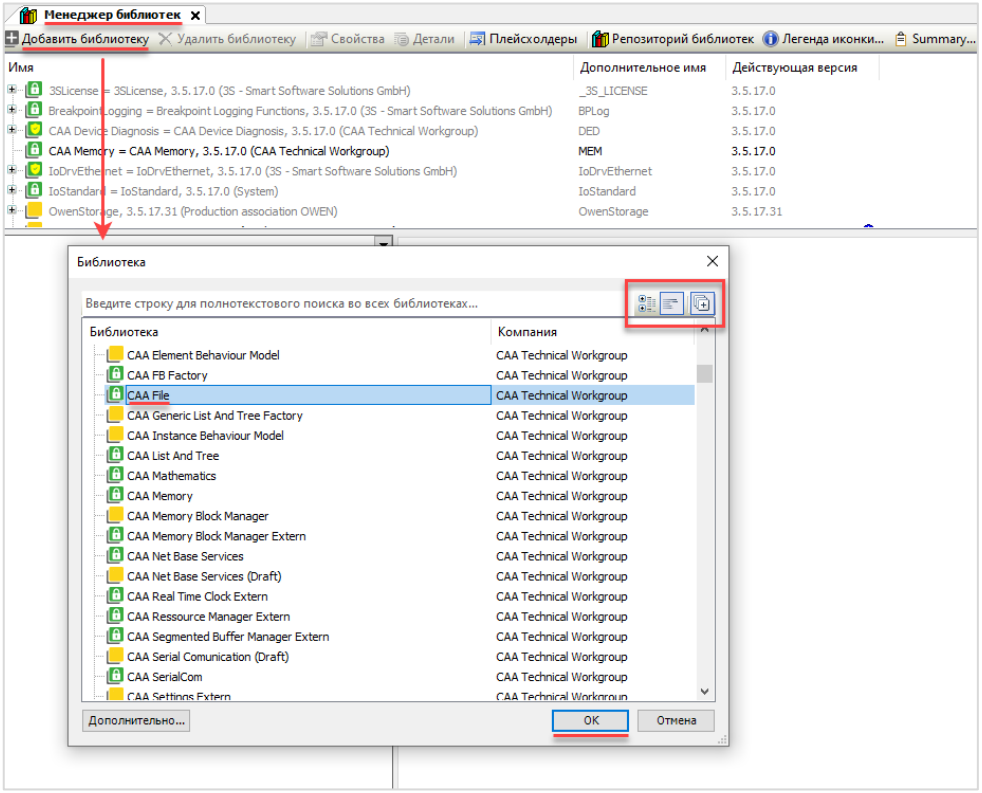


Рисунок 4.1.1 – Добавление библиотеки CAA File в проект CODESYS

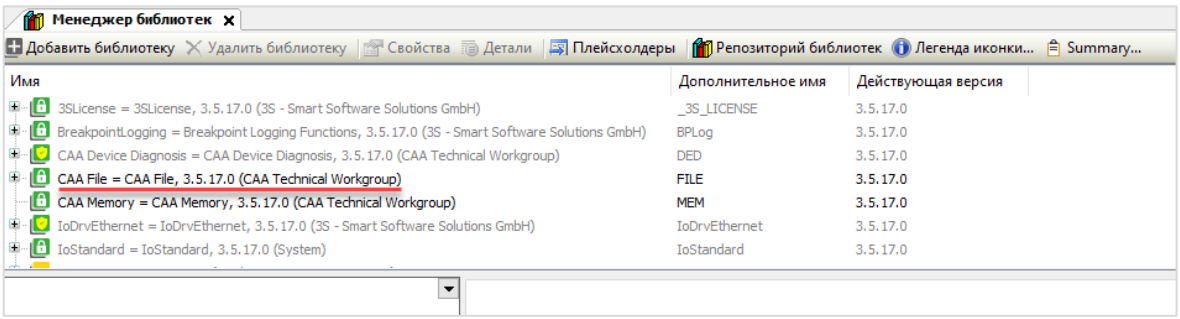


Рисунок 4.1.2 – Отображение библиотеки CAA File в менеджере библиотек

ПРИМЕЧАНИЕ

При объявлении экземпляров ФБ библиотеки следует перед их названием указывать префикс FILE**.** (пример: **FILE.Open**).

4.2 Структуры и перечисления

4.2.1 Структура FILE.FILE_DIR_ENTRY

Структура **FILE.FILE_DIR_ENTRY** описывает параметры каталога/файла и используется при работе с ФБ FILE.DirList.

Таблица 4.2.1 – Описание переменных структуры FILE.FILE_DIR_ENTRY

Название	Тип данных	Описание
sEntry	CAA.FILENAME	Имя каталога или файла
szSize	CAA.SIZE	Размер файла в байтах. В текущих версиях библиотеки (**3.5.17.0** и ниже) размер каталога не определяется – всегда возвращается **0**
xDirectory	BOOL	**TRUE** – объект является каталогом, **FALSE** – объект является файлом
xExclusive	BOOL	Тип доступа к каталогу/файлу: **TRUE** – только однопользовательский доступ **FALSE** – возможен многопользовательский доступ
dtLastModification	DT	Дата и время последнего изменения каталога/файла. В версии библиотеки ниже **3.5.16.0** некорректно определяется дата и время последнего изменения каталогов (**CDS-68177**)

4.2.2 Перечисление FILE.ERROR

Перечисление **FILE.ERROR** описывает ошибки, которые могут возникнуть во время вызова ФБ библиотеки.

Таблица 4.2.2 – Описание элементов перечисления FILE.ERROR

Название	Значение	Описание
NO_ERROR	0	Нет ошибок
TIME_OUT	5100	Истек лимит времени для данной операции
ABORT	5101	Операция была прервана с помощью входа **xAbort**
HANDLE_INVALID	5103	Некорректный дескриптор файла
NOT_EXIST	5104	Каталог или файл не существуют
EXIST	5105	Каталог или файл уже существуют
NO_MORE_ENTRIES	5106	Получена информация о всех вложенных элементах
NOT_EMPTY	5107	Каталог или файл не являются пустыми
READ_ONLY_CAA	5108	Каталог или файл защищены от записи
WRONG_PARAMETER	5109	ФБ вызван с неверными аргументами
WRITE_INCOMPLETE	5111	Запись в файл не была завершена (возможна потеря данных)
NOT_IMPLEMENTED	5112	Операция не поддерживается устройством
ASM_CREATEJOB_FAILED	5113	Ошибка асинхронной операции (ограничение – не более 20 асинхронных операций в пределах цикла задачи ПЛК; см. п. 4.4)
FILE_OPERATION_DENIED	5114	Доступ к файлу не разрешен на уровне конфигурационного файла CODESYS (**CODESYSControl.cfg**)

4.2.3 Перечисление FILE.MODE

Перечисление **FILE.MODE** описывает режим открытия файла.

Таблица 4.2.3 – Описание элементов перечисления FILE.MODE

Название	Значение	Описание
MWRITE	0	Запись (файл будет перезаписан или создан)
MREAD	1	Чтение (существующий файл будет открыт для чтения)
MRDWR	2	Чтение/запись (файл будет перезаписан или создан)
MAPPD	3	Дозапись (существующий файл будет открыт в режиме записи, данные будут дописаны в конец файла; если файла не существует – то будет возвращена ошибка **NOT_EXIST**)
MREADPLUS[6]	4	Чтение/запись с произвольной позиции (позиция определяются с помощью ФБ FILE.SetPos)
MWRITEPLUS[7]	5	Чтение/запись (файл будет перезаписан или создан). Режим идентичен **MRDWR**, добавлен для гармонизации названий режимов с библиотекой SysFile
MAPPENDPLUS[7]	6	Дозапись (существующий файл будет открыт в режиме записи, данные будут дописаны в конец файла; если файла не существует – то он будет создан)

4.3 Пути к каталогам и файлам

При использовании ФБ библиотеки в значительном числе случаев следует указывать путь к каталогу или файлу, над которым будет производиться операция. Общая информация о путях в Linux и ограничениях для их названий приведена в п. 2.4 и п. 2.5 соответственно.

При работе с библиотекой можно указывать как относительные, так и абсолютные пути, а также использовать заместители.

Пример: ФБ File.DirCreate создает новый каталог по пути **sDirName**.

- Если sDirName=’test1’, то результатом работы ФБ является создание каталога **test1** в рабочей директории;
- Если sDirName=’/mnt/ufs/home/test2’, то результатом работы ФБ является создание каталога **test2** в каталоге **/mnt/ufs/home/**.

В первом случае был использован относительный путь, во втором – абсолютный.

4.4 Ограничения при работе с файлами

Максимальное количество одновременно выполняемых асинхронных функциональных блоков (блоков библиотек **CAA File**, **CAA SerialCom**, **CAA NetBaseServices** и т. д.) не должно превышать **20-ти** (по возможности рекомендуется в каждый момент времени работать только с одним файлом)**.** Выполнением блока считается его вызов со значением **TRUE** на входе **xExecute**. В случае нарушения этого правила при попытке вызова 21-го асинхронно выполняемого блока на выходе ФБ возникает ошибка **ASM_CREATEJOB_FAILED** или (для версии библиотеки **3.5.11.0** и ниже) 5802.

Максимальный размер файла, с котором можно работать через библиотеку – 2 Гб.

[6] Данный режим добавлен в версии библиотеки 3.5.13.40.

[7] Данный режим добавлен в версии библиотеки 3.5.17.0.

4.5 ФБ работы с каталогами

4.5.1 ФБ FILE.DirCreate

Функциональный блок **FILE.DirCreate** создает новый каталог. Без указания полного пути каталог создается внутри рабочего каталога.

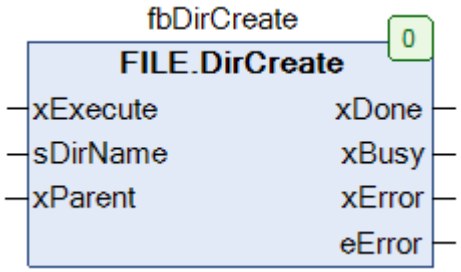


Рисунок 4.5.1 – Внешний вид ФБ FILE.DirCreate на языке CFC

Таблица 4.5.1 – Описание входов и выходов ФБ FILE.DirCreate

Название	Тип данных	Описание
Входные переменные		
xExecute	BOOL	Переменная активации блока. Запуск блока происходит по переднему фронту переменной
sDirName	STRING	Имя (или полный путь) создаваемого каталога. См. п. 2.4, п. 2.5 и п. 4.3
xParent	BOOL	Режим рекурсивного создания каталогов. **TRUE** – все несуществующие каталоги, указанные в пути, создаются автоматически **FALSE** – если в пути указан хотя бы один несуществующий каталог, то блок завершает работу с кодом ошибки **NOT_EXIST**
Выходные переменные		
xDone	BOOL	Флаг успешного завершения работы блока
xBusy	BOOL	Флаг «ФБ в процессе работы»
xError	BOOL	Флаг ошибки. Принимает значение **TRUE** в случае возникновения ошибки
eError	FILE.ERROR	Статус работы ФБ (или код ошибки)

4.5.2 ФБ FILE.DirOpen

Функциональный блок **FILE.DirOpen** открывает каталог и возвращает его дескриптор (**handle**), который требуется для последующего использования ФБ File.DirList.

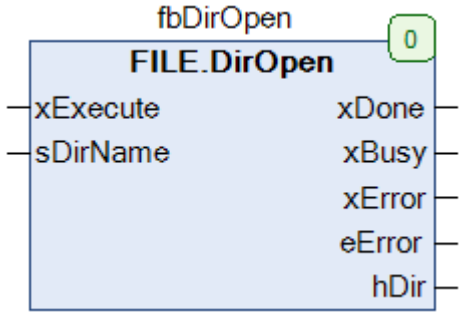


Рисунок 4.5.2 – Внешний вид ФБ FILE.DirOpen на языке CFC

Таблица 4.5.2 – Описание входов и выходов ФБ FILE.DirOpen

Название	Тип данных	Описание
Входные переменные		
xExecute	BOOL	Переменная активации блока. Запуск блока происходит по переднему фронту переменной
sDirName	STRING	Имя (или полный путь) открываемого каталога. См. п. 2.4, п. 2.5 и п. 4.3
Выходные переменные		
xDone	BOOL	Флаг успешного завершения работы блока
xBusy	BOOL	Флаг «ФБ в процессе работы»
xError	BOOL	Флаг ошибки. Принимает значение **TRUE** в случае возникновения ошибки
eError	FILE.ERROR	Статус работы ФБ (или код ошибки)
hDir	FILE.CAA.HANDLE	Дескриптор открытого каталога

4.5.3 ФБ FILE.DirList

Функциональный блок **FILE.DirList** возвращает информацию о каталоге по его дескриптору (**handle**). Предварительно каталог должен быть открыт с помощью ФБ FILE.DirOpen. Блок работает следующим образом: пока каталог открыт, каждый последующий вызов блока возвращает информацию о новом вложенном объекте (каталоге или файле). Если получена информация обо всех объектах, то при вызове блока на выходе **eError** возвращается ошибка **NO_MORE_ENTRIES**.

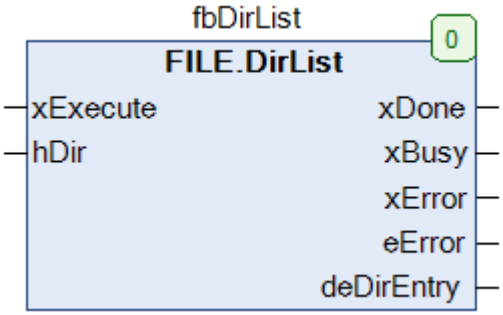


Рисунок 4.5.3 – Внешний вид ФБ FILE.DirList на языке CFC

Таблица 4.5.3 – Описание входов и выходов ФБ FILE.DirList

Название	Тип данных	Описание
Входные переменные		
xExecute	BOOL	Переменная активации блока. Запуск блока происходит по переднему фронту переменной
hDir	FILE.CAA.HANDLE	Дескриптор открытого каталога
Выходные переменные		
xDone	BOOL	Флаг успешного завершения работы блока
xBusy	BOOL	Флаг «ФБ в процессе работы»
xError	BOOL	Флаг ошибки. Принимает значение **TRUE** в случае возникновения ошибки.
eError	FILE.ERROR	Статус работы ФБ (или код ошибки)
deDirEntry	FILE.FILE_DIR_ENTRY	Информация о каталоге/файле

4.5.4 ФБ FILE.DirRemove

Функциональный блок **FILE.DirRemove** используется для удаления каталогов.

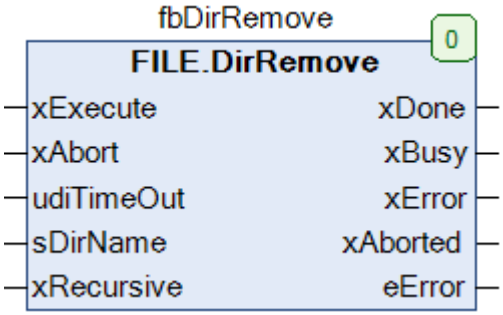


Рисунок 4.5.4 – Внешний вид ФБ FILE.DirRemove на языке CFC

Таблица 4.5.4 – Описание входов и выходов ФБ FILE.DirRemove

Название	Тип данных	Описание
Входные переменные		
xExecute	BOOL	Переменная активации блока. Запуск блока происходит по переднему фронту переменной
xAbort	BOOL	Переменная прерывания работы блока. Прерывание происходит по переднему фронту переменной
udiTimeOut	UDINT	Допустимое время операции (в мкс). Значение 0 означает, что время выполнения ФБ не ограничивается
sDirName	STRING	Имя (или полный путь) удаляемого каталога. См. п. 2.4, п. 2.5 и п. 4.3
xRecursive	BOOL	Режим рекурсивного удаления каталогов. **TRUE** – каталог удаляется вместе со всем содержимым **FALSE** – каталог удаляется только в том случае, если является пустым, в противном случае ФБ возвращает код ошибки **FILE_NOT_EMPTY**
Выходные переменные		
xDone	BOOL	Флаг успешного завершения работы блока
xBusy	BOOL	Флаг «ФБ в процессе работы»
xError	BOOL	Флаг ошибки. Принимает значение **TRUE** в случае возникновения ошибки
xAborted	BOOL	Флаг «прервано пользователем»
eError	FILE.ERROR	Статус работы ФБ (или код ошибки)

4.5.5 ФБ FILE.DirRename

Функциональный блок **FILE.DirRename** используется для переименования каталогов.

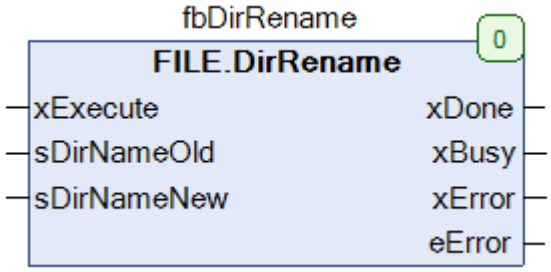


Рисунок 4.5.5 – Внешний вид ФБ FILE.DirRename на языке CFC

Таблица 4.5.5 – Описание входов и выходов ФБ FILE.DirRename

Название	Тип данных	Описание
Входные переменные		
xExecute	BOOL	Переменная активации блока. Запуск блока происходит по переднему фронту переменной
sDirNameOld	STRING	Текущее имя (или полный путь) каталога. См. п. 2.4, п. 2.5 и п. 4.3
sDirNameNew	STRING	Новое имя (или полный путь) каталога. См. п. 2.4, п. 2.5 и п. 4.3
Выходные переменные		
xDone	BOOL	Флаг успешного завершения работы блока
xBusy	BOOL	Флаг «ФБ в процессе работы»
xError	BOOL	Флаг ошибки. Принимает значение **TRUE** в случае возникновения ошибки
eError	FILE.ERROR	Статус работы ФБ (или код ошибки)

4.5.6 ФБ FILE.DirClose

Функциональный блок **FILE.DirClose** закрывает каталог. Данная операция производится после считывания информации о каталоге с помощью ФБ FILE.DirList.

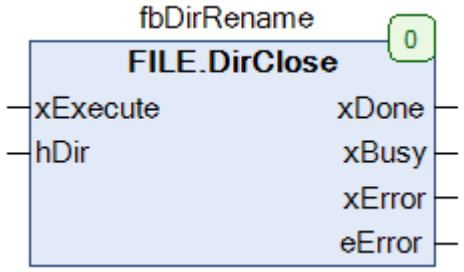


Рисунок 4.5.6 – Внешний вид ФБ FILE.DirClose на языке CFC

Таблица 4.5.6 – Описание входов и выходов ФБ FILE.DirClose

Название	Тип данных	Описание
Входные переменные		
xExecute	BOOL	Переменная активации блока. Запуск блока происходит по переднему фронту переменной
hDir	FILE.CAA.HANDLE	Дескриптор закрываемого каталога
Выходные переменные		
xDone	BOOL	Флаг успешного завершения работы блока
xBusy	BOOL	Флаг «ФБ в процессе работы»
xError	BOOL	Флаг ошибки. Принимает значение **TRUE** в случае возникновения ошибки
eError	FILE.ERROR	Статус работы ФБ (или код ошибки)

4.5.7 ФБ FILE.DirCopy[8]

Функциональный блок **FILE.DirCopy** используется для копирования содержимого каталогов.

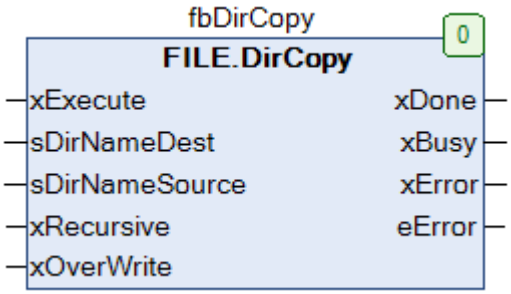


Рисунок 4.5.7 – Внешний вид ФБ FILE.DirCopy на языке CFC

Таблица 4.5.7 – Описание входов и выходов ФБ FILE.DirCopy

Название	Тип данных	Описание
Входные переменные		
xExecute	BOOL	Переменная активации блока. Запуск блока происходит по переднему фронту переменной
sDirNameDest	STRING	Путь к каталогу назначения. См. п. 2.4, п. 2.5 и п. 4.3
sDirNameSource	STRING	Путь к исходному каталогу. См. п. 2.4, п. 2.5 и п. 4.3
xRecursive	BOOL	**TRUE** – вложенные каталоги исходного каталога копируются в каталог назначения, **FALSE** – вложенные каталоги не копируются
xOverWrite	BOOL	**TRUE** – содержимое каталога назначения перезаписывается (в случае совпадения имен файлов), **FALSE** – не перезаписывается
Выходные переменные		
xDone	BOOL	Флаг успешного завершения работы блока
xBusy	BOOL	Флаг «ФБ в процессе работы»
xError	BOOL	Флаг ошибки. Принимает значение **TRUE** в случае возникновения ошибки
eError	FILE.ERROR	Статус работы ФБ (или код ошибки)

[8] Данный ФБ добавлен в версии библиотеки 3.5.13.40.

4.6 ФБ работы с файлами

4.6.1 ФБ FILE.Open

Функциональный блок **FILE.Open** открывает файл и возвращает его дескриптор (**handle**), который используется для всех остальных операций с файлом. После окончания работы с файлом следует закрыть его с помощью ФБ FILE.Close.

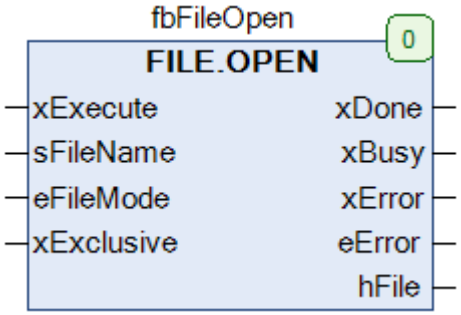


Рисунок 4.6.1 – Внешний вид ФБ FILE.Open на языке CFC

Таблица 4.6.1 – Описание входов и выходов ФБ FILE.Open

Название	Тип данных	Описание
Входные переменные		
xExecute	BOOL	Переменная активации блока. Запуск блока происходит по переднему фронту переменной
sFileName	STRING	Имя (или полный путь) открываемого файла. См. п. 2.4, п. 2.5 и п. 4.3
eFileMode	FILE.MODE	Режим открытия файла
xExclusive	BOOL	Тип доступа к открываемому файлу. **TRUE** – монопольный, **FALSE** – многопользовательский Фактически – этот вход был создан на этапе проектирования библиотеки, но не используется в ее реализации (**CDS-69925**)
Выходные переменные		
xDone	BOOL	Флаг успешного завершения работы блока
xBusy	BOOL	Флаг «ФБ в процессе работы»
xError	BOOL	Флаг ошибки. Принимает значение **TRUE** в случае возникновения ошибки
eError	FILE.ERROR	Статус работы ФБ (или код ошибки)
hFile	FILE.CAA.HANDLE	Дескриптор открытого файла

4.6.2 ФБ FILE.Close

Функциональный блок **FILE.Close** используется для закрытия файла после выполнения необходимых операций.

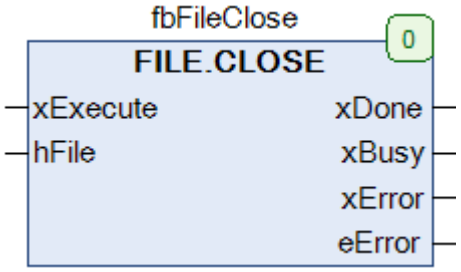


Рисунок 4.6.2 – Внешний вид ФБ FILE.Close на языке CFC

Таблица 4.6.3 – Описание входов и выходов ФБ FILE.Close

Название	Тип данных	Описание
Входные переменные		
xExecute	BOOL	Переменная активации блока. Запуск блока происходит по переднему фронту переменной
hFile	FILE.CAA.HANDLE	Дескриптор файла
Выходные переменные		
xDone	BOOL	Флаг успешного завершения работы блока
xBusy	BOOL	Флаг «ФБ в процессе работы»
xError	BOOL	Флаг ошибки. Принимает значение **TRUE** в случае возникновения ошибки
eError	FILE.ERROR	Статус работы ФБ (или код ошибки)

4.6.3 ФБ FILE.Write

Функциональный блок **FILE.Write** используется для записи данных в файл (точнее – в системный буфер, см. также ФБ FILE.Flush). Предварительно файл должен быть открыт с помощью ФБ FILE.Open.

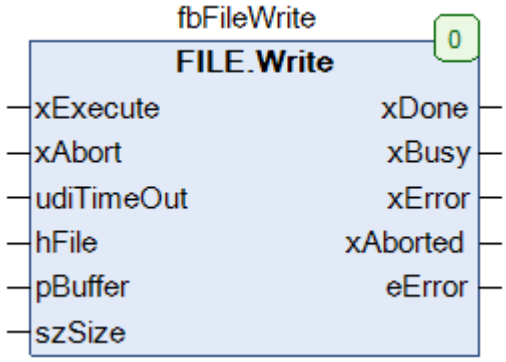


Рисунок 4.6.3 – Внешний вид ФБ FILE.Write на языке CFC

Таблица 4.6.3 – Описание входов и выходов ФБ FILE.Write

Название	Тип данных	Описание
Входные переменные		
xExecute	BOOL	Переменная активации блока. Запуск блока происходит по переднему фронту переменной
xAbort	BOOL	Переменная прерывания работы блока. Прерывание происходит по переднему фронту переменной
udiTimeOut	UDINT	Допустимое время операции (в мкс). Значение 0 означает, что время выполнения ФБ не ограничивается
hFile	FILE.CAA.HANDLE	Дескриптор файла
pBuffer	FILE.CAA.PVOID	Начальный адрес записываемых данных. Может быть указан с помощью оператора **ADR**
szSize	CAA.SIZE	Размер записываемых данных в байтах. Может быть указан с помощью оператора **SIZEOF**
Выходные переменные		
xDone	BOOL	Флаг успешного завершения работы блока
xBusy	BOOL	Флаг «ФБ в процессе работы»
xError	BOOL	Флаг ошибки. Принимает значение **TRUE** в случае возникновения ошибки
xAborted	BOOL	Флаг «прервано пользователем»
eError	FILE.ERROR	Статус работы ФБ (или код ошибки)

4.6.4 ФБ FILE.Read

Функциональный блок **FILE.Read** используется для чтения данных из файла. Предварительно файл должен быть открыт с помощью ФБ FILE.Open.

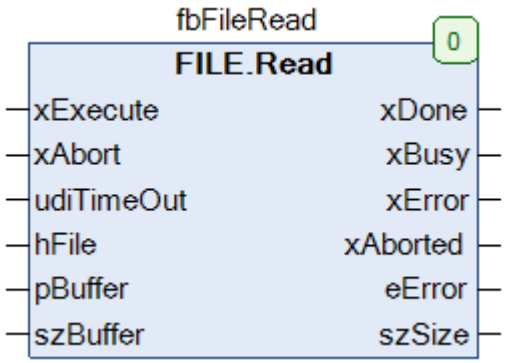


Рисунок 4.6.4 – Внешний вид ФБ FILE.Read на языке CFC

Таблица 4.6.4 – Описание входов и выходов ФБ FILE.Read

Название	Тип данных	Описание
Входные переменные		
xExecute	BOOL	Переменная активации блока. Запуск блока происходит по переднему фронту переменной
xAbort	BOOL	Переменная прерывания работы блока. Прерывание происходит по переднему фронту переменной
udiTimeOut	UDINT	Допустимое время операции (в мкс). Значение 0 означает, что время выполнения ФБ не ограничивается
hFile	FILE.CAA.HANDLE	Дескриптор файла
pBuffer	FILE.CAA.PVOID	Начальный адрес для размещения считанных данных. Может быть указан с помощью оператора **ADR**
szBuffer	CAA.SIZE	Размер считываемых данных в байтах. Может быть указан помощью оператора **SIZEOF**
Выходные переменные		
xDone	BOOL	Флаг успешного завершения работы блока
xBusy	BOOL	Флаг «ФБ в процессе работы»
xError	BOOL	Флаг ошибки. Принимает значение **TRUE** в случае возникновения ошибки
xAborted	BOOL	Флаг «прервано пользователем»
eError	FILE.ERROR	Статус работы ФБ (или код ошибки)
szSize	CAA.SIZE	Размер считанных данных в байтах

4.6.5 ФБ FILE.Rename

Функциональный блок **FILE.Rename** используется для переименования файлов.

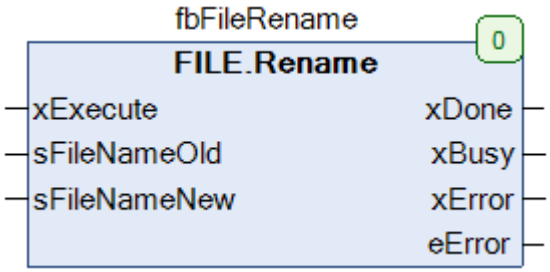


Рисунок 4.6.5 – Внешний вид ФБ FILE.Rename на языке CFC

Таблица 4.6.5 – Описание входов и выходов ФБ FILE.Rename

Название	Тип данных	Описание
Входные переменные		
xExecute	BOOL	Переменная активации блока. Запуск блока происходит по переднему фронту переменной
sFileNameOld	STRING	Текущее имя (или полный путь) файла. См. п. 2.4, п. 2.5 и п. 4.3
sFileNameNew	STRING	Новое имя (или полный путь) файла. См. п. 2.4, п. 2.5 и п. 4.3
Выходные переменные		
xDone	BOOL	Флаг успешного завершения работы блока
xBusy	BOOL	Флаг «ФБ в процессе работы»
xError	BOOL	Флаг ошибки. Принимает значение **TRUE** в случае возникновения ошибки
eError	FILE.ERROR	Статус работы ФБ (или код ошибки)

4.6.6 ФБ FILE.Copy

Функциональный блок **FILE.Copy** используется для копирования файлов.

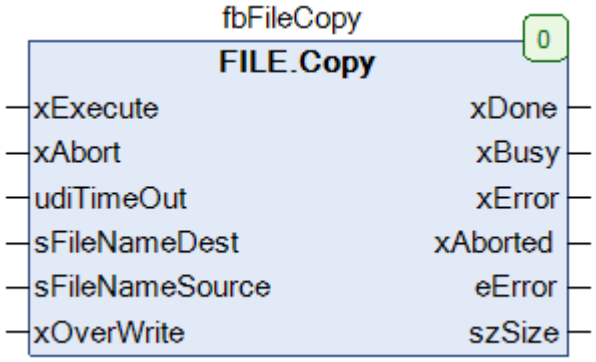


Рисунок 4.6.6 – Внешний вид ФБ FILE.Copy на языке CFC

Таблица 4.6.6 – Описание входов и выходов ФБ FILE.Copy

Название	Тип данных	Описание
Входные переменные		
xExecute	BOOL	Переменная активации блока. Запуск блока происходит по переднему фронту переменной
xAbort	BOOL	Переменная прерывания работы блока. Прерывание происходит по переднему фронту переменной
udiTimeOut	UDINT	Допустимое время операции (в мкс). Значение 0 означает, что время выполнения ФБ не ограничивается
sFileNameDest	STRING	Имя (или полный путь) копии файла. См. п. 2.4, п. 2.5 и п. 4.3
sFileNameSource	STRING	Имя (или полный путь) исходного файла. См. п. 2.4,п. 2.5 и п. 4.3
xOverWrite	BOOL	Обработка ситуации «файл с таким именем уже существует». **TRUE** – файл будет перезаписан **FALSE** – файл не будет перезаписан, блок вернет код ошибки **EXIST**
Выходные переменные		
xDone	BOOL	Флаг успешного завершения работы блока
xBusy	BOOL	Флаг «ФБ в процессе работы»
xError	BOOL	Флаг ошибки. Принимает значение **TRUE** в случае возникновения ошибки
xAborted	BOOL	Флаг «прервано пользователем»
eError	FILE.ERROR	Статус работы ФБ (или код ошибки)
szSize	CAA.FILE.SIZE	Размер скопированных данных в байтах

4.6.7 ФБ FILE.Delete

Функциональный блок **FILE.Delete** используется для удаления файлов.

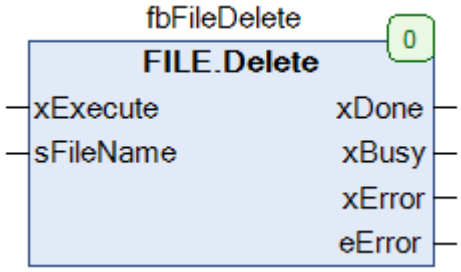


Рисунок 4.6.7 – Внешний вид ФБ FILE.Delete на языке CFC

Таблица 4.6.7 – Описание входов и выходов ФБ FILE.Delete

Название	Тип данных	Описание
Входные переменные		
xExecute	BOOL	Переменная активации блока. Запуск блока происходит по переднему фронту переменной
sFileName	STRING	Имя удаляемого файла
Выходные переменные		
xDone	BOOL	Флаг успешного завершения работы блока
xBusy	BOOL	Флаг «ФБ в процессе работы»
xError	BOOL	Флаг ошибки. Принимает значение **TRUE** в случае возникновения ошибки
eError	FILE.ERROR	Статус работы ФБ (или код ошибки)

4.6.8 ФБ FILE.Flush

Функциональный блок **FILE.Flush** используется для принудительной записи данных из системного буфера в файл. При работе ФБ FILE.Write данные сначала записываются в системный буфер, после чего ОС контроллера автоматически сохраняет их в файл. В редких специфических случаях (например, в случае возникновения в программе исключения или выключения питания сразу после вызова ФБ **FILE.Write**) сохранения данных в файл может не произойти. Использование ФБ **FILE.Flush** гарантирует, что данные сразу будут сохранены в файл. В то же время использование данной функции может привести к более быстрому истощению ресурса накопителя.

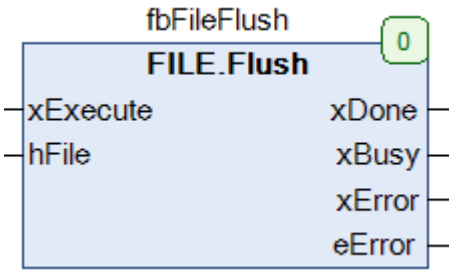


Рисунок 4.6.8 – Внешний вид ФБ FILE.Flush на языке CFC

Таблица 4.6.8 – Описание входов и выходов ФБ FILE.Flush

Название	Тип данных	Описание
Входные переменные		
xExecute	BOOL	Переменная активации блока. Запуск блока происходит по переднему фронту переменной
hFile	FILE.CAA.HANDLE	Дескриптор файла
Выходные переменные		
xDone	BOOL	Флаг успешного завершения работы блока
xBusy	BOOL	Флаг «ФБ в процессе работы»
xError	BOOL	Флаг ошибки. Принимает значение **TRUE** в случае возникновения ошибки.
eError	FILE.ERROR	Статус работы ФБ (или код ошибки)

4.6.9 ФБ FILE.GetPos

Функциональный блок **FILE.GetPos** используется для определения текущей установленной позиции в файле. Позиция представляет собой величину смещения в байтах от начала файла и используется для чтения/записи в выбранный фрагмент файла.

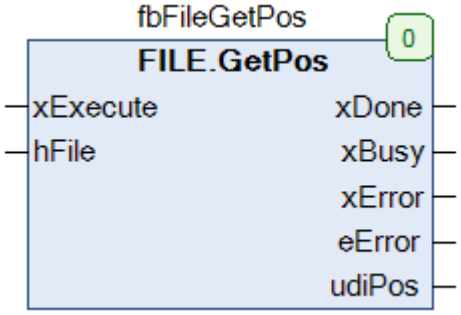


Рисунок 4.6.9 – Внешний вид ФБ FILE.GetPos на языке CFC

Таблица 4.6.9 – Описание входов и выходов ФБ FILE.GetPos

Название	Тип данных	Описание
Входные переменные		
xExecute	BOOL	Переменная активации блока. Запуск блока происходит по переднему фронту переменной
hFile	FILE.CAA.HANDLE	Дескриптор файла
Выходные переменные		
xDone	BOOL	Флаг успешного завершения работы блока
xBusy	BOOL	Флаг «ФБ в процессе работы»
xError	BOOL	Флаг ошибки. Принимает значение **TRUE** в случае возникновения ошибки
eError	FILE.ERROR	Статус работы ФБ (или код ошибки)
udiPos	UDINT	Текущая установленная позиция в файле (смещение относительно начала файла в байтах)

4.6.10 ФБ FILE.SetPos

Функциональный блок **FILE.SetPos** используется для установки позиции в файле. Позиция представляет собой величину смещения в байтах от начала файла и используется для чтения/записи в выбранный фрагмент файла.

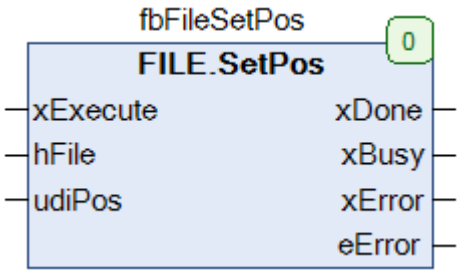


Рисунок 4.6.10 – Внешний вид ФБ FILE.SetPos на языке CFC

Таблица 4.6.10 – Описание входов и выходов ФБ FILE.SetPos

Название	Тип данных	Описание
Входные переменные		
xExecute	BOOL	Переменная активации блока. Запуск блока происходит по переднему фронту переменной
hFile	FILE.CAA.HANDLE	Дескриптор файла
udiPos	UDINT	Устанавливаемая позиция в файле (смещение относительно начала файла в байтах)
Выходные переменные		
xDone	BOOL	Флаг успешного завершения работы блока
xBusy	BOOL	Флаг «ФБ в процессе работы»
xError	BOOL	Флаг ошибки. Принимает значение **TRUE** в случае возникновения ошибки
eError	FILE.ERROR	Статус работы ФБ (или код ошибки)

4.6.11 ФБ FILE.EOF

Функциональный блок **FILE.EOF** используется для определения достижения конца файла. Конец файла считается достигнутым, если текущая установленная позиция совпадает с размером файла.

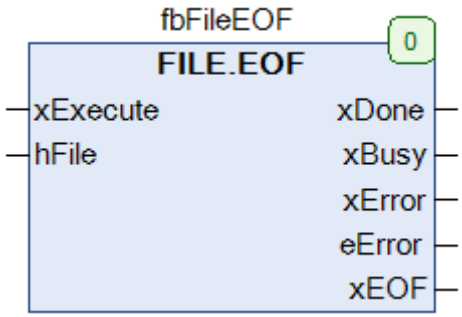


Рисунок 4.6.11 – Внешний вид ФБ FILE.EOF на языке CFC

Таблица 4.6.11 – Описание входов и выходов ФБ FILE.EOF

Название	Тип данных	Описание
Входные переменные		
xExecute	BOOL	Переменная активации блока. Запуск блока происходит по переднему фронту переменной
hFile	FILE.CAA.HANDLE	Дескриптор файла
Выходные переменные		
xDone	BOOL	Флаг успешного завершения работы блока
xBusy	BOOL	Флаг «ФБ в процессе работы»
xError	BOOL	Флаг ошибки. Принимает значение **TRUE** в случае возникновения ошибки
eError	FILE.ERROR	Статус работы ФБ (или код ошибки)
xEOF	BOOL	**TRUE** – достигнут конец файл **FALSE** – конец файла не достигнут

4.6.12 ФБ FILE.GetSize

Функциональный блок **FILE.GetSize** используется для определения размера файла.

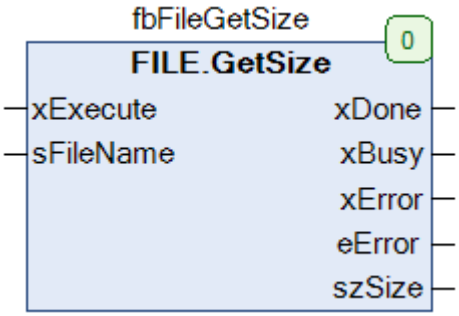


Рисунок 4.6.12 – Внешний вид ФБ FILE.GetSize на языке CFC

Таблица 4.6.12 – Описание входов и выходов ФБ FILE.GetSize

Название	Тип данных	Описание
Входные переменные		
xExecute	BOOL	Переменная активации блока. Запуск блока происходит по переднему фронту переменной
hFile	FILE.CAA.HANDLE	Дескриптор файла
Выходные переменные		
xDone	BOOL	Флаг успешного завершения работы блока
xBusy	BOOL	Флаг «ФБ в процессе работы»
xError	BOOL	Флаг ошибки. Принимает значение **TRUE** в случае возникновения ошибки
eError	FILE.ERROR	Статус работы ФБ (или код ошибки)
szSize	FILE.CAA.SIZE	Размер файла в байтах

4.6.13 ФБ FILE.GetTime

Функциональный блок **FILE.GetTime** используется для определения времени последнего изменения файла.

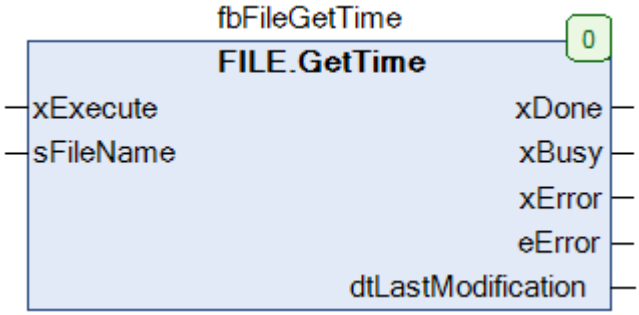


Рисунок 4.6.13 – Внешний вид ФБ FILE.GetTime на языке CFC

Таблица 4.6.13 – Описание входов и выходов ФБ FILE.GetTime

Название	Тип данных	Описание
Входные переменные		
xExecute	BOOL	Переменная активации блока. Запуск блока происходит по переднему фронту переменной
sFileName	STRING	Имя (или полный путь) файла. См. п. 2.4, п. 2.5 и п. 4.3
Выходные переменные		
xDone	BOOL	Флаг успешного завершения работы блока
xBusy	BOOL	Флаг «ФБ в процессе работы»
xError	BOOL	Флаг ошибки. Принимает значение **TRUE** в случае возникновения ошибки
eError	FILE.ERROR	Статус работы ФБ (или код ошибки)
dtLastModification	DT	Дата и время последнего изменения файла

5 Пример работы с библиотекой CAA File

5.1 Краткое описание примера

Описанный в данном пункте пример демонстрирует работу с библиотекой **CAA File** и реализацию следующего функционала:

- операции с каталогами (создание, удаление, переименование, копирование, просмотр содержимого);
- запись и чтение бинарных файлов;
- запись и чтение текстовых файлов;
- другие операции с файлами (переименование, копирование, удаление).

Основной программой примера является программа PLC_PRG, привязанная к задаче **MainTask**. В этой программе выполняется вызов программ примера:

- DIR_PRG – программа, в которой производится работа с каталогами. Содержит методы prvDirList и prvRemoveLastDirFromPath;
- FILE_PRG – программа, в которой производится работа с файлами.

Пример создан в среде **CODESYS V3.5 SP17 Patch 3** и подразумевает запуск на **СПК1xx [M01]** с таргет-файлом **3.5.17.31.** В случае необходимости запуска проекта на другом устройстве следует изменить таргет-файл в проекте (**ПКМ** на узел **Device** – **Обновить устройство).**

Пример доступен для скачивания: Example_CaaFile_3517v1.projectarchive

5.2 Использованные библиотеки

Помимо **CAA File** в примере используются следующие библиотеки (*если вы повторяете примеры «с нуля», то они должны быть добавлены в менеджере библиотек вашего проекта*):

Таблица 5.2.1 – Описание библиотек примера

Библиотека	Пространство имен	Объекты библиотеки, используемые в примере
CAA Memory	MEM	Функции для работы с блоками памяти (копирование, заполнение и т. д).
OwenStringUtils	OSU	Расширенные функции для работы со строками. Библиотека не входит в дистрибутив CODESYS и должна быть загружена с сайта ОВЕН (раздел CODESYS V3)
OwenVisuDialogs	OVD	Диалоги для визуализации (в частности, диалог отображения информации о памяти контроллера и подключенных накопителей). Библиотека не входит в дистрибутив CODESYS и должна быть загружена с сайта ОВЕН (раздел CODESYS V3)
Standard	Standard	Базовые функции для работы со строками и детекторы фронтов
Util	Util	Функция сборки метки времени из отдельных разрядов
OwenTypes	OwenTypes	Структуры узлов таргет-файла (OwenRTC, Drives и т. д.)

5.3 Перечисления и структуры

5.3.1 Перечисление FILE_DEVICE

Перечисление **FILE_DEVICE** описывает файловое устройство (накопитель или директорию в памяти контроллера).

Таблица 5.3.1 – Описание элементов перечисления FILE_DEVICE

Название	Значение	Описание
CDS_DIR	0	Рабочая директория CODESYS
USB	1	USB-накопитель
SD	2	SD-накопитель
FTP_DIR	3	Рабочая директория FTP-сервера

5.3.2 Перечисление USER_FILE_MODE

Перечисление **USER_FILE_MODE** описывает операции, поддерживаемые ФБ FileManager.

Таблица 5.3.2 – Описание элементов перечисления USER_FILE_MODE

Название	Значение	Описание
READ	0	Чтение из файла с заданной позиции
WRITE	1	Запись в файл с заданной позиции
APPEND	2	Дозапись в конец файла

5.3.3 Перечисление STATE

Перечисление **STATE** описывает шаги машины состояний ФБ FileManager и метода prvDirList программы DIR_PRG.

Таблица 5.3.3 – Описание элементов перечисления STATE

Название	Значение	Описание
IDLE	0	Ожидание команды
OPEN	1	Открытие или создание файла
SET_POS	2	Установка позиции для чтения или записи
READ	3	Чтение из файла
WRITE	4	Запись в файл
GET_SIZE	5	Определение размера файла
CLOSE	6	Закрытие файла

5.3.4 Структура ARCHIVE_RECORD

Структура **ARCHIVE_RECORD** описывает запись архива, используемую в ФБ FileManager.

Таблица 5.3.4 – Описание переменных структуры ARCHIVE_RECORD

Название	Тип данных	Описание
dtTimeStamp	DT	Метка времени
iVar	INT	Значение типа INT
rVar	REAL	Значение типа REAL

5.3.5 Структура VISU_DIR_INFO

Структура **VISU_DIR_INFO** описывает один вложенный объект каталога. Используется в программе DIR_PRG. Представляет собой версию структуры FILE.FILE_DIR_ENTRY с тем отличием, что в качестве типов используются строки (для отображения в визуализации в удобном пользователю виде).

Таблица 5.3.5 – Описание переменных структуры VISU_DIR_INFO

Название	Тип данных	Описание
sEntryName	STRING	Название объекта
wsEntryType	WSTRING(20)	Тип объекта (каталог или файл)
wsEntrySize	WSTRING(20)	Размер объекта в байтах
sLastModification	STRING(20)	Дата и время последнего изменения объекта

5.4 Функции

5.4.1 Функция BYTE_SIZE_TO_WSTRING

Функция **BYTE_SIZE_TO_WSTRING** конвертирует число байт **uliByteSize** в форматированную строку с наиболее подходящими единицами измерения и указанием размерности (например, для 305253 байт будет сформирована строка «298.09 Кбайт».

Таблица 5.4.1 – Описание переменных функции BYTE_SIZE_TO_WSTRING

Название	Тип	Описание
Входы		
uliByteSize	ULINT	Число байт
Выходы		
BYTE_SIZE_TO_WSTRING	WSTRING	Число байт в виде форматированной строки
Локальные переменные		
rByteSize	REAL	Промежуточная переменная
Константы		
c_uliBytePerKb	ULINT	Число байт в килобайте
c_uliKbPerMb	ULINT	Число килобайт в мегабайте
c_uliMbPerGb	ULINT	Число мегабайт в гигабайте
c_usiMaxDirSizeInGb	USINT	Максимальный размер каталога (выбран эмпирически)
c_usiFloatDigitCount	USINT	Число знаков после запятой в отображаемом значении
c_wsByteText	WSTRING(10)	Тексты для размерностей
c_wsKbText	WSTRING(10)	
c_wsMbText	WSTRING(10)	
c_wsGbText	WSTRING(10)	

```
// Функция преобразования числа байт в форматированную строку
// Значение отображается в наиболее подходящих единицах
// (например, 700 - в виде байт, а 1100 - килобайт)
FUNCTION BYTE_SIZE_TO_WSTRING : WSTRING
VAR_INPUT
   // Число байт
   uliByteSize:                          ULINT;

END_VAR
VAR
   // Промежуточная переменная
   rByteSize:                            REAL;

END_VAR
VAR CONSTANT
   // Число байт в килобайте
   c_uliBytePerKb:                       ULINT           := 1024;
   // Число килобайт в мегабайте
   c_uliKbPerMb:                         ULINT           := 1024 * c_uliBytePerKb;
   // Число мегабайт в гигабайте
   c_uliMbPerGb:                         ULINT           := 1024 * c_uliKbPerMb;
   // Максимальный размер каталога (выбран эмпирически)
   c_usiMaxDirSizeInGb:                  USINT           := 32;
   // Число знаков после запятой в отображаемом значении
   c_usiFloatDigitCount:                 USINT           := 2;
   // Текст для размерностей
   c_wsByteText:                         WSTRING(10) := " Байт";
   c_wsKbText:                           WSTRING(10) := " Кбайт";
   c_wsMbText:                           WSTRING(10) := " Мбайт";
   c_wsGbText:                           WSTRING(10) := " Гбайт";
END_VAR


CASE uliByteSize OF

   0  ..(c_uliBytePerKb - 1):

      BYTE_SIZE_TO_WSTRING := WCONCAT(ULINT_TO_WSTRING(uliByteSize), c_wsByteText);

   c_uliBytePerKb ..(c_uliKbPerMb - 1):

      rByteSize := ULINT_TO_REAL(uliByteSize) / ULINT_TO_REAL(c_uliBytePerKb);
      BYTE_SIZE_TO_WSTRING := WCONCAT(TO_WSTRING(OSU.REAL_TO_STRING_FORMAT(rByteSize,
         c_usiFloatDigitCount, OSU.DECIMAL_SEPARATOR.DOT) ), c_wsKbText);

   c_uliKbPerMb ..(c_uliMbPerGb - 1):

      rByteSize  := ULINT_TO_REAL(uliByteSize) / ULINT_TO_REAL(c_uliKbPerMb);
      BYTE_SIZE_TO_WSTRING := WCONCAT(TO_WSTRING(OSU.REAL_TO_STRING_FORMAT(rByteSize,
         c_usiFloatDigitCount, OSU.DECIMAL_SEPARATOR.DOT) ), c_wsMbText);

   c_uliMbPerGb ..(c_usiMaxDirSizeInGb * c_uliMbPerGb):

      rByteSize  := ULINT_TO_REAL(uliByteSize) / ULINT_TO_REAL(c_uliMbPerGb);
      BYTE_SIZE_TO_WSTRING := WCONCAT(TO_WSTRING(OSU.REAL_TO_STRING_FORMAT(rByteSize,
         c_usiFloatDigitCount, OSU.DECIMAL_SEPARATOR.DOT) ), c_wsGbText);

END_CASE
```

5.4.2 Функция GetPathToFileDevice

Функция **GetPathToFileDevice** возвращает заполнитель файлового устройства.

Таблица 5.4.2 – Описание переменных функции GetPathToFileDevice

Название	Тип	Описание
Входы		
eDevice	FILE_DEVICE	Файловое устройство
Выходы		
GetPathToFileDevice	STRING	Заполнитель устройства

```
// Функция возвращает путь для файловой системы контроллера/накопителя по ID
FUNCTION GetPathToFileDevice : STRING
VAR_INPUT
   // ID устройства
   eDevice:   FILE_DEVICE;
END_VAR
VAR
END_VAR


CASE eDevice OF
   FILE_DEVICE.CDS_DIR:
      GetPathToFileDevice := '';
   FILE_DEVICE.USB:
      GetPathToFileDevice := '$$USB$$/';
   FILE_DEVICE.SD:
      GetPathToFileDevice := '$$SD$$/';
   FILE_DEVICE.FTP:
      GetPathToFileDevice := '$$FTP$$/';
END_CASE
```

5.5 Функциональные блоки

5.5.1 ФБ FileManager

Функциональный блок **FileManager** используется для чтения и записи файлов. Данные сохраняются либо в бинарном формате, либо (если вход **xIsTextFormat** имеет значение **TRUE**) – в формате .csv. Файлы состоят из записей. Запись представляет собой набор данных, определяемый структурой ARCHIVE_RECORD. Каждая операция с файлом представляет собой добавление или считывание одной записи из файла.

Блок представляет собой «обертку» над блоками библиотеки CAA File.

Алгоритм работы блока:

По переднему фронту на входе **xExecute** происходит выполнение операции **eFileMode** (чтение, перезапись или дозапись в конце файла) с файлом, который размещен по пути **sFileName**. Вход **xIsTextFormat** определяет формат файла (**FALSE** – бинарный, **TRUE** – .csv). Записываемые данные передаются на вход **stWriteRecord**. Номер записи, требуемый для операций чтения и перезаписи, передается на вход **udiRecordNumber**.

В случае успешного завершения операции выход **xDone** принимает значение **TRUE**. Если в процессе выполнения операции возникла какая-либо ошибка – выход **xError** принимает значение **TRUE**, а выход **eError** содержит код ошибки. В случае выполнения операции чтения считанные данные размещаются на выходе **stReadRecord**. На выходе **udiRecordCount** отображается текущее число записей в файле (обновляется после каждой операции записи).

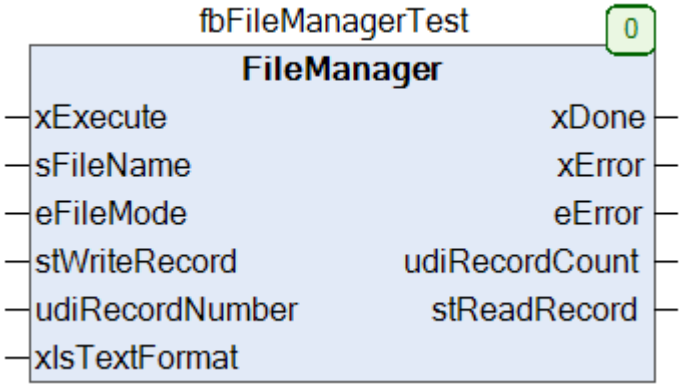


Рисунок 5.5.1 – Внешний вид ФБ FileManager на языке CFC

Ниже приведено описание входов/выходов/локальных переменных/констант блока и его листинг с комментариями к реализации.

Таблица 5.5.1 – Описание переменных ФБ FileManager

Название	Тип	Описание
Входы		
xExecute	BOOL	По переднему фронту происходит однократное выполнение операции с файлом
sFileName	STRING	Путь к файлу
eFileMode	USER_FILE_MODE	Выполняемая операция
stWriteRecord	ARCHIVE_RECORD	Записываемые данные (для **eFileMode** = **USER_FILE_MODE.WRITE** и .**APPEND**)
udiRecordNumber	UDINT	Номер записи (для **eFileMode** = **USER_FILE_MODE.READ** и .**WRITE**)
xIsTextFormat	BOOL	Формат архива (**TRUE** – текстовый, **FALSE** – бинарный)
Выходы		
xDone	BOOL	**TRUE** – операция успешно завершена
xError	BOOL	Принимает значение **TRUE** в случае возникновения ошибки
eError	FILE.ERROR	Статус работы ФБ (или код ошибки)
udiRecordCount	UDINT	Число записей в файле
stReadRecord	ARCHIVE_RECORD	Считанная запись (для **eFileMode** = **USER_FILE_MODE.READ**)
Локальные переменные		
eState	STATE	Текущий шаг машины состояний блока
eMode	FILE.MODE	Режим открытия файла
fbExecute	R_TRIG	Детектор импульса для входа **xExecute**
xResetOutputs	BOOL	Команда сброса выходов блока
fbFileOpen	FILE.Open	ФБ открытия файла
fbFileSetPos	FILE.SetPos	ФБ установки позиции в файле
fbFileRead	FILE.Read	ФБ чтения из файла
fbFileWrite	FILE.Write	ФБ записи в файл
fbFileGetSize	FILE.GetSize	ФБ определения размера файла
fbFileClose	FILE.Close	ФБ закрытия файла
hFile	FILE.CAA.HANDLE	Дескриптор файла
xCreateRecord	BOOL	Команда формирования новой записи
sRecord	STRING (2*c_usiTextRecordSize)	Буфер текстовой записи (рассчитан на 2 записи, так как при создании файла в него записывается заголовок и первая запись)
udiPos	UDINT	Позиция для записи в файл
usiTextRecordSize	USINT	Размер записываемых текстовых данных (зависит от того, записывается ли заголовок)

Таблица 5.5.2 – Описание констант ФБ FileManager

Название	Тип	Значение	Описание
c_sRecordSeparator	STRING(1)	';'	Разделитель значений в записи
c_usiTextRecordSize	USINT	80	Размер одной текстовой записи (число ее символов)
c_iRecordElemCount	INT	3	Число элементов записи (число полей структуры ARCHIVE_RECORD
c_sTitle	STRING	'Дата и время;Счетчик;Температура'	Заголовок файла
c_sEndOfLineChars	STRING(2)	'rn'	Символы перехода на новую строку (см. п. 2.6)

```
// ФБ для чтения/записи в файл структуры ARCHIVE_RECORD
// Поддерживает чтение/запись данных с произвольной позиции,
// а также дозапись в конец файла
// Поддерживаются бинарный и текстовый (.csv) формат файла
FUNCTION_BLOCK FileManager
VAR_INPUT
   // По переднему фронту происходит однократное выполнение операции eFileMode
   xExecute:                    BOOL;
   // Путь к файлу
   sFileName:                   STRING;
   // Операция, выполняемая с файлом
   eFileMode:                   USER_FILE_MODE;
   // Данные для записи в файл (для операций WRITE и APPEND)
   stWriteRecord:               ARCHIVE_RECORD;
   // Номер записи (для операций READ и WRITE)
   udiRecordNumber:             UDINT;
   // Формат архива (TRUE - текстовый, FALSE - бинарный)
   // Не должен изменяться после первого вызова экземпляра блока
   // (чтобы не нарушить структуру архива)
   xIsTextFormat:               BOOL;
END_VAR
VAR_OUTPUT
   // Флаг успешного завершения работы
   xDone:                       BOOL;
   // Флаг ошибки
   xError:                      BOOL;
   // Код ошибки
   eError:                      FILE.ERROR;
   // Текущее число записей в файле (выход обновляется после операций WRITE и APPEND)
   udiRecordCount:              UDINT;
   // Считанная запись (для операции READ)
   stReadRecord:                ARCHIVE_RECORD;
END_VAR
VAR
   // Текущий шаг машины состояний
   eState:                      STATE;
   // Режим открытия файла
   eMode:                       FILE.MODE;
   // Детектор импульса для входа xExecute
   fbExecute:                   R_TRIG;
   // Команда сброса выходов блока
   xResetOutputs:               BOOL;
   // ФБ для работы с файлами из библиотеки CAA FIle
   fbFileOpen:                  FILE.Open;
   fbFileSetPos:                FILE.SetPos;
   fbFileRead:                  FILE.Read;
   fbFileWrite:                 FILE.Write;
   fbFileGetSize:               FILE.GetSize;
   fbFileClose:                 FILE.Close;
   // Дескриптор (хэндл) открытого файла
   hFile:                       FILE.CAA.HANDLE;
   // Команда формирования новой записи
   xCreateRecord:               BOOL;
```

```
   // Буфер записи (рассчитан на 2 записи, так как при создании файла
   // в него записывается заголовок и первая запись)
   sRecord:                      STRING(2 * c_usiTextRecordSize);
   // Позиция для записи в файл
   udiPos:                       UDINT;
   // Размер записываемых текстовых данных (зависит от того, записывается ли заголовок)
   usiTextRecordSize:            USINT;
END_VAR
VAR CONSTANT
   // Разделитель значений в записи
   c_sRecordSeparator:           STRING(1) := ';';
   // Размер одной текстовой записи (число ее символов)
   c_usiTextRecordSize:          USINT     := 80;
   // Число элементов записи (число полей структуры ARCHIVE_RECORD)
   c_iRecordElemCount:           INT       := 3;
   // Заголовок файла
   c_sTitle:                     STRING    := 'Дата и время;Счетчик;Температура';
   // Символы перехода на новую строку
   c_sEndOfLineChars:            STRING(2) := '$r$n';
END_VAR


CASE eState OF

   // шаг ожидания новой команды
   STATE.IDLE:

      // сброс ошибок предыдущего вызова
      IF xResetOutputs AND NOT(xExecute) THEN
         prvResetOutputs();
      END_IF

      // детектируем команду запуска блока
      fbExecute(CLK := xExecute);

      IF fbExecute.Q THEN

         eState := STATE.OPEN;

         // в зависимости от выбранной пользователем операции определяем
         // режим открытия файла
         IF eFileMode = USER_FILE_MODE.READ OR eFileMode = USER_FILE_MODE.WRITE THEN
            eMode := FILE.MODE.MREADPLUS;
         ELSIF eFileMode = USER_FILE_MODE.APPEND THEN
            eMode := FILE.MODE.MAPPD;
         ELSE
            // значение на входе eFileMode не соответствует
            // диапазону перечисления USER_FILE_MODE
            xError := TRUE;
            eError := FILE.ERROR.WRONG_PARAMETER;
            xResetOutputs := TRUE;
         END_IF

      END_IF

   // шаг открытия файла
   STATE.OPEN:

      fbFileOpen
      (
         xExecute  := TRUE,
         sFileName := sFileName,
         eFileMode := eMode
      );

      IF fbFileOpen.xDone THEN

         hFile  := fbFileOpen.hFile;
```

```
         IF eFileMode = USER_FILE_MODE.READ OR eFileMode = USER_FILE_MODE.WRITE THEN
            // для операции чтения или записи - сначала установим позицию
            eState := STATE.SET_POS;
         ELSIF eFileMode = USER_FILE_MODE.APPEND OR xCreateRecord THEN
            // для операции дозаписи в конец файла или записи заголовка архива
            // в начало файла - сразу переходим на шаг записи
            eState := STATE.WRITE;
         END_IF

      // если файла нет - надо его создать
      ELSIF fbFileOpen.eError = FILE.ERROR.NOT_EXIST THEN

         fbFileOpen(xExecute := FALSE);
        // формируем заголовок архива для архива текстового формата
        IF xIsTextFormat THEN
           xCreateRecord := prvCreateRecord(xWriteTitle := TRUE,
              usiTextRecordSize => usiTextRecordSize);
         ELSE
         // для бинарного архива - заголовок не нужен,
         // а флаг xCreateRecord нужен, чтобы перейти на шаг записи
         // (см. перед началом ELSIF)
            xCreateRecord := TRUE;
         END_IF

         // выбираем режим для создания файла
         // и остаемся на этом же шаге для его создания файла
         eMode := FILE.MODE.MWRITE;

      ELSIF fbFileOpen.xError THEN

          prvSwitchToIdle();
          eError := fbFileOpen.eError;

     END_IF

// шаг установки позиции в файле (для операций READ и WRITE)
STATE.SET_POS:

   // позиция - это смещение в байтах от начала файла
   // определяем ее по номеру записи и ее размеру
   IF xIsTextFormat THEN
        udiPos := udiRecordNumber * c_usiTextRecordSize;
   ELSE
        udiPos := udiRecordNumber * SIZEOF(ARCHIVE_RECORD);
   END_IF

   fbFileSetPos
   (
        xExecute := TRUE,
        hFile    := hFile,
        udiPos   := udiPos
   );

   IF fbFileSetPos.xDone THEN

      IF eFileMode = USER_FILE_MODE.READ THEN
         eState := STATE.READ;
      ELSIF eFileMode = USER_FILE_MODE.WRITE THEN
         eState := STATE.WRITE;
      END_IF

   ELSIF fbFileSetPos.xError THEN

      prvSwitchToIdle();
      eError := fbFileSetPos.eError;

   END_IF
```

```
// шаг записи в файл (для операций WRITE и APPEND)
STATE.WRITE:

   IF xIsTextFormat THEN

      // однократно формируем записываемую строку
      IF NOT(xCreateRecord) THEN

      xCreateRecord := prvCreateRecord(xWriteTitle := FALSE,
         usiTextRecordSize => usiTextRecordSize);

     END_IF

      fbFileWrite
      (
         xExecute := TRUE,
         hFile    := hFile,
         pBuffer  := ADR(sRecord),
         szSize   := usiTextRecordSize
      );

   ELSE

      fbFileWrite
      (
         xExecute := TRUE,
         hFile    := hFile,
         pBuffer  := ADR(stWriteRecord),
         szSize   := SIZEOF(stWriteRecord)
      );

   END_IF

   // после записи в файл пересчитываем его размер
   IF fbFileWrite.xDone THEN

      eState := STATE.GET_SIZE;

   ELSIF fbFileWrite.xError THEN

      prvSwitchToIdle();
      eError := fbFileWrite.eError;

   END_IF

// шаг чтения из файла (для операции READ)
STATE.READ:

   IF xIsTextFormat THEN

      fbFileRead
      (
         xExecute := TRUE,
         hFile    := hFile,
         pBuffer  := ADR(sRecord),
         szBuffer := c_usiTextRecordSize
      );

   ELSE

      fbFileRead
      (
         xExecute := TRUE,
         hFile    := hFile,
         pBuffer  := ADR(stReadRecord),
         szBuffer := SIZEOF(stReadRecord)
      );

   END_IF
```

```
      IF fbFileRead.xDone THEN
         // конвертируем строку файла в структуру
         IF xIsTextFormat THEN
            stReadRecord := prvStringToRecord();
         END_IF

         eState := STATE.CLOSE;

      ELSIF fbFileRead.xError THEN

         prvSwitchToIdle();
         eError := fbFileRead.eError;

      END_IF

   // шаг определения размера файла (для операций WRITE и APPEND)
   STATE.GET_SIZE:

      fbFileGetSize
      (
         xExecute  := TRUE,
         sFileName := sFileName
      );

      IF fbFileGetSize.xDone THEN

         // рассчитываем число записей на основании размера файла и размера одной записи
         IF xIsTextFormat THEN
            udiRecordCount := fbFileGetSize.szSize / c_usiTextRecordSize;
         ELSE
           udiRecordCount := fbFileGetSize.szSize / SIZEOF(ARCHIVE_RECORD);
         END_IF

         eState := STATE.CLOSE;

      ELSIF fbFileSetPos.xError THEN

         prvSwitchToIdle();
         eError := fbFileGetSize.eError;

      END_IF

   // шаг закрытия файла
   STATE.CLOSE:

      fbFileClose
      (
         xExecute  := TRUE,
         hFile     := hFile
      );

      // выполнение блока успешно завершено
      IF fbFileClose.xDone THEN

         xDone         := TRUE;
         xResetOutputs := TRUE;
         eState        := STATE.IDLE;

      ELSIF fbFileClose.xError THEN

         prvSwitchToIdle();
         eError := fbFileClose.eError;

      END_IF

END_CASE
```

Основой блока является машина состояний, реализованная в операторе **CASE**.

На шаге **IDLE** происходит ожидание новой команды (переднего фронта входа **xExecute**). В случае детектирования фронта происходит переход на шаг открытия файла **OPEN**. Вместе с этим происходит определение режима открытия файла: в зависимости от выбранной операции (вход **eFileMode** типа USER_FILE_MODE) определяется режим открытия файла **eMode** (тип FILE.MODE). Для операций чтения (**USER_FILE_MODE.READ**) и записи (**USER_FILE_MODE.WRITE**) выбирается режим чтения/записи **FILE.MODE.MREADPLUS**, для операции дозаписи (**USER_FILE_MODE.APPEND**) – режим дозаписи (**FILE.MODE.MAPPD**). Это позволяет скрыть от пользователя блока список режимов открытия файла (в версии библиотеки **3.5.17.0** – их 7), оставив ему только выбор операции из перечисления USER_FILE_MODE (**READ/WRITE/APPEND**).

Если же на входе **eFileMode** обнаружено значение, не соответствующее ни одной из трех операций (например, это может произойти при некорректной работе с памятью), то блок формирует ошибку **WRONG_PARAMETER** и остается на шаге **IDLE**.

На шаге **IDLE** также выполняется сброс ошибок. Сброс производится в случае формирования команды на сброс **xResetOutputs** (она формируется на других шагах машины состояний блока) и только если вход **xExecute** имеет значение **FALSE** (пока он имеет значение **TRUE** – на выходах блока сохраняется информация о последней ошибке; это диктуется моделью поведения CAA Behaviour Model). Операции сброса оформлены в виде метода prvResetOutputs.

На шаге **OPEN** выполняется открытие файла. Если файл успешно открыт (выход **xDone** экземпляра ФБ FILE.Open имеет значение **TRUE**), то происходит переход на шаг записи в файл **WRITE** (для операции **APPEND**) или шаг установки позиции в файле **SET_POS** (для операций **READ** и **WRITE**).

В режиме **APPEND** возможна совершенно корректная ситуация, когда файл еще не существует, и требуется его создать. Попытка открытия несуществующего файла в режиме **FILE.MODE.MAPPD** приведет к появлению на выходе **eError** экземпляра ФБ FILE.Open ошибки **FILE_NOT_EXIST**. В этом случае выбирается режим открытия файла **FILE.MODE.MWRITE**, в котором попытка открытия несуществующего файла приводит к его созданию. Кроме того, для текстового архива (**xIsTextFormat**) формируется первая запись, включающая в себя строку заголовка (с помощью метода prvCreateRecord). Перехода на другой шаг в этом случае не происходит – в следующем цикле задачи ПЛК на этом же шаге произойдет новый вызов экземпляра ФБ FILE.Open с режимом открытия **FILE.MODE.MWRITE**.[9]

Если же в процессе открытия файла возникла другая ошибка – то происходит возвращение на шаг **IDLE** с помощью метода prvSwitchToIdle.

На шаге **SET_POS** происходит определение позиции (смещения в байтах) в файле для операций **READ** и **WRITE**. Позиция зависит от номера записи, выбранной пользователем (вход **udiRecordNumber**) и размера записи. Для бинарного формата архива (**xIsTextFormat** = **FALSE**) размер записи равен размеру структуры ARCHIVE_RECORD. Для текстового формата (**xIsTextFormat** = **TRUE**) в рамках примера для одной записи выбран размер 80 байт (константа **usiTextRecordSize**). Все текстовые записи дополняются пробелами до этого размера – как раз для того, чтобы можно было определить позицию в файле конкретной записи по ее номеру. После установки позиции в зависимости от выбранной пользователем операции происходит переход на шаг **WRITE** или **READ**.

[9] На самом деле, всех этих манипуляций можно было бы избежать при использовании режима открытия файла FILE.MODE.MAPPENDPLUS – в нем при отсутствии файла происходит его создание. Но этот режим появился только в версии библиотеки **3.5.17.0**; для возможности использование примера в ПЛК, программируемых в более ранних версиях CODESYS, в примере используется подход со сменой режима в случае отсутствия файла

На шаге **WRITE** выполняется добавление в файл одной записи. Для текстового формата архива сначала происходит формирование строки архива в переменной **sRecord** с помощью метода prvCreateRecord. Для бинарного архива происходит запись входной переменной **stWriteRecord**. После этого выполняется переход на шаг определения размера файла **GET_SIZE**.

На шаге **READ** производится чтение из файла. Для текстового формата архива считанная запись сохраняется в переменной **sRecord**, которая потом с помощью метода prvStringToRecord конвертируется в переменную типа ARCHIVE_RECORD и присваивается на выход **stReadRecord**. Для бинарного формата архива считанная запись сразу размещается в переменной **stReadRecord**. После этого выполняется переход на шаг закрытия файла **CLOSE**.

На шаге **GET_SIZE** производится определение размера файла и расчет количества сохраненных в нем записей (**udiRecordCount**). После этого выполняется переход на шаг закрытия файла **CLOSE**.

На шаге **CLOSE** выполняется закрытие файла и возвращение на шаг **IDLE**.

5.5.2 Метод prvResetOutputs

В методе **prvResetOutputs** выполняется сброс выходов и некоторых локальных переменных. В случае возникновения ошибки файл, с которым производилась операция, мог остаться открытым, поэтому в методе также выполняется попытка закрытия файла (за исключением ошибки **WRONG_PARAMETER** – она возникает на этапе открытия файла и ее наличие означает, что файл не был открыт).

```
// Сброс выходов и внутренних переменных/экземпляров ФБ
METHOD prvResetOutputs : BOOL
VAR_INPUT
END_VAR


xDone  := FALSE;
xCreateRecord := FALSE;

fbFileOpen(xExecute := FALSE);
fbFileSetPos(xExecute := FALSE);
fbFileRead(xExecute := FALSE);
fbFileWrite(xExecute := FALSE);
fbFileGetSize(xExecute := FALSE);

// если произошла ошибка - то файл мог остаться открытым
// пробуем закрыть его
// но если код ошибки WRONG_PARAMETER - то файл и не открывался
IF xError AND eError <> FILE.ERROR.WRONG_PARAMETER THEN

   fbFileClose(xExecute := xError);

   IF fbFileClose.xDone THEN
      hFile := 0;
      fbFileClose(xExecute := FALSE);
      xError := FALSE;
      eError := FILE.ERROR.NO_ERROR;
      xResetOutputs := FALSE;
   END_IF

ELSE
      hFile := 0;
      fbFileClose(xExecute := FALSE);
      xError := FALSE;
      eError := FILE.ERROR.NO_ERROR;
      xResetOutputs := FALSE;
END_IF
```

5.5.3 Метод prvCreateRecord

В методе **prvResetOutputs** выполняется формирование одной текстовой записи архива и ее сохранение в переменную **sRecord**. Если вход **xWriteTitle** имеет значение **TRUE**, то запись включает в себя заголовок (который записывается в файл вместе с первой записью). На выход **usiTextRecordSize** передается размер сформированной записи (число ее символов). В процессе формирования архивной записи выполняется конвертация полей структуры ARCHIVE_RECORD в строковое представление; для переменной типа **DT** для этого применяется функция **DT_TO_STRING_FORMAT** из библиотеки OwenStringUtils. Сформированная строка дополняется пробелами до заданной длины (длина определятся константой **usiTextRecordSize**), чтобы все записи в текстовом файле были одинаковой длины – это упрощает определение позиции записи в файле для ее чтения или перезаписи.

```
// Формирования строки архива
METHOD prvCreateRecord : BOOL
VAR_INPUT
   // TRUE - формируем заголовок архива и первую запись,
   // FALSE - формируем только архивную запись
   xWriteTitle:                  BOOL;
END_VAR
VAR
   // Метка времени в строковом формате
   m_sDateAndTime:               STRING(20);
   // Длина, до которой необходимо дополнить записываемую строку пробелами
   // (чтобы все записи в файле были одной длины)
   m_usiTargetLen:               USINT;
END_VAR
VAR_OUTPUT
   // Длина записываемых данных
   usiTextRecordSize:            USINT;
END_VAR

// очищаем буфер записи
Mem.MemFill(ADR(sRecord), SIZEOF(sRecord), 0);

// формируем заголовок архива
IF xWriteTitle THEN
   sRecord := c_sTitle;
   // дополняем строку пробелами, чтобы каждая запись архива
   // занимала одинаковое число байт
   sRecord := OSU.ADD_CHAR(sRecord,
         c_usiTextRecordSize - TO_USINT(LEN(c_sEndOfLineChars) ), ' ', TRUE);
   // добавляем символы переноса строки
   sRecord := CONCAT(sRecord, c_sEndOfLineChars);
   // строка включает в себя заголовок и запись
   usiTextRecordSize := 2 * c_usiTextRecordSize;
ELSE
   // строка включает в себя только запись
   usiTextRecordSize := c_usiTextRecordSize;
END_IF
// формируем архивную запись
m_sDateAndTime := OSU.DT_TO_STRING_FORMAT(stWriteRecord.dtTimeStamp,
   '%t[dd.MM.yyyy HH:mm:ss]');
sRecord := OSU.CONCAT8(sRecord, m_sDateAndTime, c_sRecordSeparator,
   TO_STRING(stWriteRecord.iVar), c_sRecordSeparator,
      // для корректного отображения в MS Excel
      // используем в качестве разделителя целой и дробной части запятую -
      // и оставляем 2 знака после запятой
      OSU.REAL_TO_STRING_FORMAT(stWriteRecord.rVar, 2, OSU.DECIMAL_SEPARATOR.COMMA),
         '', '');
// дополняем строку пробелами, чтобы  каждая запись архива занимала одинаковое число байт
m_usiTargetLen := usiTextRecordSize - TO_USINT(LEN(c_sEndOfLineChars) );
sRecord := OSU.ADD_CHAR(sRecord, m_usiTargetLen, ' ', TRUE);
sRecord := CONCAT(sRecord, c_sEndOfLineChars);

prvCreateRecord := TRUE;
```

5.5.4 Метод prvSwitchToIdle

Метод **prvSwithToIdle** обеспечивает возвращение на шаг **IDLE** в случае возникновения ошибки в процессе работы блока.

```
// Возвращение на шаг ожидания новой команды после возникновения ошибки
METHOD prvSwitchToIdle : BOOL
VAR_INPUT
END_VAR


xError := TRUE;
xResetOutputs := TRUE;
eState := STATE.IDLE;
```

5.5.5 Метод prvStringToRecord

Метод **prvStringToRecord** конвертирует одну текстовую запись архива в структуру типа ARCHIVE_RECORD. Конвертация производится следующим образом:

- сначала строка архива «разрезается» на отдельные строковые значения с помощью метода prvSplitStringBySeparator;
- с помощью этого же метода строковое представление метки времени «разрезается» на отдельные разряды времени;
- с помощью функции **JoinDateTime** из библиотеки Util из отдельных разрядов времени формируется переменная типа **DT**;
- остальные поля структуры (типов **INT** и **REAL**) формируются из своих строковых значений с помощью стандартных операторов **TO_INT** и **TO_REAL**.

```
// Метод конвертирует строку архива в структуру
METHOD prvStringToRecord : ARCHIVE_RECORD
VAR
        // Строковое представления даты
        m_sDate:                                STRING;
        // Строковое представления времени
        m_sTime:                                STRING;
        // Массив исходных строк
        m_astRecordValues:                      ARRAY [1..c_iRecordElemCount] OF
                                                        STRING(c_usiTextRecordSize);
        // Буфер для сохранения разрядов даты и времени
        m_asDateTimeBuffer:                     ARRAY [0..5] OF STRING;
        //
        m_uliDateAndTime:                       ULINT;
END_VAR
VAR CONSTANT
        // формат метки времени в архиве: dd.MM.yyyy HH:mm:ss
        //
        // Число разрядов даты
        mc_iDateElemCount:                      INT := 3;
        // Число разрядов времени
        mc_iTimeElemCount:                      INT := 3;
        // Разделитель между разрядами даты
        mc_sDateSeparator:                      STRING(1) := '.';
        // Разделитель между разрядами времени
        mc_sTimeSeparator:                      STRING(1) := ':';
        // Разделитель между датой и временем
        mc_sDateFromTimeSeparator:              STRING(1) := ' ';
        // Количество миллисекунд в секунде
        mc_uiMillisecondsPerSecond:             UINT := 1000;
END_VAR
```

```
// разрезаем строку архива по разделителям
prvSplitStringBySeparator(sRecord, ADR(m_astRecordValues), c_iRecordElemCount,
   c_sRecordSeparator);

// сохраняем отдельно дату и время
m_sDate := OSU.Before(m_astRecordValues[1], mc_sDateFromTimeSeparator);
m_sTime := OSU.After(m_astRecordValues[1],  mc_sDateFromTimeSeparator);

// разрезаем дату и время на разряды
prvSplitStringBySeparator(m_sDate, ADR(m_asDateTimeBuffer[0]), mc_iDateElemCount,
   mc_sDateSeparator);
prvSplitStringBySeparator(m_sTime, ADR(m_asDateTimeBuffer[0 + mc_iDateElemCount]),
   mc_iTimeElemCount, mc_sTimeSeparator);

// собираем метку времени в миллисекундах из отдельных разрядов
m_uliDateAndTime := UTIL.JoinDateTime(uiYear := TO_UINT(m_asDateTimeBuffer[2]),
   uiMonth := TO_UINT(m_asDateTimeBuffer[1]),
      uiDay := TO_UINT(m_asDateTimeBuffer[0]),
         uiHour := TO_UINT(m_asDateTimeBuffer[3]),
            uiMinute := TO_UINT(m_asDateTimeBuffer[4]),
               uiSecond := TO_UINT(m_asDateTimeBuffer[5]),
                  uiMilliseconds := 0) / mc_uiMillisecondsPerSecond;

prvStringToRecord.dtTimeStamp := TO_DT(m_uliDateAndTime);

// конвертируем остальные значения в поля структуры
prvStringToRecord.iVar := TO_INT(m_astRecordValues[2]);

// в файле мы используем запятую как разделитель целой и дробной части
// (это требуется для корректного отображения в MS Excel)
// так что при парсинге нужно заменить ее обратно на точку, чтобы оператор TO_REAL
// сработал корректно
m_astRecordValues[3] := OSU.ReplaceSubstring(m_astRecordValues[3], ',' , '.');

prvStringToRecord.rVar := TO_REAL(m_astRecordValues[3]);
```

5.5.6 Метод prvSplitStringBySeparator

Метод **prvSplitStringBySeparator** разделяет исходную строку **sSource** с разделителем **sSeparator** на отдельные строки и возвращает их в виде массива строк по указателю **pasBuffer**. Число элементов массива не должно превышать **iBufferElemsCount**.

```
// Метод разделяет одну строку с разделителями на отдельные строки
METHOD prvSplitStringBySeparator : BOOL
VAR_INPUT
   // Исходная строка с разделителями
   sSource:                      STRING;
   // Указатель на массив вырезанных строк
   pasBuffer:                    POINTER TO ARRAY [0..0] OF STRING(c_usiTextRecordSize);
   // Макс. число элементов массива
   iBufferElemsCount:            INT;
   // Разделитель
   sSeparator:                   STRING;
END_VAR
VAR
   i:                            INT;
   // Позиция предыдущего обработанного разделителя
   m_uiPrevSeparator:            UINT;
   // Позиция текущего обрабатываемого разделителя
   m_uiCurrentSeparator:         UINT;
END_VAR
```

```
m_uiPrevSeparator := 1;

FOR i := 0 TO iBufferElemsCount - 1 DO

   // ищем следующий разделитель
   m_uiCurrentSeparator := OSU.FindSubstringPosAfterN(sSource, sSeparator,
      m_uiPrevSeparator);

   // нашли
   IF m_uiCurrentSeparator <> 0 THEN
      // вырезаем строку между предыдущим и текущим разделителем
      pasBuffer^[i] := MID(sSource, TO_INT(m_uiCurrentSeparator - m_uiPrevSeparator),
         TO_INT(m_uiPrevSeparator) );
      // переходим к первому символу следующего значения
      m_uiPrevSeparator := m_uiCurrentSeparator + 1;
   // больше разделителей нет - вырезаем конец строки
   ELSE
      pasBuffer^[i] := MID(sSource, LEN(sSource) - TO_INT(m_uiPrevSeparator) + 1,
         TO_INT(m_uiPrevSeparator) );
      EXIT;
  END_IF

END_FOR
```

5.6 Программы

5.6.1 Программа FILE_PRG

В программе **FILE_PRG** реализована работа с бинарными и текстовыми файлами с помощью экземпляров ФБ FileManager, а также выполнение других операций с файлами (копирование, переименование и удаление).

```
PROGRAM FILE_PRG
VAR
   (* переменные и ФБ примера работы с бинарным файлом *)

   // Команда записи в файл
   xWriteToBinFile:                    BOOL;
   // Команда чтения из файла
   xReadFromBinFile:                   BOOL;
   // Путь к файлу
   sBinFilePath:                       STRING := 'test.bin';
   // Выбранное файловое устройство
   eBinFileDevice:                     FILE_DEVICE;
   // Путь к выбранному файловому устройству
   sPathToBinFileDevice:               STRING;
   // Полный путь к файлу
   sBinFileFullPath:                   STRING;
   // Структура записываемых данных
   stWriteBinArchiveRecord:            ARCHIVE_RECORD;
   // Структура считываемых данных
   stReadBinArchiveRecord:             ARCHIVE_RECORD;
   // Номер записи (используется для перезаписи записей файла)
   // Нумерация - с 0
   udiWriteBinRecordNumber:            UDINT;
   // Число записей в файле
   udiBinRecordCount:                  UDINT;
   // Операция, выполняемая с файлом
   eBinFileMode:                       USER_FILE_MODE;
   // Переменная для элемента визуализации Радио-кнопка
   iVisuBinOverwriteRadioButton:       INT;
   // Режим записи (FALSE - запись в конец файла,
   // TRUE - перезапись записи с номером udiWriteBinRecordNumber)
   xBinOverwrite:                      BOOL;
   // ФБ работы с файлом
   fbBinFileManager:                   FileManager;

   ////////////////////////////////////////////

    (* переменные и ФБ примера работы с текстовым файлом *)

   // Команда записи в файл
   xWriteToTextFile:                   BOOL;
   // Команда чтения из файла
   xReadFromTextFile:                  BOOL;
   // Путь к файлу
   sTextFilePath:                      STRING := 'test.csv';
   // Выбранное файловое устройство
   eTextFileDevice:                    FILE_DEVICE;
   // Путь к выбранному файловому устройству
   sPathToTextFileDevice:              STRING;
   // Полный путь к файлу
   sTextFileFullPath:                  STRING;
   // Структура записываемых данных
   stWriteTextArchiveRecord:           ARCHIVE_RECORD;
   // Структура считываемых данных
   stReadTextArchiveRecord:            ARCHIVE_RECORD;
```

```
   // Номер записи (используется для перезаписи записей файла)
   // Нумерация - с 1 (потому что 0 - это номер заголовка)
   udiWriteTextRecordNumber:              UDINT := 1;
   // Число записей в файле
   udiTextRecordCount:                    UDINT;
   // Операция, выполняемая с файлом
   eTextFileMode:                         USER_FILE_MODE;
   // Переменная для элемента визуализации Радио-кнопка
   iVisuTextOverwriteRadioButton:         INT;
   // Режим записи (FALSE - запись в конец файла,
   // TRUE - перезапись записи с номером udiWriteBinRecordNumber)
   xTextOverwrite:                        BOOL;
   // ФБ работы с файлом
   fbTextFileManager:                     FileManager;

   ////////////////////////////////////////

    (* переменные и ФБ для других операций с файлами *)

   // ФБ копирования файла
   fbFileCopy:                            FILE.Copy;
   // ФБ переименования файла
   fbFileRename:                          FILE.Rename;
   // ФБ удаления файла
   fbFileDelete:                          FILE.Delete;

   // Команда копирования файла
   xFileCopy:                             BOOL;
   // Команда переименования файла
   xFileRename:                           BOOL;
   // Команда удаления файла
   xFileDelete:                           BOOL;

   // Путь к существующему файлу в пределах файлового устройства
   sCurrentFilePath:                      STRING;
   // Текущее выбранное файловое устройство
   eCurrentFileDevice:                    FILE_DEVICE;
   // Путь к текущему выбранному файловому устройству
   sPathToCurrentFileDevice:              STRING;
   // Полный путь к существующему файлу
   sCurrentFileFullPath:                  STRING;
   // Путь к существующему файлу в пределах файлового устройства
   sNewFilePath:                          STRING;
   // Текущее выбранное файловое устройство
   eNewFileDevice:                        FILE_DEVICE;
   // Путь к текущему выбранному файловому устройству
   sPathToNewFileDevice:                  STRING;
   // Полный путь к существующему файлу
   sNewFileFullPath:                      STRING;
END_VAR


// Работа с бинарным файлом

sPathToBinFileDevice := GetPathToFileDevice(eBinFileDevice);
sBinFileFullPath := CONCAT(sPathToBinFileDevice, sBinFilePath);

stWriteBinArchiveRecord.dtTimeStamp := TargetVars.stRtc.dtDateAndTime;

xBinOverwrite := TO_BOOL(iVisuBinOverwriteRadioButton);

IF xReadFromBinFile THEN
   eBinFileMode := USER_FILE_MODE.READ;
ELSF xWriteToBinFile AND NOT(xBinOverwrite) THEN
   eBinFileMode := USER_FILE_MODE.APPEND;
ELSIF xWriteToBinFile AND xBinOverwrite THEN
   eBinFileMode := USER_FILE_MODE.WRITE;
END_IF
```

```
fbBinFileManager
(
   xExecute         := xWriteToBinFile OR xReadFromBinFile,
   sFileName        := sBinFileFullPath,
   eFileMode        := eBinFileMode,
   stWriteRecord    := stWriteBinArchiveRecord,
   udiRecordNumber  := udiWriteBinRecordNumber,
   xIsTextFormat    := FALSE,
   udiRecordCount   => udiBinRecordCount,
   stReadRecord     => stReadBinArchiveRecord
);


// Работа с текстовым файлом

sPathToTextFileDevice := GetPathToFileDevice(eTextFileDevice);
sTextFileFullPath := CONCAT(sPathToTextFileDevice, sTextFilePath);

stWriteTextArchiveRecord.dtTimeStamp := TargetVars.stRtc.dtDateAndTime;

xTextOverwrite := TO_BOOL(iVisuTextOverwriteRadioButton);

IF xReadFromTextFile THEN
   eTextFileMode := USER_FILE_MODE.READ;
ELSIF xWriteToTextFile AND NOT(xTextOverwrite) THEN
   eTextFileMode := USER_FILE_MODE.APPEND;
ELSIF xWriteToTextFile AND xTextOverwrite THEN
   eTextFileMode := USER_FILE_MODE.WRITE;
END_IF

fbTextFileManager
(
   xExecute         := xWriteToTextFile OR xReadFromTextFile,
   sFileName        := sTextFileFullPath,
   eFileMode        := eTextFileMode,
   stWriteRecord    := stWriteTextArchiveRecord,
   udiRecordNumber  := udiWriteTextRecordNumber,
   xIsTextFormat    := TRUE,
   udiRecordCount   => udiTextRecordCount,
   stReadRecord     => stReadTextArchiveRecord
);


// Другие операции с файлами

sPathToCurrentFileDevice := GetPathToFileDevice(eCurrentFileDevice);
sCurrentFileFullPath     := CONCAT(sPathToCurrentFileDevice, sCurrentFilePath);
sPathToNewFileDevice     := GetPathToFileDevice(eNewFileDevice);
sNewFileFullPath         := CONCAT(sPathToNewFileDevice, sNewFilePath);

fbFileCopy
(
   xExecute         := xFileCopy,
   sFileNameSource  := sCurrentFileFullPath,
   sFileNameDest    := sNewFileFullPath,
   xOverWrite       := TRUE
);

fbFileRename
(
   xExecute      := xFileRename,
   sFileNameOld  := sCurrentFileFullPath,
   sFileNameNew  := sNewFileFullPath
);

fbFileDelete
(
   xExecute      := xFileDelete,
   sFileName     := sCurrentFileFullPath
);
```

Программа состоит из трех фрагментов:

- работа с бинарным файлом;
- работа с текстовым файлом;
- другие операции с файлами.

Ниже описывается фрагмент кода, связанный с работой с бинарным файлом.

В визуализации примера (см. рисунок 5.6.1) пользователь выбирает используемое файловое устройство (например, директорию CODESYS или USB-накопитель) с помощью выпадающего списка, к которому привязана переменная **eBinFileDevice** (экземпляр перечисления FILE_DEVICE), и определяет путь к файлу, записывая его в переменную **sCurrentFilePath**. В программе с помощью функции GetPathToFileDevice определяется заместитель файлового устройства (**sPathToBinFileDevice**), после чего формируется полный путь к файлу (**sBinFileFullPath**).

В случае работы с бинарным и текстовым файлом пользователь может выбрать одну из трех операций: чтение, перезапись или дозапись в конец файла. В визуализации примера присутствуют две кнопки – **Добавить запись** и **Прочитать запись**, к которым привязаны переменные **xWriteToBinFile** и **xReadToBinFile**. Будет ли запись добавлена в конец файла или перезапишет существующую – определяется элементом **Радио-кнопка**. К этому элементу привязана переменная **iVisuBinOverwriteRadioButton** типа **INT** – в случае выбора режима **В конец файла** она имеет значение **0**, а в случае режима **Перезапись** – **1**. В коде программы эта переменная конвертируется в булевскую переменную **xBinOverwrite**. В зависимости от значений переменных **xWriteToBinFile**, **xBinOverwrite** и **xReadToBinFile** определяется операция, которая будет произведена с файлом (переменная **eBinFileMode**).

Вызов экземпляра ФБ FileManager, который называется **fbBinFileManager**, происходит по команде **xWriteToBinFile** или **xReadToBinFile**. В случае записи – значения полей **INT** и **REAL** экземпляра структуры ARCHIVE_RECORD с названием **stWriteTextArchiveRecord** задаются пользователем в визуализации, а в качестве метки времени (поле **dtTimeStamp**) используется системное время контроллера из узла **OwenRTC**. Пример создан на базе шаблона проекта, в котором к узлам таргет-файла уже привязаны глобальные переменные – поэтому для получения системного времени достаточно обратиться к переменной **TargetVars.stRtc.dtDateAndTime** (**TargetVars** – название списка глобальных переменных, **stRtc** – имя экземпляра структуры **TRG_RTC**, содержащей информацию о системном времени; сама структура объявлена в библиотеке OwenTypes).

Работа с текстовым файлом производится аналогично (единственное существенное отличие – при вызове экземпляра блока FileManager на его вход **xIsTextFormat** передается значение **TRUE**).

Другие операции с файлами (копирование и т. д.) не имеют какой-то специфики – в коде программы просто выполняется вызов экземпляров соответствующих ФБ.

Рисунок 5.6.1 – Внешний вид экрана работы с бинарными файлами (экран работы с текстовыми файлами выглядит аналогично)

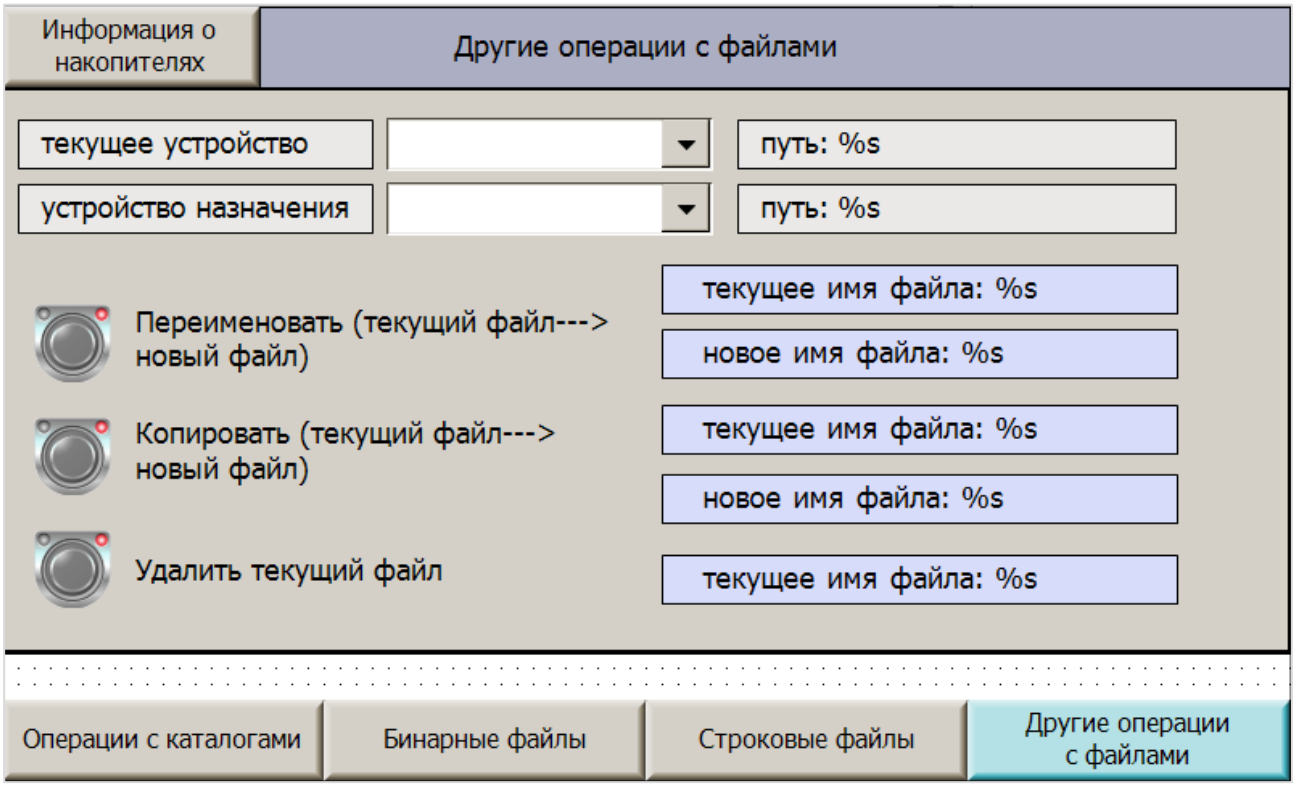


Рисунок 5.6.2 – Внешний вид экрана других операций с файлами

5.6.2 Программа DIR_PRG

В программе **DIR_PRG** реализована работа с каталогами (создание, удаление и т. д.), а также просмотр их содержимого (с помощью метода prvDirList).

```
PROGRAM DIR_PRG
VAR

   (* блоки и переменные для операций с каталогами *)

   // ФБ создания каталога
   fbDirCreate:                  FILE.DirCreate;
   // ФБ удаления каталога
   fbDirRemove:                  FILE.DirRemove;
   // ФБ переименования каталога
   fbDirRename:                  FILE.DirRename;
   // ФБ копирования каталога
   fbDirCopy:                    FILE.DirCopy;
   // ФБ открытия каталога
   fbDirOpen:                    FILE.DirOpen;
   // ФБ получения информации о содержимом каталога
   fbDirList:                    FILE.DirList;
   // ФБ закрытия каталога
   fbDirClose:                   FILE.DirClose;

   // Команда создания каталога
   xDirCreate:                   BOOL;
   // Команда удаления каталога
   xDirRemove:                   BOOL;
   // Команда переименования каталога
   xDirRename:                   BOOL;
   // Команда копирования каталога
   xDirCopy:                     BOOL;

   // Путь к существующему каталогу в предалах файлового устройства
   sCurrentDirPath:              STRING;
   // Путь к новому каталогу в пределах файлового устройства
   sNewDirNamePath:              STRING;
   // Текущее выбранное файловое устройство
   eCurrentFileDevice:           FILE_DEVICE;
   // Путь к текущему выбранному файловому устройству
   sPathToCurrentFileDevice:     STRING;
   // Полный путь к существующему каталогу
   sCurrentDirFullPath:          STRING;
   // Полный путь к новому каталогу
   sNewDirFullPath:              STRING;
   // Предыдущее выбранное файловое устройство
   ePrevFileDevice:              FILE_DEVICE;

   ////////////////////////////////////////////////

   (* переменные для получения информации о каталоге *)

   // Счетчик циклов
   i:                            DINT;

   // Полный путь к выбранному в визуализации каталогу
   sDirListPath:                 STRING;
   // Полный путь к выгружаемому из визулизации файлу
   sFileTransferPath:            STRING;

   // Массив данных об объектах каталога
   astDirInfo:                   ARRAY [1..c_uiMaxDirEntries] OF FILE.FILE_DIR_ENTRY;

   // Массив данных об объектах каталога визуализации
   astVisuDirInfo:               ARRAY [1..c_uiMaxDirEntries] OF VISU_DIR_INFO;
   // Количество обработанных объектов
   diEntryPos:                   DINT;
```

```
   // Дескриптор каталога
   hDirHandle:              FILE.CAA.HANDLE;
   // Шаг машины состояний
   eState:                  STATE;

   // Номер выбранной в визуализации строки таблицы
   iSelectedEntry:          INT := 1;
   // Детектор команды перехода в выбранный каталог
   fbCmdGoToNextDir:        R_TRIG;
   // Детектор перехода из текущего каталога на уровень выше
   fbCmdBackToPrevDir:      R_TRIG;
   // Команда команды перехода в выбранный каталог
   xGoToNextDir:            BOOL;
   // Команда перехода из текущего каталога на уровень выше
   xBackToPrevDir:          BOOL;
   // Сигнал отключения возможности перехода в каталог (если выбран файл, не каталог)

   xDisableNextDirButton:   BOOL;
   // Флаг выполнения первого цикла задачи
   xFirstCycle:             BOOL;

   // Детектор команлы сбора информации об объектах каталога
   fbNeedUpdateTrig:        R_TRIG;
   // Команда сбора информации об объектах каталога
   xNeedUpdate:             BOOL;
   // Флаг завершения сбора информации об объектах каталога
   xDone:                   BOOL;
END_VAR

VAR CONSTANT
   // Максимальное количество обрабатываемых вложенных элементов каталога
   c_uiMaxDirEntries:       UINT          := 120;
   // Разделитель каталогов в файловой системе ОС Linux
   c_sCharSlash:            STRING(1)     :='/';
END_VAR


// Операции с каталогами (экран Visu01_DirExample)

// определяем пути к выбранным пользователем каталогам
sPathToCurrentFileDevice := GetPathToFileDevice(eCurrentFileDevice);

sCurrentDirFullPath := CONCAT(sPathToCurrentFileDevice, sCurrentDirPath);
sNewDirFullPath     := CONCAT(sPathToCurrentFileDevice, sNewDirNamePath);

// вызов ФБ создания каталога
fbDirCreate
(
   xExecute := xDirCreate,
   sDirName := sNewDirFullPath,
   // с созданием указанных в пути подкаталогов
   xParent  := TRUE
);

// вызов ФБ удаления каталога
fbDirRemove
(
   xExecute   := xDirRemove,
   sDirName   := sCurrentDirFullPath,
   // с удалением всех вложенных каталогов и файлов
   xRecursive := TRUE
);

// вызов ФБ переименования каталога
fbDirRename
(
   xExecute    := xDirRename,
   sDirNameOld := sCurrentDirFullPath,
   sDirNameNew := sNewDirFullPath
);
```

```
// вызов ФБ копирования каталога
fbDirCopy
(
   xExecute       := xDirCopy,
   sDirNameSource := sCurrentDirFullPath,
   sDirNameDest   := sNewDirFullPath,
   // с перезаписью файлов
   xOverWrite     := TRUE,
   // с копированием вложенных каталогов
   xRecursive     := TRUE
);

// Просмотр каталогов (экран Visu_DirList)

// отключаем кнопку перехода в каталог для "специальных" каталогов
// (текущего и родительского) и в случае выбора файла
xDisableNextDirButton := astVisuDirInfo[iSelectedEntry].sEntryName = '.' OR
   astVisuDirInfo[iSelectedEntry].sEntryName = '..' OR
      astVisuDirInfo[iSelectedEntry].wsEntryType = "Файл";

// однократно считываем информацию о каталогах при старте проекта
IF NOT(xFirstCycle) THEN

   xFirstCycle := TRUE;
   xNeedUpdate := TRUE;

END_IF

// считываем информацию при изменении файлового устройства
IF (eCurrentFileDevice <> ePrevFileDevice) THEN

   sDirListPath    :=  sPathToCurrentFileDevice;
   ePrevFileDevice :=  eCurrentFileDevice;
   xNeedUpdate     :=  TRUE;

END_IF

// по сигналу переходим в выбранный каталог
fbCmdGoToNextDir(CLK := xGoToNextDir);

IF fbCmdGoToNextDir.Q THEN

   sDirListPath := CONCAT(sDirListPath, astVisuDirInfo[iSelectedEntry].sEntryName);

   IF sDirListPath <> sPathToCurrentFileDevice THEN
      sDirListPath := CONCAT(sDirListPath, c_sCharSlash);
   END_IF

   xNeedUpdate := TRUE;

END_IF

// по сигналу переходим на уровень выше, контролируя, что продолжается работа с прежним
устройством
fbCmdBackToPrevDir(CLK := xBackToPrevDir);

IF fbCmdBackToPrevDir.Q AND sDirListPath <> sPathToCurrentFileDevice THEN

   prvRemoveLastDirFromPath();
   xNeedUpdate := TRUE;

END_IF

// получение информации о каталоге
fbNeedUpdateTrig(CLK := xNeedUpdate);
xDone := prvDirList(fbNeedUpdateTrig.Q, sDirListPath);

IF xDone THEN
   xNeedUpdate := FALSE;
END_IF
```

Программа состоит из двух фрагментов:

- операции с каталогами;
- просмотр содержимого каталогов.

В визуализации примера (см. рисунок 5.6.3) пользователь выбирает используемое файловое устройство (например, директорию CODESYS или USB-накопитель) с помощью выпадающего списка, к которому привязана переменная **eCurrentFileDevice** (экземпляр перечисления FILE_DEVICE), и определяет пути к «существующему» и «новому» каталогу, записывая их в переменные **sCurrentDirPath** и **sNewDirPath** (существующий каталог, например, можно удалить или переименовать, а новый – создать). В программе с помощью функции GetPathToFileDevice определяется заместитель файлового устройства (**sPathToCurrentFileDevice**), после чего формируются полные пути к существующему и новому каталогу (**sCurrentDirFullPath** и **sNewDirFullPath**).

Операции с каталогами (создание, удаление и т. д.) не имеют какой-то специфики – в коде программы просто выполняется вызов экземпляров соответствующих ФБ.

Рисунок 5.6.3 – Внешний вид экрана операций с каталогами

Во втором фрагменте программы выполняется код, реализующий просмотр содержимого каталогов. Отображение содержимого каталогов производится на экране **Visu_DirList** с помощью элемента **Таблица**. Обновление данных таблицы происходит в следующих случаях:

- в момент старта проекта;
- при изменении файлового устройства;
- при переходе во вложенный каталог;
- при переходе из текущего каталога на уровень выше.

В коде программы производится проверка всех этих ситуаций и случае необходимости обновления информации в таблице формируется команда **xNeedUpdate**, которая передается на вход метода prvDirList, реализующего сбор информации о выбранном каталоге (путь к каталогу передается на второй вход метода).

После получения информации метод возвращает на свой выход значение **TRUE**. Это значение попадает в переменную **xDone**, после чего происходит сброс переменной **xNeedUpdate**.

Процесс сбора информации может занять несколько секунд (в зависимости от числа вложенных каталогов и файлов анализируемого каталога). В течение этого времени переменная **xNeedUpdate** сохраняет значение **TRUE**, за счет чего поверх таблицы отображается пиктограмма обновления данных (к ее переменной невидимости привязано инвертированное значение переменной **xNeedUpdate**).

Переход во вложенный каталог должен быть доступен только в том случае, если в таблице выбран каталог. Поэтому в случае выбора файла кнопки **Открыть каталог** и **На уровень выше** блокируются с помощью переменной **xDisableNextDirButton**, привязанной к параметру кнопки **Переменные состояний/Отключение ввода**. Кроме того, кнопка блокируется в случае выбора каталогов «.» и «..» – это специальные каталоги Linux, обозначающие текущий и родительский каталог (см. подробнее, например, в этой статье), которые в рамках примера не обрабатываются.

При переходе на уровень выше вызывается метод prvRemoveLastDirFromPath, который удаляет из текущего выбранного пути последний каталог.

Кнопка **Выгрузить на ПК** позволяет выгрузить выделенный в таблице файл и сохранить его на ПК (или другом устройстве, на котором открыта web-визуализация – например, смартфоне). Это реализовано с помощью действия **Передача файла**, привязанного к кнопке. Перед выполнением этого действия происходит вызов ST-кода с формированием пути к выделенному в таблице файлу (см. рисунок 5.6.5). Данный функционал доступен только в web-визуализации контроллера. Кнопка **Выгрузить на ПК** блокируется, если выбранный в таблице элемент не является файлом.

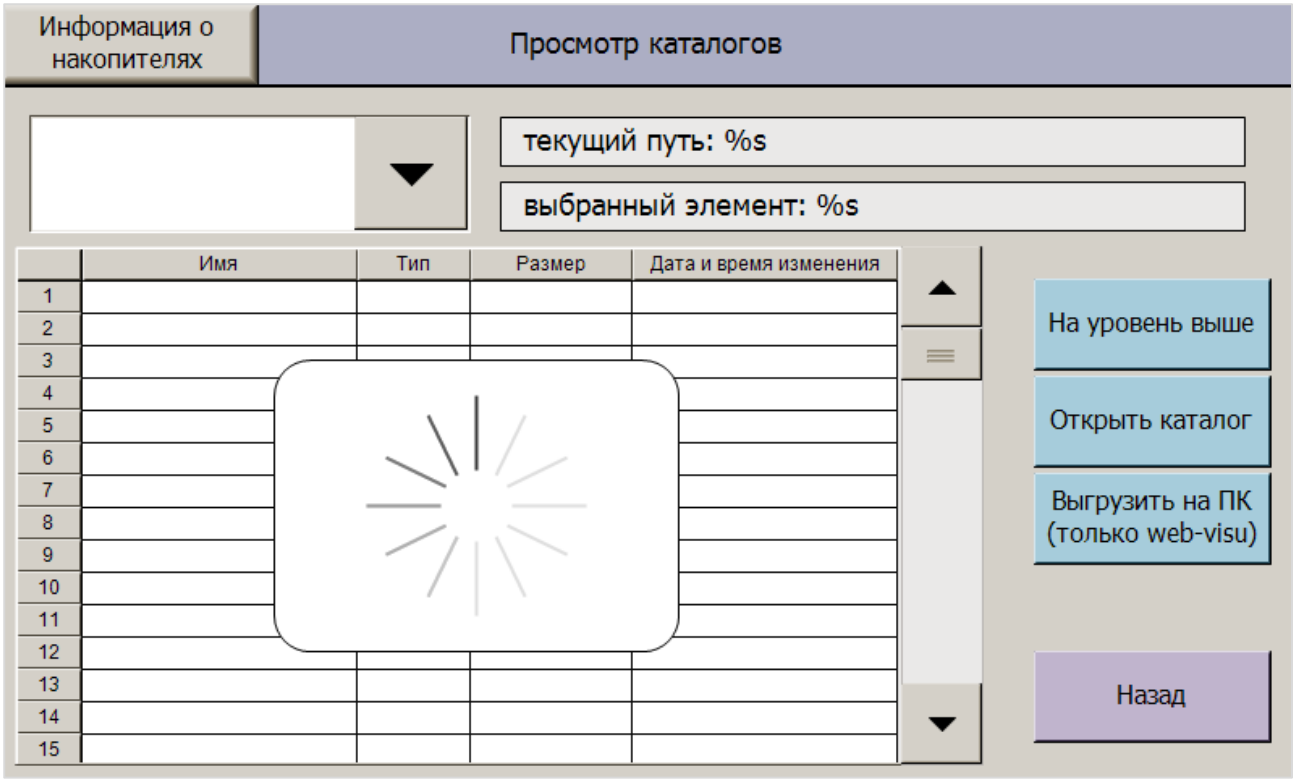


Рисунок 5.6.4 – Внешний вид экрана просмотра содержимого каталогов

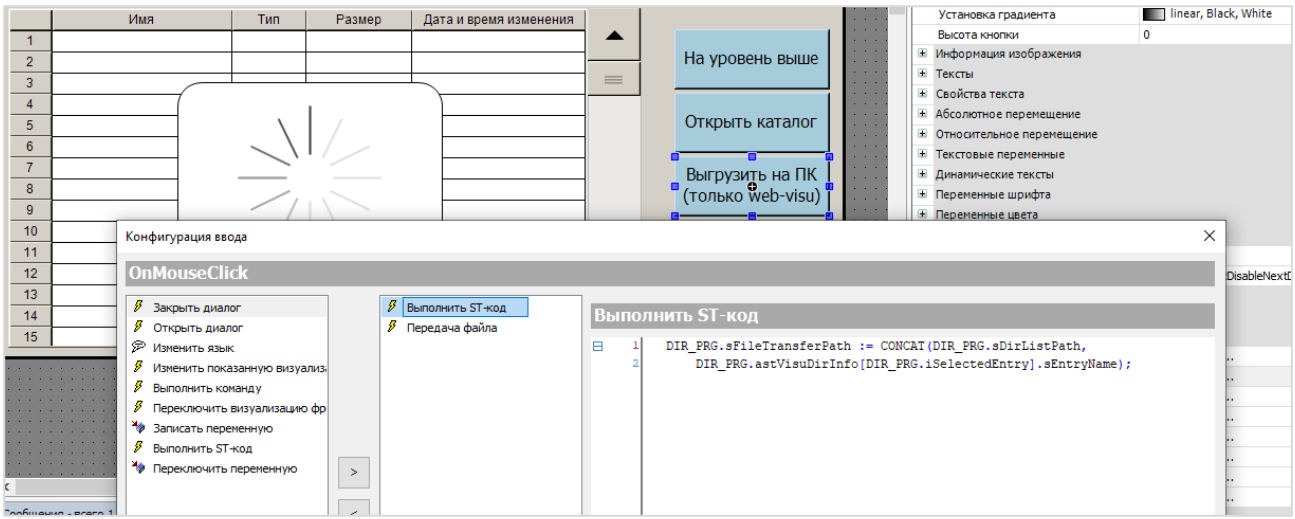


Рисунок 5.6.5 – Код кнопки Выгрузить на ПК

5.6.3 Метод prvDirList

Метод **prvDirList** используется для получения информации об объектах выбранного каталога.

```
// Метод получения информации о вложенных объектах каталога (каталогах и файлах)
METHOD prvDirList: BOOL
VAR_INPUT
   // Команда запуска блока
   xExecute:              BOOL;
   // Обрабатываемый каталог
   sDirPath:              STRING;
END_VAR


CASE eState OF

   // шаг ожидания команды
   STATE.IDLE:

      prvDirList := FALSE;

      IF xExecute THEN
         eState := STATE.OPEN;
      END_IF

   // шаг открытия каталога
   STATE.OPEN:

      // обнуляем позицию для записи информации о файлах/каталогах
      diEntryPos := LOWER_BOUND(astDirInfo, 1);

      fbDirOpen
      (
         xExecute := TRUE,
        sDirName  := sDirPath
      );

      IF fbDirOpen.xDone THEN

         hDirHandle      :=      fbDirOpen.hDir;
          fbDirOpen(xExecute := FALSE);
          eState := STATE.READ;

      ELSIF fbDirOpen.xError THEN
```

```
         fbDirOpen(xExecute := FALSE);
         eState := STATE.IDLE;

      END_IF

// шаг получения информации об объектах каталога
STATE.READ:

   fbDirList
   (
      xExecute := TRUE,
      hDir     := hDirHandle
   );

   // пока нет ошибок, получаем информацию о текущем файле/каталоге...
   IF fbDirList.xDone AND fbDirList.eError = FILE.ERROR.NO_ERROR THEN
      astDirInfo[diEntryPos] := fbDirList.deDirEntry;

      // информацию о каждом обработанном файле/каталоге записываем
      // в следующую ячейку массива
      diEntryPos := diEntryPos + 1;

      // если число вложенных файлов/каталогов больше, чем размер массива...
      // ...то начинаем перезаписывать его с нуля
      // можно реализовать другую обработку -
      // например, не добавлять последующие записи в массив и сгенерировать ошибку
      IF diEntryPos > c_uiMaxDirEntries THEN
         diEntryPos   :=      LOWER_BOUND(astDirInfo, 1);
      END_IF

      fbDirList(xExecute := FALSE);

   // если код ошибки - "NO_MORE_ENTRIES", то обработаны все файлы/каталоги...
   // ...и можно завершать работу блока
   ELSIF fbDirList.eError = FILE.ERROR.NO_MORE_ENTRIES THEN

      Mem.MemFill(ADR(astVisuDirInfo), SIZEOF(astVisuDirInfo), 0);
      // переходим к первой строке таблицы
      iSelectedEntry  :=      TO_INT(LOWER_BOUND(astDirInfo, 1) );

      // заполняем массив структур информацией о содержимом каталога
      // так как diEntryPos инкрементируется и после обработки последнего объекта,
      // то надо сделать -1
      FOR i := LOWER_BOUND(astDirInfo, 1) TO diEntryPos - 1 DO

         astVisuDirInfo[i].sEntryName   := astDirInfo[i].sEntry;
         astVisuDirInfo[i].wsEntrySize  := BYTE_SIZE_TO_WSTRING(astDirInfo[i].szSize);
         astVisuDirInfo[i].wsEntryType  :=      SEL(astDirInfo[i].xDirectory,
            "Файл", "Каталог");
         astVisuDirInfo[i].sLastModification :=
            OSU.DT_TO_STRING_FORMAT(astDirInfo[i].dtLastModification,
                '%t[dd.MM.yyyy HH:mm:ss]');
      END_FOR

         fbDirList(xExecute := FALSE);
         eState := STATE.CLOSE;

   ELSIF fbDirList.xError THEN

      fbDirList(xExecute := FALSE);
      eState  := STATE.CLOSE;

   END_IF
```

```
   // шаг закрытия каталога
   STATE.CLOSE:

      fbDirClose
      (
         xExecute := TRUE,
         hDir     := hDirHandle
      );

      IF fbDirClose.xDone THEN

         fbDirClose(xExecute := FALSE);
         eState      := STATE.IDLE;
         prvDirList  := TRUE;

      ELSIF fbDirClose.xError THEN

         fbDirClose(xExecute := FALSE);
         eState := STATE.IDLE;

      END_IF

END_CASE
```

Сбор информации о содержимом каталога производится с помощью экземпляра ФБ File.DirList. Каждый вызов блока возвращает информацию об одном вложенном объекте (каталоге или файле). Когда информация обо всех объектах получена – вызов блока вернет ошибку **NO_MORE_ENTRIES**.

Информация об объектах сохраняется в массиве структур **astDirInfo** (типа FILE.FILE_DIR_ENTRY). Метод конвертирует этот массив в массив структур **astVisuDirInfo** (типа VISU_DIR_INFO). Структура **VISU_DIR_INFO** состоит из строк – это удобно для отображения информации в таблице на экране визуализации, так как, например, незаполненные элементы массива в этом случае отображаются в виде пустых строк (для массива типа FILE.FILE_DIR_ENTRY в этом случае бы отображались бы «нулевые» значения: 0 для числовых типов, 1970-1-1-00:00:00 для типа DT и т. д.).

Размерность массивов определяется константой **c_uiMaxDirEntries**. Если число объектов каталога превышает значение этой константы, то массив начнет перезаписываться с первого элемента (такова обработка этой ситуации в рамках примера; в реальном проекте можно реализовать другую обработку – например, не добавлять в массив «не влезающие» записи и сгенерировать ошибку).

5.6.4 Метод prvRemoveLastDirFromPath

Метод **prvRemoveLastDirFromPath** удаляет из пути название последнего каталога (например, строка 'USB/2021/04/' после вызова метода превратится в 'USB/2021/').

```
// Метод удаления названия последнего каталога из пути
METHOD prvRemoveLastDirFromPath
VAR_INPUT
END_VAR

// удаляем последний символ в текущем пути (это "/")
sDirListPath[LEN(sDirListPath) - 1]       :=      0;

// справа налево стираем символы из пути до тех пор, пока не найдем "/"
// таким образом, из текущего пути будет удалено имя самого последнего каталога
FOR i:=LEN(sDirListPath) - 1 TO 0 BY -1 DO

   IF sDirListPath[i] = c_sCharSlash[0] THEN
      EXIT;
   ELSE
      sDirListPath[i] := 0;
   END_IF

END_FOR
```

5.6.5 Программа PLC_PRG

Программа **PLC_PRG** является основной задачей примера. Она привязана к задаче **MainTask**. Эта программа включает в себя только вызовы программ DIR_PRG и FILE_PRG.

```
PROGRAM PLC_PRG
VAR
END_VAR


DIR_PRG();
FILE_PRG();
```

5.7 Визуализации

Пример включает в себя 5 экранов визуализации:

- **Visu01_DirExample** – экран операций с каталогами;
- **Visu_DirList** – экран просмотра содержимого каталогов;
- **Visu02_BinFileExample** – экран работы с бинарным файлом;
- **Visu03_TextFileExample** – экран работы с текстовым файлом;
- **Visu04_FilesActionsExample** – экран других операций с файлами (копирование, удаление и т. д.).

Каждый экран, за исключением **Visu_DirList**, включает в себя кнопки перехода на остальные экраны. Переход на экран **Visu_DirList** осуществляется только с экрана **Visu01_DirExample** – именно поэтому его название отличается от других отсутствием порядкового номера.

Информация о нюансах реализации конкретных экранов приведена в описании программ DIR_PRG и FILE_PRG.

Каждый экран содержит кнопку **Информация о накопителях**. По нажатию на эту кнопку открывается диалог **DrivesOwen** из библиотеки OwenVisuDialogs. В качестве аргумента в диалог передается экземпляр структуры **TRG_Drives** из библиотеки OwenTypes. В рамках шаблона проекта (а пример создан на базе шаблона) этот экземпляр объявлен в списке глобальных переменных **TargetVars** и имеет название **stDrives**. Его переменные привязаны к каналам узла **Drives**, который находится в дереве проекта.

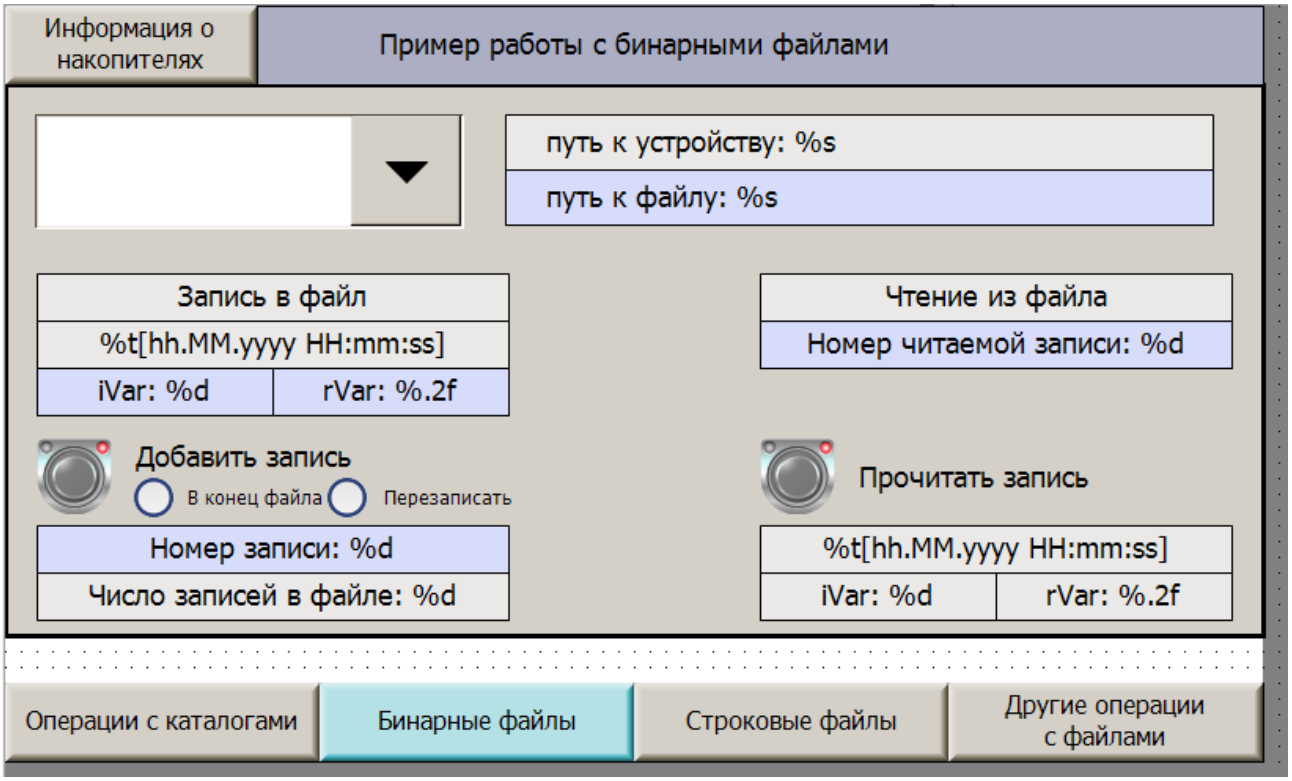


Рисунок 5.7.1 – Внешний вид экрана работы с бинарными файлами

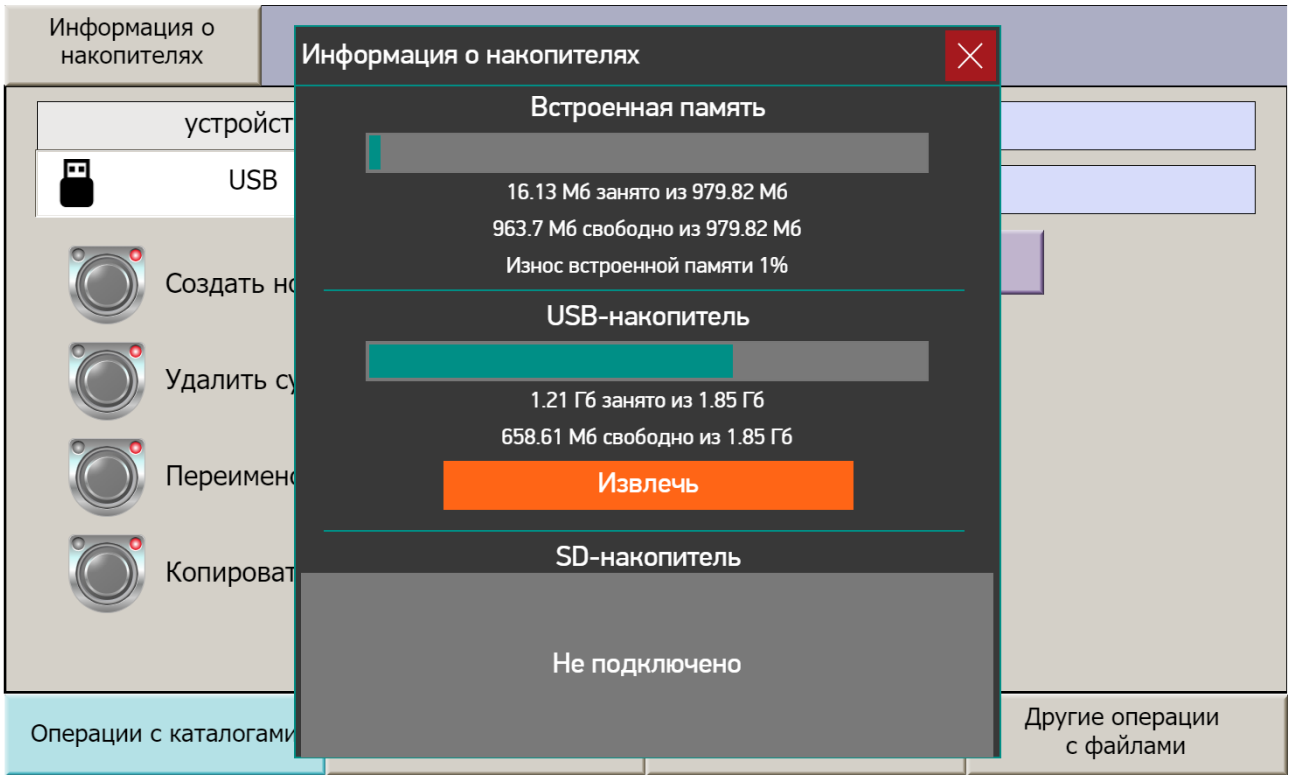


Рисунок 5.7.2 – Внешний вид диалога DrivesOwen

5.8 Работа с примером

Для работы с примером следует загрузить его в контроллер. Желательно одновременно подключиться к контроллеру с помощью утилиты WinSCP, чтобы получить возможность видеть содержимое его файловой системы.

Стартовым экраном проекта является экран операций с каталогами (**Visu01_DirExample**). После нажатия на кнопку **Информация о накопителях** будет отображен диалог с информацией о памяти контроллера и подключенных накопителей (см. рисунок 5.7.2).

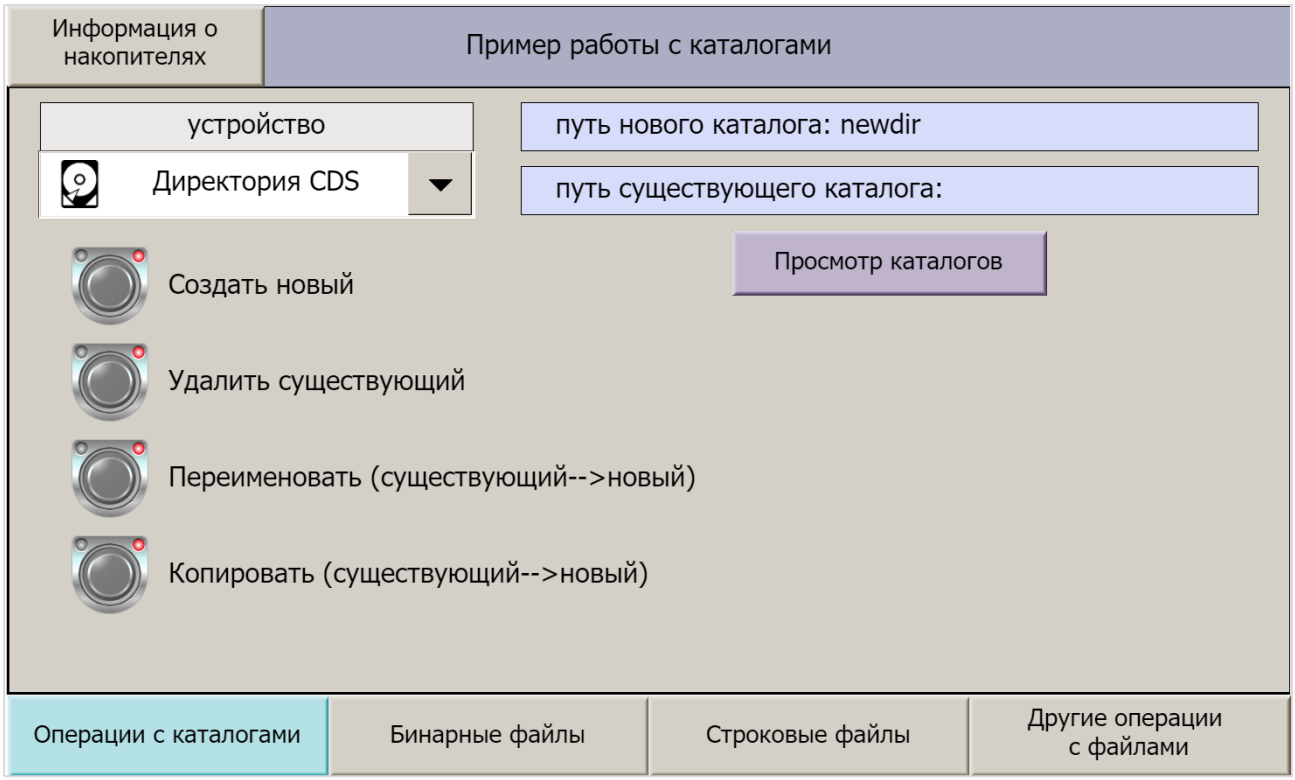


Рисунок 5.8.1 – Внешний вид экрана операций с каталогами

По умолчанию в качестве файлового устройства выбрана рабочая директория CODESYS.

Введите имя нового каталога (например, **newdir**) и нажмите кнопку **Создать**. В памяти контроллера будет создан новый каталог:

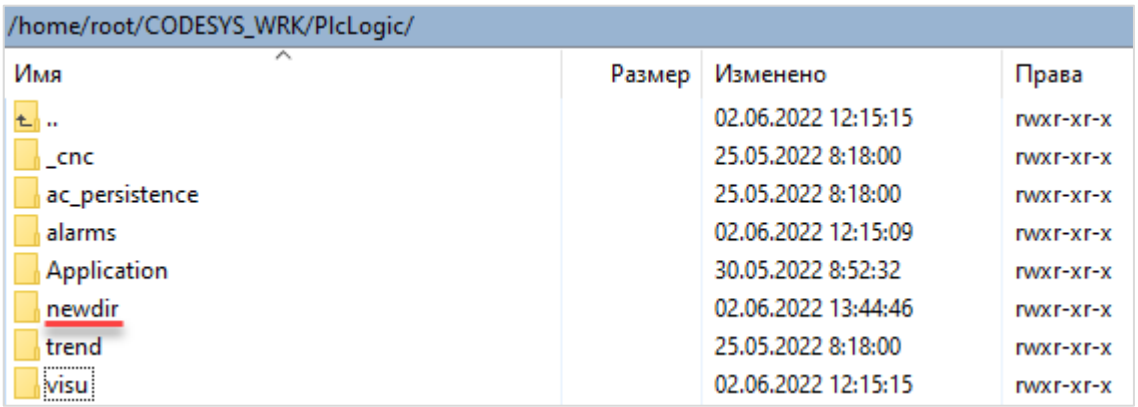


Рисунок 5.8.2 – Результат создания каталога

Теперь в качестве имени существующего каталога введите **newdir**, а в качестве имени нового каталога – например, **newdir2**. Нажмите кнопку **Копировать**. В результате будет создан каталог **newdir2**, являющийся копией каталога **newdir**.

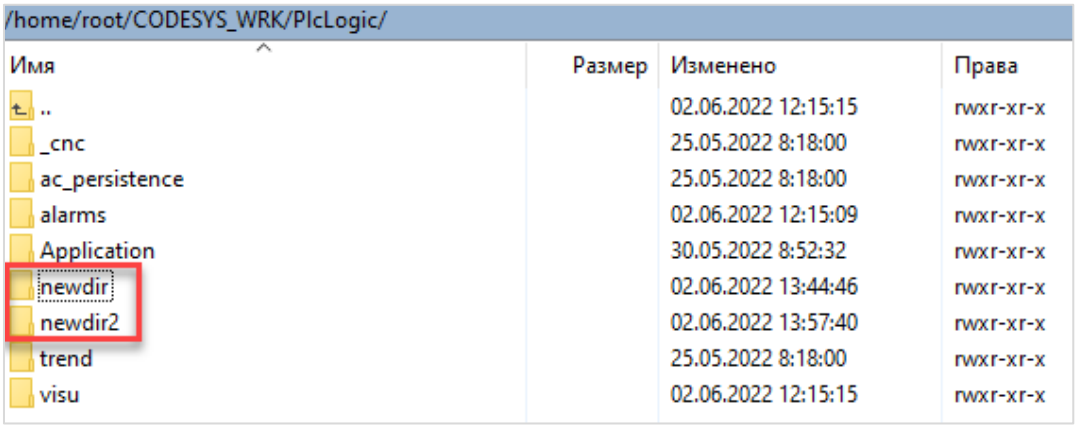


Рисунок 5.8.3 – Результат копирования каталогов

Теперь в качестве имени нового каталога введите **newdir3** и нажмите кнопку **Переименовать**. В результате каталог **newdir** будет переименован в **newdir3**.

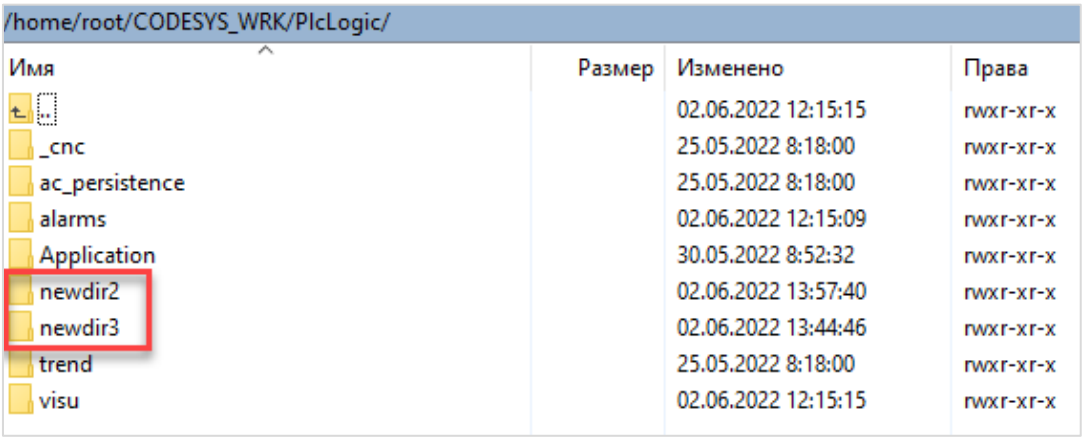


Рисунок 5.8.4 – Результат переименования каталогов

Введите в качестве имени существующего каталога **newdir3** и нажмите кнопку **Удалить**. В результате каталог **newdir3** будет удален.

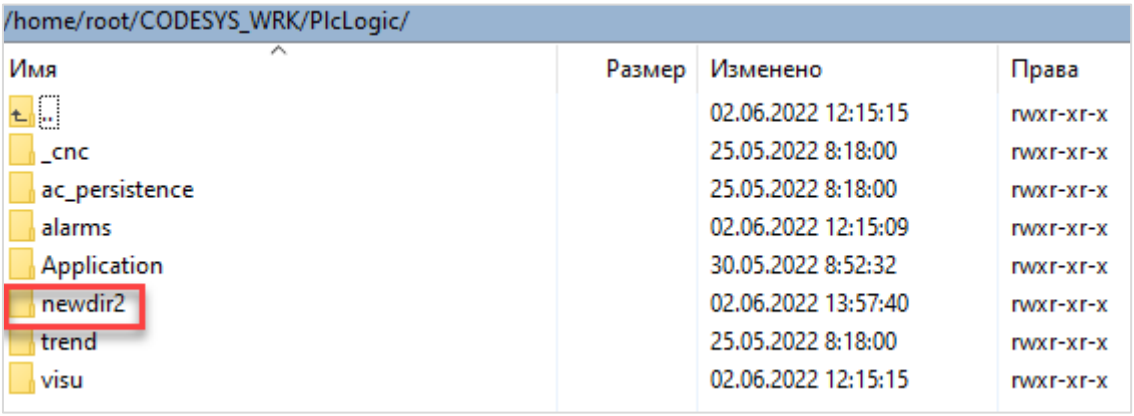


Рисунок 5.8.5 – Результат удаления каталога newdir3

Нажмите кнопку **Просмотр каталогов** для перехода на экран просмотра содержимого каталогов. Выберите нужное файловое устройство и используйте кнопки **Открыть каталог** и **На уровень выше** для перемещения по его каталогам. Кнопка **Выгрузить на ПК** позволяет скачать выбранный файл (он будет загружен в web-браузере; этот функционал поддержан только в web-визуализации контроллера).

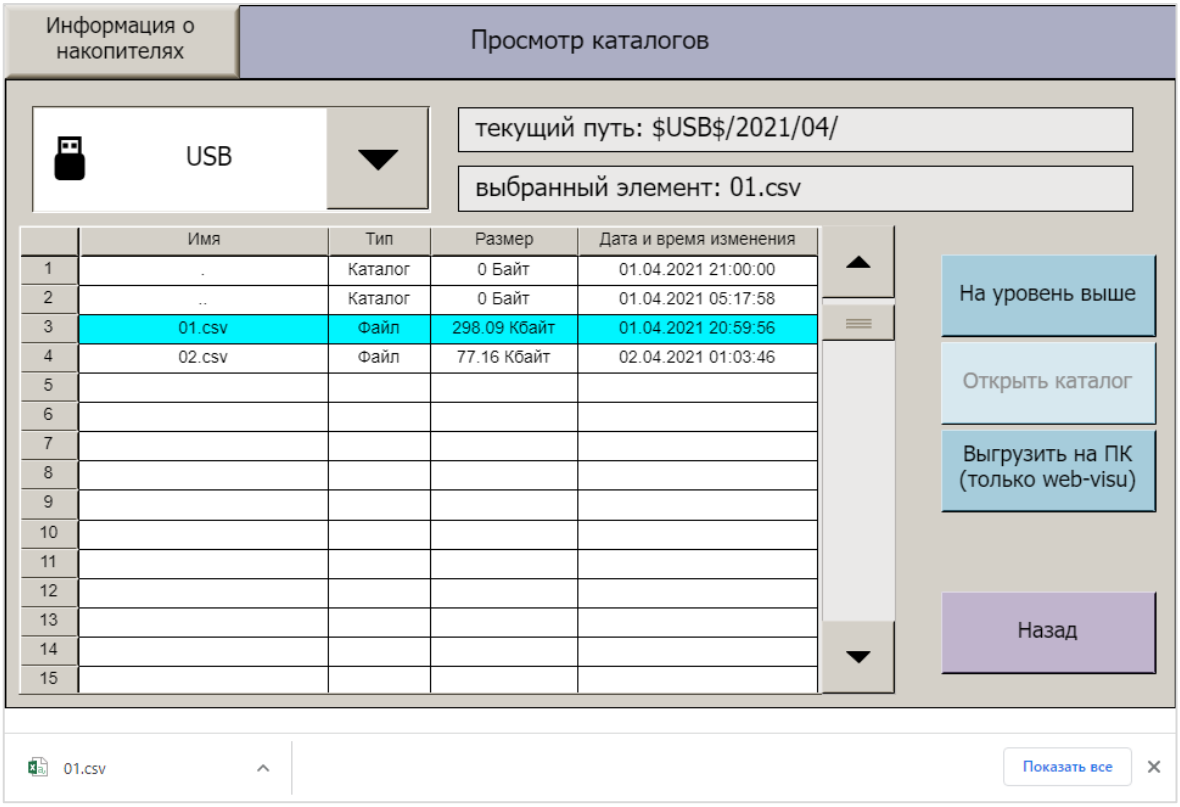


Рисунок 5.8.6 – Внешний вид экрана просмотра каталогов

Стоит обратить внимание на следующие моменты:

- названия файлов и каталогов, содержащих кириллицу, отображаются некорректно. Это связано с тем, что поле **sEntry** структуры FILE.FILE_DIR_ENTRY имеет тип **STRING** – а в визуализации этот тип подходит только для отображения строк в кодировке **ASCII**;
- размер всех каталогов отображается нулевым. Это особенность CODESYS – для каталогов размер не определяется. При необходимости можно рассчитывать его в коде как сумму размеров всех файлов данного каталога (стоит учесть, что каталог может содержать вложенные каталоги). В рамках примера такой расчет не реализован.

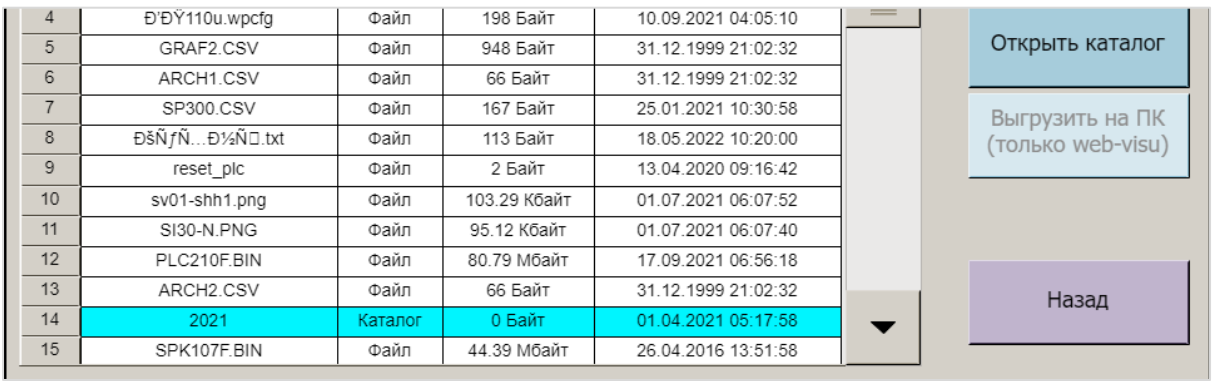


Рисунок 5.8.7 – Некорректное отображение имен файлов/каталогов, содержащих кириллицу, и размера каталогов

Нажмите **Назад**, чтобы перейти на экран операций с каталогами, а уже с него перейдите на экран **Бинарные файлы**.

По умолчанию в качестве файлового устройства выбрана рабочая директория CODESYS, а путь к файлу – **test.bin** (то есть файл будет создан в корне рабочей директории). Введите значения для переменных **iVar** и **rVar** (например, 1 и 1.11) и нажмите кнопку **Добавить запись** в режиме **В конец файла**. Добавьте еще несколько записей с разными значениями. После каждой записи значений поле **Число записей в файле** будет увеличиваться на 1.

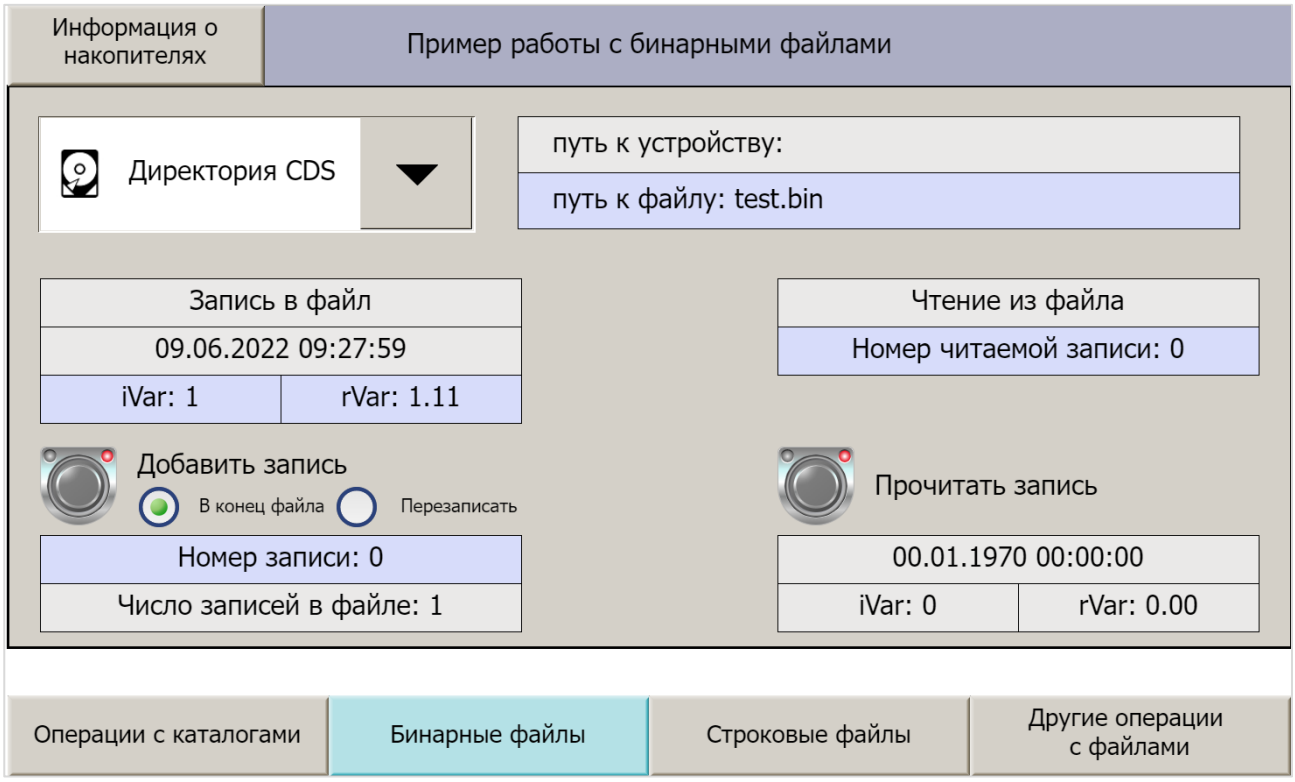


Рисунок 5.8.8 – Внешний вид экрана работы с бинарными файлами

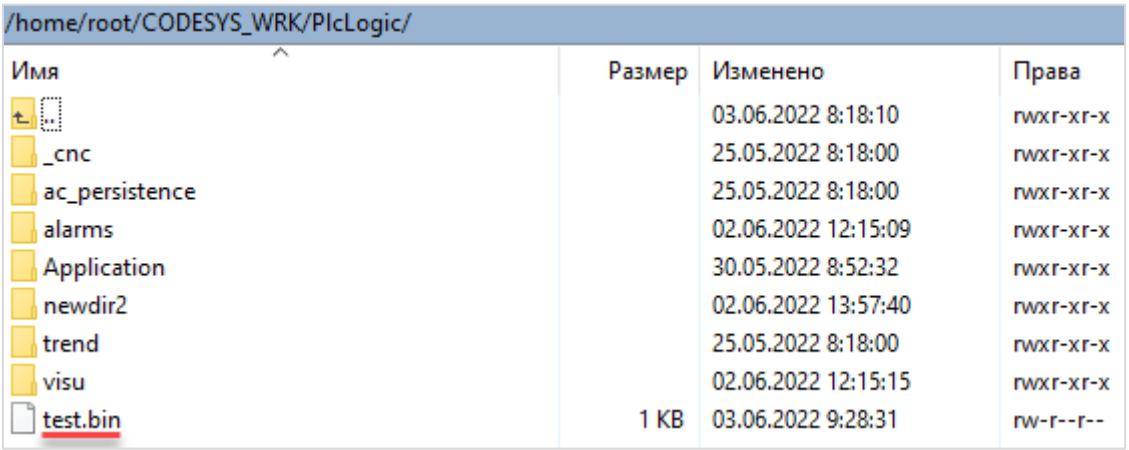


Рисунок 5.8.9 – Создание бинарного файла

Выберите режим **Перезапись**, оставьте **Номер записи** в значении **0**, введите новые значения **iVar** и **rVar** (например, 11 и 111.222) и нажмите кнопку **Добавить запись**. После этого нажмите кнопку **Прочитать запись** (**Номер читаемой записи** также оставьте в значении 0). В полях под кнопкой отобразятся считанные значения.

Перейдите на экран работы с текстовыми файлами.

Экран работы с текстовыми файлами выглядит аналогично экрану работы с бинарными файлами. Повторите на нем те же самые процедуры. Обратите внимание, что число записей в файле будет на **1** превышать число добавленных в конец файла записей – потому что вместе с первой записью в файл добавляется заголовок, который воспринимается как отдельная запись. Для полей **Номер записи** и **Номер читаемой записи** нельзя указать значение **0**, потому что оно как раз соответствует заголовку – в рамках примера его перезапись и чтение не реализовано.

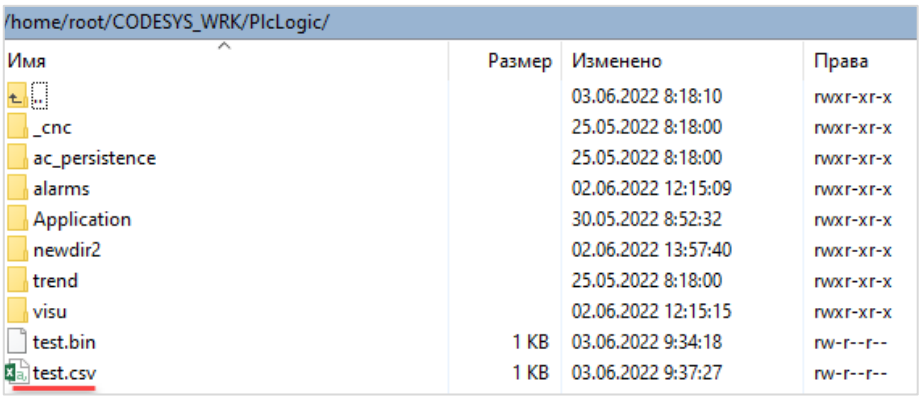


Рисунок 5.8.10 – Создание текстового файла

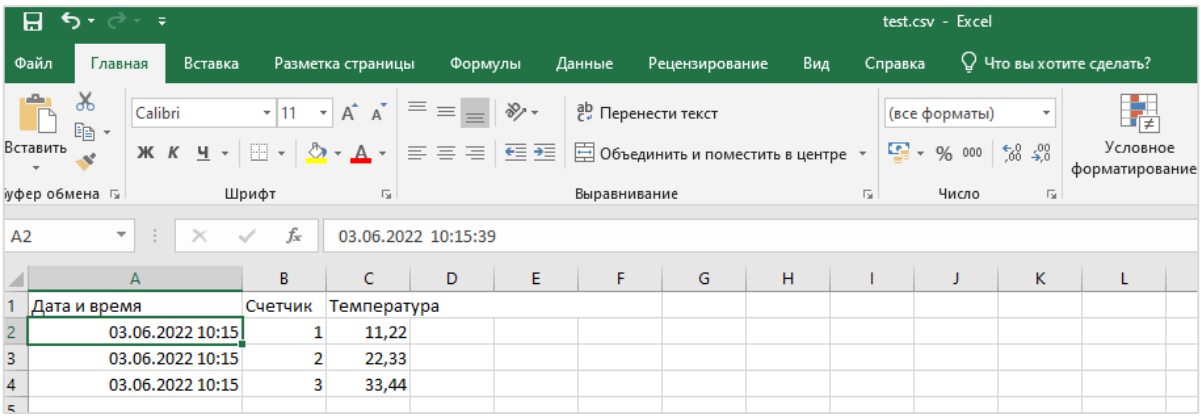


Рисунок 5.8.11 – Отображение созданного текстового файла в Microsoft Excel

Перейдите на экран других операций с файлами.

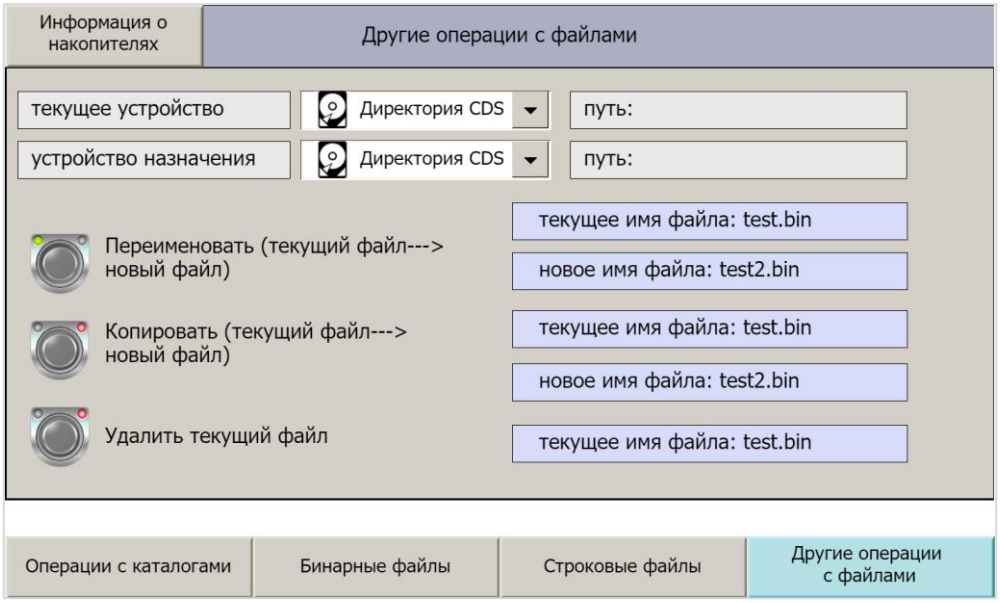


Рисунок 5.8.12 – Внешний вид экрана других операций с файлами

По умолчанию в качестве файловых устройств выбрана директория CODESYS. В качестве текущего имени файла введите **test.bin** (этот файл был создан на экране работы с бинарным файлом), в качестве имени нового файла – **test2.bin**. Нажмите кнопку **Переименовать**.

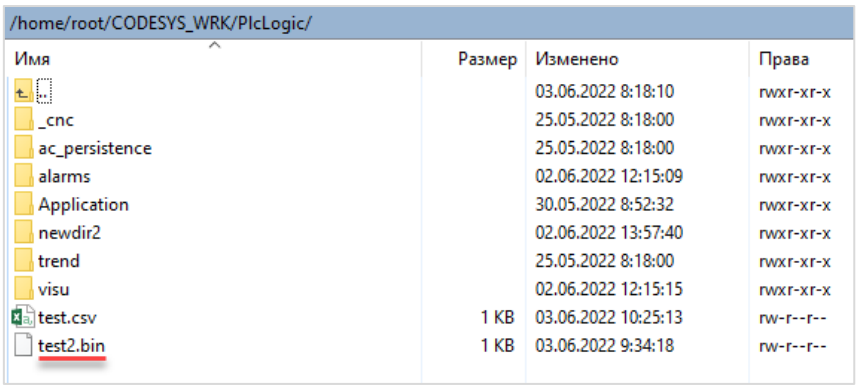


Рисунок 5.8.13 – Результат переименования файла

Введите в качестве текущего имени – **test2.bin**, а в качестве имени нового файла – **test3.bin**. Нажмите кнопку **Копировать**.

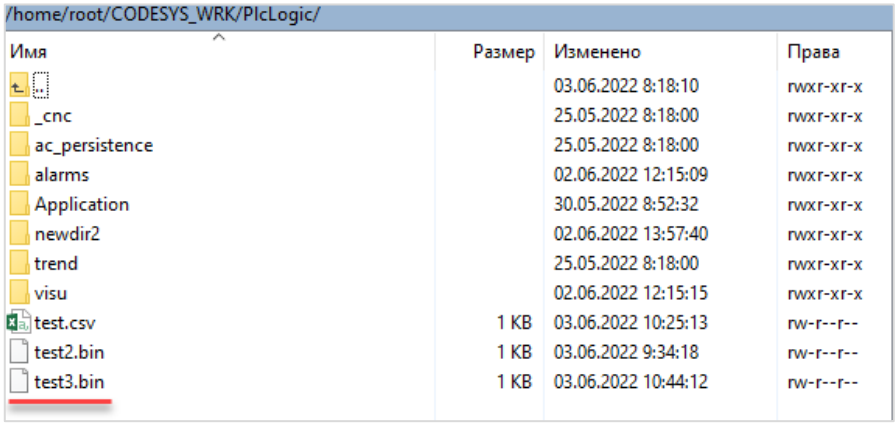


Рисунок 5.8.14 – Результат копирования файла

Теперь нажмите кнопку **Удалить**. В результате файл **test2.bin** будет удален.

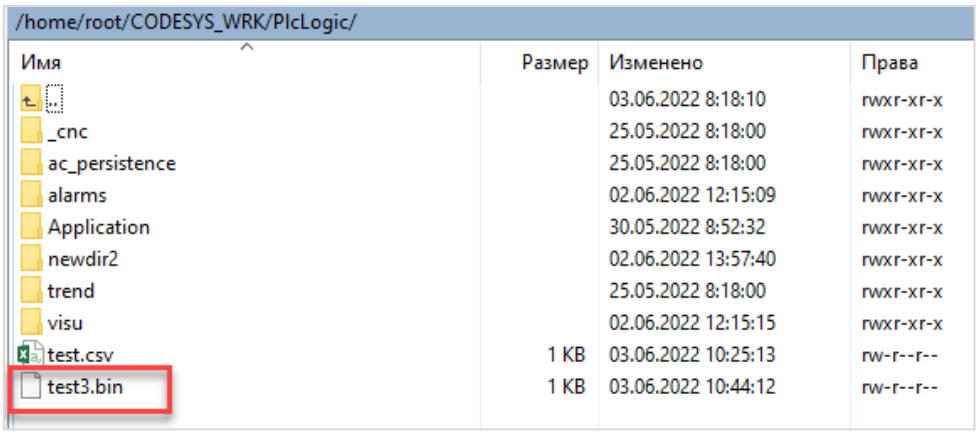


Рисунок 5.8.15 – Результат удаления файла

Таким образом, в рамках данного пункта был рассмотрен весь функционал примера.

6 Дополнительные вопросы

6.1 Библиотека SysFile

В данном документе рассматривается библиотека CAA File. В CODESYS есть и другие библиотеки работы с файлами – в частности, **SysFile**. Эта библиотека является аналогом библиотеки **SysLibFile** из **CoDeSys V2.3**. Ее основной особенностью является то, что каждая операция с файлом представлена в виде синхронно выполняемой функции (в отличие от **CAA File**, блоки которой являются асинхронными) – то есть выполнение задачи контроллера, в которой вызывается эта функция, блокируется до ее завершения. Это может привести к непредсказуемым изменениям времени выполнения цикла данной задачи. Поэтому в большинстве случаев данная библиотека не рекомендуется к использованию. Описание библиотеки доступно в справке CODESYS. Пример использования библиотеки приведен в данном видео и этой серии видео.

Библиотека **SysFile** не содержит функций для работы с каталогами – они входят в состав библиотеки **SysDir**.

6.2 Сохранение текстовых файлов в кодировке Unicode

В рамках примера работы с текстовыми файлами использовались переменные типа **STRING**. Таким образом, сохраняемый файл имеет кодировку, основанную на **ASCII**. Поскольку в заголовке файла использованы символы кириллицы, то кодировка файла – Win1251 (в этом можно убедиться, открыв созданный в примере .csv-файл в редакторе, отображающем кодировку – например, Notepad++).

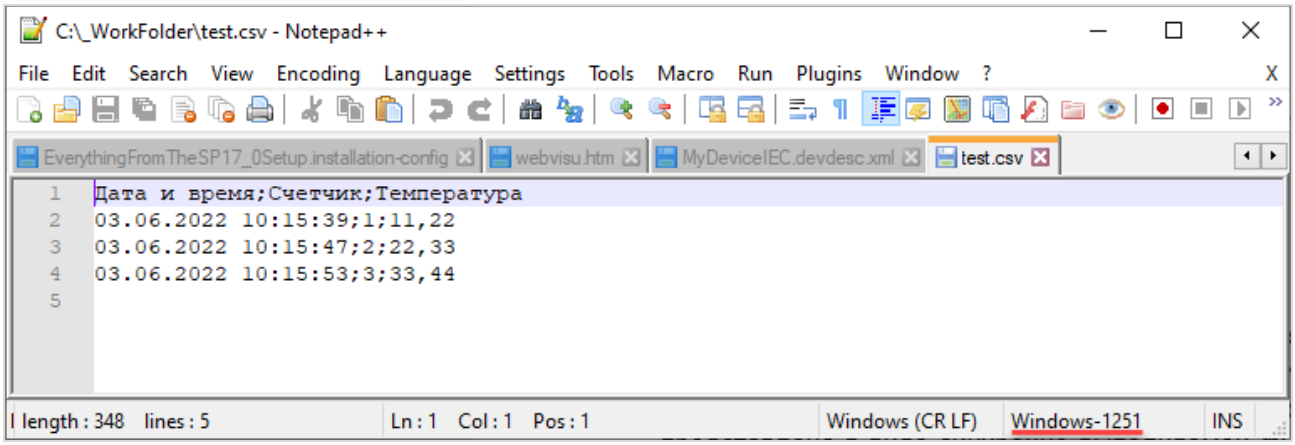


Рисунок 6.1 – Отображение кодировки текстового файла примера

Если требуется сохранять файлы в кодировке UTF-16/UCS-2, то для этого следует:

- заменить для всех используемых строковых переменных тип со **STRING** на **WSTRING**. Обратите внимание, что для типа **WSTRING** размер каждого символа (в том числе, NULL-терминатора) составляет 2 байта. Функции для работы со строками типа **WSTRING** доступны в библиотеке **Standard64** (**WCONCAT** и т. д.);
- в начале файла потребуется разместить маркер последовательности байтов.

Если требуется сохранять файлы в кодировке UTF-8, то следует сначала сформировать строку типа **WSTRING**, а затем конвертировать ее в массив байтов с кодовыми точками **UTF-8** с помощью функции **ConvertUTF16toUTF8** из библиотеки **StringUtils**. В начало массива потребуется добавить маркер последовательности байтов. Указатель на этот массив следует передать в вызове ФБ FILE.Write.

6.3 Сохранение «длинных» строк в текстовый файл

CODESYS не накладывает явного ограничения на длину строковых переменных – например, вы можете объявить строку длиной в 100000 символов:

```
VAR
   sVar: STRING(100000);
END_VAR
```

Но функции библиотек **Standard** и **Standard64** поддерживают только обработку строк, длина которых не превышает **255** символов. Для обработки более длинных строк (например, их объединения) следует использовать функции библиотеки **StringUtls** – они не имеют подобных ограничений.

6.4 Работа со строками с помощью утилит Linux

Контроллеры ОВЕН, программируемые в среде CODESYS 3.5, содержат ряд полезных утилит Linux для работы со строками – sed, iconv, jq и другие. Вы можете использовать их для решения специфических задач – например, с помощью утилиты **jq** очень просто распарсить файл формата JSON.

Вызов утилит производится с помощью библиотеки **CmpSysExec**. Документация на библиотеку и примеры ее использования доступны на сайте ОВЕН в разделе CODESYS V3.

Кроме того, начиная с прошивок **2.4.xxxx.xxxx** в состав прошивки входит интерпретатор языка Python – поэтому работу с файлами может реализовать и с помощью скриптов, написанных на этом языке.