CODESYS V3.5

Архивация

Руководство пользователя

01.12.2018
версия 2.0

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** или 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.

Особенности библиотеки **CAA File**:

- рассчитана на продвинутых пользователей;
- требует хороших навыков программирования;
- дает доступ к низкоуровневым функциям и ФБ работы с файлами, позволяя решить практическую любую задачу.

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 рассмотрены примеры использования библиотеки.

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 строго последовательно.

2 Основные сведения о работе с файлами

2.1 Общие сведения о памяти контроллеров

Файл – это именованная область памяти на носителе информации, 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. Файловая система контроллеров ОВЕН – UBIFS.

У контроллеров ОВЕН имеется три физически разных области памяти:

- энергонезависимая память (Flash);
- оперативная память (RAM);
- retain-память (область памяти retain-переменных).

Говоря о работе с файлами, мы будем подразумевать 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. 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 **конфигураторе** и таргет-файле.

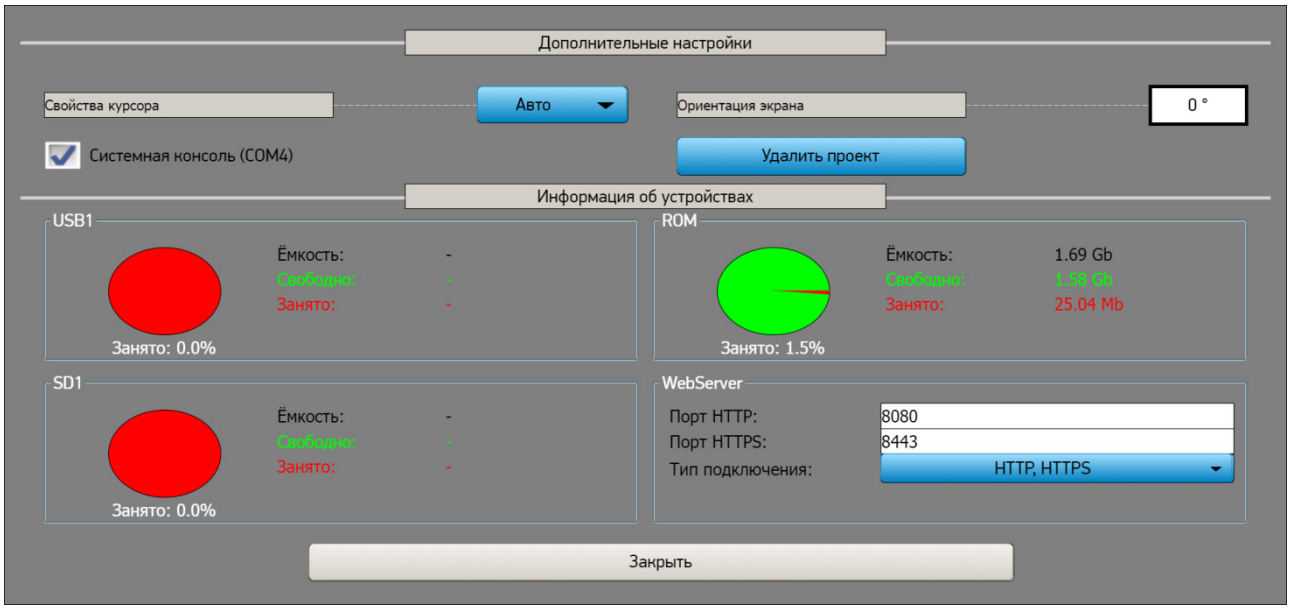


Рисунок 2.1 – Информация о памяти контроллера и накопителей в конфигураторе СПК

2.2 Операции с файлами

Во время работы с файлами используются четыре основные операции:

- открытие файла (если файл 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 операциях с ним.

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

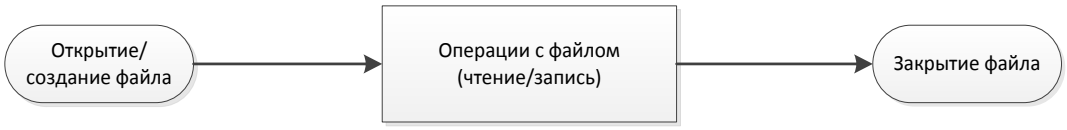


Рисунок 2.2 – Упрощенная схема работы с файлами

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.

Попытка работы с несуществующими файлами, а также, например, открытие уже открытого (или закрытие уже закрытого) файла могут привести к сбоям в работе контроллера – поэтому программист должен учитывать возможность возникновения этих ситуаций и реализовать их обработку.

Библиотека **CAA File** реализует асинхронный доступ к файлам – в связи с этим выполнение блоков может занять несколько циклов, но остальные задачи (визуализация, обмен и т. д.) в течение этого времени будут продолжать выполняться в штатном режиме. В большинстве случаев каждая отдельная операция с файлом (открытие, чтение, запись, закрытие) реализуется 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 – FAT16/FAT32, NTFS, 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

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 накопителям выглядят следующим образом:

- для USB1 **/mnt/ufs/media/sda1**
- для SD **/mnt/ufs/media/mmcblk0p1**
- рабочий каталог контроллера **/mnt/ufs/home/root/CODESYS_WRK/PlcLogic**
- каталог FTP-сервера **/mnt/ufs/home/ftp/in**

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

Рабочая директория для виртуального контроллера версии **SP11 Patch 5:**
C:\ProgramData\CODESYS\CODESYSControlWinV3\35A42129

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.

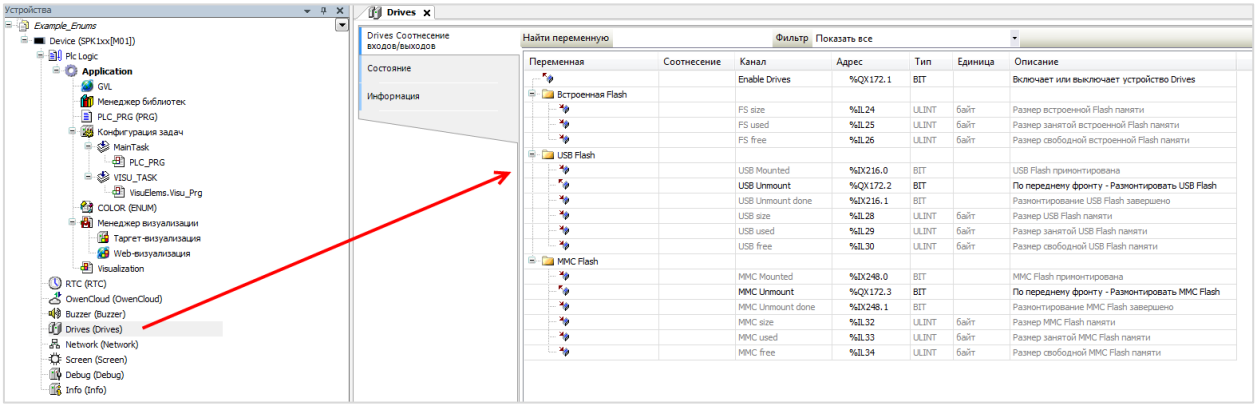


Рисунок 2.3 – Каналы узла Drives

Таблица 2.1 – 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 size	ULINT	Объем Flash-памяти контроллера в байтах[1]
FS used	ULINT	Количество занятой Flash-памяти контроллера в байтах[1]
FS free	ULINT	Количество свободной Flash-памяти контроллера в байтах[1]
USB Flash		
USB Mounted	BOOL	Принимает значение **TRUE** после монтирования USB Flash накопителя, **FALSE** – при демонтировании
USB Unmount	BOOL	По переднему фронту переменной происходит демонтирование USB накопителя
USB Unmount done	BOOL	Принимает значение **TRUE** после демонтирования USB накопителя
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	По переднему фронту переменной происходит демонтирование MMC накопителя
MMC Unmount done	BOOL	Принимает значение **TRUE** после демонтирования MMC накопителя
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

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.

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

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**);
- String Utils (дополнительные функции работы со строками);
- OwenStringUtils (конвертация строки из ASCII в UNICODE и обратно);
- OSCAT (дополнительные функции работы со строками).

Следует отметить, что контроллеры ОВЕН ***не поддерживают кодировку Win1251*** – таким образом, переменные и константы типа **STRING** не могут использоваться для отображения в визуализации кириллических символов. В этом случае следует использовать переменные 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 2.2 – 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**)

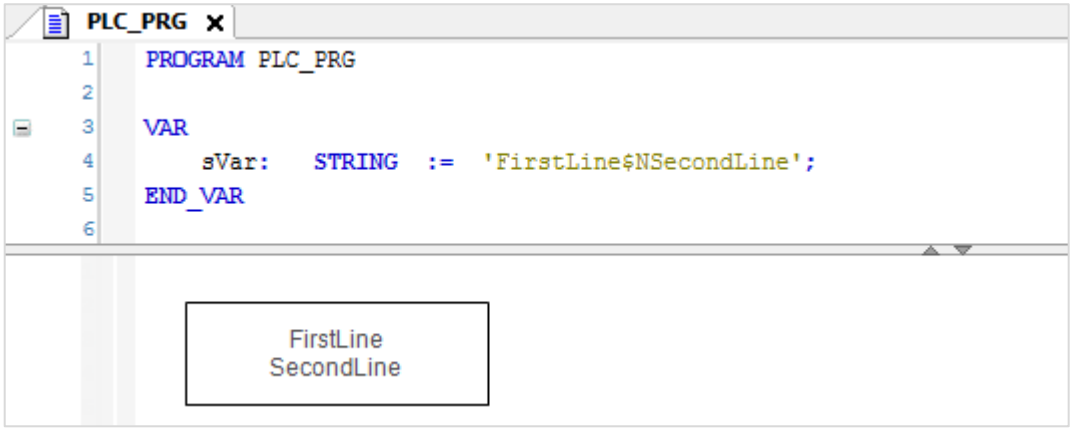


Рисунок 2.4 – Использование управляющих последовательностей

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. Попытку открытия уже открытого файла.
3. Попытку закрытия уже закрытого файла.
4. Проверку монтирования накопителя перед работой с ним.
5. Проверку демонтирования накопителя перед извлечением.
6. Наличие свободного места для архива на накопителе.

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 и имя пользователя – **root**. Поле пароля должно остаться пустым (если только ранее пароль был задан средствами Linux). Чтобы подключиться к контроллеру, следует нажать **Войти**.

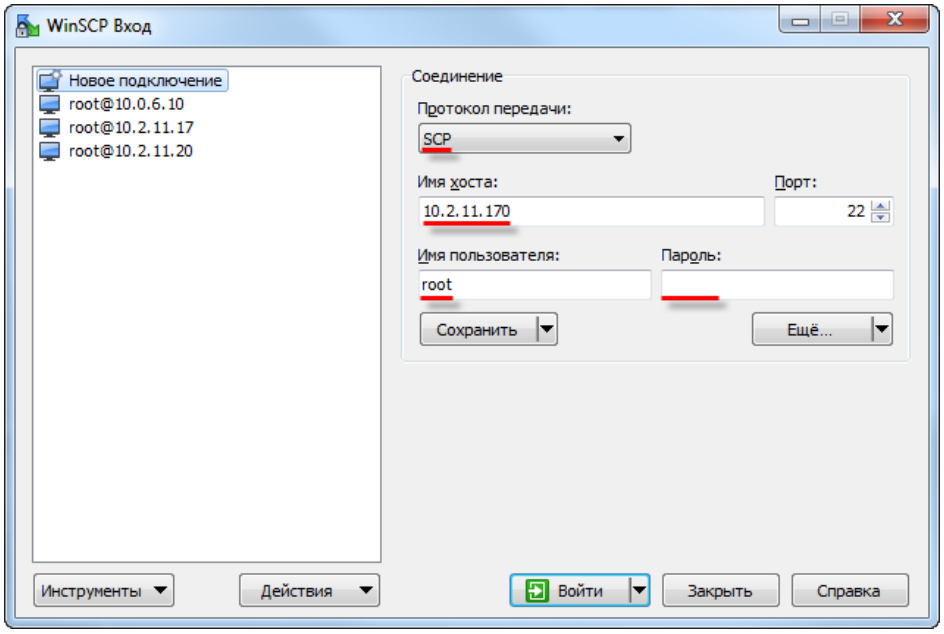


Рисунок 2.5 – Настройки соединения в WinSCP

После появления окна аутентификации пользователя следует нажать кнопку **ОК** (поле **Пароль** следует оставить пустым).

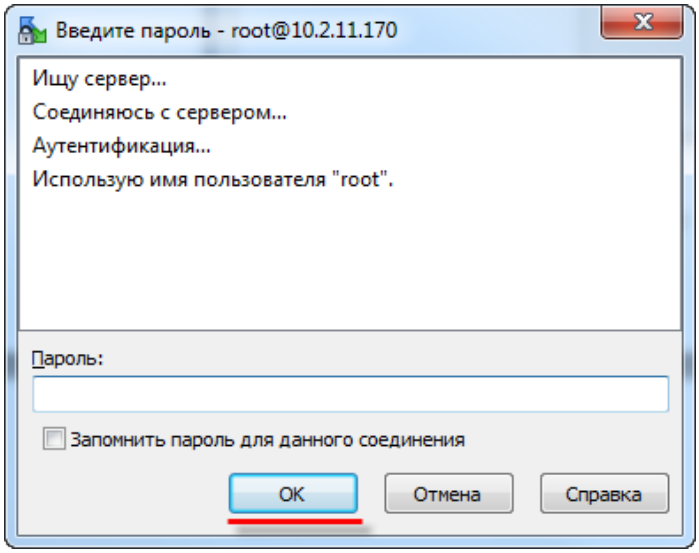


Рисунок 2.6 – Окно аутентификации в WinSCP

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.

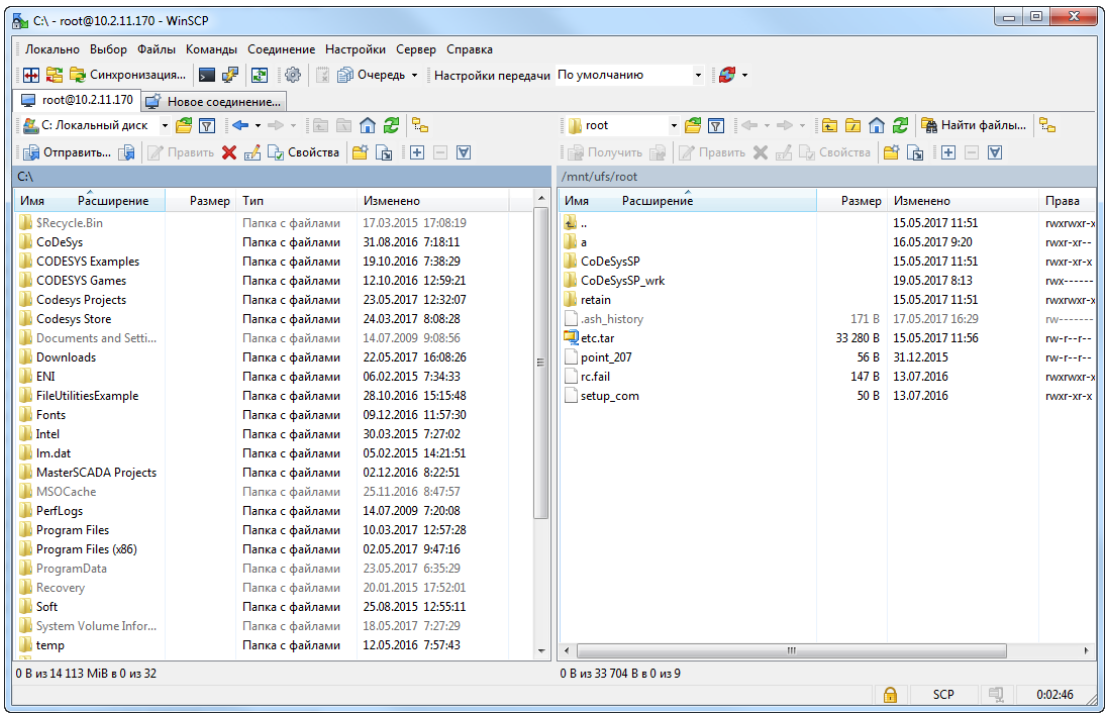


Рисунок 2.7 – Окно файлового менеджера WinSCP

2.9 Работа с FTP

Контроллер может использоваться в режиме FTP-сервера. Логин для доступа: **ftp**, пароль задается в конфигураторе на вкладке **Изменить пароли**. См. более подробную информацию в руководстве **CODESYS V3.5. FAQ**.

Директория FTP-сервера: **/mnt/ufs/home/ftp/in**

Для работы в режиме FTP-клиента следует использовать утилиту **cURL**. Пакет доступен на диске с ПО из комплекта поставки и 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.x.x.** В настоящем руководстве описывается работа с компонентом версии **3.5.4.9**.

Архиватор распространяется в виде пакета формата **.package**. Пакет доступен на диске с ПО из комплекта поставки и 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**.

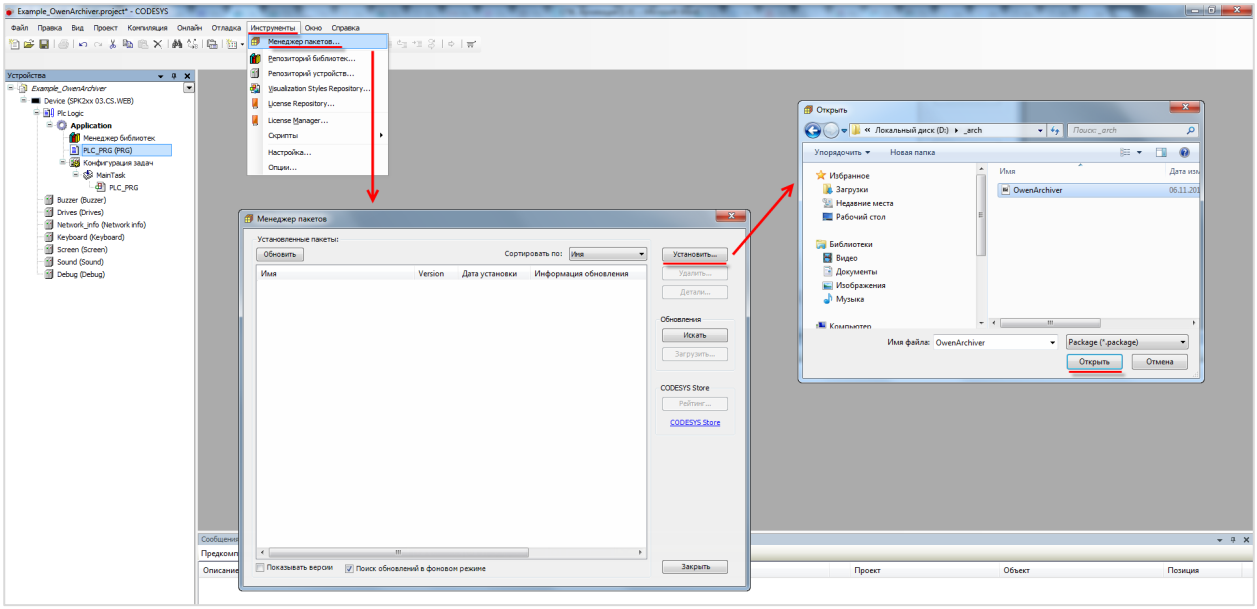


Рисунок 3.1 – Установка пакета OwenArchiver в среду CODESYS

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

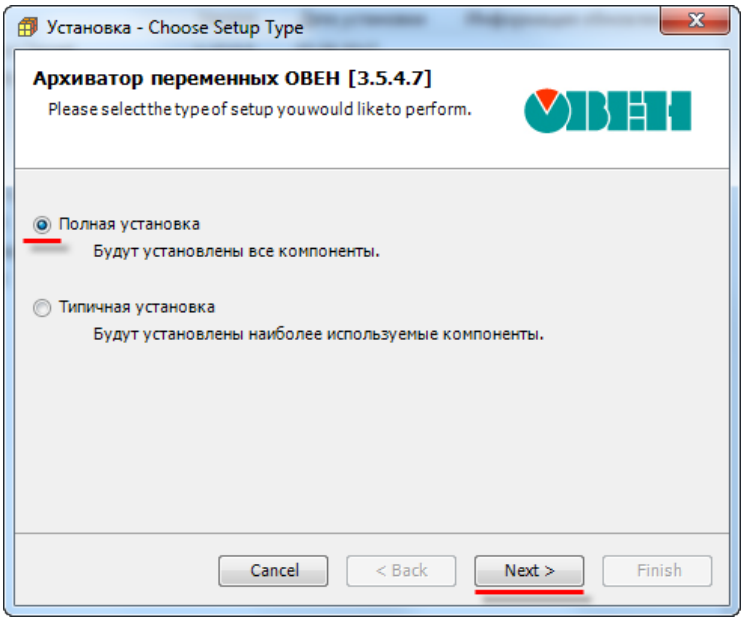


Рисунок 3.2 – Начало установки архиватора

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

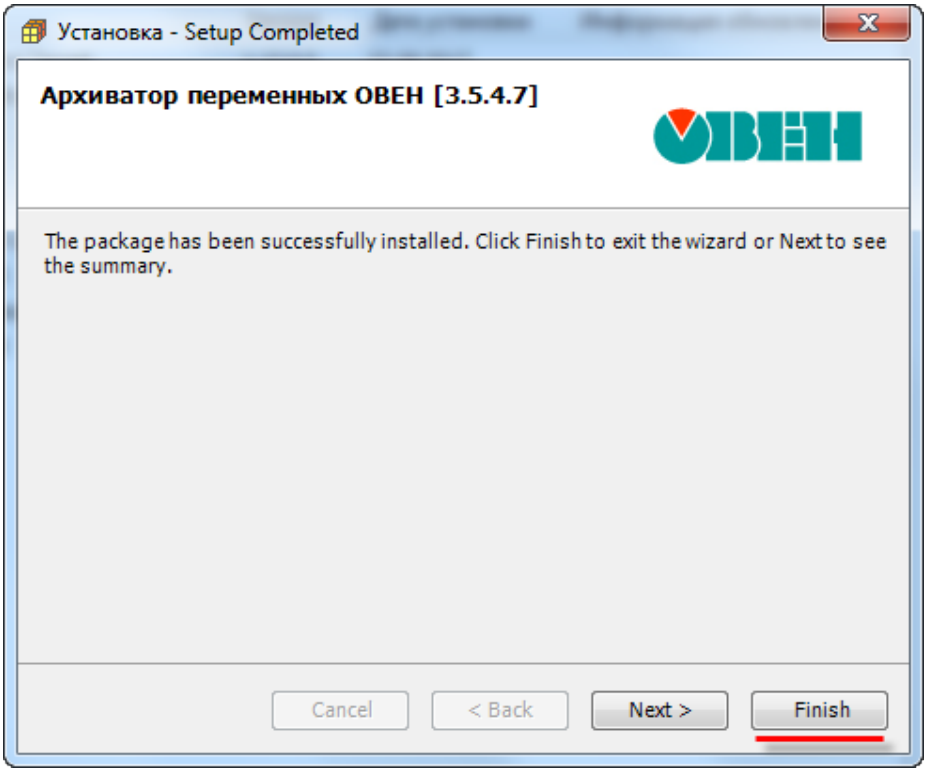


Рисунок 3.3 – Завершение установки архиватора

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**):

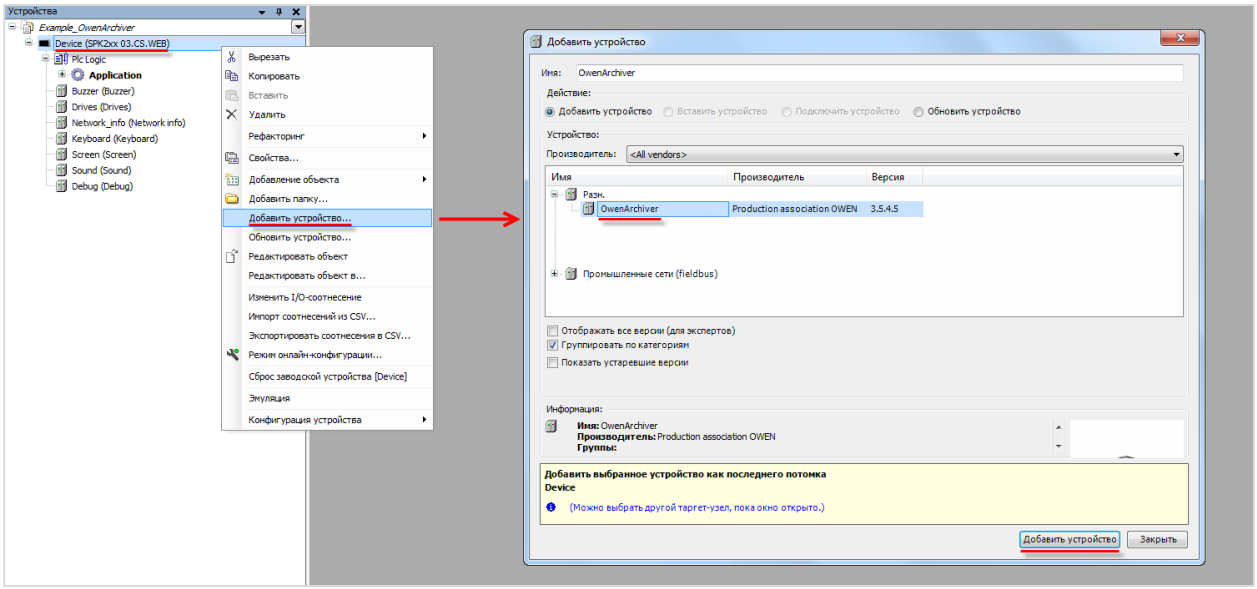


Рисунок 3.4 – Добавление архиватора в проект CODESYS

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

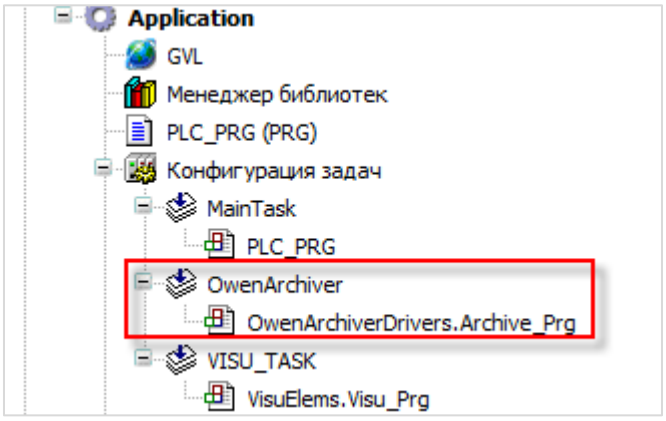


Рисунок 3.5 – Внешний вид дерева проекта после добавления архиватора

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** – запись будет добавлять 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 вестись во внутреннюю память контроллера (в рабочий каталог в папку **/archives/<имя_архива>**);
- **USB-flash** – архив будет вестись на USB-накопитель (в папку **/archives/<имя_архива>**);
- **SD-карта** – архив будет вестись на SD-накопитель (в папку **/archives/<имя_архива>**);
- **Директория FTP** – архив будет вестись во внутреннюю память 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).

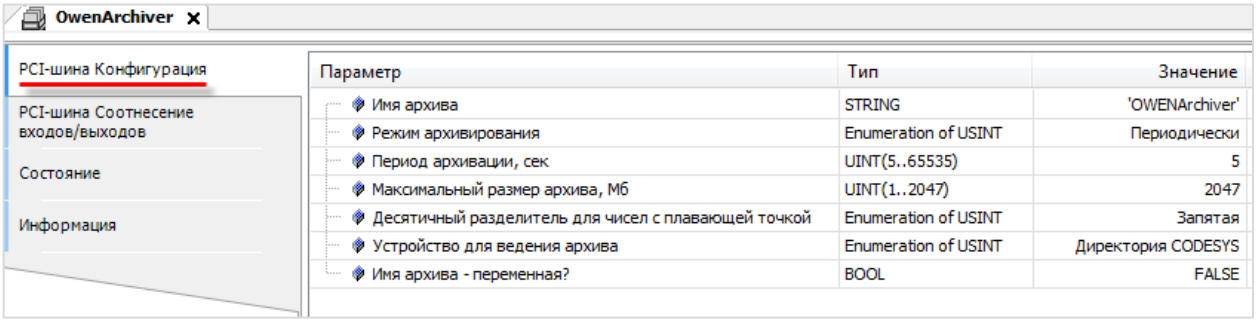


Рисунок 3.6 – Настройки архиватора, вкладка PCI-шина Конфигурация

3. В настройках компонента **OwenArchiver** на вкладке **PCI-шина Соотнесение входов/выходов** привязать к нужным каналам переменные:
4.

Таблица 3.1 – 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)
Запись лога отладки	BOOL	Если параметр имеет значение **TRUE**, то в память контроллера будет вестись 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. Изменения вступают в силу по переднему фронту канала **Запустить архиватор**
Статус архиватора		
Код последней ошибки	USINT	Код последней ошибки архиватора. См. библиотеку **OwenArchiveErrors,** содержащую функции декодирования 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

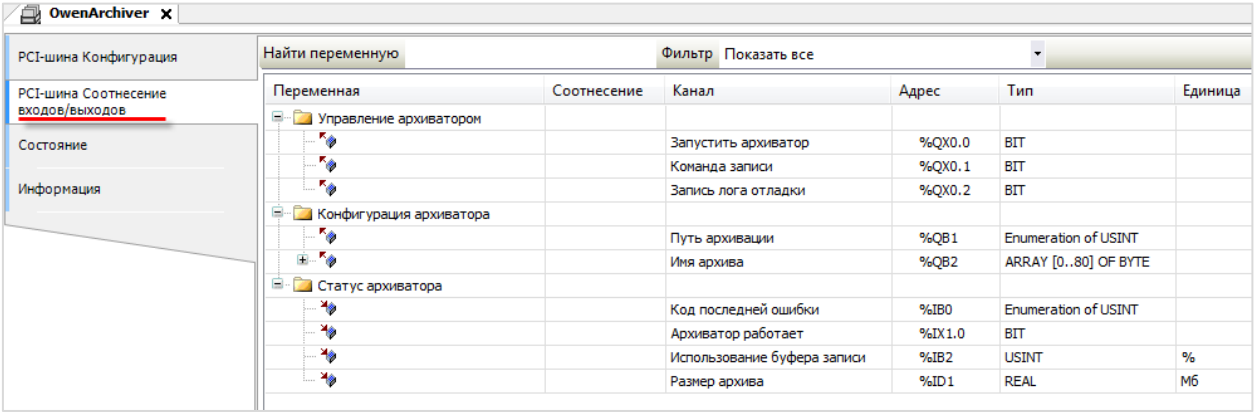


Рисунок 3.7 – Настройки архиватора, вкладка PCI-шина Соотнесение входов/выходов

5. В настройках компонента **CSVFormat** на вкладке **CSVFormat Конфигурация** выбрать структуру архива:
6.

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 папке (название – *номер месяца*), папки за каждый год будут сохранены в корневой папке (название – *номер года*). По достижению максимального размера архива (см. пп. 3) самые старые файлы будут последовательно удаляться. Если в результате удаление файлов папка какого-либо месяца окажется пустой, то она будет удалена;

Год/Месяц_День – архив за каждые сутки будет записываться в отдельный файл (название – *номер месяца_номер дня*), файлы за каждый год будут сохранены в корневой папке (название – *номер года*).

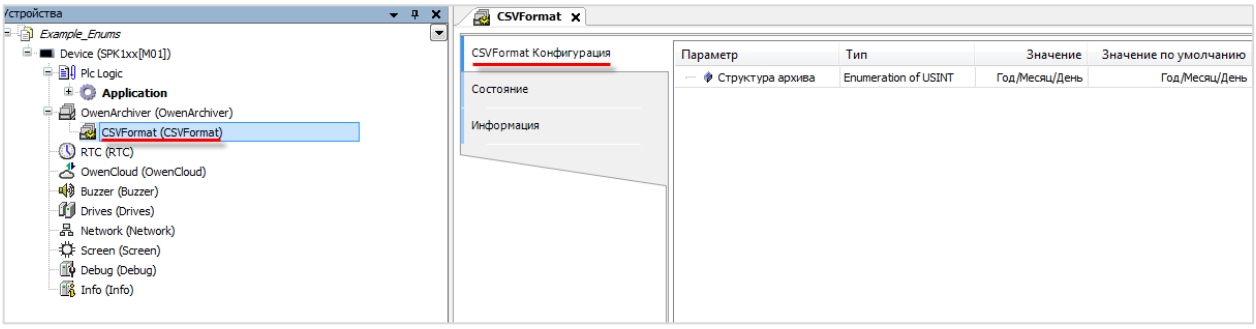


Рисунок 3.8 – Настройки архиватора, вкладка CSVFormat Конфигурация

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

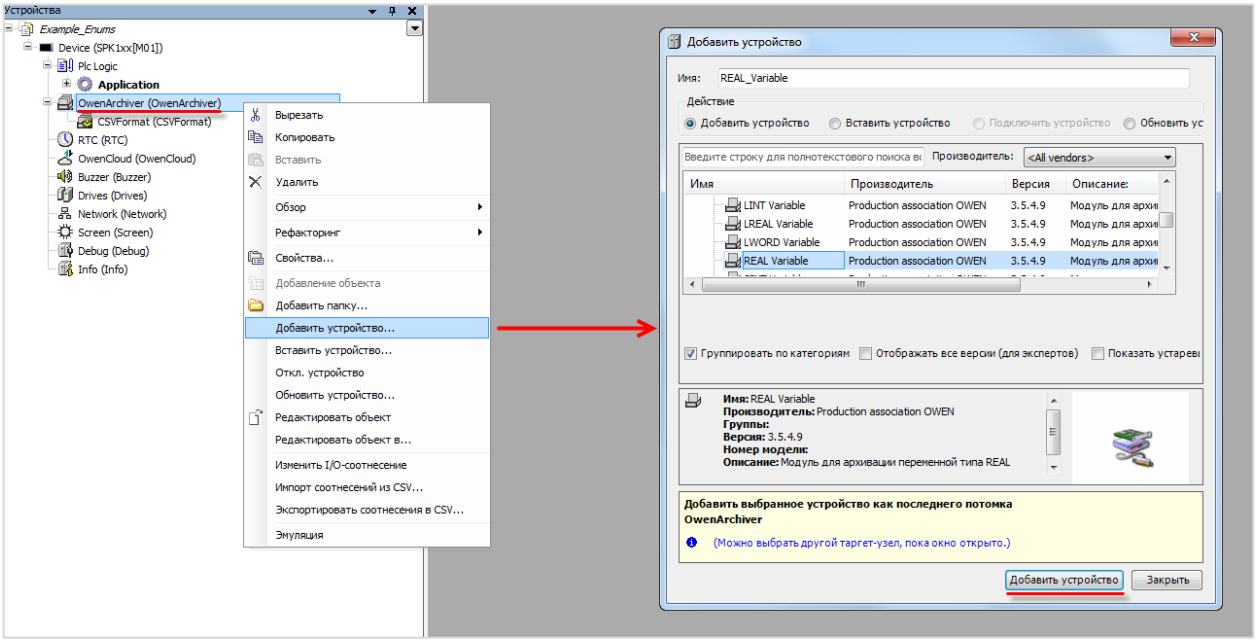


Рисунок 3.9 – Добавление каналов архивируемых переменных

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

Описание переменной – используется при формировании заголовка архива;
Кол-во знаков после десятичного разделителя – количество знаков после запятой для переменных типа **REAL/LREAL**.

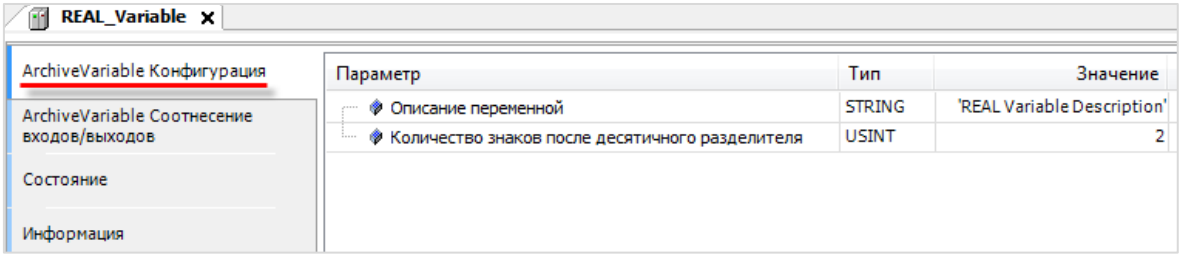


Рисунок 3.10 – Настройки канала архивируемой переменной

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

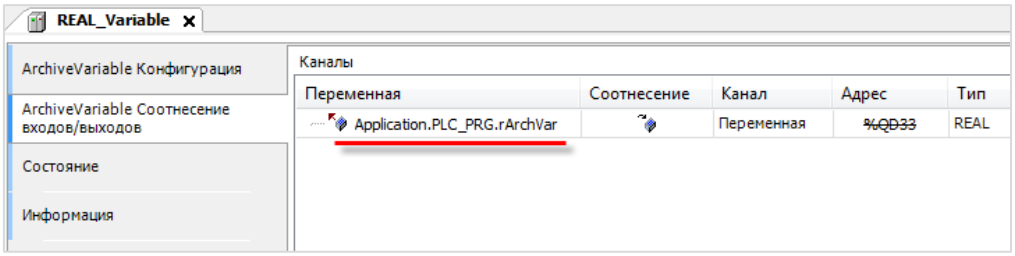


Рисунок 3.11 – Привязка архивируемой переменной к каналу

3.3 Ограничения, связанные с использованием архиватора

1. Максимальное количество переменных для одного архиватора – **64**.
2. 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 операторов может привести к значительной нагрузке на процессор и высоким затратам оперативной памяти – корректная работа контроллера в данном случае не гарантируется.
3. Архиватор использует **память ввода-вывода CODESYS**. Ее количество ограничено и зависит от модели контроллера. 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.
4. 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**.
5. Для архивируемых строковых переменных максимальный размер составляет **80** символов [**STRING(80)**].

3.4 Пример работы с архиватором

Пример создан в среде **CODESYS V3.5 SP11 Patch 5** и подразумевает запуск на **СПК1xx [M01]** с таргет-файлом **3.5.11.x.** 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 3.12 – 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**:

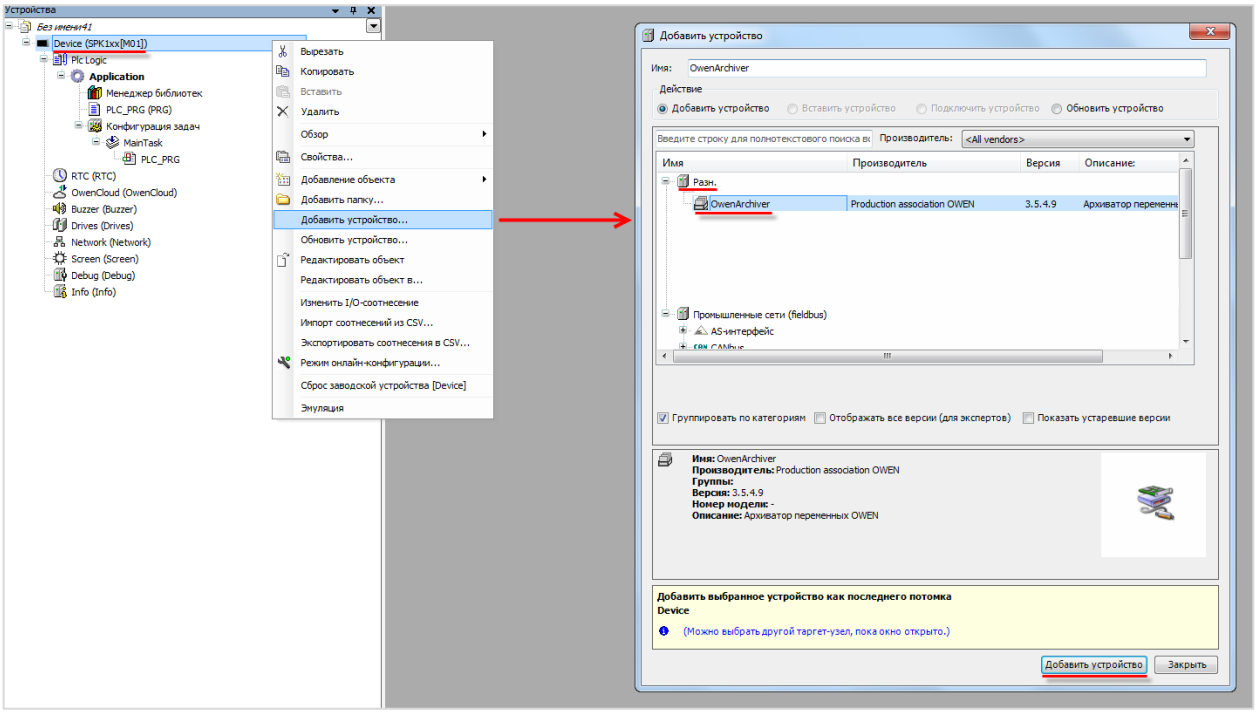


Рисунок 3.13 – Добавление компонента OwenArchiver

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

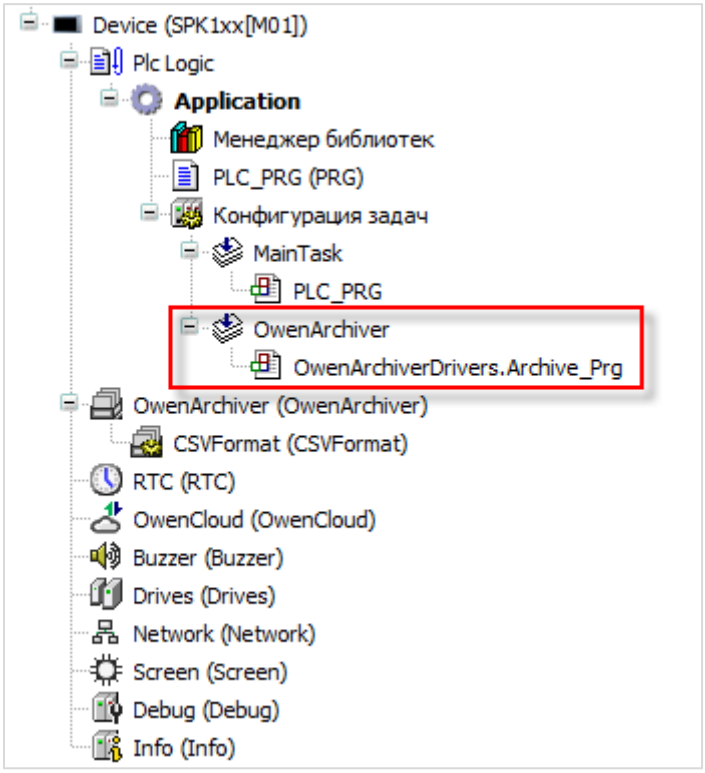


Рисунок 3.14 – Внешний вид дерева проекта после добавления архиватора

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**.

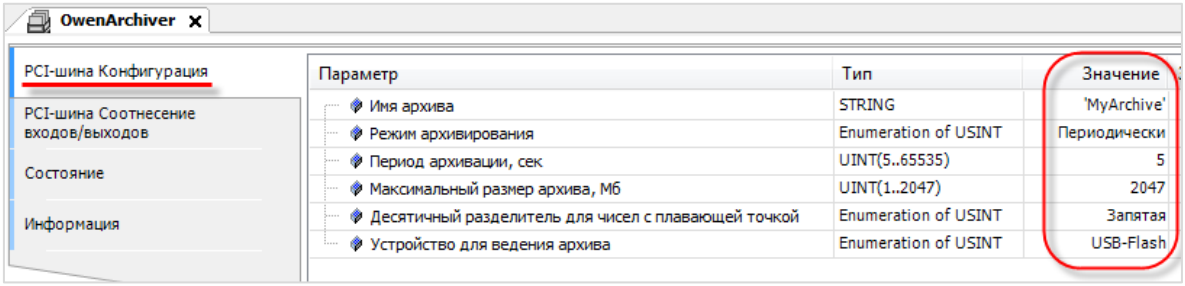


Рисунок 3.15 – Настройка параметров архивации

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

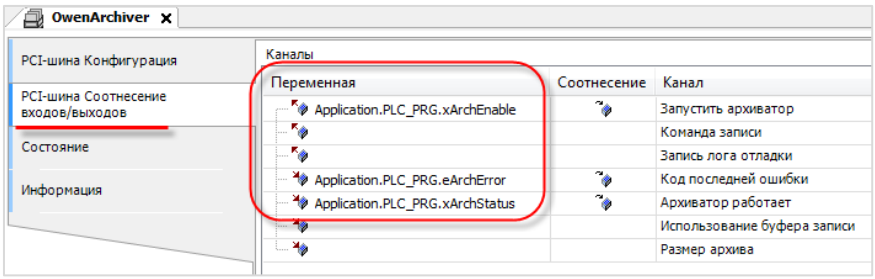


Рисунок 3.16 – Привязка переменных контроля архиватора

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

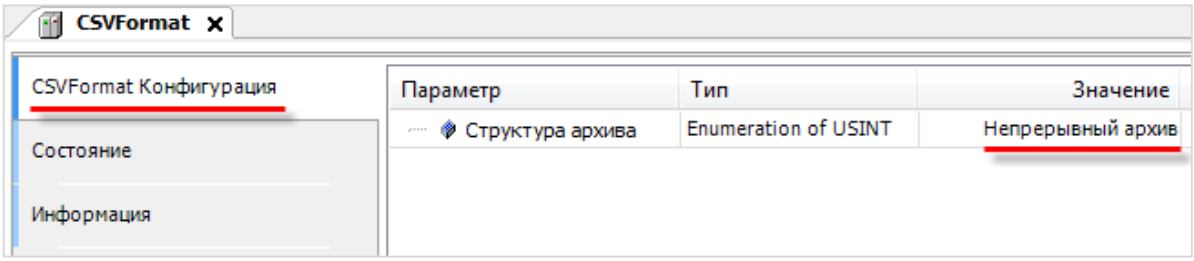


Рисунок 3.17 – Выбор режима архивации

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 продолжит вестись 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).

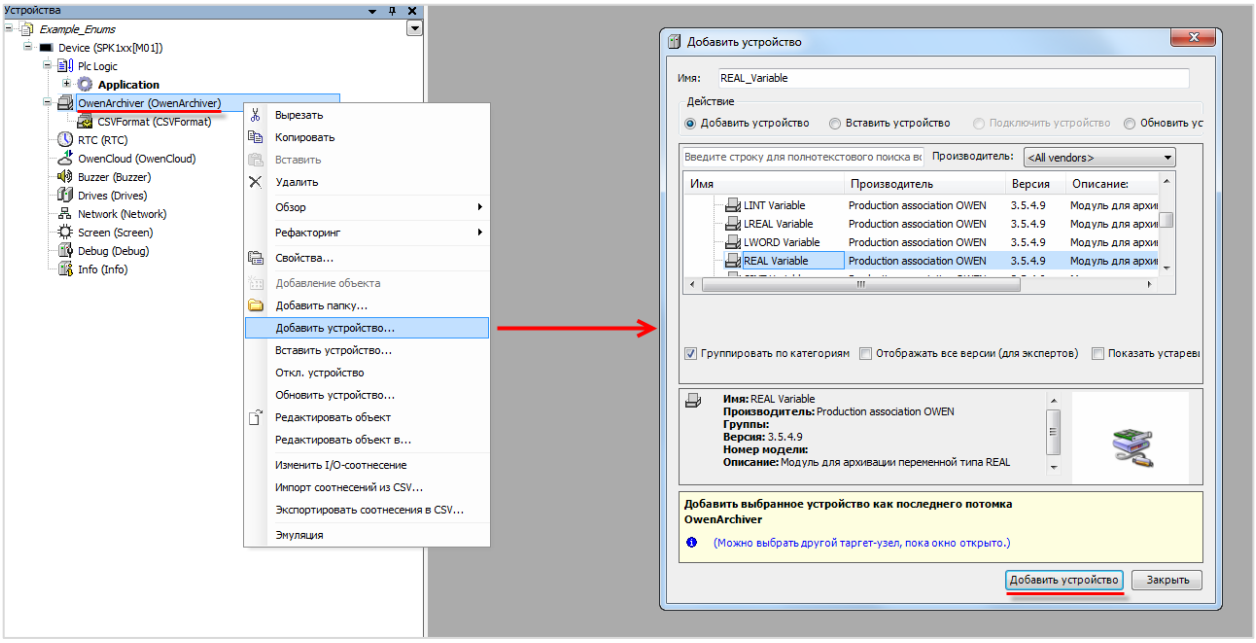


Рисунок 3.18 – Добавление каналов архивации

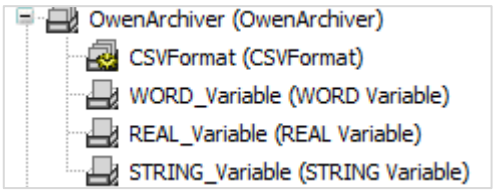


Рисунок 3.19 – Внешний вид дерева проекта после добавления каналов архивации

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

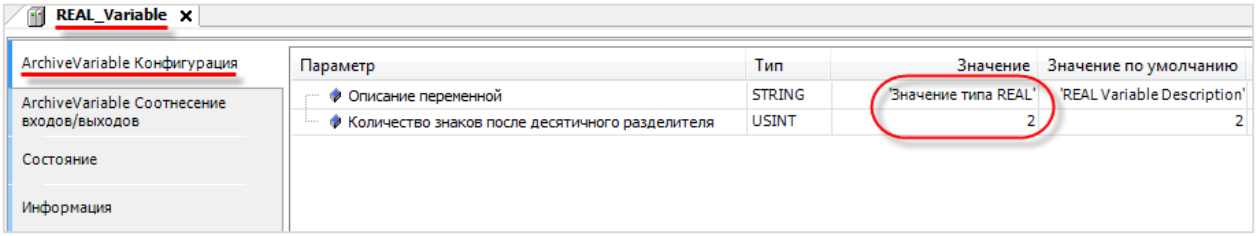


Рисунок 3.20 – Настройка канала архивации

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**.

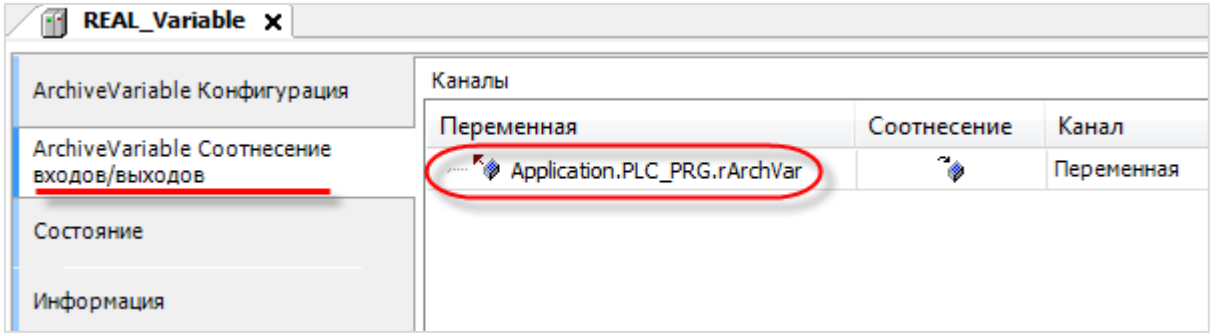


Рисунок 3.21 – Привязка переменных к каналам архивации

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 3.22 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 3.22 – Внешний вид экрана визуализации

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

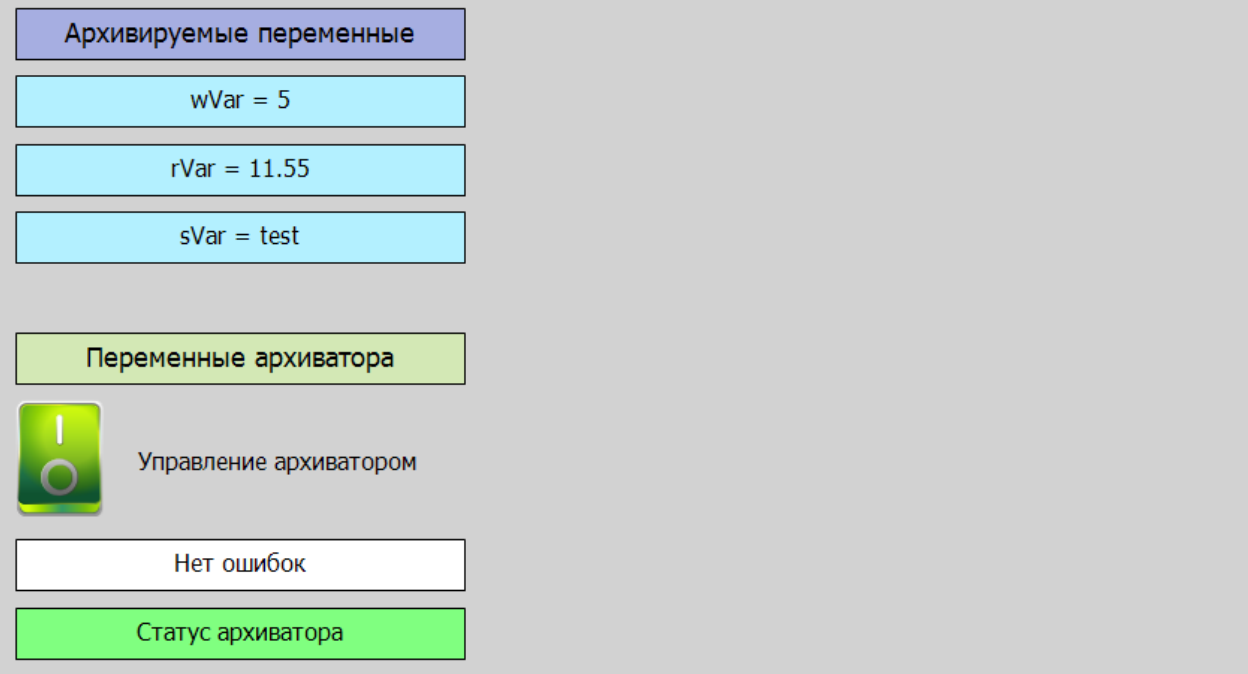
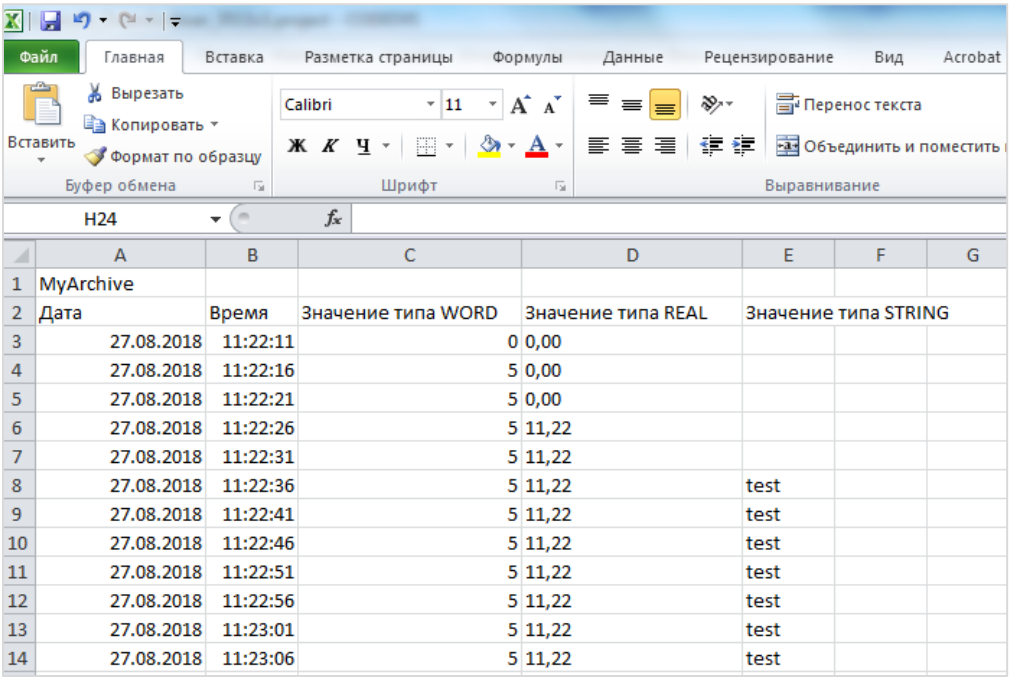


Рисунок 3.23 – Внешний вид экрана визуализации

8. В корневой папке 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 3.24 – 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.

Библиотека реализует асинхронный доступ к файлам – поэтому выполнение блоков может занять несколько циклов 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**, расположенную в папке **Intern/CAA/System**.

ПРИМЕЧАНИЕ

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

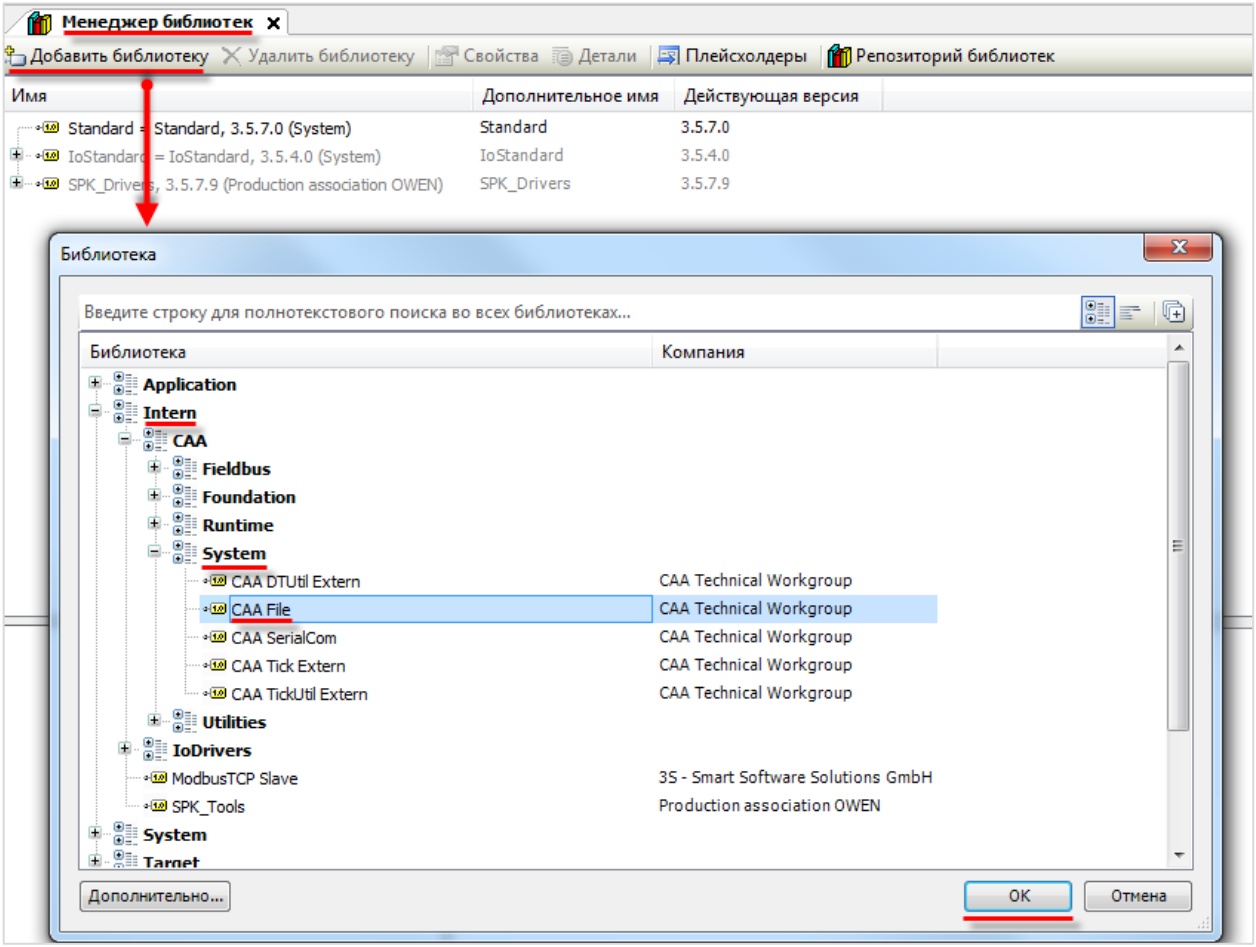


Рисунок 4.1 – Добавление библиотеки CAA File в проект CODESYS

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 в случае работы с ФБ FILE.DirList.

Таблица 4.1 – Описание переменных структуры FILE.FILE_DIR_ENTRY

Название	Тип данных	Описание
sEntry	CAA.FILENAME	Имя каталога или файла
szSize	CAA.SIZE	Размер каталога/файла в байтах. В версии библиотеки **3.5.11.0** и ниже некорректно определяется размер каталогов
xDirectory	BOOL	**TRUE** – каталог, **FALSE** – файл
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 **3.5.11.0** и ниже некорректно определяется дата и время последнего изменения каталогов

4.2.2 Перечисление FILE.ERROR

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

Таблица 4.2 – 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

4.2.3 Перечисление FILE.MODE

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

Таблица 4.3 – 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)

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.

При этом рабочим каталогом CODESYS в контроллерах ОВЕН является **/mnt/ufs/home/root/CODESYS_WRK/PlcLogic.**

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

- Если sDirName='test1', то результатом работы ФБ является создание каталога **test1** в каталоге **/mnt/ufs/home/root/CODESYS_WRK/PlcLogic** (т. е. в рабочем каталоге);
- Если sDirName='/mnt/ufs/home/root/test2', то результатом работы ФБ является создание каталога **test2** в каталоге **/mnt/ufs/home/root**.

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

4.4 Ограничения при работе с файлами

Максимальное количество одновременно выполняемых операций (открытие, чтение, запись и др.) с каталогами и файлами не должно превышать **20-ти** (по возможности рекомендуется в каждый момент времени работать только с одним файлом)**.** Операция считается незавершенной, пока вход **xExecute** имеет значение **TRUE** – поэтому рекомендуется запускать работу блоков с помощью единичных импульсов по переднему фронту. В случае нарушения этого правила при попытке открытия **21-го** файла на выходе ФБ FILE.OPEN возникает ошибка **5802.**

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.

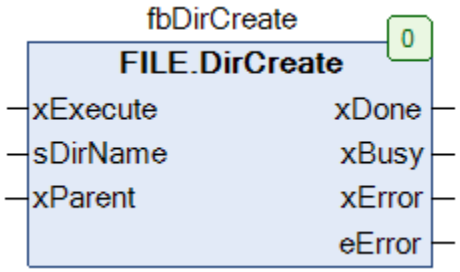


Рисунок 4.2 – Внешний вид ФБ FILE.DirCreate на языке CFC

Таблица 4.1 – 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 указано более одного несуществующего каталога, то блок завершает работу с сообщением об ошибке
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 имя ошибки)

4.5.2 ФБ FILE.DirOpen

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

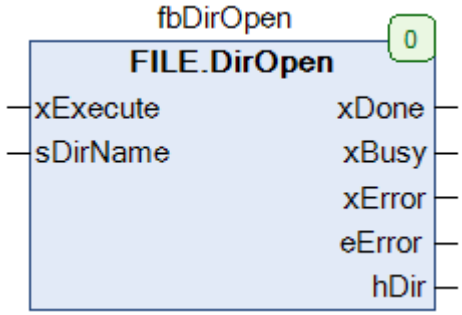


Рисунок 4.3 – Внешний вид ФБ FILE.DirOpen на языке CFC

Таблица 4.2 – 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 имя 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**.

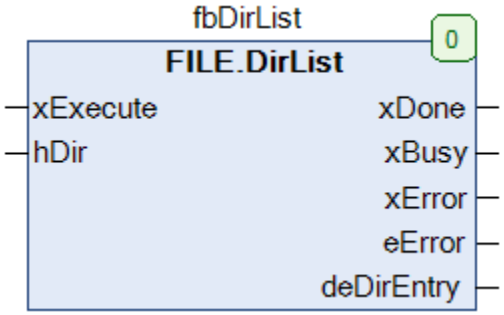


Рисунок 4.4 – Внешний вид ФБ FILE.DirList на языке CFC

Таблица 4.3 – 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 имя ошибки)
deDirEntry	FILE.FILE_DIR_ENTRY	Информация о каталоге/файле

4.5.4 ФБ FILE.DirRemove

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

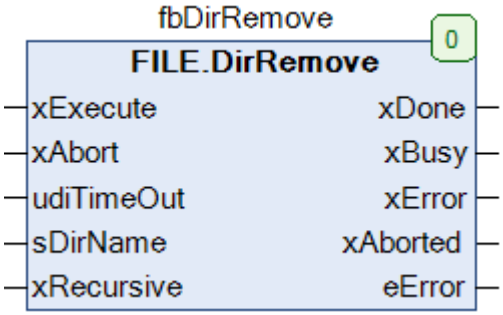


Рисунок 4.5 – Внешний вид ФБ FILE.DirRemove на языке CFC

Таблица 4.4 – 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 сообщение об ошибке
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 имя ошибки)

4.5.5 ФБ FILE.DirRename

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

Рисунок 4.6 – Внешний вид ФБ FILE.DirRename на языке CFC

Таблица 4.5 – 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 имя 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.

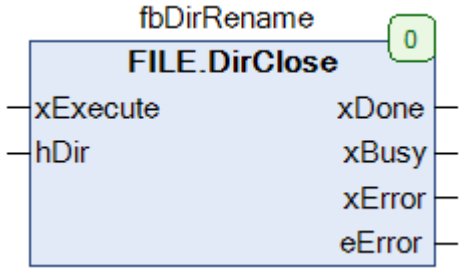


Рисунок 4.7 – Внешний вид ФБ FILE.DirClose на языке CFC

Таблица 4.6 – 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 имя ошибки)

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.

ПРИМЕЧАНИЕ
Попытка открытия ранее открытого (и не закрытого) файла может привести к ошибкам в работе контроллера.

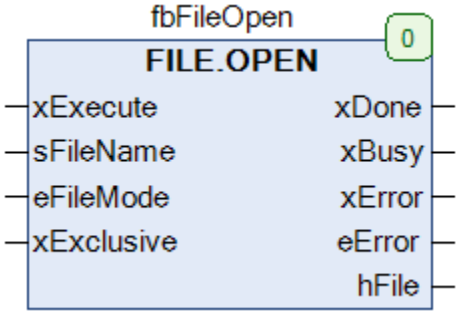


Рисунок 4.8 – Внешний вид ФБ FILE.OPEN на языке CFC

Таблица 4.7 – 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
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 имя 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.

ПРИМЕЧАНИЕ
Попытка закрытия ранее закрытого файла (или еще не открытого файла) может привести к ошибкам в работе контроллера.

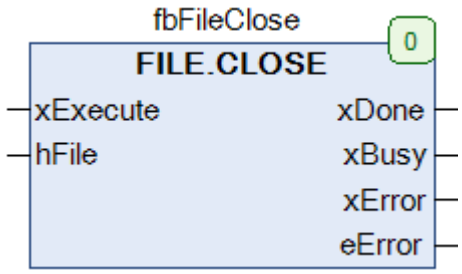


Рисунок 4.9 – Внешний вид ФБ FILE.Close на языке CFC

Таблица 4.8 – 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 имя 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.

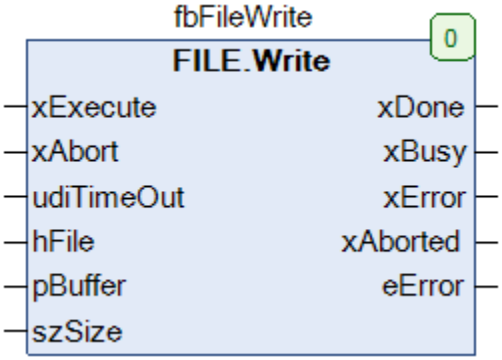


Рисунок 4.10 – Внешний вид ФБ FILE.WRITE на языке CFC

Таблица 4.9 – 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 имя 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.

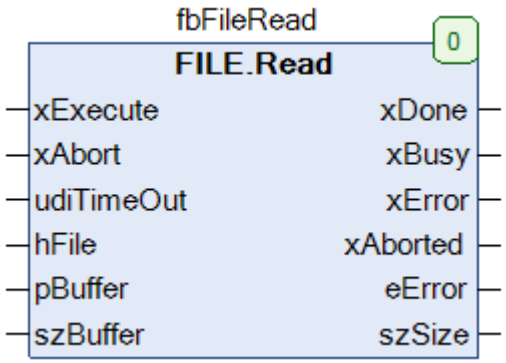


Рисунок 4.11 – Внешний вид ФБ FILE.READ на языке CFC

Таблица 4.10 – 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	Максимально допустимый 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 имя ошибки)
szSize	CAA.SIZE	Размер считанных данных в байтах

4.6.5 ФБ FILE.RENAME

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

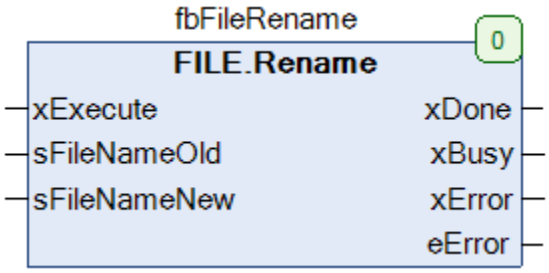


Рисунок 4.12 – Внешний вид ФБ FILE.RENAME на языке CFC

Таблица 4.11 – 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 имя ошибки)

4.6.6 ФБ FILE.COPY

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

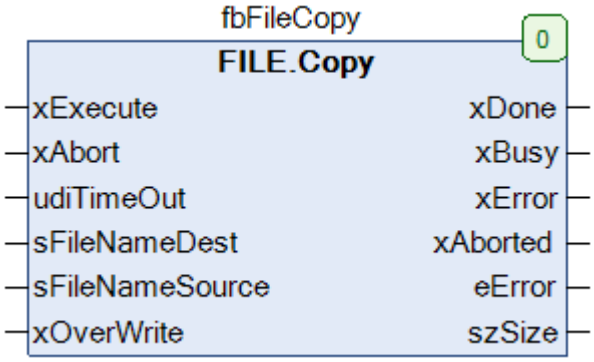


Рисунок 4.13 – Внешний вид ФБ FILE.COPY на языке CFC

Таблица 4.12 – 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 переменно
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 выдаст сообщение об ошибке **ERROR.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 имя ошибки)
szSize	CAA.FILE.SIZE	Размер скопированных данных в байтах

4.6.7 ФБ FILE.DELETE

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

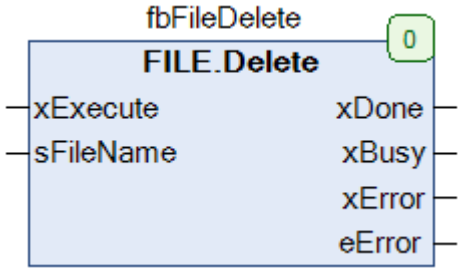


Рисунок 4.14 – Внешний вид ФБ FILE.DELETE на языке CFC

Таблица 4.13 – 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 имя 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) сохранения данных в файл может не произойти. Использование Flush гарантирует, что данные сразу будут сохранены в файл. В то же время использование данной функции может привести к более быстрому истощению ресурса накопителя.

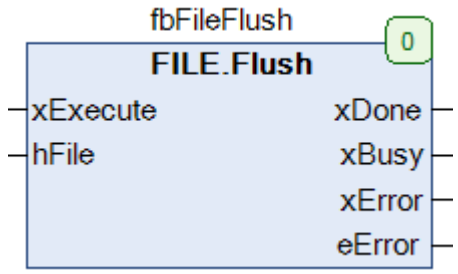


Рисунок 4.15 – Внешний вид ФБ FILE.FLUSH на языке CFC

Таблица 4.14 – 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 имя 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.

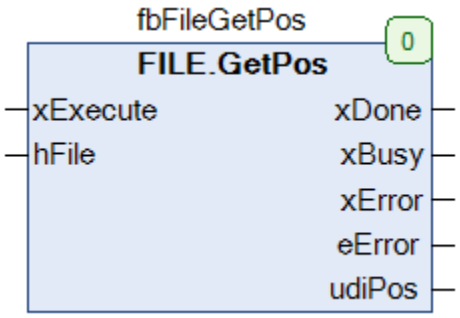


Рисунок 4.16 – Внешний вид ФБ FILE.GetPos на языке CFC

Таблица 4.15 – Описание входов и выходов ФБ FILE.GetPost

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 имя 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.

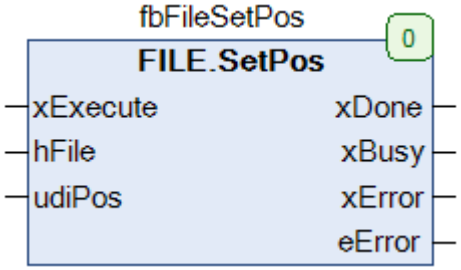


Рисунок 4.17 – Внешний вид ФБ FILE.SetPos на языке CFC

Таблица 4.16 – 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 имя 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.

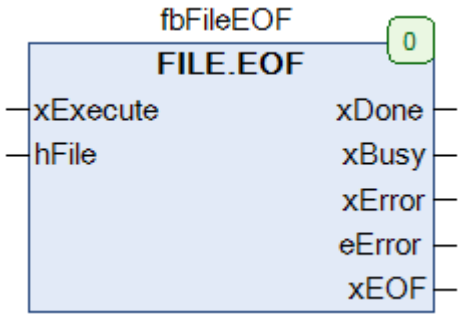


Рисунок 4.18 – Внешний вид ФБ FILE.EOF на языке CFC

Таблица 4.17 – 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 имя 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.

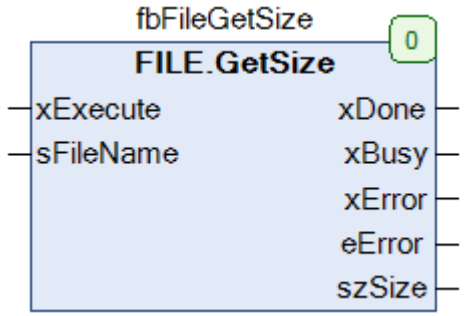


Рисунок 4.19 – Внешний вид ФБ FILE.GetSize на языке CFC

Таблица 4.18 – 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 имя 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.

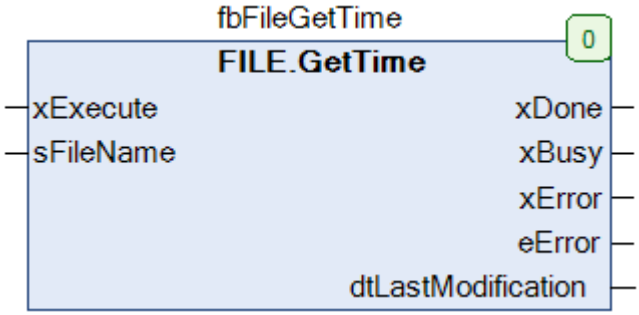


Рисунок 4.20 – Внешний вид ФБ FILE.GetTime на языке CFC

Таблица 4.19 – 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 имя 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:

- все три программы привязаны к задаче **MainTask** с временем цикла **20 мс**;
- все программы, ФБ и функции написаны на языке ST;
- все рисунки, приведенные в документе, хорошо масштабируются;
- листинг POU примера приведен в Приложении.

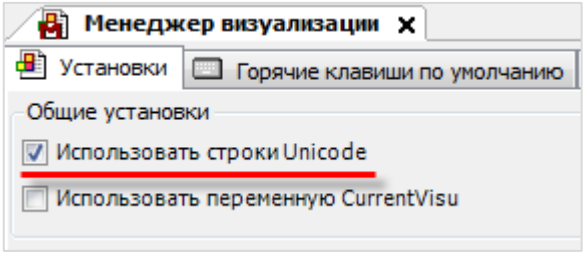
Таблица 5.1 – Структура примера

№ п/п	Функционал	Программа
1	Получение информации о памяти контроллера и подключенных накопителей	PLC_PRG
2	Работу с каталогами (создание, удаление, переименование, просмотр содержимого)	PLC_PRG
3	Запись и чтение архивов в бинарном формате	BinFileExample_PRG
4	Запись архивов в формате **.csv**	StringFileExample_PRG
5	Работу с файлами (копирование, удаление и т. д.)	PLC_PRG

Пример создан в среде **CODESYS V3.5 SP11 Patch 5** и подразумевает запуск на **СПК1xx [M01]** с таргет-файлом **3.5.11.x.** 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_CAA_File.projectarchive

Для отображения в визуализации русскоязычных символов необходимо в **Менеджере визуализации** поставить галочку **Использовать строки Unicode**. Следует помнить, что для вывода кириллического текста должны использоваться переменные типа **WSTRING**.



5.2 Использованные библиотеки

Для создания примера были использованы следующие библиотеки:

- **CAA File** (3.5.11.0) – для работы с файлами;
- **CAA DTUtil** (3.5.11.0) – для работы с системным временем;
- **Standard64** (3.5.2.0) – для работы со строками типа **WSTRING**.

Для повторения примера из документа следует добавить эти библиотеки в проект CODESYS.

5.3 Содержимое примера

Таблица 5.2. – Описание POU примера

Компонент	Где используется	Описание
Программы		
PLC_PRG	Пример получения информации о накопителях, работе с каталогами и базовых операций с файлами	
BinFileExample_PRG	Пример экспорта и импорта бинарного файла	
StringFileExample_PRG	Пример экспорта строкового файла	
Действия		
act01_DriveInfo	PLC_PRG	Получение информация о накопителях
act02_DirExample		Работа с каталогами
act03_DirList		Просмотр содержимого каталогов
act04_ActionsWithFiles		Дополнительные операции с файлами
Структуры		
ArchData	BinFileExample_PRG, StringFileExample_PRG	Архивируемые данные
DriveInfo	PLC_PRG	Информация о накопителе
VisuDirInfo	PLC_PRG	Информация о содержимом каталога
Перечисления		
FileDevice	DEVICE_PATH	Названия накопителей
FileWork	BinFileExample_PRG, StringFileExample_PRG, DIR_INFO	Имена шагов работы с файлами
Функции и ФБ		
BYTE_SIZE_TO_WSTRING	PLC_PRG (act01, act03)	Конвертация числа байт в формат. строку
CONCAT11	PLC_PRG (act03), StringFileExample_PRG	Склеивание 11-ти строковых переменных
DEVICE_PATH	PLC_PRG (act02,act03,act04) StringFileExample_PRG, BinFileExample_PRG	Определение пути к выбранному устройству
DIR_INFO	PLC_PRG (act03)	Получение информации о каталоге
LEAD_ZERO	SPLIT_DT_TO_FSTRINGS	Добавление ведущего нуля к числу
REAL_TO_FSTRING	StringFileExample_PRG	Конвертация REAL в формат. строку
REAL_TO_FWSTRING	PLC_PRG (act01)	Конвертация REAL в формат. строку
SPLIT_DT_TO_FSTRINGS	PLC_PRG (act03), StringFileExample_PRG	Выделение из метки времени отдельных разрядов

5.4 Получение информации о накопителях (PLC_PRG, действие act01_DriveInfo)

Таргет-файлы контроллеров ОВЕН содержат узел **Drives,** который используется для получения информации о памяти контроллера и подключенных накопителей. Список каналов узла и их описание приведены в п. 2.4.

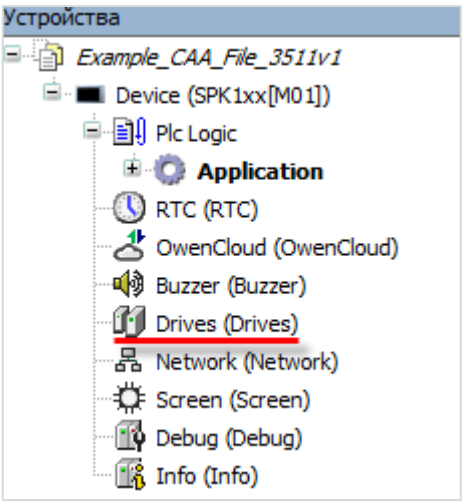


Рисунок 5.4.1 – Узел Drives в дереве проекта

5.4.1 Объявление переменных

Сначала в проекте следует объявить структуру **DriveInfo**, которая будет описывать параметры накопителя (**Application – Добавление объекта – DUT – Структура**):

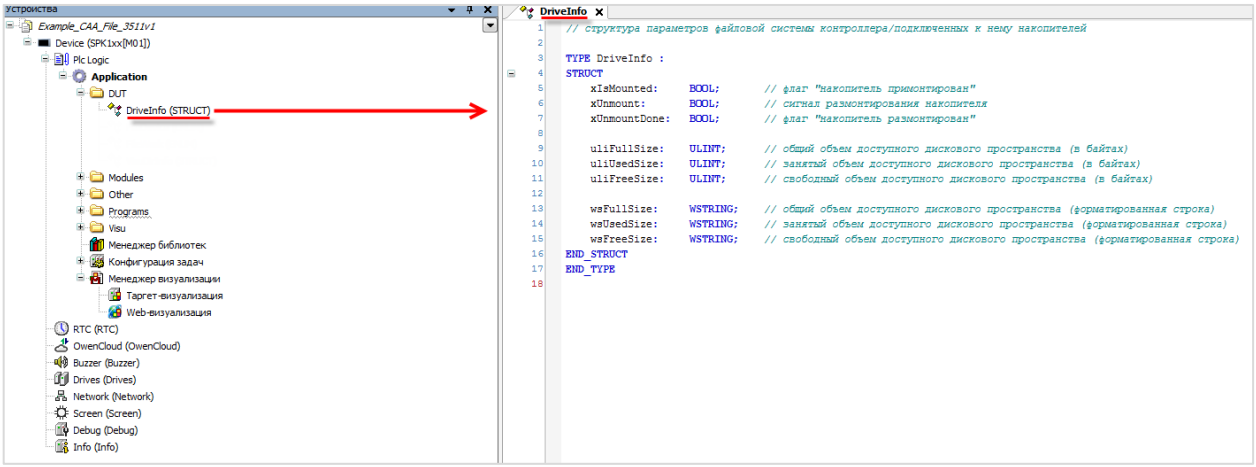


Рисунок 5.4.2 – Объявление структуры DriveInfo

Помимо шести переменных, соответствующих каналам вкладки **Drives,** следует дополнительно объявить три **WSTRING** переменных для отображения общего/занятого/свободного объема накопителей в визуализации – так как отображение объема в виде числа байт не будет удобным для оператора.

Во время работы с файлами в контроллерах ОВЕН можно использовать следующие места хранения файлов:

- Память контроллера;
- USB-накопитель;
- SD-накопитель.

Следует объявить в программе **PLC_PRG** три экземпляра структуры **DriveInfo**. Также следует объявить логическую переменную **xDriveInfo** с начальным значением **TRUE**, которая будет использоваться для запуска процесс сбора данных о накопителях, и два таймера **TON** (необходимость их объявления будет пояснена чуть позднее).

PLC_PRG

```
1  // пример действий с каталогами и файлами (помимо чтения и записи)
2
3  PROGRAM PLC_PRG
4  VAR
5      (*act01_DriveInfo | информация о памяти СПК и накопителей*)
6
7      xDriveInfo:                 BOOL    :=  TRUE;       // режим сбора данных (TRUE - вкл.)
8
9      stPlcMemory:                DriveInfo;              // структура параметров памяти контроллера
10     stUsbMemory:                DriveInfo;              // структура параметров памяти USB-накопителя
11     stSdMemory:                 DriveInfo;              // структура параметров памяти SD-накопителя
12
13     fbUsbUnmountTimeout:        TON;                    // таймер сброса флага "USB отмонтирован"
14     fbSdUnmountTimeout:         TON;                    // таймер сброса флага "SD отмонтирован"
```

Рисунок 5.4.3 – Объявление переменных в программе PLC_PRG

Затем следует привязать переменные объявленных экземпляров структур к соответствующим каналам узла **Drives**. Следующие переменные останутся непривязанными:

- в структуре **stSpkMemory** – xIsMounted, xUnmount, xUnmoundDone (память контроллера нельзя монтировать и демонтировать);
- во всех структурах – переменные типа **WSTRING** (они будут использоваться в визуализации).

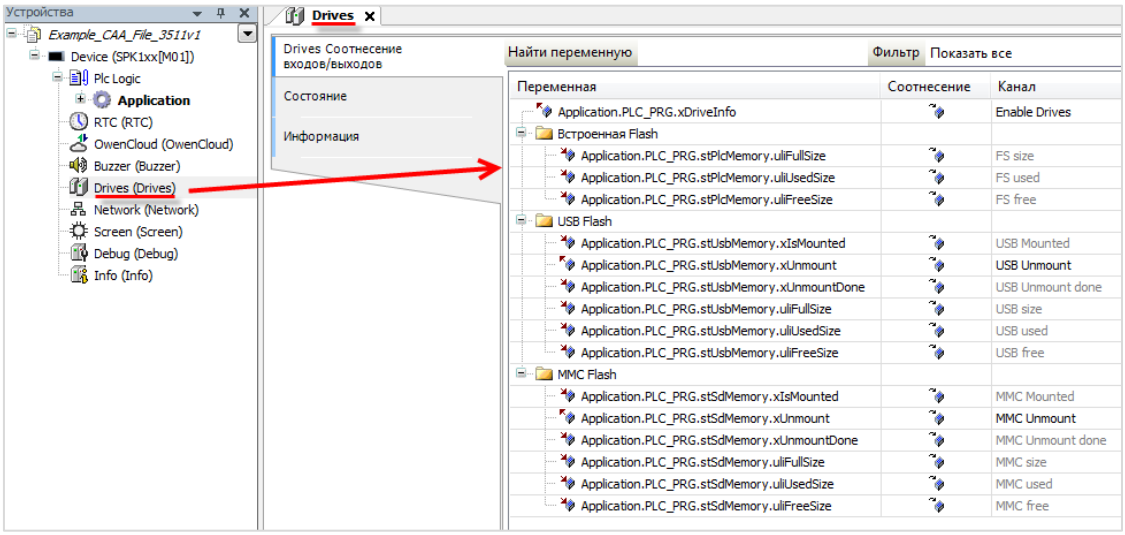


Рисунок 5.4.4 – Привязка переменных к каналам узла Drives

Уже на этом этапе разработки примера после загрузки проекта в привязанные переменные будет считана информация о накопителях:

Device.Application.PLC_PRG

Выражение	Тип	Значение
xDriveInfo	BOOL	TRUE
stPlcMemory	DriveInfo	
stUsbMemory	DriveInfo	
xIsMounted	BOOL	TRUE
xUnmount	BOOL	FALSE
xUnmountDone	BOOL	FALSE
uliFullSize	ULINT	1989115904
uliUsedSize	ULINT	1467666432
uliFreeSize	ULINT	521449472
wsFullSize	WSTRING	" "
wsUsedSize	WSTRING	" "
wsFreeSize	WSTRING	" "

Рисунок 5.4.5 – Значение переменных, привязанных к каналам узла Drives, во время работы проекта

5.4.2 Разработка программы

Отображение объема накопителя в виде числа байт не будет наглядным для оператора. Поэтому его следует преобразовать в более читабельный формат (например, «11.22 Мбайт»). Для преобразования следует создать две функции – **BYTE_SIZE_TO_WSTRING** и **REAL_TO_FWSTRING**.

Функция **BYTE_SIZE_TO_WSTRING** преобразует число байт в форматированную строку. В зависимости от диапазона, в котором находится значение, оно будет конвертировано в наиболее подходящие единицы: например, 1023 байта будут конвертированы в строку «1023 Байт», а 1030 байта – в строку «1.006 Кбайт». Код функции приведен на рисунке 5.4.6:

BYTE_SIZE_TO_WSTRING

```
// функция преобразования числа байт в форматированную строку

FUNCTION BYTE_SIZE_TO_WSTRING : WSTRING
VAR_INPUT
    uliByteSize:                ULINT;                                      // число байт
END_VAR
VAR CONSTANT
    c_uliBytePerKilobyte:       ULINT := 1024;                              // число байт в килобайте
    c_uliKilobytePerMegabyte:   ULINT := 1024 * c_uliBytePerKilobyte;       // число килобайт в мегабайте
    c_uliMegabytePerGigabyte:   ULINT := 1024 * c_uliKilobytePerMegabyte;   // число мегабайт в гигабайте
END_VAR
VAR
    rByteSize:                  REAL;                                       // промежуточная переменная
END_VAR
```

```
CASE uliByteSize OF

    0  ..(c_uliBytePerKilobyte - 1):
            BYTE_SIZE_TO_WSTRING    :=  WCONCAT(ULINT_TO_WSTRING(uliByteSize), " Байт");

    c_uliBytePerKilobyte ..(c_uliKilobytePerMegabyte - 1):
            rByteSize   :=  ULINT_TO_REAL(uliByteSize)  /   ULINT_TO_REAL(c_uliBytePerKilobyte);
            BYTE_SIZE_TO_WSTRING    :=  WCONCAT(REAL_TO_FWSTRING(rByteSize, 2), " Кбайт");

    c_uliKilobytePerMegabyte ..(c_uliMegabytePerGigabyte - 1):
                        rByteSize   :=  ULINT_TO_REAL(uliByteSize)  /   ULINT_TO_REAL(c_uliKilobytePerMegabyte);
                        BYTE_SIZE_TO_WSTRING    :=  WCONCAT(REAL_TO_FWSTRING(rByteSize, 2), " Мбайт");

    c_uliMegabytePerGigabyte ..(32 * c_uliMegabytePerGigabyte):
                        rByteSize   :=  ULINT_TO_REAL(uliByteSize)  /   ULINT_TO_REAL(c_uliMegabytePerGigabyte);
                        BYTE_SIZE_TO_WSTRING    :=  WCONCAT(REAL_TO_FWSTRING(rByteSize, 2), " Гбайт");

END_CASE
```

Рисунок 5.4.6 – Код функции BYTE_SIZE_TO_WSTRING

В коде функции **BYTE_SIZE_TO_WSTRING** используется вспомогательная функция **REAL_TO_FWSTRING**, которая округляет переменную типа **REAL** до нужного количества знаков после запятой и преобразует ее в строку. Например, вызов функции со следующими аргументами

REAL_TO_FWSTRING(11.2288, 2)

вернет строку «11.23».

Код функции **REAL_TO_FWSTRING** приведен на рисунке ниже:

REAL_TO_FWSTRING

```
// функция конвертирует значение типа REAL в строку с n знаков после запятой

FUNCTION REAL_TO_FWSTRING :     WSTRING
VAR_INPUT
    rVar:                       REAL;       // входное значение
    usiPrecision:               USINT;      // нужное кол-во знаков после запятой
END_VAR
VAR
    uliVar:                     ULINT;      // промежуточная переменная
    lrVar:                      LREAL;      // промежуточная переменная

END_VAR
```

```
uliVar              :=  LREAL_TO_ULINT( (rVar) * EXPT(10, usiPrecision) );
lrVar               :=  ULINT_TO_LREAL(uliVar) / EXPT(10, usiPrecision);
REAL_TO_FWSTRING    :=  LREAL_TO_WSTRING(lrVar);
```

Рисунок 5.4.7 – Код функции REAL_TO_FWSTRING

Принцип работы функции заключается в следующем:

- пусть имеется значение rVar=**11.2266**, которое необходимо округлить до usiPrecision=**2** знаков после запятой;
- запятая смещается на **две** позиции вправо (с помощью умножения на $\mathbf{10^2}$), результат – число **1122.66**;
- выполняется конвертация в целочисленное значение, результат – число **1123**;
- выполняется обратная конвертация в REAL, результат – число **1123.0**;
- запятая смещается на две позиции влево (с помощью деления на $\mathbf{10^2}$), результат – искомое округленное значение **11.23**.

Затем следует оптимизировать индикацию демонтирования накопителя. Флаг демонтирования накопителя **xUnmountDone** взводится в **TRUE** на время, пока сигнал демонтирования **xUnmount** имеет значение **TRUE**. Соответственно, если **xUnmount** получит импульс по переднему фронту – накопитель будет успешно демонтирован, но флаг демонтирования примет значение **TRUE** только на один цикл ПЛК – что не будет детектировано человеческим глазом.

Поэтому следует использовать следующий алгоритм демонтирования: нажатие оператором кнопки в визуализации будет переключать переменную **xUnmount** в состояние **TRUE**, что приведет к переключению в **TRUE** переменной **xUnmountDone**, в результате чего будет загораться индикатор, сообщающий об успешном демонтировании накопителя. Спустя заданный интервал времени (например, 5 секунд) **xUnmount** будет сброшен в **FALSE** из программы, что приведет к отключению индикатора.

Код для обеих операций (конвертации объемов накопителей в форматированные строки и сброс сигнала демонтирования) следует добавить в программу **PLC_PRG**:

```
1
2   // преобразование размеров полной/занятой/свободной памяти в форматированную строку
3
4   stPlcMemory.wsFullSize      :=  BYTE_SIZE_TO_WSTRING(stPlcMemory.uliFullSize);
5   stPlcMemory.wsUsedSize      :=  BYTE_SIZE_TO_WSTRING(stPlcMemory.uliUsedSize);
6   stPlcMemory.wsFreeSize      :=  BYTE_SIZE_TO_WSTRING(stPlcMemory.uliFreeSize);
7
8   stUsbMemory.wsFullSize      :=  BYTE_SIZE_TO_WSTRING(stUsbMemory.uliFullSize);
9   stUsbMemory.wsUsedSize      :=  BYTE_SIZE_TO_WSTRING(stUsbMemory.uliUsedSize);
10  stUsbMemory.wsFreeSize      :=  BYTE_SIZE_TO_WSTRING(stUsbMemory.uliFreeSize);
11
12  stSdMemory.wsFullSize       :=  BYTE_SIZE_TO_WSTRING(stSdMemory.uliFullSize);
13  stSdMemory.wsUsedSize       :=  BYTE_SIZE_TO_WSTRING(stSdMemory.uliUsedSize);
14  stSdMemory.wsFreeSize       :=  BYTE_SIZE_TO_WSTRING(stSdMemory.uliFreeSize);
15
16
17
18  // сброс флагов "устройство отмонтировано" через 5 секунд после отмонтирования устройства
19
20  fbUsbUnmountTimeout(IN := stUsbMemory.xUnmountDone, PT := T#5S);
21
22  IF fbUsbUnmountTimeout.Q THEN
23      stUsbMemory.xUnmount := FALSE;
24  END_IF
25
26  fbSdUnmountTimeout(IN := stSdMemory.xUnmountDone, PT := T#5S);
27
28  IF fbSdUnmountTimeout.Q THEN
29      stSdMemory.xUnmount := FALSE;
30  END_IF
```

Рисунок 5.4.8 – Код операций с переменными вкладки Drives

В следующих пунктах в **PLC_PRG** будет добавлен новый код; чтобы разграничить его фрагменты, связанные с разными пунктами документа, будут созданы действия (**action**). Действие представляет собой изолированный фрагмент кода. Сначала следует создать действие (**PLC_PRG – Добавление объекта – Действие**) с названием **act01_DriveInfo** и вынести в него код из рисунка 5.4.8.

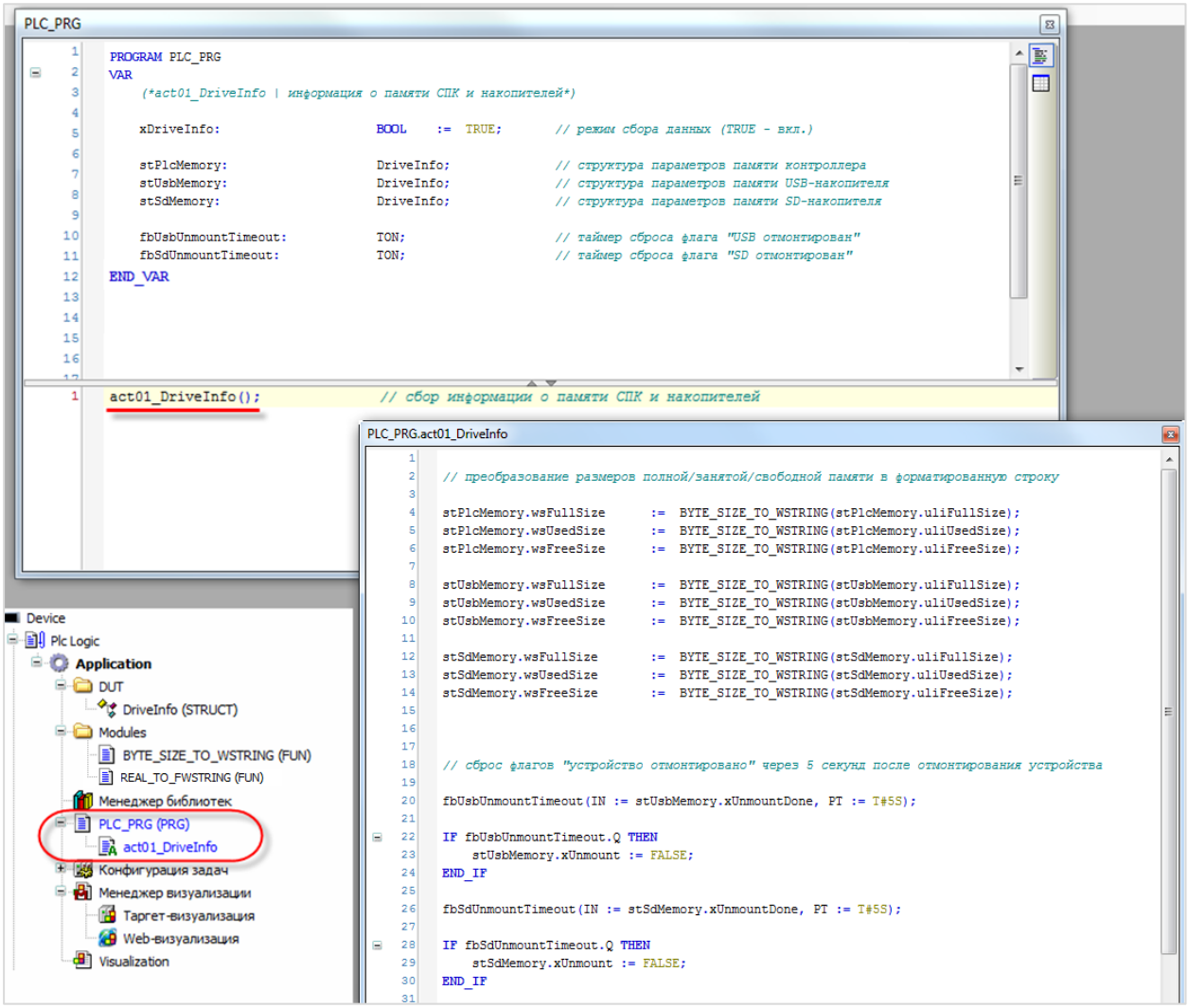


Рисунок 5.4.9 – Код действия act01_DriveInfo и его вызов в программе PLC_PRG

5.4.3 Создание визуализации

Затем следует создать интерфейс оператора. Здесь и в следующих пунктах не будет рассматриваться процесс разработки визуализации (вся необходимая информация приведена в документе **CODESYS V3.5. Визуализация**). На рисунке 5.4.10 приведен внешний вид экрана **Visu01_DriveInfo**, который включает в себя:

- 9 прямоугольников, отображающих информацию о полном/занятом/свободном объеме каждого накопителя (переменные типа **WSTRING**);
- 2 индикатора, отображающих статус USB- и SD-накопителей (с привязанными переменными **xIsMounted**);
- 2 кнопки для демонтирования накопителей (с привязанными переменными **xUnmount**, поведение – **Переключатель изображения**);
- 2 индикатора, отображающих флаги успешного демонтирования накопителей (с привязанными переменными **xUnmountDone**).

Визуализация также содержит кнопки переключения экранов (описание других экранов проекта приведено в соответствующих пунктах). Пример работы с экраном приведен в п. 5.10.

Рисунок 5.4.10 – Внешний вид экрана Visu01_DriveInfo

5.5 Работа с каталогами (PLC_PRG, действие act02_DirExample)

В данном пункте приведен пример работы с каталогами. Каталоги позволяют разделять файлы на группы, что упрощает работу с ними. Каталоги могут создаваться, переименовываться и удаляться. Также пользовать может получить информацию о содержимом каталога.

5.5.1 Объявление переменных

Каталоги, с которыми работает пользователь, могут быть расположены в памяти контроллера или подключенных к нему накопителей. Для упрощения программы следует объявить перечисление **FileDevice**, описывающее эти накопители (**Application – Добавление объекта – DUT – Перечисление**):

FileDevice

```
{attribute 'strict'}

// тип устройства для архивации
TYPE FileDevice :
(
    PLC_MEMORY :=  0,
    USB        :=  10,
    SD         :=  20,
    FTP        :=  30
);
END_TYPE
```

Рисунок 5.5.1 – Объявление перечисления FileDevice

В программе **PLC_PRG** следует объявить следующие переменные:

```
(*act02_DirExample | операции с каталогами*)

fbDirCreate:                FILE.DirCreate;         // ФБ создания каталога
fbDirRemove:                FILE.DirRemove;         // ФБ удаления каталога
fbDirRename:                FILE.DirRename;         // ФБ переименования каталога

sDirName:                   STRING;                 // полный путь к текущему каталогу
sDirNameNew:                STRING;                 // полный путь для создаваемого каталога
sVisuDirName:               STRING;                 // имя текущего каталога
sVisuDirNameNew:            STRING;                 // имя создаваемого каталога
sDeviceDirPath:             STRING;                 // путь к устройству
iDeviceDirPath:             INT := 0;               // ID устройства
```

Рисунок 5.5.2 – Объявление переменных в программе PLC_PRG

5.5.2 Разработка программы

Затем следует создать функцию **DEVICE_PATH**, которая в качестве аргумента принимает ID (идентификатор) накопителя и возвращает путь к его файловой системе. Пути приведены в п. 2.4.

DEVICE_PATH

```
// функция возвращает путь для файловой системы контроллера или накопителя по ID

FUNCTION DEVICE_PATH : STRING
VAR_INPUT
    iDevice:            INT;            // ID устройства
END_VAR
VAR
END_VAR
```

```
CASE iDevice OF
    FileDevice.PLC_MEMORY:
        DEVICE_PATH:='/mnt/ufs/home/root/CODESYS_WRK/';
    FileDevice.USB:
        DEVICE_PATH:='/mnt/ufs/media/sda1/';
    FileDevice.SD:
        DEVICE_PATH:='/mnt/ufs/media/mmcblk0p1/';
    FileDevice.FTP:
        DEVICE_PATH:='/var/lib/ftp/in/';
END_CASE
```

Рисунок 5.5.3 – Код функции DEVICE_PATH

Оператор должен выбрать ID устройства (например, через элемент **Комбинированное окно/Combobox**), чтобы программа автоматически сформировала путь к нему. В противном случае ввод полного пути осуществлялся бы с экранной клавиатуры.

Следует в программе **PLC_PRG** действие **act02_DirExample** (**PLC_PRG – Добавление объекта – Действие**) и вынести в него следующий код:

PLC_PRG.act02_DirExample

```
// получаем путь к выбранному устройству
sDeviceDirPath  :=  DEVICE_PATH(iDeviceDirPath);

// склеиваем его с именами каталогов
sDirName        :=  CONCAT(sDeviceDirPath, sVisuDirName);
sDirNameNew     :=  CONCAT(sDeviceDirPath, sVisuDirNameNew);

// выполняем ФБ операций с каталогами
fbDirCreate(xExecute:=, sDirName:=sDirNameNew, xParent := TRUE);
fbDirRename(xExecute:=, sDirNameOld:=sDirName, sDirNameNew:=sDirNameNew);
fbDirRemove(xExecute:=, sDirName:=sDirName, xRecursive := TRUE);
```

Рисунок 5.5.4 – Код действия act02_DirExample

В программе **PLC_PRG** следует добавить вызов данного действия:

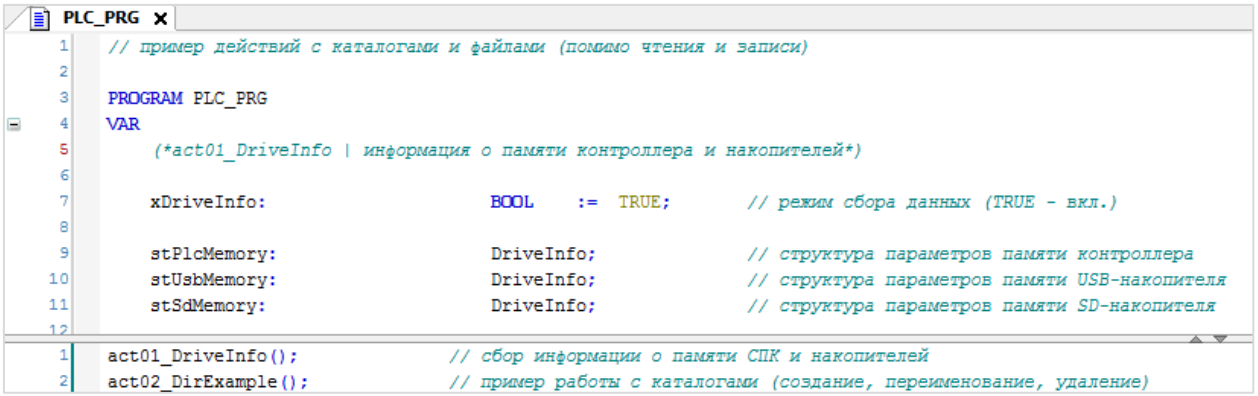


Рисунок 5.5.5 – Вызов действия act02_DirExample в программе PLC_PRG

Действие **act02_DirExample** (см. рисунок 5.5.4) производит следующие операции:

- получение путь к выбранному накопителю по его ID;
- склеивание пути к накопителю с именами текущего и создаваемого каталога;
- вызов экземпляров функциональных блоков создания, переименования и удаления каталогов.

ПРИМЕЧАНИЕ

В рамках примера вызов ФБ осуществляется без соотнесения входа **xExecute** с какой-либо переменной. Оператор с помощью нажатия кнопок будет воздействовать напрямую на входы блоков. Пользователю следует реализовать свой алгоритм работы с данными блоками, который позволит решить его конкретную задачу.

ПРИМЕЧАНИЕ

В рамках примера в качестве строковых аргументов ФБ используются одни и те же переменные. В большинстве практических задач разумно использовать уникальные переменные для каждого ФБ.

5.5.3 Создание визуализации

Затем следует создать интерфейс оператора для работы с каталогами. На рисунке 5.5.6 приведен внешний вид экрана **Visu02_DirExample**, который включает в себя:

- элемент **Комбинированное окно – целочисленный**, используемый для выбора накопителя, с каталогами которого будет работать программа. К элементу привязана переменная **iDeviceDirPath**. Настройки элемента описаны в п. 5.5.4;
- прямоугольник **Путь к устройству**, отображающий значение переменной **sDeviceDirPath**;
- два прямоугольника **Имя нового каталога** с привязанной переменной **sVisuDirNameNew**. В настройках элементов на вкладке **InputConfiguration** для действия **OnMouseClick** задана операция **Записать переменную** (тип ввода – диалог **VisuKeypad**);
- два прямоугольника **Имя существующего каталога** с привязанной переменной **sVisuDirName**. В настройках элементов на вкладке **InputConfiguration** для действия **OnMouseClick** задана операция **Записать переменную** (тип ввода – диалог **VisuKeypad**);
- три кнопки для выполнения операций с каталогами с поведением **Клавиша изображения**. К кнопке **Создать новый** привязана переменная **fbDirCreate.xExecute**, к кнопке **Удалить существующий** - **fbDirRemove.xExecute**, к кнопке **Переименовать** – **fbDirRename.xExecute**.

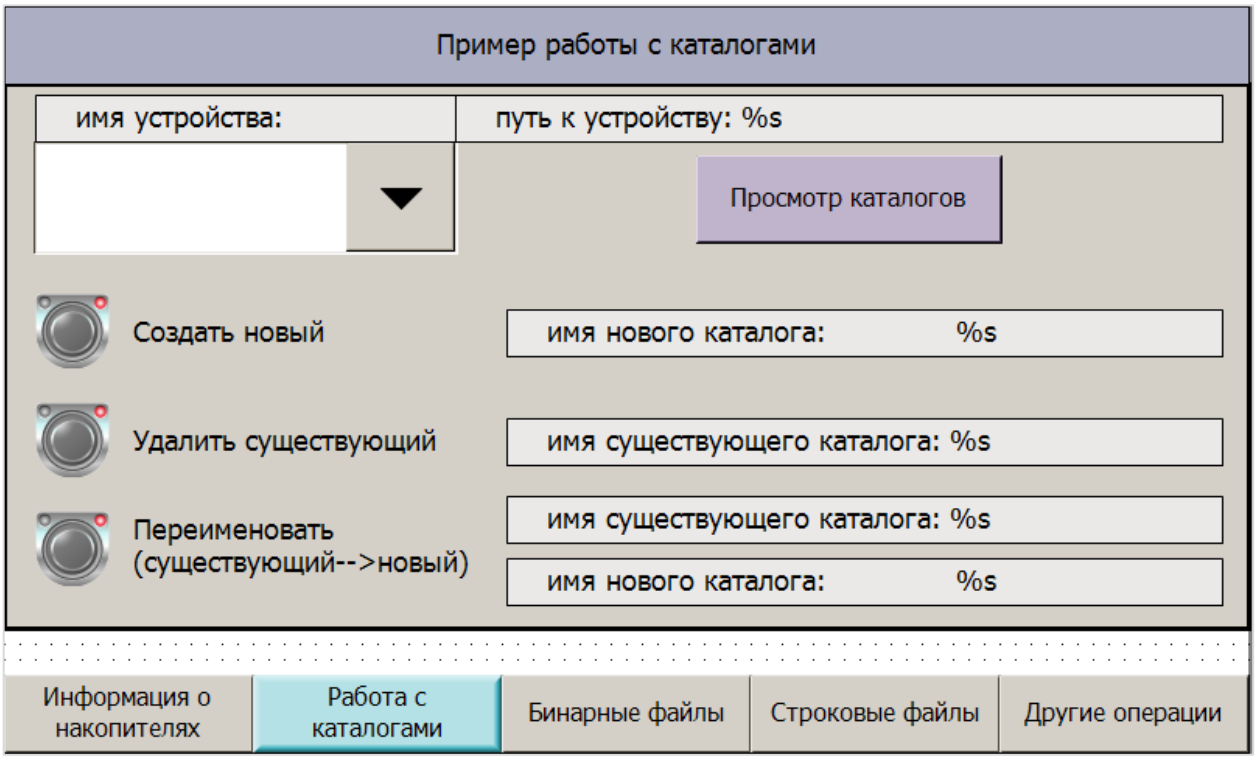


Рисунок 5.5.6. Внешний вид экрана Visu02_DirExample

Визуализация также содержит кнопки переключения экранов (описание других экранов проекта приведено в соответствующих пунктах). Пример работы с экраном приведен в п. 5.10.

5.5.4 Настройка элемента Комбинированное окно

В визуализации этого и следующих пунктов используется элемент **Комбинированное окно – Целочисленный** для выбора оператором нужного накопителя. Настройки элемента приведены на рисунке ниже:

Свойства

Фильтр | Сортировать по | Порядок сортировки | Эксперт

Свойство	Значения
Имя элемента	GenElemInst_223
Тип элемента	Комбинированное окно - Целочисленный
⊟ Позиция	
X	20
Y	88
Ширина	270
Высота	70
Переменная	PLC_PRG.iDeviceDirPath
Список текстов	'FileDevice'
Пул изображений	'ImagePool'
⊞ Параметры списка	
⊟ Тексты	
Подсказка	
⊟ Поддиапазон	
Использовать поддиапазон	☐
Минимальное значение	0
Конечный индекс	30
Отбрасывать недостающие тексты	☑
⊞ Свойства текста	
⊞ Переменные состояний	

Рисунок 5.5.7 – Настройки элемента Комбинированное окно – Целочисленный

Элемент использует компоненты **Список текстов** (**FileDevice**) и **Пул изображений** (**ImagePool**). Компоненты следует добавить в проект (**Application – Добавление объекта**). Содержимое компонентов приведено ниже.

FileDevice

ID	По умолчанию
0	PLC_MEMORY
10	USB
20	SD
30	FTP

ImagePool

ID	Имя файла	Изображение	Тип ссылки
0	HDD.png		Связь с файлом
10	USB.png		Связь с файлом
20	SD.png		Связь с файлом
30	FTP.png	FTP	Связь с файлом

Рисунок 5.5.8 – Содержимое компонентов Список Текстов и Пул изображений

5.6 Просмотр содержимого каталогов (PLC_PRG, действие act03_DirList)

В некоторых случаях оператору может потребоваться возможность просмотра содержимого накопителя (например, чтобы выбрать файл с нужным рецептом). В совокупности с функционалом, описанным в п. 5.5 (создание/удаление/переименование каталогов), это позволит создать простейший файловый менеджер.

5.6.1 Объявление переменных

Как упоминалось в п. 2.2, работу с файлами/каталогами можно представить в виде последовательности шагов, выполняемых с помощью оператора **CASE**. В качестве меток оператора **CASE** можно использовать обычные числа (0, 1, 2 и т. д.) – но это затруднит чтение программы. Поэтому следует объявить перечисление **FileWork** (**Application – Добавление объекта – DUT – Перечисление**), в котором номера шагов связываются с символьными именами. В данном пункте используется лишь несколько элементов этого перечисления. Все остальные будут использованы в п. 5.7 и 5.8 во время создания архиваторов.

FileWork

```
{attribute 'strict'}

// имена шагов работы с файлами
TYPE FileWork :
(
    OPEN            :=  0,
    CREATE          :=  10,
    READ            :=  20,
    SET_READ_POS    :=  30,
    WRITE           :=  40,
    FLUSH           :=  50,
    CLOSE           :=  60,
    GET_SIZE        :=  70
);
END_TYPE
```

Рисунок 5.6.1 – Объявление перечисления FileWork

Во время просмотра каталогов оператор будет получать о каждом вложенном каталоге/файле в виде экземпляра структуры FILE.FILE_DIR_ENTRY. Чтобы отображать данные в визуализации следует привести их к удобному для оператора виду. Для этого объявим структуру **VisuDirInfo**:

VisuDirInfo

```
// структура информации о каталога/файла, отображаемой в визуализации

TYPE VisuDirInfo :
STRUCT
    sEntryName:             STRING;     // имя каталога/файл
    wsEntryType:            WSTRING;    // тип (каталог или файл)
    wsEntrySize:            WSTRING;    // размер файла в байтах
    sLastModification:      STRING;     // дата последнего изменения файла
END_STRUCT
END_TYPE
```

Рисунок 5.6.2 – Объявление структуры VisuDirInfo

Затем следует объявить в программе **PLC_PRG** следующие переменные:

```
    (*act03_DirList | информация о выбранном каталоге*)

    fbDirInfo:                  DIR_INFO;               // ФБ сбора информации о каталоге
    xDirList:                   BOOL;                   // сигнал сбора информации о каталоге
    i:                          INT;                    // счетчик для цикла

    // путь к выбранному каталогу
    sDirListPath:               STRING  :='/mnt/ufs/home/root/CODESYS_WRK/';

    // путь к предыдущему выбранному каталогу
    sLastDevice:                STRING;

    // массив данных о вложенных файлах/каталогов для визуализации
    astVisuDirInfo:             ARRAY [0..c_MAX_ENTRIES] OF VisuDirInfo;

    fbSplitDT:                  SPLIT_DT_TO_FSTRINGS;   // ФБ конвертации времени в строку
    asEntryDT:                  ARRAY [0..10] OF STRING; // метка времени в виде отдельных строковых разрядов

    iSelectedEntry:             INT;                    // номер выбранной строки таблицы

    xDown:                      BOOL;                   // сигнал "Открыть каталог"
    xUp:                        BOOL;                   // сигнал "Перейти на уровень выше"
    xHideUp:                    BOOL;                   // переменная неактивности кнопки "Открыть каталог"
    xFirstScan:                 BOOL;                   // сигнал "Сканирование каталога"
```

Рисунок 5.6.3 – Объявление переменных в программе PLC_PRG

Также следует объявить несколько констант:

```
VAR CONSTANT
    c_MAX_ENTRIES:              UINT        :=100;      // максимальное числов вложенных элементов каталога
    c_sCharSlash:               STRING(1)   :='/';      // разделитель для пути в файловой системе
    c_byCodeSlash:              BYTE        :=16#2F;    // ASCII-код разделителя

    // пустая структура для очистки таблицы
    c_astVisuDirInfoNull:           ARRAY [0..c_MAX_ENTRIES] OF VisuDirInfo;
END_VAR
```

Рисунок 5.6.4 – Объявление констант в программе PLC_PRG

5.6.2 Разработка программы

На рисунке 5.6.3 были объявлены экземпляры функциональных блоков **DIR_INFO** и **SPLIT_DT_TO_FSTRING**, но сами блоки еще не созданы. Первый из них будет использоваться непосредственно для получения информации о содержимом каталога, второй – для преобразования метки времени типа **DT** в строковые представления отдельных разрядов.

Следует создать ФБ **DIR_INFO** со следующим интерфейсом:

DIR_INFO

```
1  // ФБ для получения информации о содержимом каталога (о вложенных файлах/каталогах)
2
3  FUNCTION_BLOCK DIR_INFO
4  VAR_INPUT
5      xExecute:           BOOL;               // сигнал запуска блока
6      sDirName:           STRING;             // имя обрабатываемого каталога
7  END_VAR
8  VAR_OUTPUT
9      xDone:              BOOL;               // флаг "данные получены"
10     // информация о вложенных файлах/каталогах
11     astDirInfo:         ARRAY [0..c_MAX_ENTRIES] OF FILE.FILE_DIR_ENTRY;
12     uiEntryPos:         UINT;               // кол-во обработанных файлов и каталогов
13 END_VAR
14 VAR
15     fbDirOpen:          FILE.DirOpen;       // ФБ открытия каталога
16     fbDirList:          FILE.DirList;       // ФБ получения информации о содержимом каталога
17     fbDirClose:         FILE.DirClose;      // ФБ закрытия каталога
18
19     hDirHandle:         FILE.CAA.HANDLE;    // дескриптор открытого каталога
20     eState:             FileWork;           // перечисление с именами шагов
21     fbStart:            R_TRIG;             // триггер запуска блока
22 END_VAR
23 VAR CONSTANT
24     c_MAX_ENTRIES:      UINT    :=100;      // максимальное число обрабатываемых файлов/каталогов
25 END_VAR
```

Рисунок 5.6.5 – Объявление переменных ФБ DIR_INFO

ПРИМЕЧАНИЕ

ФБ содержит константу **c_MAX_ENTRIES**. Одноименная константа уже была объявлена в программе **PLC_PRG** (см. рисунок 5.6.4). Значения обоих констант должны совпадать. Вариант с двумя константами является достаточно простым, но следует отметить, что оптимальным решением было бы обойтись одной глобальной константой, объявленной в **Списке глобальных переменных**.

Код блока **DIR_INFO** будет выглядеть следующим образом:

```
// детектируем сигнал запуска блока
fbStart(CLK:=xExecute);

// сбрасываем сигнал завершения работы
xDone:=FALSE;

CASE eState OF


    FileWork.OPEN:  // открываем каталог

        // обнуляем позицию для записи информации о файлах/каталогах
        uiEntryPos:=0;

        fbDirOpen(xExecute:=fbStart.Q, sDirName:=sDirName);

        IF fbDirOpen.xDone THEN
            hDirHandle  :=  fbDirOpen.hDir;
            fbDirOpen(xExecute:=FALSE);
            eState      :=  FileWork.READ;
        END_IF


    FileWork.READ:  // получаем информацию о вложенных файлах и каталогах

        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[uiEntryPos]  :=  fbDirList.deDirEntry;

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

            // если число вложенных файлов/каталогов больше, чем размер массива...
            // ...то начинаем перезаписывать его с нуля
            IF uiEntryPos>c_MAX_ENTRIES THEN
                uiEntryPos  :=  0;
            END_IF

            fbDirList(xExecute:=FALSE);
        END_IF

        // если код ошибки - "NO_MORE_ENTRIES", то обработаны все файлы/каталоги...
        // ...и можно завершать работу блока
        IF  fbDirList.eError=FILE.ERROR.NO_MORE_ENTRIES  THEN
            fbDirList(xExecute:=FALSE);
            eState  :=  FileWork.CLOSE;
        END_IF


    FileWork.CLOSE:   //завершение работы блока

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

        IF fbDirClose.xDone THEN
            fbDirClose(xExecute:=FALSE);

            // устанавливаем флаг завершения работы
            xDone   :=  TRUE;

            eState  :=  FileWork.OPEN;
        END_IF

END_CASE
```

Рисунок 5.6.6 – Код ФБ DIR_INFO

Блок **DIR_INFO** работает по следующему алгоритму: по переднему фронту на входе **xExecute** начинается получение информации о каталоге, расположенному по пути **sDirName**.

- на шаге **OPEN** выполняется открытие каталога с помощью ФБ FILE.DirOpen. Если каталог успешно открыт, то происходит переход на шаг **READ**;
- на шаге **READ** начинается получение информации о вложенных файлах/каталогов с помощью ФБ FILE.DirList. Полученные данные записываются на выход **astDirInfo**, который представляет собой массив структур типа FILE.FILE_DIR_ENTRY. Если число полученных данных превышает размер массива (верхняя граница которого определяется константой **c_MAX_ENTRIES**), то массив перезаписывается начиная с нулевой записи. То есть если каталог включает в себя 102 файла, то блок вернет информацию о файлах 1–102, причем информация о файле 102 будет записана в ячейку 0. Если получена информация обо всех вложенных элементах каталога (об этом сигнализирует ошибка **NO_MORE_ENTRIES** на выходе **xError** экземпляра блока **fbDirList**), то происходит переход к шагу **CLOSE**;
- на шаге **CLOSE** каталог закрывается.

Для работы с блоком **DIR_INFO** следует:

- записать путь к нужному каталогу на вход **sDirName**;
- сформировать импульс по переднему фронту на входе **xExecute**;
- ожидать формирования импульса на выходе **xDone**. Когда **xDone** примет значение **TRUE,** можно забрать полученную информацию о вложенных элементах каталога с выхода **astDirInfo**, число обработанных элементов – с выхода **uiEntryPos**.

После создания блока следует вызвать его в программе **PLC_PRG**. Но предварительно следует создать еще один блок, который будет конвертировать значение даты и времени типа **DT** в строковые представления отдельных разрядов с ведущими нулями. Имя блока – **SPLIT_DT_TO_FSTRINGS**.

SPLIT_DT_TO_FSTRINGS

```
// ФБ разделяет метку времени типа DT на строковые представления отдельных разрядов с ведущими нулями

FUNCTION_BLOCK SPLIT_DT_TO_FSTRINGS
VAR_INPUT
    dtDateAndTime:          DT;         // метка времени в формате DT
END_VAR
VAR_OUTPUT
    sYear:                  STRING;     // разряды времени в строковом представлении
    sMonth:                 STRING;     //
    sDay:                   STRING;     //
    sHour:                  STRING;     //
    sMinute:                STRING;     //
    sSecond:                STRING;     //
END_VAR
VAR
```

```
DTU.DTSplit
(
    dtDateAndTime,
    ADR(uiYear),
    ADR(uiMonth),
    ADR(uiDay),
    ADR(uiHour),
    ADR(uiMinute),
    ADR(uiSecond)
);

sYear       :=      UINT_TO_STRING(uiYear);
sMonth      :=      LEAD_ZERO(uiMonth);
sDay        :=      LEAD_ZERO(uiDay);
sHour       :=      LEAD_ZERO(uiHour);
sMinute     :=      LEAD_ZERO(uiMinute);
sSecond     :=      LEAD_ZERO(uiSecond);
```

Рисунок 5.6.7 – Объявление переменных и код ФБ SPLIT_DT_TO_FSTRINGS

Блок использует функцию **DTU.DTSplit**, которая входит в библиотеку **CAA DTUtil** (ее необходимо добавить в проект), а также вспомогательную функцию **LEAD_ZERO**, которую пользователь должен создать самостоятельно:

LEAD_ZERO

```
// функция преобразует число в строку с ведущим нулем

FUNCTION LEAD_ZERO : STRING
VAR_INPUT
    uiInput:         UINT;
END_VAR
VAR
END_VAR
```

```
IF uiInput>9 THEN
    LEAD_ZERO:=UINT_TO_STRING(uiInput);
ELSE
    LEAD_ZERO:=CONCAT('0', UINT_TO_STRING(uiInput));
END_IF
```

Рисунок 5.6.8 – Объявление переменных и код функции LEAD_ZERO

ФБ **SPLIT_DT_TO_FSTRINGS** получает на вход переменную типа **DT**, выделяет из нее значения отдельных разрядов времени в виде переменных типа **UINT**, после чего преобразует их в строки с ведущими нулями с помощью функции **LEAD_ZERO**.

Пример работы ФБ: вход **dtDateAndTIme** имеет значение **DT#2017-7-27-7:32:5**

Тогда выходы блока будут иметь следующие значения:

- sYear = ’2017’;
- sMonth = ’07’;
- sDay = ’27’;
- sHour = ’07’;
- sMinute = ’32’;
- sSecond = ’05’.

Имея в наличии значения отдельных разрядов времени, можно склеить из них строковую метку времени в нужном пользователю формате. Для этого следует создать функцию **CONCAT11**, которая собирает 11 отдельных **STRING** переменных в одну:

CONCAT11

```
1  // функция склеивает заданное число строковых переменных, помещенных в массив
2
3  FUNCTION CONCAT11 : STRING(255)
4  VAR_INPUT
5      asSTR:          ARRAY [0..c_MAX_STR] OF STRING;
6  END_VAR
7  VAR
8      sBuffer:        STRING;         // промежуточная переменная
9      i:              INT;            // счетчик для цикла
10 END_VAR
11
12 VAR CONSTANT
13     c_MAX_STR:      INT:=10;        // размер массива строковых переменных
14 END_VAR
```

```
1  FOR i:=0 TO c_MAX_STR DO
2      sBuffer:=CONCAT(sBuffer, asSTR[i]);
3  END_FOR
4
5  CONCAT11:=sBuffer;
```

Рисунок 5.6.9 – Объявление переменных и код функции CONCAT11

В программе **PLC_PRG** следует создать действие **act03_DirList** (**PLC_PRG – Добавление объекта – Действие**) и вынести в него код, приведенный на рисунке 5.6.10.

PLC_PRG.act03_DirList

```
// получаем путь к выбранному устройству
sDeviceDirPath:=DEVICE_PATH(iDeviceDirPath);

// при загрузке проекта и при выборе нового устройства сканируем его корневой каталог
IF NOT(xFirstScan) OR sDeviceDirPath<>sLastDevice THEN
    sDirListPath    :=  sDeviceDirPath;
    sLastDevice     :=  sDeviceDirPath;
    xDirList        :=  TRUE;
    xFirstScan      :=  TRUE;
END_IF


// если выбранный элемент - файл или символическая ссылка, то скрываем кнопку "Открыть каталог"
xHideUp := astVisuDirInfo[iSelectedEntry].sEntryName='..'
    OR astVisuDirInfo[iSelectedEntry].sEntryName='.' OR astVisuDirInfo[iSelectedEntry].wsEntryType="файл";


// по сигналу переходим в выбранный каталог
IF xDown THEN

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

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

    xDown       :=  FALSE;
    xDirList    :=  TRUE;
END_IF


// по сигналу переходим на уровень выше, контролируя, что продолжается работа с прежним устройством
IF xUp AND sDirListPath<>sDeviceDirPath THEN

    // 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 удален самый нижний каталог
    FOR i:=LEN(sDirListPath)-1 TO 0 BY -1 DO

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

    xUp         :=  FALSE;
    xDirList    :=  TRUE;
END_IF

// получаем информацию о содержимом каталога
fbDirInfo(xExecute:=xDirList, sDirName:=sDirListPath);

IF fbDirInfo.xDone THEN

    // стираем информацию о предыдущем открытом каталоге
    astVisuDirInfo  :=  c_astVisuDirInfoNull;
    // переходим к верхней строке таблицы
    iSelectedEntry  :=  0;

    // заполняем массив структур информацией о содержимом каталога
    FOR i:=0 TO UINT_TO_INT(fbDirInfo.uiEntryPos-1) DO
        astVisuDirInfo[i].sEntryName    :=  fbDirInfo.astDirInfo[i].sEntry;
        astVisuDirInfo[i].wsEntrySize   :=  BYTE_SIZE_TO_WSTRING(fbDirInfo.astDirInfo[i].szSize);
        astVisuDirInfo[i].wsEntryType   :=  SEL(fbDirInfo.astDirInfo[i].xDirectory, "файл", "Каталог");

        // преобразуем дату и время последнего изменения файла в форматированную строку
        fbSplitDT(dtDateAndTime:=fbDirInfo.astDirInfo[i].dtLastModification);

        asEntryDT[0]    :=  fbSplitDT.sDay;
        asEntryDT[1]    :=  '.';
        asEntryDT[2]    :=  fbSplitDT.sMonth;
        asEntryDT[3]    :=  '.';
        asEntryDT[4]    :=  fbSplitDT.sYear;
        asEntryDT[5]    :=  ' ';
        asEntryDT[6]    :=  fbSplitDT.sHour;
        asEntryDT[7]    :=  ':';
        asEntryDT[8]    :=  fbSplitDT.sMinute;
        asEntryDT[9]    :=  ':';
        asEntryDT[10]   :=  fbSplitDT.sSecond;

        astVisuDirInfo[i].sLastModification :=  CONCAT11(asEntryDT);

        xDirList    :=  FALSE;
    END_FOR
END_IF
```

Рисунок 5.6.10 – Код действия act03_DirList

Код выполняет следующие операции:

1. При загрузке проекта однократно (с помощью переменной **xFirstStart**) происходит запуск ФБ **fbDirInfo** для получения информации о корневом каталоге выбранного устройства (по умолчанию – памяти контроллера).
2. Если оператор выберет другое устройство (это можно определить по несоответствию значений переменных **sDeviceDirPath** и **sLastDevice**), то будет получена информация о корневом каталоге этого устройства.

   Полученная в пп. 1–2 информация будет представлена в табличном виде (более подробно см. в п. 5.6.3) – в виде набора файлов и каталогов, доступных для выделения.

   Если оператор выделит каталог, то сможет просмотреть его содержимое, нажав кнопку **Открыть каталог**. Для возвращения в предыдущий каталог следует нажать кнопку **На уровень выше.**

   В случае выделения **файла** кнопка **Открыть каталог** должна быть неактивной. В ОС Linux также существуют специальные каталоги «.» и «..», представляющие собой ссылки на текущий и родительский каталог. В рамках примера оператору запрещено работать с этими каталогами.

   Логическая переменная **xHideUp**, характеризующая неактивность кнопки, будет принимать значение **TRUE** в вышеописанных случаях.
3. По команде оператора (**xDown**) происходит формирование пути к следующему вложенному каталогу (выбранному с помощью выделения строки таблицы на экране визуализации, см. ниже) и запуск ФБ **fbDirInfo**, который получит информацию о данном каталоге.
4. По команде оператора (**xUp**) происходит формирование пути к родительскому каталогу (расположенному на уровень выше по отношению к текущему) и запуск ФБ **DirInfo,** который получит информацию о данном каталоге. У пользователя нет возможности перейти на уровень выше относительно корневого каталога.
5. Программа обрабатывает информацию, полученную от блока **fbDirInfo** – в частности, преобразовывает размер вложенных каталогов/файлов в строковый вид с помощью функции **BYTE_SIZE_TO_WSTRING** (которая была создана в п. 5.4.2) и метки времени последнего изменения каталога/файла в форматированную строку с помощью ФБ **SPLIT_DT_TO_FSTRINGS**.

5.6.3 Создание визуализации

Затем следует создать интерфейс оператора для просмотра каталогов. На рисунке 5.6.11 приведен внешний вид экрана **Visu06_DirList**, который включает в себя:

- элемент **Комбинированное окно – целочисленный**, используемый для выбора накопителя, с которым будет работать программа. Настройки элемента описаны в п. 5.5.4. К элементу привязана переменная **iDeviceDirPath**;
- прямоугольник **Текущий путь**, отображающий значение переменной **sDirListPath**;
- прямоугольник **Выбранный элемент**, отображающий значение переменной **astVisuDirInfo[iSelectedEntry].sEntryName** – т. е. имя элемента, выбранного пользователем в таблице;
- таблицу, к которой привязан массив структур **astVisuDirInfo**. На вкладке **Выбор** к параметру **Переменная для выбранной строки** привязана переменная **iSelectedEntry**;
- элемент **Полоса прокрутки**, к которому привязана переменная **iSelectedEntry**. Элемент используется для прокрутки таблицы. Следует отметить, что у таблицы есть встроенная полоса прокрутки, но ее размер зависит от размера таблицы, и в некоторых случаях может быть слишком мал. В данном примере поверх встроенной полосы прокрутки наложен отдельный элемент **Полоса прокрутки** увеличенного размера. Настройки элемента приведены ниже. Настройки должны соответствовать фактическому размеру таблицы;

Свойство	Значения
Имя элемента	GenElemInst_280
Тип элемента	Полоса прокрутки
Значение	PLC_PRG.iSelectedEntry
Минимальное значен...	0
Максимальное значе...	99
Размер страницы	10
Прокрутка выполнена	☐

Рисунок 5.6.11 – Настройки элемента Полоса прокрутки

- кнопка **Открыть каталог** с привязанной переменной **xDown** (**Конфигурация ввода – Нажать – xDown**) и переменной отключения ввода **xHideUp**;
- кнопка На уровень выше с привязанной переменной **xUp** (**Конфигурация ввода – Нажать – xUp**).

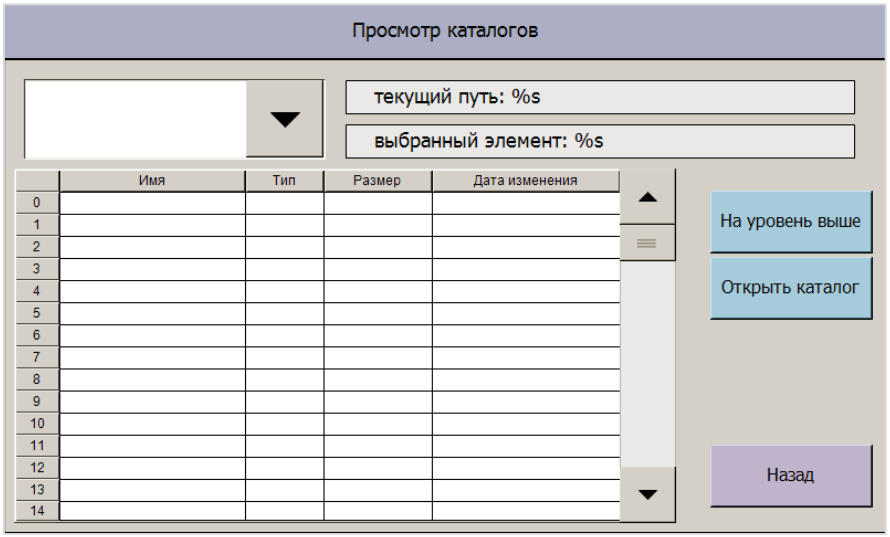


Рисунок 5.6.12 – Внешний вид экрана Visu06_DirList

Визуализация также содержит кнопки переключения экранов (описание других экранов проекта приведено в соответствующих пунктах). Пример работы с экраном приведен в п. 5.10.

5.7 Экспорт и импорт бинарных файлов (BinFileExample_PRG)

Информация о различиях бинарных и текстовых файлов приведена в п. 2.6.

5.7.1 Объявление переменных

Как упоминалось в п. 2.2, работу с файлами/каталогами можно представить в виде последовательности шагов, выполняемых с помощью оператора **CASE**. В качестве меток оператора **CASE** можно использовать обычные числа (0, 1, 2 и т. д.) – но это затруднит чтение программы. Соответствующее перечисление **FIleWork** уже было объявлено в п. 5.6.1:

FileWork

```
{attribute 'strict'}

// имена шагов работы с файлами
TYPE FileWork :
(
    OPEN            :=  0,
    CREATE          :=  10,
    READ            :=  20,
    SET_READ_POS    :=  30,
    WRITE           :=  40,
    FLUSH           :=  50,
    CLOSE           :=  60,
    GET_SIZE        :=  70
);
END_TYPE
```

Рисунок 5.7.1 – Объявление перечисления FileWork

Следует объявить структуру данных для записи в файл и чтения из него. В данном примере эта структура содержит одну переменную типа **WORD** и одну переменную типа **REAL** (**Application – Добавление объекта – DUT – Структура**). Структура будет иметь имя **ArchData**:

ArchData

```
// структура архивируемых данных

TYPE ArchData :
STRUCT
    wValue:         WORD;
    rValue:         REAL;
END_STRUCT
END_TYPE
```

Рисунок 5.7.2 – Объявление структуры ArchData

Следует объявить в программе **BinFileExample_PRG** следующие переменные:

BinFileExample_PRG

```
1  // пример экспорта и импорта данных из бинарного файла
2
3  PROGRAM BinFileExample_PRG
4  VAR
5      fbFileOpen:                 FILE.Open;                      // ФБ открытия файла
6      fbFileClose:                FILE.Close;                     // ФБ закрытия файла
7      fbFileWrite:                FILE.Write;                     // ФБ записи в файл
8      fbFileRead:                 FILE.Read;                      // ФБ чтения из файла
9      fbFileFlush:                FILE.Flush;                     // ФБ сброса буфера в файл
10     fbFileSetPos:               FILE.SetPos;                    // ФБ установки позиции для чтения
11     fbFileGetSize:              FILE.GetSize;                   // ФБ получения размера файла
12
13     hFile:                      FILE.CAA.HANDLE;                // дескриптор открытого файла
14     stExportBinData:            ArchData;                       // структура экспортируемых данных
15     stImportBinData:            ArchData;                       // структура для импорта данных
16     udiWriteEntry:              UDINT;                          // число записей в файле
17     udiReadEntry:               UDINT       := 1;               // позиция для чтения из файла
18     sFileName:                  STRING;                         // полный путь к файлу
19     sDevicePath:                STRING;                         // путь к устройству
20     iDevicePath:                INT;                            // ID устройства
21     sVisuFileName:              STRING      :=  'test.bin';     // имя файла
22
23     xWrite:                     BOOL;                           // сигнал записи в файл
24     xRead:                      BOOL;                           // сигнал чтения из файла
25     xWBusy:                     BOOL;                           // флаг "запись в файл"
26     xRBusy:                     BOOL;                           // флаг "чтение из файла"
27     eState:                     FileWork    := FileWork.GET_SIZE;   // шаг операции с файлом
28
29     fbWriteTrig:                F_TRIG;                         // триггер записи в файл
30     fbReadTrig:                 F_TRIG;                         // триггер чтения из файла
31 END_VAR
32
```

Рисунок 5.7.3 – Объявление переменных в программе BinFileExample_PRG

В программе объявлены два экземпляра структуры **ArchData** – один из них будет содержать данные, записываемые в файл, другой – данные, прочитанные из файла.

5.7.2 Разработка программы

Структура программы **BinFileExample_PRG** приведена рисунке 5.7.5. Перед началом работы с файлом программа получает путь к выбранному устройству (с помощью функции **DEVICE_PATH** из п. 5.5.2) и детектирует задний фронт управляющего сигнала. Управляющих сигналов в данном случае может быть два – сигнал записи в файл (**xWrite**) и сигнал чтения из файла (**xRead**). Сигнал записи в файл имеет больший приоритет – если оба сигнала станут активными в течение одного цикла контроллера, то будет произведена запись данных в файл. Чтения из файла в этом случае произведено не будет. В зависимости от детектированного сигнала, соответствующая логическая переменная получит значение **TRUE** (**xWBusy** – в случае записи, **xRBusy** – в случае чтения).

```
// получаем путь к выбранному устройству
sDevicePath :=  DEVICE_PATH(iDevicePath);

// склеиваем его с именем выбранного файла
sFileName   :=  CONCAT(sDevicePath, sVisuFileName);

// детектируем сигнал записи в файл или чтения из файла
fbWriteTrig(CLK:=xWrite);
fbReadTrig(CLK:=xRead);

// в зависимости от пришедшего сигнала взводим соответствующий флаг
IF fbWriteTrig.Q THEN
    xWBusy  :=  TRUE;
ELSIF fbReadTrig.Q THEN
    xRBusy  :=  TRUE;
END_IF


CASE eState OF


    FileWork.OPEN:  // шаг открытия файла

    FileWork.CREATE:    // шаг создания файла

    FileWork.WRITE: // шаг записи в буфер

    FileWork.FLUSH: // шаг сброса буфера в файл

    FileWork.SET_READ_POS:  // шаг установки позиции для чтения из файла

    FileWork.READ:  // шаг чтения данных

    FileWork.CLOSE: // шаг закрытия файла

    FileWork.GET_SIZE:  // шаг определения размера файла

END_CASE
```

Рисунок 5.7.4 – Структура программы BinFileExample_PRG

Работа с файлами происходит в управляющем операторе **CASE**. На рисунке 5.7.4. приведены только имена шагов без раскрытия их программного кода (он будет приведен ниже). Алгоритм работы с файлами:

- перед началом работы файл следует открыть (шаг **OPEN**);
- если файл не существует, то его следует создать (шаг **CREATE**);
- если был детектирован сигнал записи в файл, то следует произвести запись в буфер (шаг **WRITE**), после чего записать буфер в файл (шаг **FLUSH**);
- если был детектирован сигнал чтения из файла, то следует определить позицию чтения из файла (шаг **SET_READ_POS**), после чего прочитать данные из файла (шаг **READ**);
- после окончания работы с файлом его следует закрыть (шаг **CLOSE**);
- если была произведена запись, то после закрытия файла можно узнать его новый размер (шаг **GET_SIZE**).

Ниже приведен код и комментарии для каждого из шагов.

На шаге **OPEN** происходит открытие файла с помощью экземпляра ФБ FILE.OPEN. В зависимости от управляющего сигнала (запись или чтение) файл открывается в режиме **MAPPD** (дозапись в конец файла) или **MREAD** (чтение из файла). В случае обращения к несуществующему файлу на выходе

eError блока **fbFileOpen** появляется ошибка **NOT_EXIST**. При попытке записи в несуществующий файл его следует создать (перейдя на шаг **CREATE**). В рамках примера не будет рассматриваться и обрабатываться ситуация чтения из несуществующего файла. Результатом успешного открытия файла будет получение дескриптора (**hFile**), который будет использоваться при всех следующих действиях с данным файлом. Если файл успешно открыт, то происходит переход на шаг **WRITE** или **SET_READ_POS** (в зависимости от полученного управляющего сигнала).

```
22      FileWork.OPEN:  // шаг открытия файла
23
24              // в зависимости от команды выбираем нужный режим работы с файлом (чтение или запись)
25              IF xWBusy THEN
26                  fbFileOpen(xExecute:=TRUE, sFileName:=sFileName, eFileMode:=FILE.MODE.MAPPD);
27              ELSIF xRBusy THEN
28                  fbFileOpen(xExecute:=TRUE, sFileName:=sFileName, eFileMode:=FILE.MODE.MREAD);
29              END_IF
30
31              // если файл, в который производится запись, не существует, то создадим его
32              IF xWBusy AND fbFileOpen.eError=FILE.ERROR.NOT_EXIST THEN
33                  fbFileOpen(xExecute:=FALSE);
34                  eState  :=  FileWork.CREATE;
35              END_IF
36
37              //  если файл существует и был успешно открыт, то переходим к нужному шагу
38              // (записи в файл или установки позиции для чтения)
39              IF fbFileOpen.xDone THEN
40                  hFile   :=  fbFileOpen.hFile;
41                  fbFileOpen(xExecute:=FALSE);
42
43                  IF xWBusy THEN
44                      eState  :=  FileWork.WRITE;
45                  ELSIF xRBusy THEN
46                      eState  :=  FileWork.SET_READ_POS;
47                  END_IF
48
49              END_IF
50
```

Рисунок 5.7.5 – Код шага OPEN

На шаге **CREATE** происходит создание файла с помощью экземпляра ФБ FILE.OPEN. Для создания файла следует открыть его в режиме **MWRITE** – он будет автоматически создан при первой записи. Результатом успешного создания файла будет получение дескриптора (**hFile**), который будет использоваться во всех следующих действиях с данным файлом. После создания файла происходит переход на шаг **WRITE**. В рамках примера обработка ошибок на данном шаге не производится.

```
52      FileWork.CREATE:    // шаг создания файла
53
54              fbFileOpen(xExecute:=TRUE, sFileName:=sFileName, eFileMode:=FILE.MODE.MWRITE);
55
56              IF fbFileOpen.xDone THEN
57                  hFile   :=  fbFileOpen.hFile;
58                  fbFileOpen(xExecute:=FALSE);
59
60                  // после создания файла можно перейти к шагу записи данных
61                  eState  :=  FileWork.WRITE;
62              END_IF
63
64              IF fbFileOpen.xError THEN
65                  // обработка ошибок
66              END_IF
67
```

Рисунок 5.7.6 – Код шага CREATE

На шаге **WRITE** происходит запись данных структуры **stExportBinData** в системный буфер с помощью экземпляра ФБ FILE.WRITE. После записи осуществляется переход на шаг **FLUSH**. В рамках примера обработка ошибок на данном шаге не производится.

```
69      FileWork.WRITE: // шаг записи в буфер
70
71              fbFileWrite(xExecute:=TRUE, hFile:=hFile, pBuffer:=ADR(stExportBinData), szSize:=SIZEOF(stExportBinData));
72
73              IF fbFileWrite.xDone THEN
74                  fbFileWrite(xExecute:=FALSE);
75
76                  // теперь данные записаны в системный буфер; операционная система сама запишет их в файл...
77                  // ...но мы можем сразу сделать это принудительно, чтобы гарантировать сохранность данных
78                  eState  :=  FileWork.FLUSH;
79              END_IF
80
81              IF fbFileWrite.xError THEN
82                  // обработка ошибок
83              END_IF
```

Рисунок 5.7.7 – Код шага WRITE

На шаге **FLUSH** происходит сброс системного буфера в файл с помощью ФБ FILE.FLUSH. Данный шаг не является обязательным – после шага **WRITE** данные также будут записаны в файл. Подробнее о целесообразности применения данного ФБ см. в его описании. После сброса буфера в файл происходит переход на шаг **CLOSE**. В рамках примера обработка ошибок на данном шаге не производится.

```
86      FileWork.FLUSH: // шаг сброса буфера в файл
87
88              fbFileFlush(xExecute:=TRUE, hFile:=hFile);
89
90              IF fbFileFlush.xDone THEN
91                  fbFileFlush(xExecute:=FALSE);
92
93                  // теперь можно перейти к шагу закрытия файла
94                  eState  :=  FileWork.CLOSE;
95              END_IF
96
97              IF fbFileFlush.xError THEN
98                  // обработка ошибок
99              END_IF
```

Рисунок 5.7.8 – Код шага FLUSH

На шаге **SET_READ_POS** происходит установка позиции для чтения с помощью ФБ FILE.SetPos. Позиция представляет собой смещение в байтах между началом файла и читаемой записью. Оператор **SIZEOF** позволяет вычислить размер одной записи файла. Переменная **udiReadEntry** определяет номер считываемой записи. Так как первая запись в файле расположена с нулевого байта, то из значения **udiReadEntry** следует вычесть единицу. После установки позиции осуществляется переход на шаг **READ**. В рамках примера обработка ошибок на данном шаге не производится.

```
    FileWork.SET_READ_POS:  // шаг установки позиции для чтения из файла

            fbFileSetPos(xExecute:=TRUE, hFile:=hFile, udiPos:=SIZEOF(stExportBinData)*(udiReadEntry-1));

            IF fbFileSetPos.xDone THEN
                fbFileSetPos(xExecute:=FALSE);

                // позиция для чтения выбрана, теперь можно перейти к шагу чтения данных
                eState  :=  FileWork.READ;
            END_IF

            IF fbFileSetPos.xError THEN
                // обработка ошибок
            END_IF
```

Рисунок 5.7.9 – Код шага SET_READ_POS

На шаге **READ** из файла считываются данные с помощью экземпляра ФБ FILE.READ. Считанные данные записываются в структуру **stImportData**. После чтения осуществляется переход на шаг **CLOSE**. В рамках примера обработка ошибок на данном шаге не производится.

```
    FileWork.READ:  // шаг чтения данных

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

            IF fbFileRead.xDone THEN
                fbFileRead(xExecute:=FALSE);

                // теперь можно перейти к шагу закрытия файла
                eState  :=  FileWork.CLOSE;
            END_IF

            IF fbFileRead.xError THEN
                // обработка ошибок
            END_IF
```

Рисунок 5.7.10 – Код шага READ

На шаге **CLOSE** происходит закрытие файла с помощью экземпляра ФБ FILE.CLOSE. Выбор следующего шага зависит от произведенной операции – после записи в файл происходит переход на шаг **GET_SIZE** для определения нового размера файла, после чтения – переход на шаг **OPEN** для ожидания следующего управляющего сигнала. В рамках примера обработка ошибок на данном шаге не производится.

```
    FileWork.CLOSE: // шаг закрытия файла

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

            IF fbFileClose.xDone THEN
                fbFileClose(xExecute:=FALSE);

                IF xWBusy THEN
                    // после записи в файл узнаем его новый размер
                    eState  :=  FileWork.GET_SIZE;
                ELSE
                    // после чтения из файла его размер не изменится, так что...
                    // ...вернемся на шаг открытия файла для ожидания следующего управляющего сигнала
                    eState  :=  FileWork.OPEN;
                END_IF

                xWBusy  :=  FALSE;
                xRBusy  :=  FALSE;

            END_IF
```

Рисунок 5.7.11 – Код шага CLOSE

На шаге **GET_SIZE** происходит определение размера файла с помощью экземпляра ФБ FILE.GetSize. После определения размера файла осуществляется переход на шаг **OPEN** для ожидания следующего управляющего сигнала. Если блок **fbFileGetSize** возвращает ошибку **NOT_EXIST** (файл не существует), то размер файла можно принять за **0**.

```
156     FileWork.GET_SIZE:  // шаг определения размера файла
157
158             fbFileGetSize(xExecute:=TRUE, sFileName:=sFileName);
159
160             IF fbFileGetSize.xDone THEN
161
162                 // узнаем число записей в файле - оно равно отношению размера файла к размеру одной записи
163                 udiWriteEntry:=fbFileGetSize.szSize / SIZEOF(stExportBinData);
164                 fbFileGetSize(xExecute:=FALSE);
165
166                 // вернемся на шаг открытия файла для ожидания следующего управляющего сигнала
167                 eState  :=  FileWork.OPEN;
168             END_IF
169
170             // размер несуществующего файла...
171             IF fbFileGetSize.eError=FILE.ERROR.NOT_EXIST THEN
172
173                 // очевидно, можно интерпретировать как ноль
174                 udiWriteEntry   :=  0;
175                 fbFileGetSize(xExecute:=FALSE);
176
177                 // вернемся на шаг открытия файла для ожидания следующего управляющего сигнала
178                 eState          :=  FileWork.OPEN;
179             ELSIF  fbFileGetSize.xError THEN
180                 fbFileGetSize(xExecute:=FALSE);
181                 eState          :=  FileWork.OPEN;
182             END_IF
183
184 END_CASE
```

Рисунок 5.7.12 – Код шага GET_SIZE

5.7.3 Создание визуализации

Сначала следует создать интерфейс оператора для работы с бинарными файлами. На рисунке 5.7.13 приведен внешний вид экрана **Visu03_BinFileExample**, который включает в себя:

- элемент **Комбинированное окно – целочисленный**, используемый для выбора накопителя, с которым будет работать программа. Настройки элемента описаны в п. 5.5.4. К элементу привязана переменная **iDevicePath**;
- прямоугольник **Путь к устройству**, отображающий значение переменной **sDevicePath**;
- прямоугольник **Имя файла** с привязанной переменной **sVisuFileName**. В настройках элемента на вкладке **InputConfiguration** для действия **OnMouseClick** задана операция **Записать переменную** (тип ввода – диалог **VisuKeypad**);
- прямоугольники **wValue** и **rValue** (**Запись в файл**) с привязанными переменными **stExportBinData.wValue** и **stExportBinData.rValue** соответственно. В настройках элементов на вкладке **InputConfiguration** для действия **OnMouseClick** задана операция **Записать переменную** (тип ввода – диалог **VisuNumpad**);
- кнопка **Записать** с привязанной переменной **xWrite**, поведение элемента – **Клавиша изображения**;
- прямоугольник **Число записей в файле** с привязанной переменной **udiWriteEntry**;
- прямоугольник **Номер читаемой записи** с привязанной переменной **udiReadEntry**. В настройках элемента на вкладке **InputConfiguration** для действия **OnMouseClick** задана операция **Записать переменную** (тип ввода – диалог **VisuNumpad**), минимальное вводимое значение – **1**;
- кнопка **Прочитать** с привязанной переменной **xRead**, поведение элемента – **Клавиша изображения**;
- прямоугольники **wValue** и **rValue** (**Чтение из файла**) с привязанными переменными **stImportBinData.wValue** и **stImportBinData.rValue** соответственно.

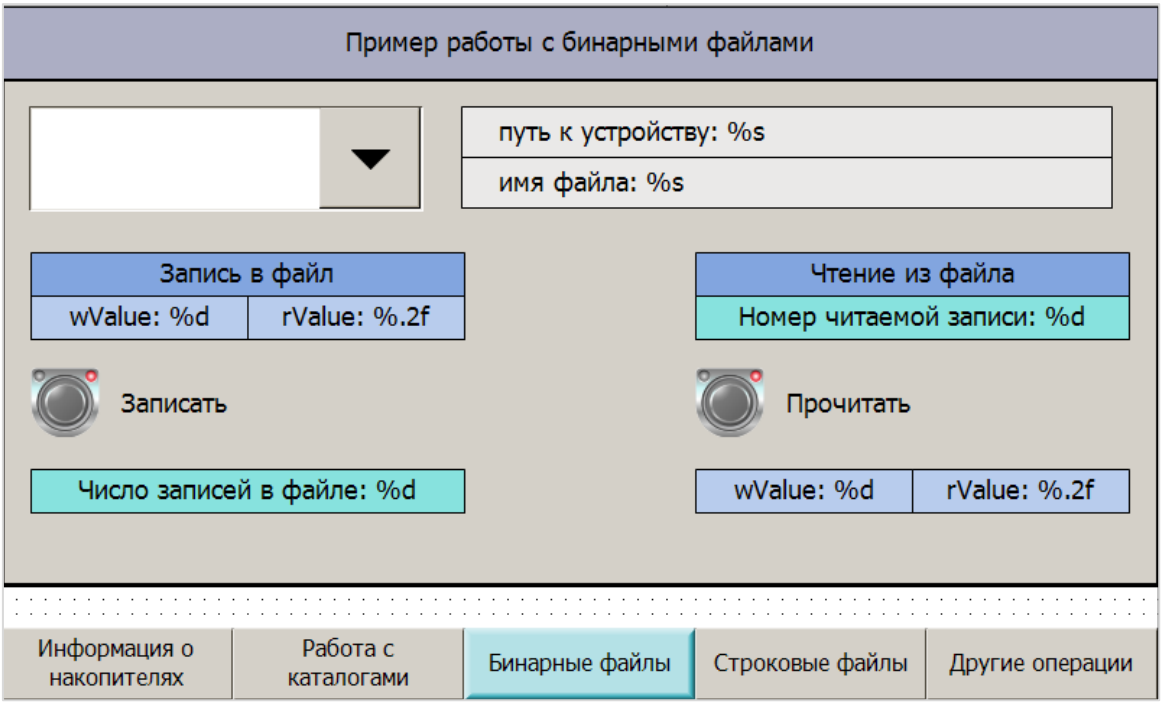


Рисунок 5.7.13 – Внешний вид экрана Visu03_BinFileExample

5.8 Экспорт текстовых файлов (StringFileExample_PRG)

Формат .csv используется для представления табличных данных и состоит из текстовых записей, разграниченных символом-разделителем. В русской локали таким символом по умолчагию является точка с запятой (;). Информация о различиях бинарных и текстовых файлов приведена в п. 2.6.

5.8.1 Объявление переменных

Как упоминалось в п. 2.2, работа с файлами/каталогами можно представить в виде последовательности шагов, выполняемых с помощью оператора **CASE.** В качестве меток оператора **CASE** можно использовать обычные числа (0, 1, 2 и т. д.) – но это затруднит чтение программы. Соответствующее перечисление **FIleWork** уже было объявлено в п. 5.6.1:

FileWork

```
{attribute 'strict'}

// имена шагов работы с файлами
TYPE FileWork :
(
    OPEN            :=  0,
    CREATE          :=  10,
    READ            :=  20,
    SET_READ_POS    :=  30,
    WRITE           :=  40,
    FLUSH           :=  50,
    CLOSE           :=  60,
    GET_SIZE        :=  70
);
END_TYPE
```

Рисунок 5.8.1 – Объявление перечисления FileWork

Структура архивируемых данных **ArchData**, содержащая одну переменную типа **WORD** и одну переменную типа **REAL**, уже была объявлена в п. 5.7.1:

ArchData

```
// структура архивируемых данных

TYPE ArchData :
STRUCT
    wValue:         WORD;
    rValue:         REAL;
END_STRUCT
END_TYPE
```

Рисунок 5.8.2 – Объявление структуры ArchData

В программе **StringFileExample_PRG** следует объявить следующие переменные и константы:

StringFileExample_PRG

```
1  // пример экспорта данных в текстовый файл
2
3  PROGRAM StringFileExample_PRG
4  VAR
5      fbFileOpen:                 FILE.Open;                      // ФБ открытия файла
6      fbFileClose:                FILE.Close;                     // ФБ закрытия файла
7      fbFileWrite:                FILE.Write;                     // ФБ записи в файл
8      fbFileFlush:                FILE.Flush;                     // ФБ сброса буфера в файл
9      fbFileGetSize:              FILE.GetSize;                   // ФБ получения размера файла
10
11     hFile:                      FILE.CAA.HANDLE;                // дескриптор открытого файла
12     stExportData:               ArchData;                       // структура экспортируемых данных
13     asExportStringData:         ARRAY [0..10] OF STRING;        // структура данных архива в виде строк
14     sArchEntry:                 STRING(255);                    // строка, записываемая в архив
15     xTitle:                     BOOL;                           // флаг "запись заголовка произведена"
16     udiArchSize:                UDINT;                          // размер архива в байтах
17     uiArchEntry:                UINT;                           // кол-во строк архива
18     sFileName:                  STRING;                         // имя файла
19     xWrite:                     BOOL;                           // сигнал записи в файл
20     xWBusy:                     BOOL;                           // флаг "чтение из файла"
21     eState:                     FileWork    :=  FileWork.GET_SIZE;  // шаг операции с файлом
22
23     sDevicePath:                STRING;                         // путь к устройству
24     iDevicePath:                INT;                            // ID устройства
25     sVisuFileName:              STRING      :=  'test.csv';     // имя файла архива
26
27     fbWriteTrig:                F_TRIG;                         // триггер записи в файл
28
29     fbGetCurrentDT:             DTU.GetDateAndTime;             // ФБ считывания системного времени
30     fbSplitDT:                  SPLIT_DT_TO_FSTRINGS;           // ФБ конвертации времени в строку
31     asDateTimeStrings:          ARRAY [0..10] OF STRING;        // метка времени в виде отдельных строковых разрядов
32     sTimeStamp:                 STRING(20);                     // метка времени в виде форматированной строки
33 END_VAR
34
35 VAR CONSTANT
36     // заголовок архива
37     c_sTitle:                   STRING(60)  :=  'Дата;Время;Значение типа WORD;Значение типа REAL;$N';
38     c_sDelimiter:               STRING(1)   :=  ';';
39 END_VAR
```

Рисунок 5.8.3 – Объявление переменных в программе StringFileExample_PRG

Для строковых переменных, значения которых будут заноситься в файл архива, длина указывается в явном виде (например, **STRING(20)**). Это важно для переменной **sArchEntry**, которая представляет собой строку архива. Если бы ее размер не был указан (**255**), то максимальное количество символов в строке составило бы **80** (таков размер по умолчанию для переменной типа **STRING**). Выбор значения **255** связан с ограничениями библиотеки **Standard**, функции которой не позволяют работать со строками большего размера.

5.8.2 Разработка программы

Структура программы **StringFileExample_PRG** приведена на рисунке 5.8.4:

```
1   // считываем системное время
2   fbGetCurrentDT(xExecute:=NOT(fbGetCurrentDT.xDone));
3
4   IF fbGetCurrentDT.xDone THEN
5       // вырезаем отдельные разряды времени и конвертируем их в строки
6       fbSplitDT(dtDateAndTime:=fbGetCurrentDT.dtDateAndTime);
7   END_IF
8
9   // подготавливаем метку времени в виде форматированной строки
10  asDateTimeStrings[0]    :=  fbSplitDT.sDay;
11  asDateTimeStrings[1]    :=  '.';
12  asDateTimeStrings[2]    :=  fbSplitDT.sMonth;
13  asDateTimeStrings[3]    :=  '.';
14  asDateTimeStrings[4]    :=  fbSplitDT.sYear;
15  asDateTimeStrings[5]    :=  c_sDelimiter;
16  asDateTimeStrings[6]    :=  fbSplitDT.sHour;
17  asDateTimeStrings[7]    :=  ':';
18  asDateTimeStrings[8]    :=  fbSplitDT.sMinute;
19  asDateTimeStrings[9]    :=  ':';
20  asDateTimeStrings[10]   :=  fbSplitDT.sSecond;
21
22  // собираем строку, которая будет записана в архив
23  asExportStringData[0]   :=  CONCAT11(asDateTimeStrings);
24  asExportStringData[1]   :=  c_sDelimiter;
25  asExportStringData[2]   :=  WORD_TO_STRING(stExportData.wValue);
26  asExportStringData[3]   :=  c_sDelimiter;
27  asExportStringData[4]   :=  REAL_TO_FSTRING(stExportData.rValue,2);
28  asExportStringData[5]   :=  '$N';
29
30  sArchEntry  :=  CONCAT11(asExportStringData);
31
32  // получаем путь к выбраенному устройству
33  sDevicePath :=  DEVICE_PATH(iDevicePath);
34
35  // склеиваем его с именем выбранного файла
36  sFileName   :=  CONCAT(sDevicePath, sVisuFileName);
37
38  // детектируем сигнал записи в файл
39  fbWriteTrig(CLK:=xWrite);
40
41  // если получен сигнал записи, то взводим соответствующий флаг
42  IF fbWriteTrig.Q THEN
43      xWBusy  :=  TRUE;
44  END_IF
45
46
47  CASE eState OF
48
49      FileWork.OPEN:  // шаг открытия файла
50  [20 lines]
71      FileWork.CREATE:    // шаг создания файла
72       [18 lines]
91      FileWork.WRITE: // шаг записи в буфер
92              [26 lines]
119     FileWork.FLUSH: // шаг сброса буфера в файл
120      [14 lines]
135     FileWork.CLOSE: // шаг закрытия файла
136      [11 lines]
148     FileWork.GET_SIZE:  // шаг определения размера файла
149      [26 lines]
176 END_CASE
```

Рисунок 5.8.4 – Структура программы StringFileExample_PRG

В программе можно выделить два основных блока:

1. Подготовка архивных данных (весь код до оператора **CASE**);
2. Запись в файл (содержимое оператора **CASE**).

В начале первого блока следует организовать циклическое чтение системного времени с помощью экземпляра ФБ **GetDateAndTime**. Время считывается в формате **DT** и разрезается на отдельные разряды с помощью ФБ **SPLIT_DT_TO_FSTRINGS** (который уже создан в п. 5.6.2), которые склеиваются в форматированную строку с разделителями – метку времени для текущей записи архива.

ПРИМЕЧАНИЕ

Таргет-файлы контроллеров ОВЕН содержат узел RTC, который позволяет получить дату и время в виде форматированных строк без дополнительных преобразований.

Затем метка времени и архивируемые значения склеиваются в одну строку, формируя архивную запись. Для этого используются вспомогательные функции **CONCAT11** (создана в п. 5.6.2) и **REAL_TO_FSTRING**. Функция **REAL_TO_FSTRING** используется для преобразования значения с плавающей точкой в строковое представление с заданным количеством знаков после запятой. Также функция заменяет символ разделителя целой и дробной части с точки на запятую (для корректного отображения значений в **Microsoft Excel** и др. ПО в русских локалях). Код функции выглядит следующим образом:

REAL_TO_FSTRING

```
// функция конвертирует значение типа REAL в строку с n знаков после запятой

FUNCTION REAL_TO_FSTRING :      STRING
VAR_INPUT
    rVar:                       REAL;       // входное значение
    usiPrecision:               USINT;      // нужное кол-во знаков после запятой
END_VAR
VAR
    uliVar:                     ULINT;      // промежуточная переменная
    lrVar:                      LREAL;      // промежуточная переменная
    sVar:                       STRING;     // промежуточная переменная
    xSign:                      BOOL;       // знак входного значения. TRUE - минус
END_VAR
```

```
// определяем знак
xSign := (rVar<0.0);

// оставляем нужное кол-во знаков после запятой
uliVar  :=  LREAL_TO_ULINT( ABS(rVar)*EXPT(10, usiPrecision) );
lrVar   :=  ULINT_TO_LREAL(uliVar) / EXPT(10, usiPrecision);
sVar    :=  LREAL_TO_STRING(lrVar);

// если нужно - возвращаем знак "минус"
IF xSign THEN
    sVar := CONCAT('-', sVar);
END_IF

// меняем точку на запятую для корректного отображения в MS Excel
REAL_TO_FSTRING := REPLACE(sVar, ',', 1, FIND(sVar, '.'));
```

Рисунок 5.8.5 – Код функции REAL_TO_FSTRING

Код функции почти полностью совпадает с кодом функции **REAL_TO_FWSTRING** из п. 5.4.2. Оптимальным решением было бы объединить обе функции в одну, но для упрощения примера рассматривается вариант с двумя отдельными функциями.

Архив следует сохранять в формате **.csv**. В русских локалях разделителем для этого формата является символ ‘**;**’, ASCII-код которого содержится в константе **c_sDelimiter**. Последним символом строки архива является спецсимвол **$N**, который соответствует переходу на новую строку (см. п. 2.6). Соответственно, после выполнения приведенного ниже фрагмента кода в переменную **sArchEntry** будет записана одна строка архива. Затем следует записать эту строку в файл:

```
// собираем строку, которая будет записана в архив
asExportStringData[0]   :=  CONCAT11(asDateTimeStrings);
asExportStringData[1]   :=  c_sDelimiter;
asExportStringData[2]   :=  WORD_TO_STRING(stExportData.wValue);
asExportStringData[3]   :=  c_sDelimiter;
asExportStringData[4]   :=  REAL_TO_FSTRING(stExportData.rValue,2);
asExportStringData[5]   :=  c_sDelimiter;
asExportStringData[6]   :=  '$N';

sArchEntry  :=  CONCAT11(asExportStringData);
```

Перед началом работы с файлом программа получает путь к выбранному устройству (с помощью функции **DEVICE_PATH** из п. 5.5.2), склеивает его с именем файла архива и детектирует задний фронт сигнала записи.

```
// получаем путь к выбраенному устройству
sDevicePath :=  DEVICE_PATH(iDevicePath);

// склеиваем его с именем выбранного файла
sFileName   :=  CONCAT(sDevicePath, sVisuFileName);

// детектируем сигнал записи в файл
fbWriteTrig(CLK:=xWrite);

// если получен сигнал записи, то взводим соответствующий флаг
IF fbWriteTrig.Q THEN
    xWBusy  :=  TRUE;
END_IF
```

Рисунок 5.8.6 – Фрагмент программы StringFileExample_PRG

Работа с файлами происходит в управляющем операторе **CASE**. На рисунке 5.8.4 были приведены только имена шагов без раскрытия их программного кода (он будет приведен ниже). Алгоритм работы с файлами:

- перед началом работы файл следует открыть (шаг **OPEN**);
- если файл не существует, то его следует создать (шаг **CREATE**);
- если был детектирован сигнал записи в файл, то следует произвести запись в буфер (шаг **WRITE**), после чего записать буфер в файл (шаг **FLUSH**);
- после окончания работы с файлом его следует закрыть (шаг **CLOSE**);
- если была произведена запись, то после закрытия файла можно узнать его новый размер (шаг **GET_SIZE**).

Ниже приведен код и комментарии для каждого из шагов.

На шаге **OPEN** происходит открытие файла с помощью экземпляра ФБ FILE.OPEN. Файл открывается в режиме **MAPPD** (дозапись в конец файла). В случае обращения к несуществующему файлу на выходе **eError** блока **fbFileOpen** появляется ошибка **NOT_EXIST**. При попытке записи в несуществующий файл его следует создать (перейдя на шаг **CREATE**) и записать в него заголовок (для этого используется флаг **xTitle**). Результатом успешного открытия файла будет получение дескриптора (**hFile**), который будет использоваться во всех следующих действиях с данным файлом. Если файл успешно открыт, то происходит переход на шаг **WRITE**.

```
52              IF xWBusy THEN
53                  fbFileOpen(xExecute:=TRUE, sFileName:=sFileName, eFileMode:=FILE.MODE.MAPPD);
54              END_IF
55
56              // если файл, в который производится запись, не существует, то создадим его и запишем в него заголовок архива
57              IF fbFileOpen.eError=FILE.ERROR.NOT_EXIST THEN
58                  fbFileOpen(xExecute:=FALSE);
59                  eState  :=  FileWork.CREATE;
60                  xTitle  :=  TRUE;
61              END_IF
62
63              // если файл существует и был успешно открыт, то переходим к шагу записи в файл
64              IF fbFileOpen.xDone THEN
65                  hFile   :=  fbFileOpen.hFile;
66                  fbFileOpen(xExecute:=FALSE);
67
68                  eState  :=  FileWork.WRITE;
69              END_IF
70
```

Рисунок 5.8.7 – Код шага OPEN

На шаге **CREATE** происходит создание файла с помощью экземпляра ФБ FILE.OPEN. Для создания файла следует открыть его в режиме **MWRITE** – он будет автоматически создан при первой записи. При создании файла значение переменной **udiArchEntry**, характеризующей количество записей в архиве, обнуляется. Результатом успешного создания файла будет получение дескриптора (**hFile**), который будет использоваться во всех следующих действиях с данным файлом. После создания файла происходит переход на шаг **WRITE**. В рамках примера обработка ошибок на данном шаге не производится.

```
72      FileWork.CREATE:    // шаг создания файла
73
74              // в созданном файле еще нет записей
75              uiArchEntry:=0;
76
77              fbFileOpen(xExecute:=TRUE, sFileName:=sFileName, eFileMode:=FILE.MODE.MWRITE);
78
79              IF fbFileOpen.xDone THEN
80                  hFile   :=  fbFileOpen.hFile;
81                  fbFileOpen(xExecute:=FALSE);
82
83                  // после создания файла можно перейти к шагу записи данных
84                  eState  :=  FileWork.WRITE;
85              END_IF
86
87              IF fbFileOpen.xError THEN
88                  // обработка ошибок
89              END_IF
```

Рисунок 5.8.8 – Код шага CREATE

На шаге **WRITE** происходит запись строки архива **sArchEntry** (или строки заголовки и строки архива, если файл был создан на предыдущем шаге **CREATE**) в системный буфер с помощью ФБ FILE.WRITE. После записи осуществляется переход на шаг **FLUSH**. В рамках примера обработка ошибок на данном шаге не производится.

```
92      FileWork.WRITE: // шаг записи в буфер
93
94              // если это первая запись в файле - то перед ней запишем заголовок
95              IF xTitle THEN
96                  sArchEntry  :=  CONCAT(c_sTitle, sArchEntry);
97
98                  // после первой записи заголовок записывать уже не нужно
99                  xTitle  :=  FALSE;
100             END_IF
101
102             // запись строки архива в файл
103             fbFileWrite(xExecute:=TRUE, hFile:=hFile, pBuffer:=ADR(sArchEntry), szSize:=INT_TO_UDINT(LEN(sArchEntry)));
104
105             IF fbFileWrite.xDone THEN
106                 fbFileWrite(xExecute:=FALSE);
107
108                 // после записи число строк в архиве увеличилось на одну
109                 uiArchEntry:=uiArchEntry+1;
110
111                 // теперь можно перейти к шагу сброса буфера в файл
112                 eState  :=  FileWork.FLUSH;
113             END_IF
114
115             IF fbFileWrite.xError THEN
116                 // обработка ошибок
117             END_IF
```

Рисунок 5.8.9 – Код шага WRITE

На шаге **FLUSH** происходит сброс системного буфера в файл с помощью ФБ FILE.FLUSH. Данный шаг не является обязательным – после шага **WRITE** данные также будут записаны в файл. Подробнее о целесообразности применения данного ФБ см. в его описании. После сброса буфера в файл происходит переход на шаг **CLOSE.** В рамках примера обработка ошибок на данном шаге не производится.

```
120     FileWork.FLUSH: // шаг сброса буфера в файл
121
122             fbFileFlush(xExecute:=TRUE, hFile:=hFile);
123
124             IF fbFileFlush.xDone THEN
125                 fbFileFlush(xExecute:=FALSE);
126
127                 // теперь можно перейти к шагу закрытия файла
128                 eState:=FileWork.CLOSE;
129             END_IF
130
131             IF fbFileFlush.xError THEN
132                 // обработка ошибок
133             END_IF
```

Рисунок 5.8.10 – Код шага FLUSH

На шаге **CLOSE** происходит закрытие файла с помощью ФБ FILE.CLOSE. После закрытия файла происходит переход на шаг **GET_SIZE**.

```
136     FileWork.CLOSE: // шаг закрытия файла
137
138             fbFileClose(xExecute:=TRUE, hFile:=hFile);
139
140             IF fbFileClose.xDone THEN
141                 fbFileClose(xExecute:=FALSE);
142                 xWBusy  :=  FALSE;
143
144                 // теперь можно перейти к шагу определения размера файла
145                 eState  :=  FileWork.GET_SIZE;
146             END_IF
147
```

Рисунок 5.8.11 – Код шага CLOSE

На шаге **GET_SIZE** происходит определение размера файла с помощью ФБ FILE.GetSize. После определения размер файла осуществляется переход на шаг **OPEN** для ожидания следующего управляющего сигнала. Если блок **fbFileGetSize** возвращает ошибку **NOT_EXIST** (файл не существует), то размер файла можно принять за **0**.

```
149     FileWork.GET_SIZE:  // шаг определения размера файла
150
151             fbFileGetSize(xExecute:=TRUE, sFileName:=sFileName);
152
153                 // определяем размер файла
154                 IF fbFileGetSize.xDone THEN
155                     udiArchSize:=fbFileGetSize.szSize;
156                     fbFileGetSize(xExecute:=FALSE);
157
158                     // вернемся на шаг открытия файла для ожидания следующего управляющего сигнала
159                     eState  :=  FileWork.OPEN;
160                 END_IF
161
162             // размер несуществующего файла...
163             IF fbFileGetSize.eError=FILE.ERROR.NOT_EXIST THEN
164
165                 // очевидно, можно интерпретировать как ноль
166                 udiArchSize :=  0;
167                 fbFileGetSize(xExecute:=FALSE);
168
169                 // вернемся на шаг открытия файла для ожидания следующего управляющего сигнала
170                 eState          :=  FileWork.OPEN;
171             ELSIF  fbFileGetSize.xError THEN
172                 fbFileGetSize(xExecute:=FALSE);
173                 eState          :=  FileWork.OPEN;
174             END_IF
175
```

Рисунок 5.8.12 – Код шага GET_SIZE

5.8.3 Создание визуализации

Следует создать интерфейс оператора для работы с каталогами. На рисунке 5.8.13 приведен внешний вид экрана **Visu04_StringFileExample**, который включает в себя:

- элемент **Комбинированное окно – целочисленный**, используемый для выбора накопителя, с которым будет работать программа. Настройки элемента описаны в п. 5.5.4. К элементу привязана переменная **iDevicePath**;
- прямоугольник **Путь к устройству**, отображающий значение переменной **sDevicePath**;
- прямоугольник **Имя файла** с привязанной переменной **sVisuFileName**. В настройках элемента на вкладке **InputConfiguration** для действия **OnMouseClick** задана операция **Записать переменную** (тип ввода – диалог **VisuKeypad**);
- прямоугольники **wValue** и **rValue** с привязанными переменными **stExportData.wValue** и **stExportData.rValue** соответственно. В настройках элементов на вкладке **InputConfiguration** для действия **OnMouseClick** задана операция **Записать переменную** (тип ввода – диалог **VisuNumpad**).
- кнопка **Записать** с привязанной переменной **xWrite**, поведение элемента – **Клавиша изображения**;
- прямоугольник **Размер архива** с привязанной переменной **udiArchSize**;
- прямоугольник **Кол-во записей в архиве** с привязанной переменной **uiArchEntry**.

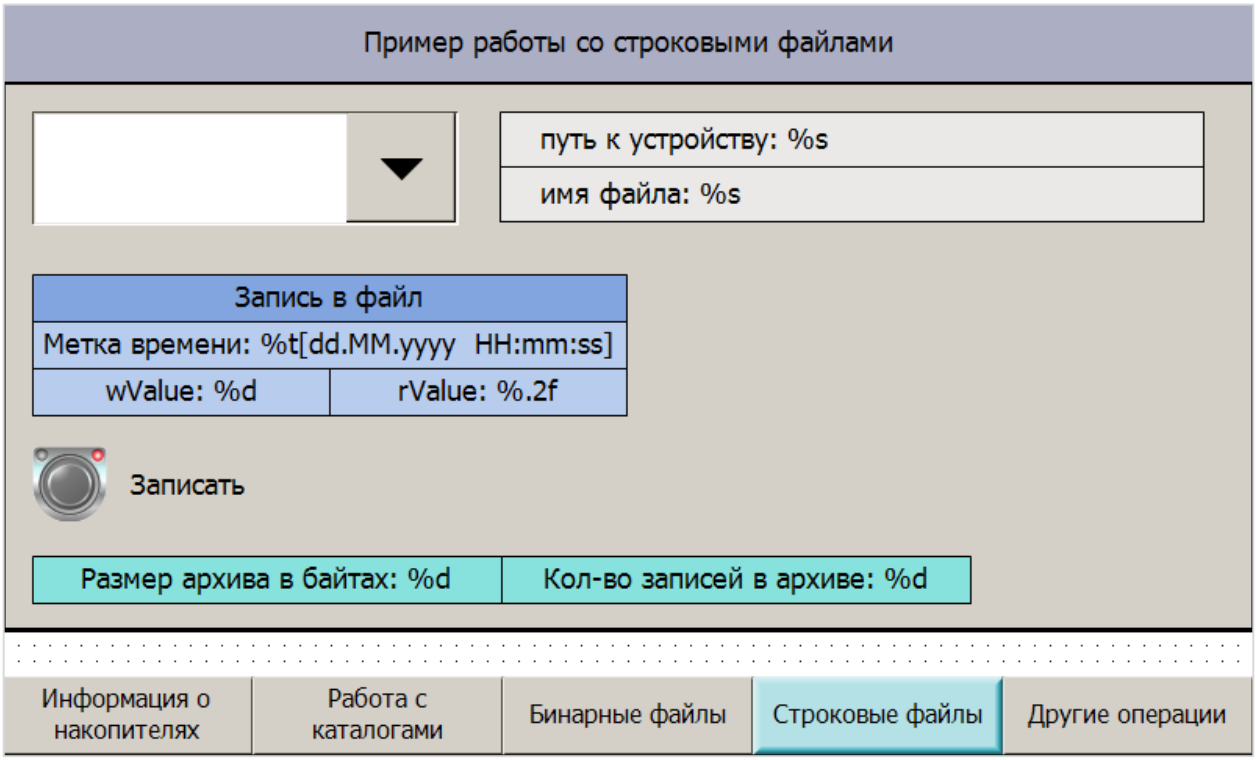


Рисунок 5.8.13 – Внешний вид экрана Visu04_StringFileExample

5.9 Дополнительные операции с файлами (PLC_PRG, действие act04_ActionsWithFiles)

В данном пункте рассмотрены другие доступные операции с файлами – переименование, копирование, удаление файлов.

5.9.1 Объявление переменных

В программе **PLC_PRG** следует объявить следующие переменные:

```
(*act04_ActionsWithFiles | операции с файлами *)

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

sFileName:                  STRING;             // полный путь к текущему файлу
sFileNameNew:               STRING;             // полный путь к создаваемому файлу
sVisuFileName:              STRING;             // имя текущего файла
sVisuFileNameNew:           STRING;             // имя создаваемого файла
sDeviceFilePath:            STRING;             // путь к текущему устройству
iDeviceFilePath:            INT;                // ID текущего устройства
sDeviceFilePathCopy:        STRING;             // путь к устройству для копирования файла
iDeviceFilePathCopy:        INT;                // ID устройства для копирования файла
sFileNameCopy:              STRING;             // полный путь для копирования файла
```

Рисунок 5.9.1 – Объявление переменных в программе PLC_PRG

5.9.2 Разработка программы

В программе **PLC_PRG** следует создать действие **act04_ActionWithFiles** (**PLC_PRG – Добавление объекта – Действие**) и вынести в него следующий код:

PLC_PRG.act04_ActionsWithFiles

```
// получаем путь к выбранному устройству
sDeviceFilePath     :=  DEVICE_PATH(iDeviceFilePath);

// склеиваем его с именами файлов
sFileName           :=  CONCAT(sDeviceFilePath, sVisuFileName);
sFileNameNew        :=  CONCAT(sDeviceFilePath, sVisuFileNameNew);

// получаем путь к выбранному устройству для копирования
sDeviceFilePathCopy :=  DEVICE_PATH(iDeviceFilePathCopy);
// склеиваем его с именем файла
sFileNameCopy       :=  CONCAT(sDeviceFilePathCopy, sVisuFileName);

// выполняем ФБ операций с файлами
fbFileRename(xExecute:=, sFileNameOld:=sFileName, sFileNameNew:=sFileNameNew);
fbFileCopy  (xExecute:=, xOverWrite:=TRUE, sFileNameDest:=sFileNameCopy, sFileNameSource:=sFileName);
fbFileDelete(xExecute:=, sFileName:=sFileName);
```

Рисунок 5.9.2 – Код действия act04_ActionWithFiles

Действие производит следующие операции:

- возвращает путь к выбранному накопителю по его **ID** с помощью функции **DEVICE_PATH,** созданной в п. 5.5.2;
- склеивает путь к накопителю с именами текущего файла и его копии;
- вызов экземпляров функциональных блоков переименования, копирования и удаления файлов.

ПРИМЕЧАНИЕ

В рамках примера вызов ФБ осуществляется без соотнесения входа **xExecute** с какой-либо переменной. Оператор с помощью нажатия кнопок будет воздействовать напрямую на входы блока. Пользователю необходимо реализовать свой алгоритм работы с данными блоками, который позволит решить его конкретную задачу.

ПРИМЕЧАНИЕ

В рамках примера в качестве строковых аргументов ФБ используются одни и те же переменные. В большинстве практических задач разумно использовать уникальные переменные для каждого ФБ.

Вызов созданного действия следует добавить в программу **PLC_PRG**:

PLC_PRG

```
46      fbSplitDT:                  SPLIT_DT_TO_FSTRINGS;    // ФБ конвертации времени в строку
47      asEntryDT:                  ARRAY [0..10] OF STRING; // метка времени в виде отдельных строковых разрядов
48
49      iSelectedEntry:             INT;                     // номер выбранной строки таблицы
50
51      xDown:                      BOOL;                    // сигнал "Открыть каталог"
52      xUp:                        BOOL;                    // сигнал "Перейти на уровень выше"
53      xFirstScan:                 BOOL;                    // сигнал "Сканирование каталога"
54
```

```
1  act01_DriveInfo();           // сбор информации о памяти СПК и накопителей
2  act02_DirExample();          // пример работы с каталогами (создание, переименование, удаление)
3  act03_DirList();             // пример получения информации о содержимом каталога
4  act04_ActionsWithFiles();    // пример работы с файлами (переименование, копирование, удаление)
5
```

Рисунок 5.9.3 – Вызов действия act04_ActionsWithFiles в программе PLC_PRG

5.9.3 Создание визуализации

Следует создать интерфейс оператора для дополнительных операций с файлами. На рисунке 5.9.4 приведен внешний вид экрана **Visu05_FilesActionsExample**, который включает в себя:

- два элемента **Комбинированное окно – целочисленный,** используемый для выбора накопителей, с файлами которых будет работать программа. К элементам привязаны переменные **iDeviceFilePath** и **iDeviceFilePathCopy**. Настройки элемента описаны в п. 5.5.4.
- два прямоугольника **Путь**, отображающие значения переменных **sDeviceFilePath** и **sDeviceFilePathCopy**;
- три прямоугольника **Текущее имя файла** с привязанной переменной **sVisuFileName**. В настройках элементов на вкладке **InputConfiguration** для действия **OnMouseClick** задана операция **Записать переменную** (тип ввода – диалог **VisuKeypad**).
- прямоугольник **Новое имя файла** с привязанной переменной **sVisuFileNameNew**. В настройках элемента на вкладке **InputConfiguration** для действия **OnMouseClick** задана операция **Записать переменную** (тип ввода – диалог **VisuKeypad**).
- три кнопки для выполнения операций с файлами с поведением **Клавиша изображения**. К кнопке **Переименовать** привязана переменная **fbFileRename.xExecute**, к кнопке **Копировать** – **fbFileCopy.xExecute**, к кнопке **Удалить файл** – **fbFileDelete.xExecute**.

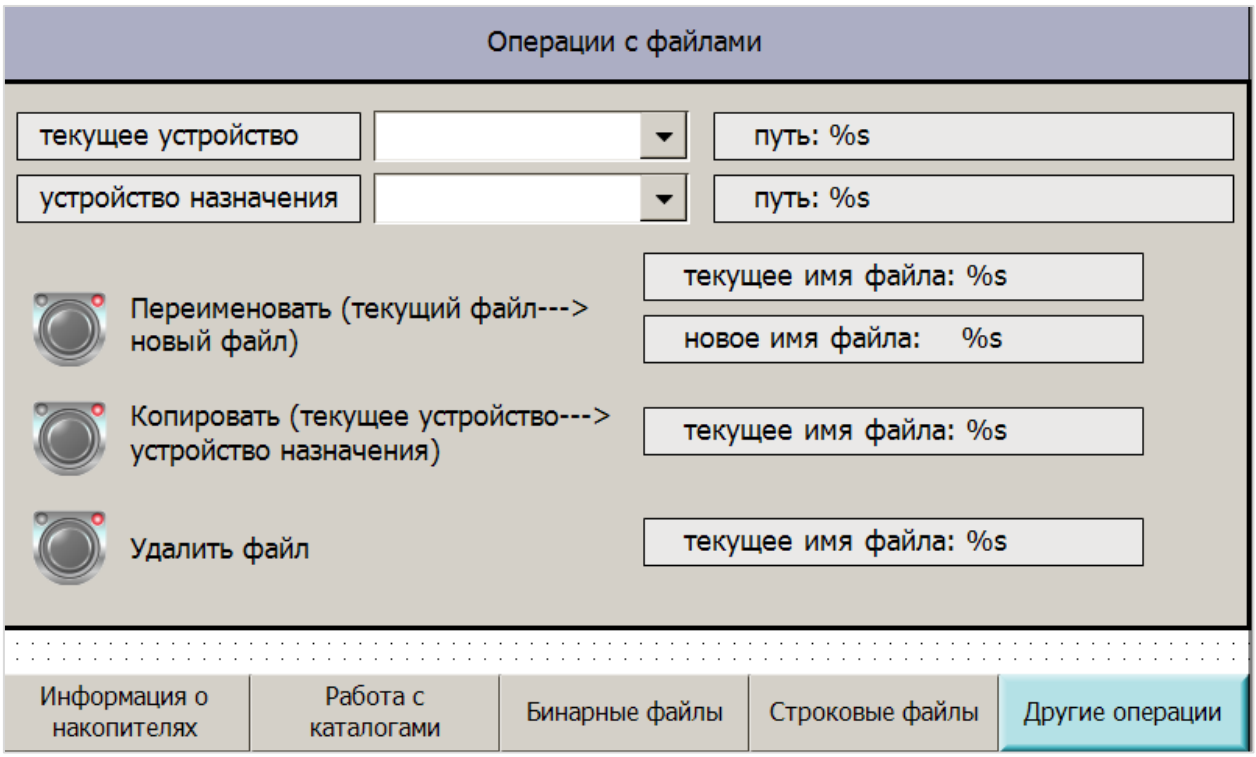


Рисунок 5.9.4 – Внешний вид экрана Visu05_FilesActionsExample

Визуализация также содержит кнопки переключения экранов (описание других экранов проекта приведено в соответствующих пунктах). Пример работы с экраном приведен в п. 5.10.

5.10 Работа с примером

Для работы с проектом следует:

1. Подключиться к контроллеру и загрузить в него проект. Если модель контроллера отличается от использованной в примере (**СПК1xx [M01]**), то следует выбрать нужный таргет-файл (**Device – Обновить устройство**).
2. После загрузки проекта будет отображен экран **Информация о накопителях**, содержащий информацию о накопителях. Первоначальный сбор информации о накопителях может занять несколько секунд. В это время не должно происходить переключения визуализаций проекта.

Рисунок 5.10.1 – Внешний вид экрана Информация о накопителях (Visu01_DriveInfo)

Нажать кнопку **Размонтировать**, чтобы демонтировать накопитель. Информация о занятой/доступной памяти накопителя обнулится, а индикатор «**Накопитель размонтирован**» загорится на 5 секунд, после чего погаснет. Для повторного монтирования накопителя следует извлечь его из контроллера и подключить снова.

Нажать кнопку **Работа с каталогами**, чтобы перейти на следующий экран.

3. На экране **Работа с каталогами** выбрать нужное устройство и ввести имя нового каталога (например, **test**). Нажать кнопку **Создать новый**.

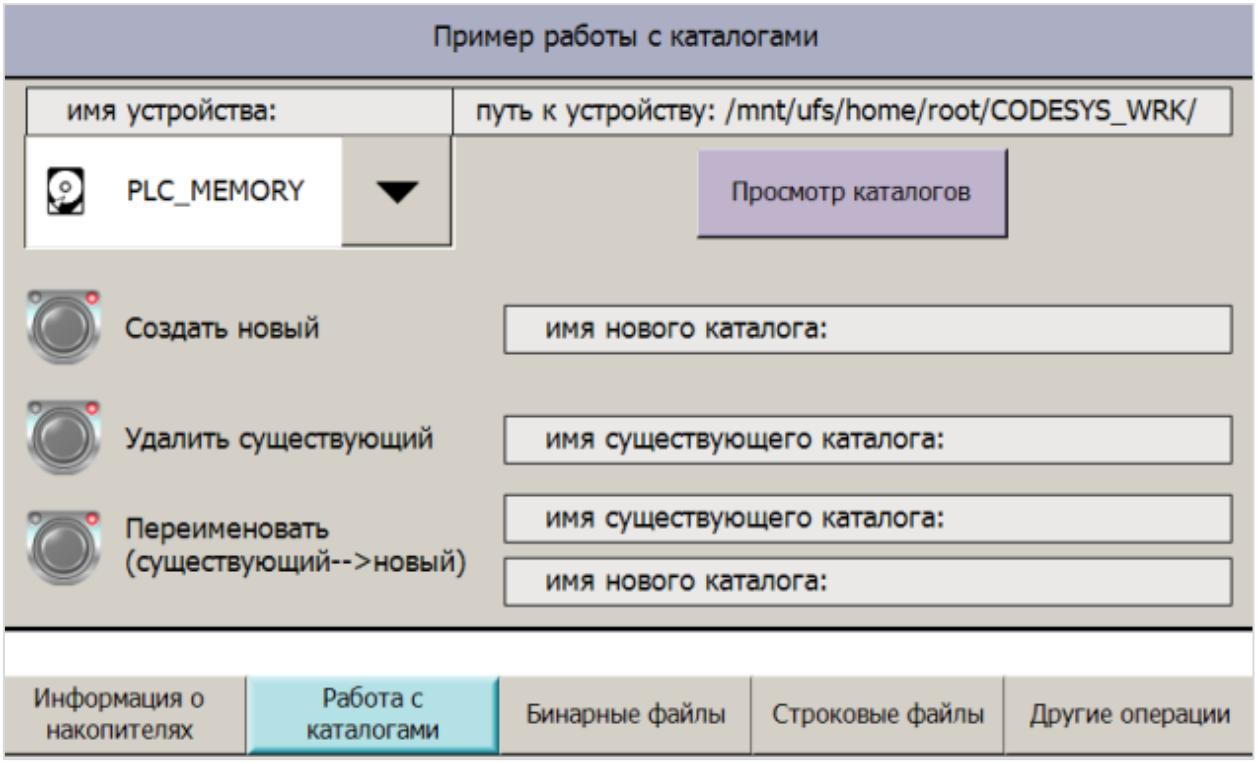


Рисунок 5.10.2 – Внешний вид экрана Работа с каталогами (Visu02_DirExample)

Затем следует подключиться к контроллеру с помощью утилиты **WinSCP** (см. п. 2.8), перейти по отображаемому пути (на рисунке 5.10.2 это **/mnt/ufs/home/root/CODESYS_WRK**) и убедится, что был создан каталог с названием **test.**

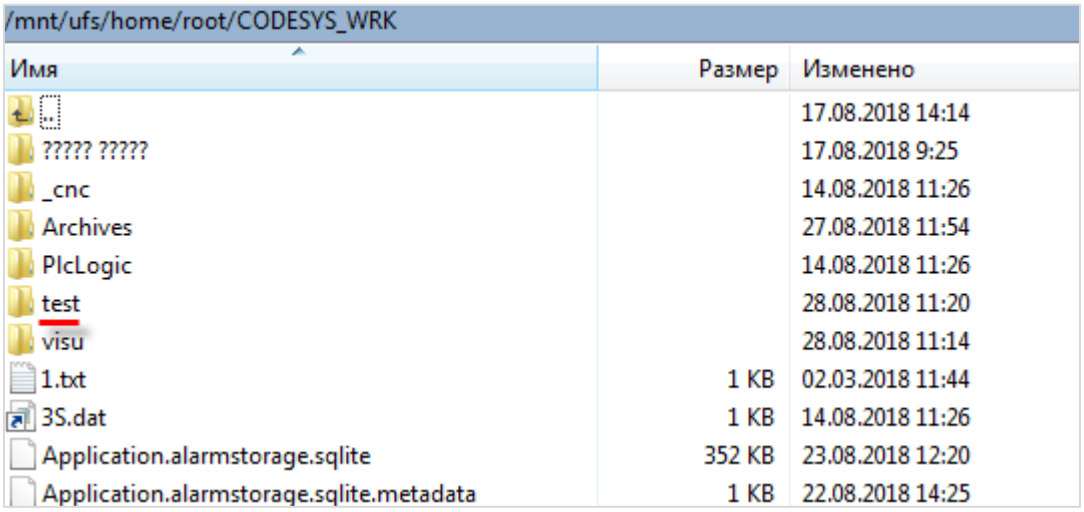
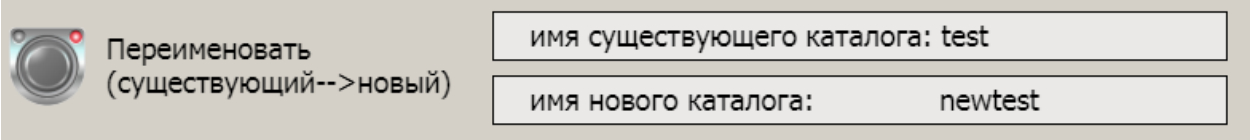


Рисунок 5.10.3 – Проверка создания нового каталога

Затем следует ввести имя существующего каталога **test** и имя нового каталога **newtest.**



Нажать кнопку **Переименовать**.

Проверить через **WinSCP**, что каталог был переименован:

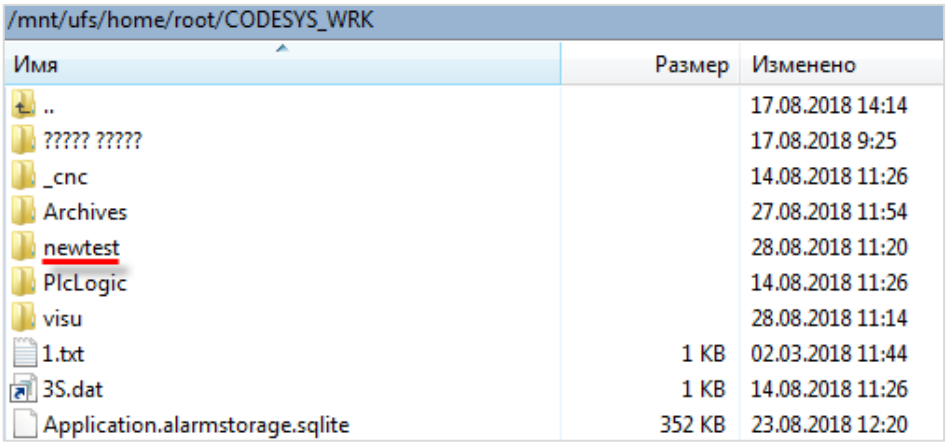
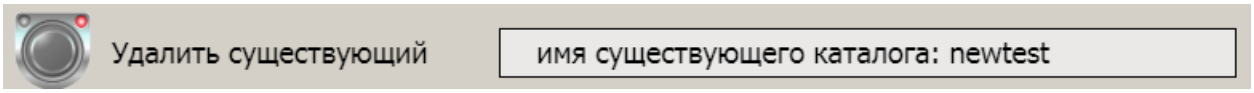


Рисунок 5.10.4 – Проверка переименования каталога

Затем следует ввести имя существующего каталога **newtest** и нажать кнопку **Удалить**. Проверить через **WinSCP**, что каталог был удален:



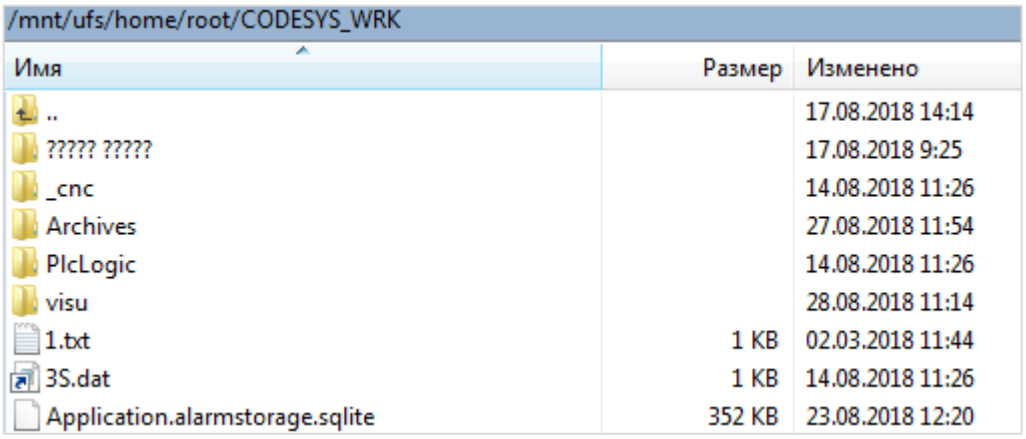
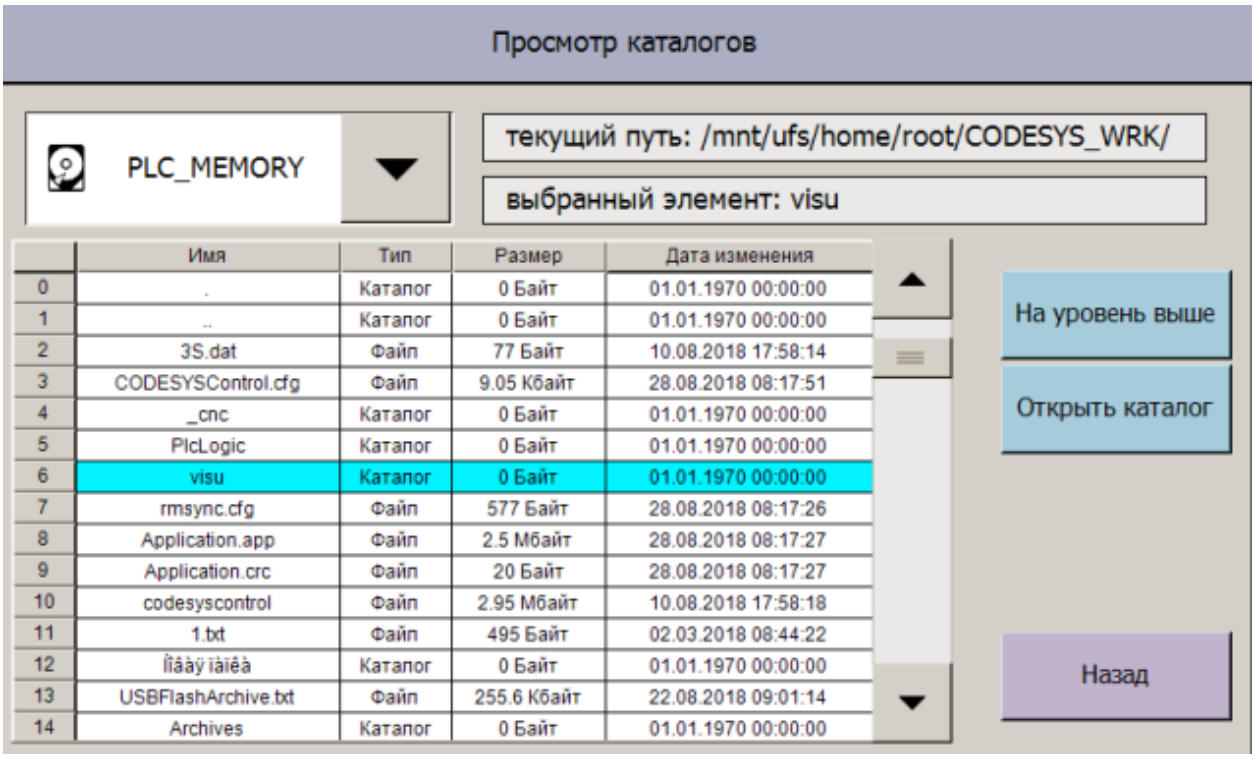


Рисунок 5.10.5 – Проверка удаления каталога

Затем следует нажать кнопку **Просмотр каталогов** (см. рисунок 5.10.2) и выбрать нужный накопитель. Будет автоматически произведено сканирование его корневого каталога. Результаты будут представлены в таблице. Следует выбрать нужный каталог с помощью курсора или элемента **Полоса прокрутки**, после чего нажать кнопку **Открыть каталог**. Чтобы выйти из каталога, следует нажать кнопку **На уровень выше**. Выйти из рабочего каталога нельзя.



	Имя	Тип	Размер	Дата изменения
0	.	Каталог	0 Байт	01.01.1970 00:00:00
1	..	Каталог	0 Байт	01.01.1970 00:00:00
2	3S.dat	Файл	77 Байт	10.08.2018 17:58:14
3	CODESYSControl.cfg	Файл	9.05 Кбайт	28.08.2018 08:17:51
4	_cnc	Каталог	0 Байт	01.01.1970 00:00:00
5	PlcLogic	Каталог	0 Байт	01.01.1970 00:00:00
6	visu	Каталог	0 Байт	01.01.1970 00:00:00
7	rmsync.cfg	Файл	577 Байт	28.08.2018 08:17:26
8	Application.app	Файл	2.5 Мбайт	28.08.2018 08:17:27
9	Application.crc	Файл	20 Байт	28.08.2018 08:17:27
10	codesyscontrol	Файл	2.95 Мбайт	10.08.2018 17:58:18
11	1.txt	Файл	495 Байт	02.03.2018 08:44:22
12	Îîâàÿ ïàïêà	Каталог	0 Байт	01.01.1970 00:00:00
13	USBFlashArchive.txt	Файл	255.6 Кбайт	22.08.2018 09:01:14
14	Archives	Каталог	0 Байт	01.01.1970 00:00:00

Рисунок 5.10.6 – Внешний вид экрана Просмотр каталогов (Visu06_DirList)

На рисунке выше размер и дата изменения каталогов отображается некорректно. Также не поддерживается отображения русскоязычных символов. Подробнее см. в п. 5.2.1.

Нажать кнопку **Назад**, чтобы вернуться на экран **Работа с каталогами.**

Нажать кнопку **Бинарные файлы**, чтобы перейти на следующий экран.

4. На экране **Бинарные файлы** выбрать нужное устройство и ввести имя файла (по умолчанию – **test.bin**). Задать значения **wValue** и **rValue** (*запись в файл*) и нажать кнопку **Записать**. Повторить операцию несколько раз, меняя значения переменных. Счетчик числа записей будет увеличиваться после каждой записи.

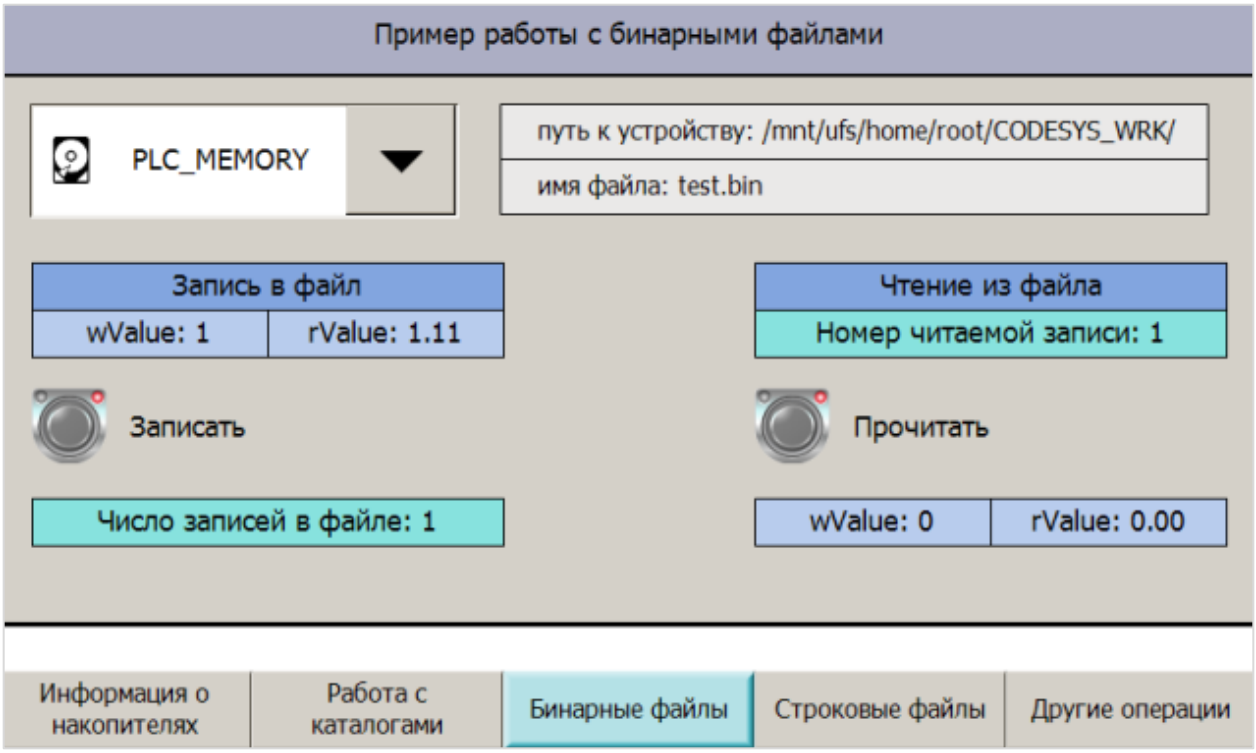


Рисунок 5.10.7 – Внешний вид экрана Бинарные файлы (Visu03_BinFileExample)

Проверить через **WinSCP,** что файл был создан:

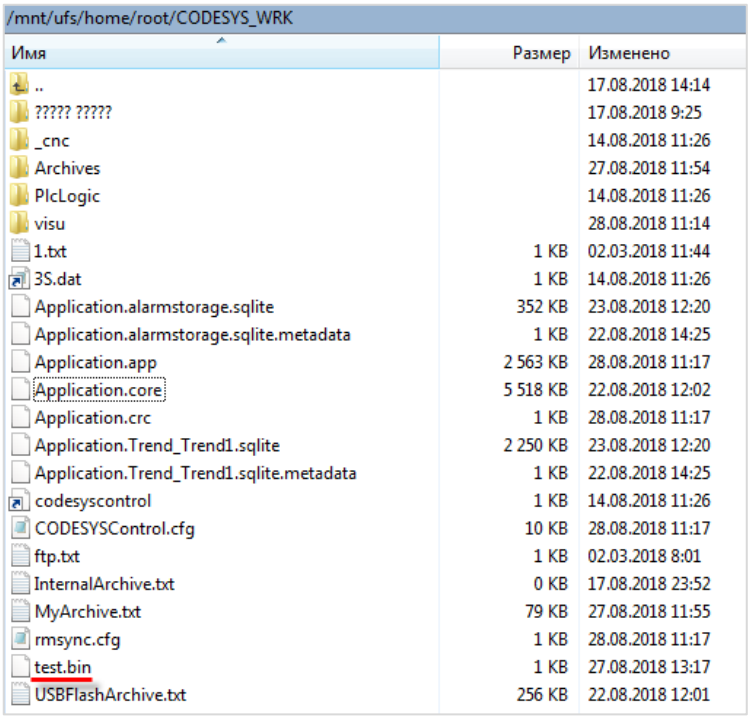


Рисунок 5.10.8 – Проверка создания файла

Выбрать номер читаемой записи (по умолчанию – **1**) и нажать кнопку **Прочитать**. Данные из файла будут считаные в переменные **wValue** и **rValue** (*чтение из файла*).

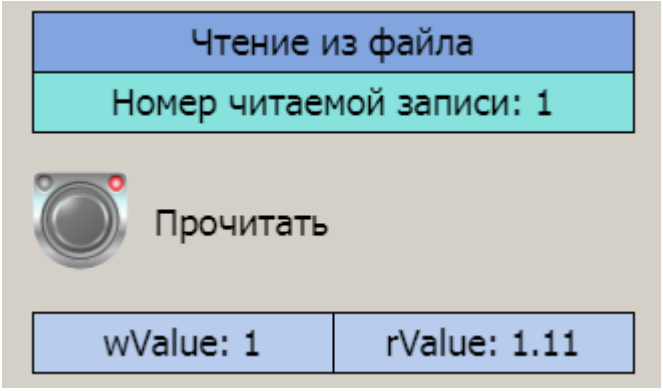


Рисунок 5.10.9 – Чтение данных из файла

Нажать кнопку **Строковые файлы**, чтобы перейти на следующий экран.

5. На экране **Строковые файлы** выбрать нужное устройство и ввести имя файла (по умолчанию – **test.csv**). Нажать кнопку **Записать,** чтобы создать файл и записать в него заголовок архива. Задать значения **wValue** и **rValue** и нажать кнопку **Записать**. Повторить операцию несколько раз, меняя значения переменных. Счетчик числа записей и размера архива будет увеличиваться после каждой записи.

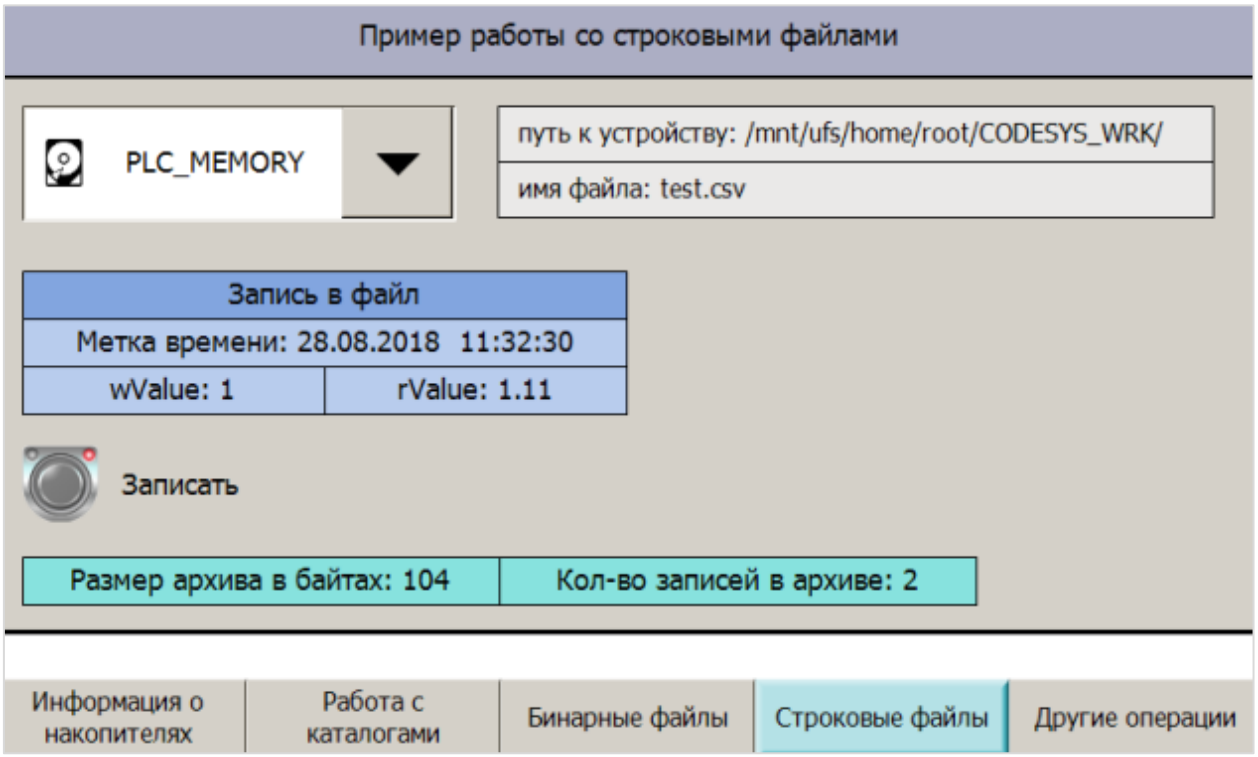


Рисунок 5.10.10 – Внешний вид экрана Строковые файлы (Visu04_StringFileExample)

Проверить через **WinSCP**, что файл был создан:

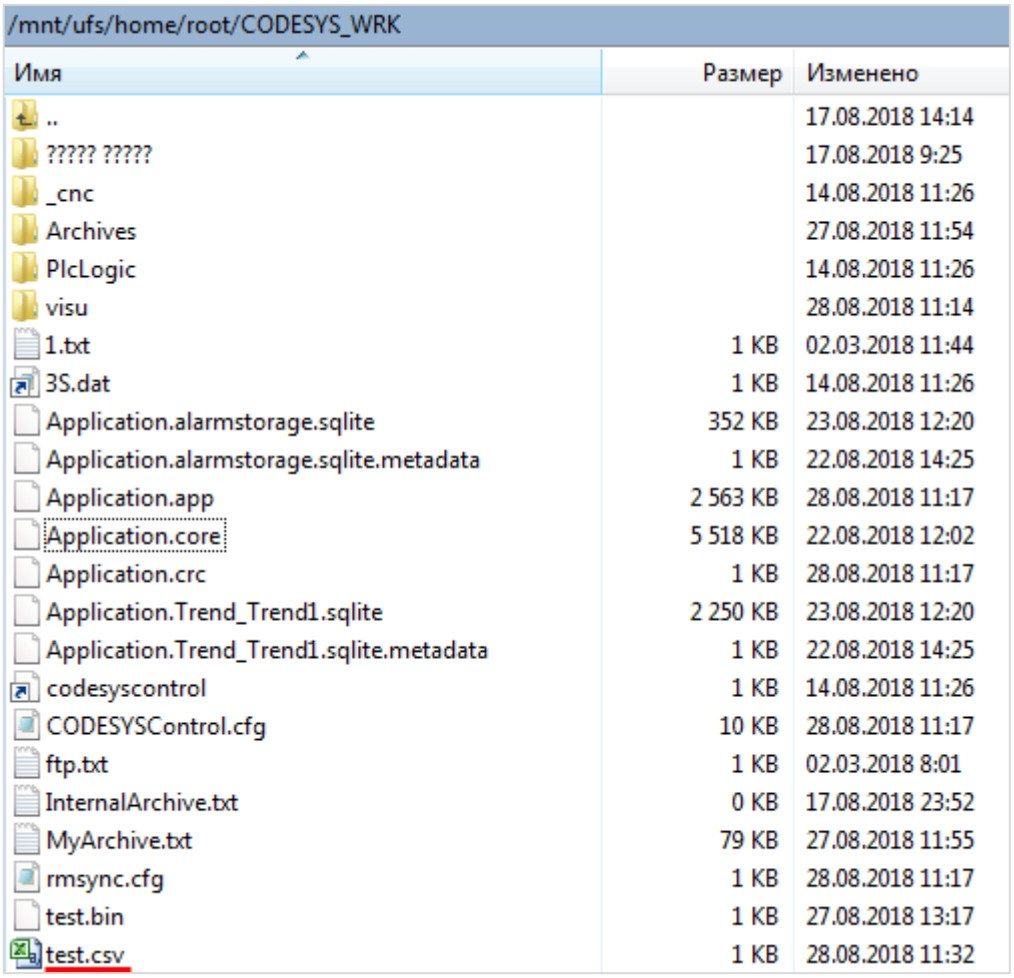


Рисунок 5.10.11 – Проверка создания файла

Скопировать его на ПК и открыть с помощью **Microsoft Excel** или другого ПО:

F17

	A	B	C	D
1	Дата	Время	Значение	Значение типа REAL
2	02.08.2017	12:23:51	1	1,22
3	02.08.2017	12:24:00	2	3,44
4	02.08.2017	12:24:01	2	3,44
5	02.08.2017	12:24:01	2	3,44
6	02.08.2017	12:24:02	2	3,44

Рисунок 5.10.12 – Пример содержимого файла

Нажать кнопку **Другие операции**, чтобы перейти на следующий экран.

6. На экране **Другие операции** выбрать текущее устройство. Ввести текущее и новое имя файла (например**, test.csv** и **new.csv**) и нажать кнопку **Переименовать**.

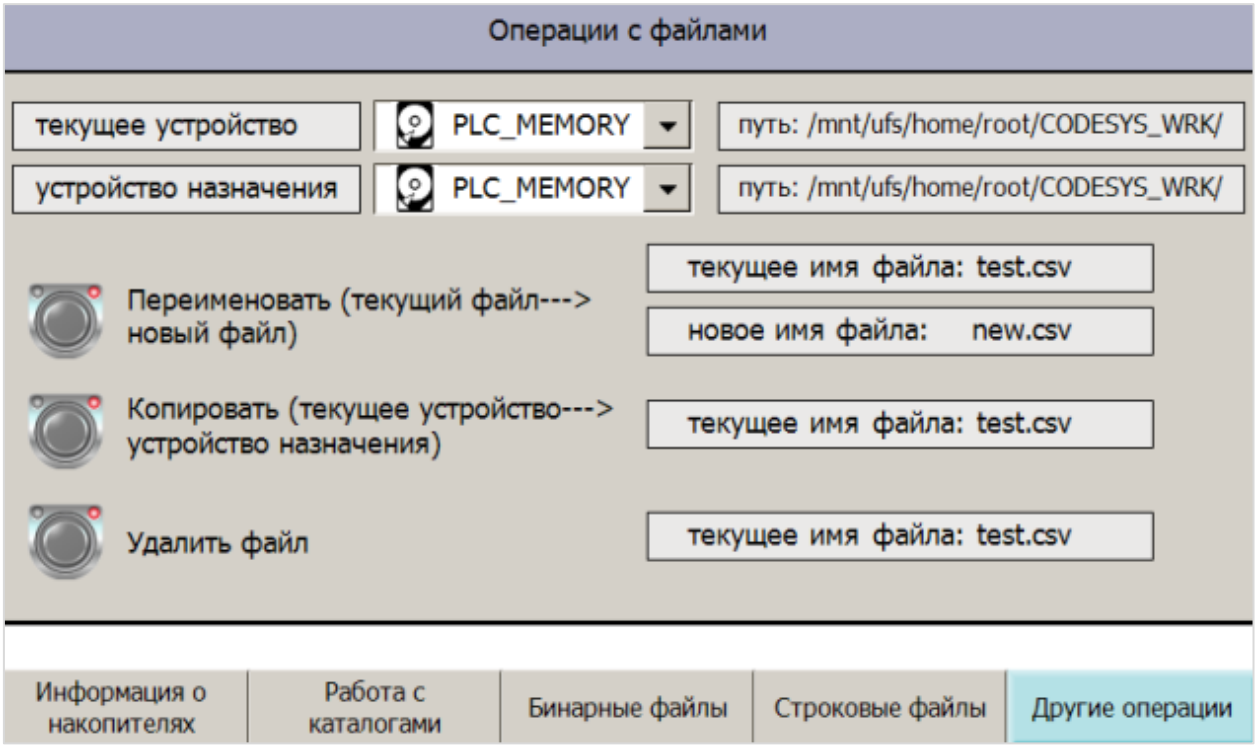


Рисунок 5.10.13 – Внешний вид экрана Другие операции (Visu05_FileActionsExample)

Проверить через **WinSCP**, что файл был переименован:

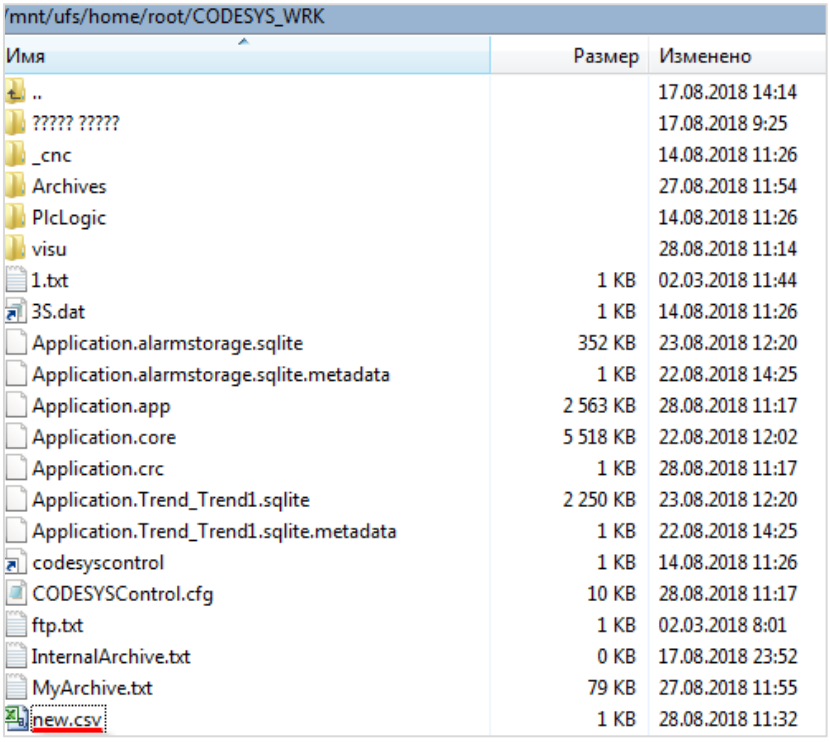


Рисунок 5.10.14 – Проверка переименования файла

Ввести текущее имя файла **test.bin** и выбрать устройство назначения (например, USB). Нажать кнопку **Копировать**.

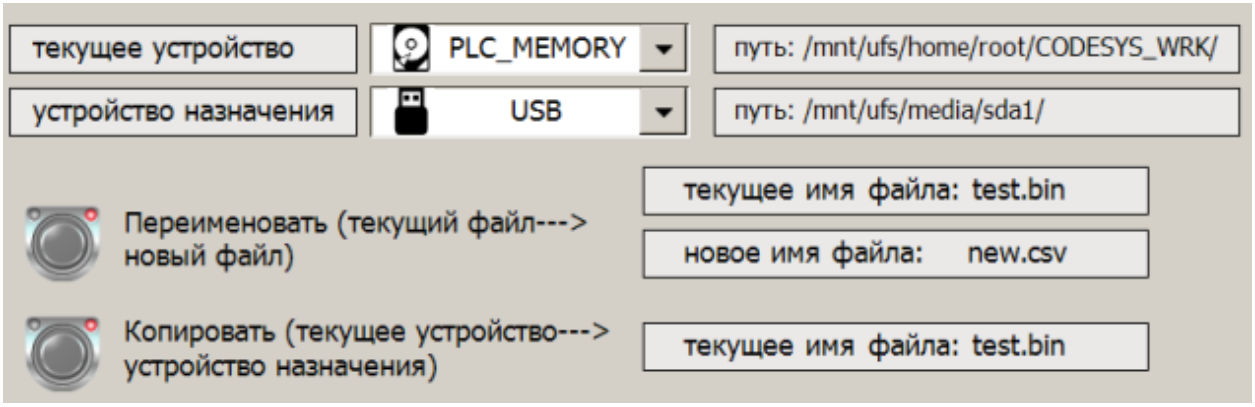


Рисунок 5.10.15 – Копирование файлов

Проверить через **WinSCP,** что файл был скопирован (на рисунке 5.10.15 выбран USB-накопитель и указан путь к нему – **/mnt/ufs/media/sda1**).

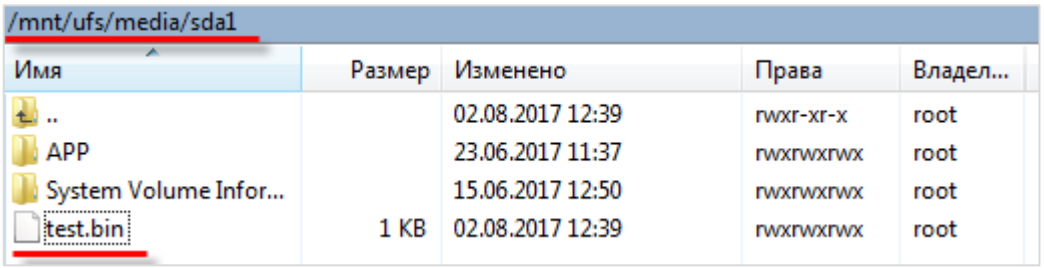


Рисунок 5.10.15 – Проверка копирования файла

Выбрать текущее устройство (например, USB) и нажать кнопку **Удалить**. Текущий файл (**test.bin**) будет удален с USB-накопителя:

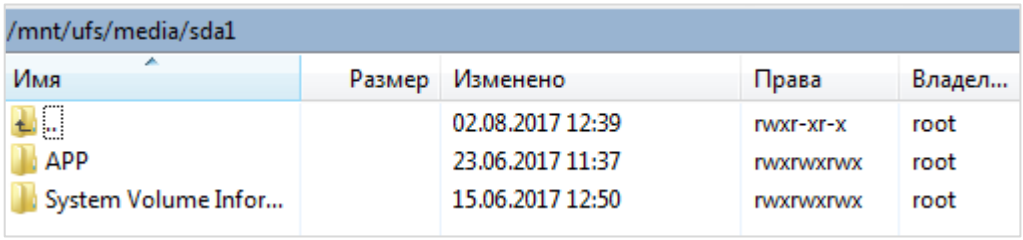


Рисунок 5.10.16 – Проверка удаления файла

5.11 Рекомендации и замечания

Ниже перечислены основные тезисы и рекомендации по разработке программ, работающих с файлами, использованные в данном документе.

- ФБ и программы, работающие с файлами, разбиваются на шаги, которые выполняются через оператор **CASE**;
- чтобы сделать прозрачным переходы между шагами, можно использовать **перечисления**;
- чтобы упростить отладку и повысить читабельность кода, можно выделять его законченные фрагменты в **действия**;
- после завершения каждой операции с файлом следует завершить работу соответствующего ФБ (обычно под этим понимается их вызов с параметром **xExecute=FALSE**);
- переход к следующему шагу должен происходить только после окончания предыдущего. Контроль окончания шага, в частности, может осуществляться с помощью выходов **xDone** соответствующих ФБ;
- текстовые и бинарные файлы отличаются форматом представления данных. Размер записи бинарного файла определяется размером записываемых данных (см. оператор **SIZEOF**), размер записи текстового файла можно определить с помощью функции **LEN** из библиотеки **Standard**;
- перед началом работы с файлом необходимо убедиться, что он существует;
- при работе с текстовыми файлами следует помнить о максимальной длине строк (см. рисунок 5.8.3).

Следует также отметить ряд моментов, оставшихся за пределами примеров документа:

- в рамках примера действия с файлами (запись, чтение и т. д.) происходят при нажатии соответствующих кнопок в визуализации, которые генерируют единичные импульсы в соответствующих переменных (**xWrite, xRead** и т. д.). Пользователь может создать алгоритм воздействия на эти переменные, который требуется для решения его конкретной задачи (циклическая запись в архив, запись по изменению, ежедневное создание нового файла архива и т. д.);
- в некоторых случаях требуется тщательная обработка ошибок. Следует контролировать выходы **xError** и **eError** соответствующих ФБ. См. описание кодов ошибок в п. 5.2.2;
- не следует пытаться открыть уже открытый или закрыть уже закрытый файл;
- перед началом работы с файлом, расположенном на внешнем накопителе, следует проверить, смонтирован ли этот накопитель;
- перед извлечением накопителя его следует демонтировать. Следует предусмотреть соответствующие окна/сообщения в визуализации, чтобы это было очевидно для оператора;
- следует контролировать доступное свободное место на контроллере/накопителе. Если оно заканчивается, следует остановить архивацию или начать перезаписывать файл;
- оптимальным решением является в каждый момент времени работать только с одним файлом. Попытка архивировать данные в несколько файлов одновременно может привести к снижению стабильности работы основной программы.

Приложение А. Листинг примера

А.1 Структуры и перечисления

А.1.1. Структура ArchData

```
// структура архивируемых данных

TYPE ArchData :
STRUCT
    wValue:         WORD;
    rValue:         REAL;
END_STRUCT
END_TYPE
```

А.1.2. Структура DriveInfo

```
// структура параметров файловой системы контроллера/подключенных к нему накопителей

TYPE DriveInfo :
STRUCT
    xIsMounted:     BOOL;       // флаг "накопитель примонтирован"
    xUnmount:   BOOL;       // сигнал размонтирования накопителя
    xUnmountDone:    BOOL;       // флаг "накопитель размонтирован"

    uliFullSize:   ULINT;       // общий объем доступного пространства (в байтах)
    uliUsedSize: ULINT;       // занятый объем доступного пространства (в байтах)
    uliFreeSize:  ULINT;       // свободный объем доступного пространства (в байтах)

    wsFullSize:  WSTRING;  // общий объем накопителя (формат. строка)
    wsUsedSize:    WSTRING;  // занятый объем накопителя (формат. строка)
    wsFreeSize: WSTRING;  // свободный объем накопителя (формат. строка)
END_STRUCT
END_TYPE
```

A.1.3. Перечисление FileDevice

```
{attribute 'strict'}

// тип устройства для архивации
TYPE FileDevice :
(
    PLC_MEMORY      :=    0,
    USB         :=    10,
    SD      :=    20,
    FTP         :=    30
);
END_TYPE
```

A.1.4. Перечисление FileDevice

```
{attribute 'strict'}

// имена шагов работы с файлами
TYPE FileWork :
(
    OPEN        := 0,
    CREATE      := 10,
    READ    := 20,
    SET_READ_POS  := 30,
    WRITE       := 40,
    FLUSH       := 50,
    CLOSE       := 60,
    GET_SIZE    := 70
);
END_TYPE
```

A.1.5. Структура VisuDirInfo

```
// структура информации о каталога/файла, отображаемой в визуализации

TYPE VisuDirInfo :
STRUCT
    sEntryName:         STRING;     // имя каталога/файл
    wsEntryType:        WSTRING;    // тип (каталог или файл)
    wsEntrySize:        WSTRING;    // размер файла в байтах
    sLastModification:  STRING;     // дата последнего изменения файла
END_STRUCT
END_TYPE
```

A.2 Структуры и перечисления

A.2.1. Функция BYTE_SIZE_TO_WSTRING

```
// функция преобразования числа байт в форматированную строку

FUNCTION BYTE_SIZE_TO_WSTRING : WSTRING
VAR_INPUT
    uliByteSize: ULINT;              // число байт
END_VAR
VAR CONSTANT
    c_uliBytePerKilobyte:      ULINT := 1024; // число байт в килобайте
    c_uliKilobytePerMegabyte:  ULINT := 1024 * c_uliBytePerKilobyte;
    c_uliMegabytePerGigabyte:  ULINT := 1024 * c_uliKilobytePerMegabyte;
END_VAR
VAR
    rByteSize:          REAL;        // промежуточная переменная
END_VAR



CASE uliByteSize OF

0  ..(c_uliBytePerKilobyte - 1):
BYTE_SIZE_TO_WSTRING := WCONCAT(ULINT_TO_WSTRING(uliByteSize), " Байт");

c_uliBytePerKilobyte ..(c_uliKilobytePerMegabyte - 1):
    rByteSize    := ULINT_TO_REAL(uliByteSize) /
    ULINT_TO_REAL(c_uliBytePerKilobyte);
    BYTE_SIZE_TO_WSTRING := WCONCAT(REAL_TO_FWSTRING(rByteSize, 2), " Кбайт");

c_uliKilobytePerMegabyte ..(c_uliMegabytePerGigabyte - 1):
rByteSize    := ULINT_TO_REAL(uliByteSize) /
    ULINT_TO_REAL(c_uliKilobytePerMegabyte);
    BYTE_SIZE_TO_WSTRING := WCONCAT(REAL_TO_FWSTRING(rByteSize, 2), " Мбайт");

c_uliMegabytePerGigabyte ..(32 * c_uliMegabytePerGigabyte):
    rByteSize    := ULINT_TO_REAL(uliByteSize) /
    ULINT_TO_REAL(c_uliMegabytePerGigabyte);
    BYTE_SIZE_TO_WSTRING := WCONCAT(REAL_TO_FWSTRING(rByteSize, 2), " Гбайт");

END_CASE
```

A.2.2. Функция CONCAT11

```
// функция склеивает заданное число строковых переменных, помещенных в массив

FUNCTION CONCAT11 : STRING(255)
VAR_INPUT
    asSTR:          ARRAY [0..c_MAX_STR] OF STRING;
END_VAR
VAR
    sBuffer:      STRING;         // промежуточная переменная
    i:            INT;            // счетчик для цикла
END_VAR

VAR CONSTANT
    c_MAX_STR:      INT:=10;      // размер массива строковых переменных
END_VAR



FOR i:=0 TO c_MAX_STR DO
    sBuffer:=CONCAT(sBuffer, asSTR[i]);
END_FOR

CONCAT11:=sBuffer;
```

A.2.3. Функция DEVICE_PATH

```
// функция возвращает путь для файловой системы контроллера или накопителя по ID

FUNCTION DEVICE_PATH : STRING
VAR_INPUT
    iDevice:  INT;              // ID устройства
END_VAR
VAR
END_VAR


CASE iDevice OF
    FileDevice.PLC_MEMORY:
        DEVICE_PATH:='/mnt/ufs/home/root/CODESYS_WRK/';
    FileDevice.USB:
        DEVICE_PATH:='/mnt/ufs/media/sda1/';
    FileDevice.SD:
        DEVICE_PATH:='/mnt/ufs/media/mmcblk0p1/';
          FileDevice.FTP:
        DEVICE_PATH:='/var/lib/ftp/in';
END_CASE
```

А.2.4. ФБ DIR_INFO

```
// ФБ для получения информации о содержимом каталога (о вложенных файлах/каталогах)

FUNCTION_BLOCK DIR_INFO
VAR_INPUT
    xExecute:       BOOL;               // сигнал запуска блока
    sDirName:       STRING;             // имя обрабатываемого каталога
END_VAR
VAR_OUTPUT
    xDone:          BOOL;               // флаг "данные получены"
    // информация о вложенных файлах/каталогах
    astDirInfo:     ARRAY [0..c_MAX_ENTRIES] OF FILE.FILE_DIR_ENTRY;
    uiEntryPos:     UINT;     // кол-во обработанных файлов и каталогов
END_VAR
VAR
    fbDirOpen:      FILE.DirOpen;   // ФБ открытия каталога
    fbDirList:      FILE.DirList; // ФБ получения информации о содержимом каталога
    fbDirClose:     FILE.DirClose;  // ФБ закрытия каталога

    hDirHandle:     FILE.CAA.HANDLE;   // дескриптор открытого каталога
    eState:         FileWork;     // перечисление с именами шагов
    fbStart:        R_TRIG;         // триггер запуска блока
END_VAR
VAR CONSTANT
    c_MAX_ENTRIES:     UINT  :=100;      // макс. число обрабатываемых файлов/каталогов
END_VAR



// детектируем сигнал запуска блока
fbStart(CLK:=xExecute);

// сбрасываем сигнал завершения работы
xDone:=FALSE;

CASE eState OF


    FileWork.OPEN:    // открываем каталог

        // обнуляем позицию для записи информации о файлах/каталогах
        uiEntryPos:=0;

        fbDirOpen(xExecute:=fbStart.Q, sDirName:=sDirName);

        IF fbDirOpen.xDone THEN
            hDirHandle  :=  fbDirOpen.hDir;
            fbDirOpen(xExecute:=FALSE);
            eState      :=  FileWork.READ;
        END_IF


    FileWork.READ:  // получаем информацию о вложенных файлах и каталогах

        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[uiEntryPos] :=  fbDirList.deDirEntry;
```

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

            // если число вложенных файлов/каталогов больше, чем размер массива...
            // ...то начинаем перезаписывать его с нуля
            IF uiEntryPos>c_MAX_ENTRIES THEN
                uiEntryPos   :=  0;
            END_IF

            fbDirList(xExecute:=FALSE);
        END_IF

        // если код ошибки - "NO_MORE_ENTRIES", то обработаны все файлы/каталоги...
        // ...и можно завершать работу блока
        IF  fbDirList.eError=FILE.ERROR.NO_MORE_ENTRIES  THEN
            fbDirList(xExecute:=FALSE);
            eState     :=  FileWork.CLOSE;
        END_IF


    FileWork.CLOSE:   // завершение работы блока

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

        IF fbDirClose.xDone THEN
            fbDirClose(xExecute:=FALSE);

            // устанавливаем флаг завершения работы
            xDone     :=  TRUE;

            eState     :=  FileWork.OPEN;
        END_IF

END_CASE
```

A.2.5. Функция LEAD_ZERO

```
// функция преобразует число в строку с ведущим нулем

FUNCTION LEAD_ZERO : STRING
VAR_INPUT
    uiInput:   UINT;
END_VAR
VAR
END_VAR


IF uiInput>9 THEN
    LEAD_ZERO:=UINT_TO_STRING(uiInput);
ELSE
    LEAD_ZERO:=CONCAT('0', UINT_TO_STRING(uiInput));
END_IF
```

A.2.6. Функция REAL_TO_FSTRING

```
// функция конвертирует значение типа REAL в строку с n знаков после запятой

FUNCTION REAL_TO_FSTRING :        STRING
VAR_INPUT
    rVar:            REAL;       // входное значение
    usiPrecision:        USINT;      // нужное кол-во знаков после запятой
END_VAR
VAR
    uliVar:          ULINT;      // промежуточная переменная
    lrVar:           LREAL;      // промежуточная переменная
    sVar:            STRING;     // промежуточная переменная
END_VAR

// определяем знак
xSign := (rVar<0.0);

// оставляем нужное кол-во знаков после запятой
uliVar :=  LREAL_TO_ULINT(ABS (rVar)*EXPT(10, usiPrecision) );
lrVar  :=  LINT_TO_LREAL(uliVar) / EXPT(10, usiPrecision);
sVar   :=  LREAL_TO_STRING(lrVar);

// если нужно - возвращаем знак "минус"
IF xSign THEN
    sVar := CONCAT('-', sVar);
END_IF

// меняем точку на запятую для корректного отображения в MS Excel
REAL_TO_FSTRING:=REPLACE(sVar, ',', 1, FIND(sVar, '.'));
```

A.2.7. Функция REAL_TO_FWSTRING

```
// функция конвертирует значение типа REAL в строку с n знаков после запятой

FUNCTION REAL_TO_FWSTRING :        WSTRING
VAR_INPUT
    rVar:            REAL;       // входное значение
    usiPrecision:        USINT;      // нужное кол-во знаков после запятой
END_VAR
VAR
    uliVar:          ULINT;      // промежуточная переменная
    lrVar:           LREAL;      // промежуточная переменная

END_VAR


uliVar          :=  LREAL_TO_ULINT( (rVar)*EXPT(10, usiPrecision) );
lrVar           :=  LINT_TO_LREAL(uliVar) / EXPT(10, usiPrecision);
REAL_TO_FWSTRING    :=  LREAL_TO_WSTRING(lrVar);
```

А.2.8. ФБ SPLIT_DT_TO_FSTRINGS

```
// ФБ разделяет метку времени типа DT на строковые представления отдельных разрядов с
ведущими нулями

FUNCTION_BLOCK SPLIT_DT_TO_FSTRINGS
VAR_INPUT
    dtDateAndTime:          DT;       // метка времени в формате DT
END_VAR
VAR_OUTPUT
    sYear:              STRING;     // разряды времени в строковом представлении
    sMonth:             STRING;     //
    sDay:           STRING;     //
    sHour:              STRING;     //
    sMinute:        STRING;     //
    sSecond:        STRING;     //
END_VAR
VAR
    uiYear:             UINT;     // разряды времени в десятичном представлении
    uiMonth:        UINT;     //
    uiDay:          UINT;     //
    uiHour:             UINT;     //
    uiMinute:       UINT;     //
    uiSecond:           UINT;     //
END_VAR


DTU.DTSplit
(
    dtDateAndTime,
    ADR(uiYear),
    ADR(uiMonth),
    ADR(uiDay),
    ADR(uiHour),
    ADR(uiMinute),
    ADR(uiSecond)
);

sYear    :=    UINT_TO_STRING(uiYear);
sMonth      :=     LEAD_ZERO(uiMonth);
sDay    :=    LEAD_ZERO(uiDay);
sHour    :=    LEAD_ZERO(uiHour);
sMinute      :=     LEAD_ZERO(uiMinute);
sSecond    :=    LEAD_ZERO(uiSecond);
```

А.3 Программа PLC_PRG

```
// пример действий с каталогами и файлами (помимо чтения и записи)

PROGRAM PLC_PRG
VAR
    (*act01_DriveInfo | информация о памяти контроллера и накопителей*)

    xDriveInfo:       BOOL:= TRUE;    // режим сбора данных (TRUE - вкл.)

    stPlcMemory:      DriveInfo;      // структура параметров памяти контроллера
    stUsbMemory:      DriveInfo;      // структура параметров памяти USB-накопителя
    stSdMemory:       DriveInfo;      // структура параметров памяти SD-накопителя

    fbUsbUnmountTimeout:  TON;        // таймер сброса флага "USB отмонтирован"
    fbSdUnmountTimeout:   TON;        // таймер сброса флага "SD отмонтирован"


    (*act02_DirExample | операции с каталогами*)

    fbDirCreate:      FILE.DirCreate; // ФБ создания каталога
    fbDirRemove:      FILE.DirRemove;  // ФБ удаления каталога
    fbDirRename:      FILE.DirRename;  // ФБ переименования каталога

    sDirName:         STRING;         // полный путь к текущему каталогу
    sDirNameNew:      STRING;         // полный путь для создаваемого каталога
    sVisuDirName:     STRING;         // имя текущего каталога
    sVisuDirNameNew:  STRING;         // имя создаваемого каталога
    sDeviceDirPath:   STRING;         // путь к устройству
    iDeviceDirPath: INT;              // ID устройства


    (*act03_DirList | информация о выбранном каталоге*)

    fbDirInfo:        DIR_INFO;       // ФБ сбора информации о каталоге
    xDirList:       BOOL;             // сигнал сбора информации о каталоге
    i:            INT;                // счетчик для цикла

    // путь к выбранному каталогу
    sDirListPath:     STRING :='/mnt/ufs/home/root/CODESYS_WRK/';

    // путь к предыдущему выбранному каталогу
    sLastDevice:      STRING;

    // массив данных о вложенных файлах/каталогов для визуализации
    astVisuDirInfo:  ARRAY [0..c_MAX_ENTRIES] OF VisuDirInfo;

    fbSplitDT:        SPLIT_DT_TO_FSTRINGS;    // ФБ конвертации времени в строку
        asEntryDT:        ARRAY [0..10] OF STRING; // метка времени в виде отдельных
                            // строковых разрядов

    iSelectedEntry:     INT;            // номер выбранной строки таблицы

    xDown:            BOOL;         // сигнал "Открыть каталог"
    xUp:            BOOL;         // сигнал "Перейти на уровень выше"
        xHideUp:        BOOL;         // переменная неактивности кнопки "Открыть каталог"
    xFirstScan:       BOOL;         // сигнал "Сканирование каталога"
```

```
    (*act04_ActionsWithFiles | операции с файлами *)

    fbFileRename:       FILE.RENAME;     // ФБ переименования файла
    fbFileCopy:       FILE.COPY;     // ФБ копирования файла
    fbFileDelete:     FILE.DELETE;       // ФБ удаления файла

    sFileName:       STRING;          // полный путь к текущему файлу
    sFileNameNew:     STRING;          // полный путь к создаваемому файлу
    sVisuFileName:     STRING;          // имя текущего файла
    sVisuFileNameNew:   STRING;          // имя создаваемого файла
    sDeviceFilePath:    STRING;          // путь к текущему устройству
    iDeviceFilePath:    INT;           // ID текущего устройства
    sDeviceFilePathCopy: STRING;          // путь к устройству для копирования файла
    iDeviceFilePathCopy:  INT;           // ID устройства для копирования файла
    sFileNameCopy:    STRING;          // полный путь для копирования файла

END_VAR


VAR CONSTANT
    // максимальное число вложенных элементов каталога
    c_MAX_ENTRIES:          UINT       :=100;

        // разделитель для пути в файловой системе
    c_sCharSlash:           STRING(1)  :='/';

        c_byCodeSlash:          BYTE      :=16#2F; // ASCII-код разделителя

    // пустая структура для очистки таблицы
    c_astVisuDirInfoNull:       ARRAY [0..c_MAX_ENTRIES] OF VisuDirInfo;
END_VAR

// код программы PLC_PRG

act01_DriveInfo();       // сбор информации о памяти контроллера и накопителей
act02_DirExample();        // пример работы с каталогами (создание, переименование, удаление)
act03_DirList();      // пример получения информации о содержимом каталога
act04_ActionsWithFiles();// пример работы с файлами (переименование, копирование, удаление)
```

A.3.1. Действие act01_DriveInfo

```
// преобразование размеров полной/занятой/свободной памяти в форматированную строку

stSpkMemory.wsFullSize    := BYTE_SIZE_TO_WSTRING(stSpkMemory.lwFullSize);
stSpkMemory.wsUsedSize    := BYTE_SIZE_TO_WSTRING(stSpkMemory.lwUsedSize);
stSpkMemory.wsFreeSize    := BYTE_SIZE_TO_WSTRING(stSpkMemory.lwFreeSize);

stUSB.wsFullSize    := BYTE_SIZE_TO_WSTRING(stUSB.lwFullSize);
stUSB.wsUsedSize    := BYTE_SIZE_TO_WSTRING(stUSB.lwUsedSize);
stUSB.wsFreeSize    := BYTE_SIZE_TO_WSTRING(stUSB.lwFreeSize);

stSD.wsFullSize    := BYTE_SIZE_TO_WSTRING(stSD.lwFullSize);
stSD.wsUsedSize    := BYTE_SIZE_TO_WSTRING(stSD.lwUsedSize);
stSD.wsFreeSize    := BYTE_SIZE_TO_WSTRING(stSD.lwFreeSize);


// сброс флагов "устройство отмонтировано" через 5 секунд после отмонтирования устройства

fbUsbUnmountTimeout(IN:=stUSB.xUnmountDone, PT:=T#5S);

IF fbUsbUnmountTimeout.Q THEN
    stUSB.xUnmount:=FALSE;
END_IF

fbSdUnmountTimeout(IN:=stSD.xUnmountDone, PT:=T#5S);

IF fbSdUnmountTimeout.Q THEN
    stSD.xUnmount:=FALSE;
END_IF
```

A.3.2. Действие act02_DirExample

```
// получаем путь к выбранному устройству
sDeviceDirPath    := DEVICE_PATH(iDeviceDirPath);

// склеиваем его с именами каталогов
sDirName        := CONCAT(sDeviceDirPath, sVisuDirName);
sDirNameNew     := CONCAT(sDeviceDirPath, sVisuDirNameNew);

// выполняем ФБ операций с каталогами
fbDirCreate(xExecute:=, sDirName:=sDirNameNew, xParent:=TRUE);
fbDirRename(xExecute:=, sDirNameOld:=sDirName, sDirNameNew:=sDirNameNew);
fbDirRemove(xExecute:=, sDirName:=sDirName, xRecursive:=TRUE);
```

A.3.3. Действие act03_DirList

```
// получаем путь к выбранному устройству
sDeviceDirPath:=DEVICE_PATH(iDeviceDirPath);

// при загрузке проекта и при выборе нового устройства сканируем его корневой каталог
IF NOT(xFirstScan) OR sDeviceDirPath<>sLastDevice THEN
    sDirListPath    := sDeviceDirPath;
    sLastDevice     := sDeviceDirPath;
    xDirList        := TRUE;
    xFirstScan      := TRUE;
END_IF
```

```
// если выбранный элемент - файл или специальный каталог, то скрываем кнопку "Открыть каталог"
xHideUp  := astVisuDirInfo[iSelectedEntry].sEntryName='..'
          OR astVisuDirInfo[iSelectedEntry].sEntryName='.' OR
          astVisuDirInfo[iSelectedEntry].wsEntryType="Файл";


// по сигналу переходим в выбранный каталог
IF xDown THEN

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

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

    xDown         :=  FALSE;
    xDirList    :=  TRUE;
END_IF


// по сигналу переходим на уровень выше
IF xUp AND sDirListPath<>sDeviceDirPath THEN

    // 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 удален самый нижний каталог
    FOR i:=LEN(sDirListPath)-1 TO 0 BY -1 DO

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

    xUp             :=  FALSE;
    xDirList        :=  TRUE;
END_IF


// получаем информацию о содержимом каталога
fbDirInfo(xExecute:=xDirList, sDirName:=sDirListPath);

IF fbDirInfo.xDone THEN

    // стираем информацию о предыдущем открытом каталоге
    astVisuDirInfo    :=  c_astVisuDirInfoNull;
    // перереходим к верхней строке таблицы
    iSelectedEntry   :=  0;



// заполняем массив структур информацией о содержимом каталога
FOR i:=0 TO UINT_TO_INT(fbDirInfo.uiEntryPos-1) DO

astVisuDirInfo[i].sEntryName     :=  fbDirInfo.astDirInfo[i].sEntry;
astVisuDirInfo[i].wsEntrySize    :=  BYTE_SIZE_TO_WSTRING(fbDirInfo.astDirInfo[i].szSize);
astVisuDirInfo[i].wsEntryType    :=  SEL(fbDirInfo.astDirInfo[i].xDirectory, "Файл", "Каталог");
```

```
    // преобразуем дату и время последнего изменения файла в форматированную строку
        fbSplitDT(dtDateAndTime:=fbDirInfo.astDirInfo[i].dtLastModification);

    asEntryDT[0]    :=  fbSplitDT.sDay;
    asEntryDT[1]    :=  '.';
    asEntryDT[2]    :=  fbSplitDT.sMonth;
    asEntryDT[3]    :=  '.';
    asEntryDT[4]    :=  fbSplitDT.sYear;
    asEntryDT[5]    :=  ' ';
    asEntryDT[6]    :=  fbSplitDT.sHour;
    asEntryDT[7]    :=  ':';
    asEntryDT[8]    :=  fbSplitDT.sMinute;
    asEntryDT[9]    :=  ':';
    asEntryDT[10]   :=  fbSplitDT.sSecond;

astVisuDirInfo[i].sLastModification  :=  CONCAT11(asEntryDT);

xDirList                :=  FALSE;

END_FOR
END_IF
```

A.3.4. Действие act04_ActionsWithFiles

```
// получаем путь к выбранному устройству
sDeviceFilePath     :=  DEVICE_PATH(iDeviceFilePath);

// склеиваем его с именами файлов
sFileName       :=  CONCAT(sDeviceFilePath, sVisuFileName);
sFileNameNew    :=  CONCAT(sDeviceFilePath, sVisuFileNameNew);

// получаем путь к выбранному устройству для копирования
sDeviceFilePathCopy :=  DEVICE_PATH(iDeviceFilePathCopy);
// склеиваем его с именем файла
sFileNameCopy       :=  CONCAT(sDeviceFilePathCopy, sVisuFileName);

// выполняем ФБ операций с файлами
fbFileRename(xExecute:=, sFileNameOld:=sFileName, sFileNameNew:=sFileNameNew);
fbFileCopy   (xExecute:=, xOverWrite:=TRUE, sFileNameDest:=sFileNameCopy,
        sFileNameSource:=sFileName);
fbFileDelete(xExecute:=, sFileName:=sFileName);
```

A.4 Программа BinFileExample

```
// пример экспорта и импорта данных из бинарного файла

PROGRAM BinFileExample_PRG
VAR
    fbFileOpen:         FILE.OPEN;      // ФБ открытия файла
    fbFileClose:        FILE.CLOSE;         // ФБ закрытия файла
    fbFileWrite:        FILE.WRITE;         // ФБ записи в файл
    fbFileRead:         FILE.READ;      // ФБ чтения из файла
    fbFileFlush:        FILE.FLUSH;         // ФБ сброса буфера в файл
    fbFileSetPos:           FILE.SetPos;        // ФБ установки позиции для чтения
    fbFileGetSize:          FILE.GetSize;       // ФБ получения размера файла

    hFile:              FILE.CAA.HANDLE;    // дескриптор открытого файла
    stExportBinData:        ArchData;       // структура экспортируемых данных
    stImportBinData:        ArchData;       // структура для импорта данных
    udiWriteEntry:          UDINT;          // число записей в файле
    udiReadEntry:           UDINT       := 1;   // позиция для чтения из файла
    sFileName:          STRING;         // полный путь к файлу
    sDevicePath:            STRING;         // путь к устройству
    iDevicePath:        INT;            // ID устройства
    sVisuFileName:          STRING  :='test.bin';   // имя файла

    xWrite:             BOOL;           // сигнал записи в файл
    xRead:              BOOL;           // сигнал чтения из файла
    xWBusy:             BOOL;           // флаг "запись в файл"
    xRBusy:             BOOL;           // флаг "чтение из файла"
    eState:             FileWork:= FileWork.GET_SIZE;   // шаг операции с файлом

    fbWriteTrig:        F_TRIG;         // триггер записи в файл
    fbReadTrig:         F_TRIG;         // триггер чтения из файла
END_VAR




// получаем путь к выбранному устройству
sDevicePath:=  DEVICE_PATH(iDevicePath);

// склеиваем его с именем выбранного файла
sFileName   :=  CONCAT(sDevicePath, sVisuFileName);

// детектируем сигнал записи в файл или чтения из файла
fbWriteTrig(CLK:=xWrite);
fbReadTrig(CLK:=xRead);

// в зависимости от пришедшего сигнала взводим соответствующий флаг
IF fbWriteTrig.Q THEN
    xWBusy  :=  TRUE;
ELSIF fbReadTrig.Q THEN
    xRBusy  :=  TRUE;
END_IF
```

```
CASE eState OF

    FileWork.OPEN:    // шаг открытия файла

    // в зависимости от команды выбираем нужный режим работы с файлом (чтение или запись)
    IF xWBusy THEN
        fbFileOpen(xExecute:=TRUE, sFileName:=sFileName, eFileMode:=FILE.MODE.MAPPD);
    ELSIF xRBusy THEN
        fbFileOpen(xExecute:=TRUE, sFileName:=sFileName, eFileMode:=FILE.MODE.MREAD);
    END_IF

            // если файл, в который производится запись, не существует, то создадим его
            IF xWBusy AND fbFileOpen.eError=FILE.ERROR.NOT_EXIST THEN
                fbFileOpen(xExecute:=FALSE);
                eState    :=  FileWork.CREATE;
            END_IF

            //  если файл существует и был успешно открыт, то переходим к нужному шагу
            // (записи в файл или установки позиции для чтения)
            IF fbFileOpen.xDone THEN
                hFile  :=  fbFileOpen.hFile;
                fbFileOpen(xExecute:=FALSE);

                IF xWBusy THEN
                    eState    :=  FileWork.WRITE;
                ELSIF xRBusy THEN
                    eState    :=  FileWork.SET_READ_POS;
                END_IF

            END_IF


    FileWork.CREATE:    // шаг создания файла

        fbFileOpen(xExecute:=TRUE, sFileName:=sFileName, eFileMode:=FILE.MODE.MWRITE);

            IF fbFileOpen.xDone THEN
                hFile  :=  fbFileOpen.hFile;
                fbFileOpen(xExecute:=FALSE);

                // после создания файла можно перейти к шагу записи данных
                eState    :=  FileWork.WRITE;
            END_IF

            IF fbFileOpen.xError THEN
                // обработка ошибок
            END_IF


    FileWork.WRITE:   // шаг записи в буфер

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

            IF fbFileWrite.xDone THEN
                fbFileWrite(xExecute:=FALSE);
```

```
// теперь данные записаны в системный буфер; операционная система сама запишет их в файл...
// ...но мы можем сразу сделать это принудительно, чтобы гарантировать сохранность данных
            eState    :=  FileWork.FLUSH;
        END_IF

        IF fbFileWrite.xError THEN
            // обработка ошибок
        END_IF


    FileWork.FLUSH:   // шаг сброса буфера в файл

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

        IF fbFileFlush.xDone THEN
            fbFileFlush(xExecute:=FALSE);

            // теперь можно перейти к шагу закрытия файла
            eState    :=  FileWork.CLOSE;
        END_IF

        IF fbFileFlush.xError THEN
            // обработка ошибок
        END_IF


    FileWork.SET_READ_POS: // шаг установки позиции для чтения из файла

            fbFileSetPos(xExecute:=TRUE, hFile:=hFile,
        udiPos:=SIZEOF(stExportBinData)*(udiReadEntry-1));

        IF fbFileSetPos.xDone THEN
            fbFileSetPos(xExecute:=FALSE);

        // позиция для чтения выбрана, теперь можно перейти к шагу чтения данных
            eState    :=  FileWork.READ;
        END_IF

        IF fbFileSetPos.xError THEN
            // обработка ошибок
        END_IF


    FileWork.READ:    // шаг чтения данных

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

        IF fbFileRead.xDone THEN
            fbFileRead(xExecute:=FALSE);

            // теперь можно перейти к шагу закрытия файла
            eState    :=  FileWork.CLOSE;
        END_IF

        IF fbFileRead.xError THEN
            // обработка ошибок
        END_IF
```

```
    FileWork.CLOSE:  // шаг закрытия файла

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

            IF fbFileClose.xDone THEN
                fbFileClose(xExecute:=FALSE);

                IF xWBusy THEN
                    // после записи в файл узнаем его новый размер
                    eState    :=  FileWork.GET_SIZE;
                ELSE
        // после чтения из файла его размер не изменится, так что...
        // ...вернемся на шаг открытия файла для ожидания следующего управляющего сигнала
                    eState    :=  FileWork.OPEN;
                END_IF

                xWBusy  :=  FALSE;
                xRBusy  :=  FALSE;

            END_IF




    FileWork.GET_SIZE:  // шаг определения размера файла

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

            IF fbFileGetSize.xDone THEN

    // узнаем число записей в файле - оно равно отношению размера файла к размеру одной записи
                udiWriteEntry:=fbFileGetSize.szSize / SIZEOF(stExportBinData);
                fbFileGetSize(xExecute:=FALSE);

        // вернемся на шаг открытия файла для ожидания следующего управляющего сигнала
                eState    :=  FileWork.OPEN;
            END_IF

            // размер несуществующего файла...
            IF fbFileGetSize.eError=FILE.ERROR.NOT_EXIST THEN

                // очевидно, можно интерпретировать как ноль
                udiWriteEntry    :=  0;
                fbFileGetSize(xExecute:=FALSE);

        // вернемся на шаг открытия файла для ожидания следующего управляющего сигнала
                eState            :=  FileWork.OPEN;
            ELSIF  fbFileGetSize.xError THEN
                fbFileGetSize(xExecute:=FALSE);
                eState            :=  FileWork.OPEN;
            END_IF

END_CASE
```

A.5 Программа StringFileExample

```
// пример экспорта данных в текстовый файл

PROGRAM StringFileExample_PRG
VAR
    fbFileOpen:         FILE.OPEN;      // ФБ открытия файла
    fbFileClose:        FILE.CLOSE;     // ФБ закрытия файла
    fbFileWrite:        FILE.WRITE;     // ФБ записи в файл
    fbFileFlush:        FILE.FLUSH;     // ФБ сброса буфера в файл
    fbFileGetSize:      FILE.GetSize;   // ФБ получения размера файла

    hFile:          FILE.CAA.HANDLE;    // дескриптор открытого файла
    stExportData:       ArchData;       // структура экспортируемых данных

    // структура данных архива в виде строк
    asExportStringData:     ARRAY [0..10] OF STRING;

    sArchEntry:         STRING(255);        // строка, записываемая в архив
    xTitle:         BOOL;           // флаг "запись заголовка произведена"
    udiArchSize:        UDINT;          // размер архива в байтах
    uiArchEntry:        UINT;       // кол-во строк архива
    sFileName:          STRING;         // имя файла
    xWrite:             BOOL;           // сигнал записи в файл
    xWBusy:             BOOL;           // флаг "чтение из файла"

    // шаг операции с файлом
    eState:         FileWork :=  FileWork.GET_SIZE;

    sDevicePath:        STRING;         // путь к устройству
    iDevicePath:        INT;            // ID устройства

        // имя файла архива
    sVisuFileName:      STRING      :=  'test.csv';

    fbWriteTrig:        F_TRIG;         // триггер записи в файл

    fbGetCurrentDT:     DTU.GetDateAndTime;   // ФБ считывания системного времени
    fbSplitDT:          SPLIT_DT_TO_FSTRINGS; // ФБ конвертации времени в строку

        // метка времени в виде отдельных строковых разрядов
    asDateTimeStrings:      ARRAY [0..10] OF STRING;
        // метка времени в виде форматированной строки
    sTimeStamp:         STRING(20);
END_VAR

VAR CONSTANT
    // заголовок архива
    c_sTitle:  STRING(60):='Дата;Время;Значение типа WORD;Значение типа REAL;$N';
    c_sDelimiter:    STRING(1)  :=';';
END_VAR
```

```
// считываем системное время
fbGetCurrentDT(xExecute:=NOT(fbGetCurrentDT.xDone));

IF fbGetCurrentDT.xDone THEN
    // вырезаем отдельные разряды времени и конвертируем их в строки
    fbSplitDT(dtDateAndTime:=fbGetCurrentDT.dtDateAndTime);
END_IF

// подготавливаем метку времени в виде форматированной строки
asDateTimeStrings[0] :=  fbSplitDT.sDay;
asDateTimeStrings[1] :=  '.';
asDateTimeStrings[2] :=  fbSplitDT.sMonth;
asDateTimeStrings[3] :=  '.';
asDateTimeStrings[4] :=  fbSplitDT.sYear;
asDateTimeStrings[5] :=  c_sDelimiter;
asDateTimeStrings[6] :=  fbSplitDT.sHour;
asDateTimeStrings[7] :=  ':';
asDateTimeStrings[8] :=  fbSplitDT.sMinute;
asDateTimeStrings[9] :=  ':';
asDateTimeStrings[10]    :=  fbSplitDT.sSecond;

// собираем строку, которая будет записана в архив
asExportStringData[0] :=  CONCAT11(asDateTimeStrings);
asExportStringData[1] :=  c_sDelimiter;
asExportStringData[2] :=  WORD_TO_STRING(stExportData.wValue);
asExportStringData[3] :=  c_sDelimiter;
asExportStringData[4] :=  REAL_TO_FSTRING(stExportData.rValue,2);
asExportStringData[5] :=  c_sDelimiter;
asExportStringData[6] :=  '$N';

sArchEntry   :=  CONCAT11(asExportStringData);

// получаем путь к выбраенному устройству
sDevicePath:=  DEVICE_PATH(iDevicePath);

// склеиваем его с именем выбранного файла
sFileName   :=  CONCAT(sDevicePath, sVisuFileName);

// детектируем сигнал записи в файл
fbWriteTrig(CLK:=xWrite);

// если получен сигнал записи, то взводим соответствующий флаг
IF fbWriteTrig.Q THEN
    xWBusy  :=  TRUE;
END_IF


CASE eState OF

    FileWork.OPEN:    // шаг открытия файла

        IF xWBusy THEN
                        fbFileOpen(xExecute:=TRUE, sFileName:=sFileName,
                        eFileMode:=FILE.MODE.MAPPD);
        END_IF
```

```
// если файл, в который производится запись, не существует…
// …то создадим его и запишем в него заголовок архива
            IF fbFileOpen.eError=FILE.ERROR.NOT_EXIST THEN
                fbFileOpen(xExecute:=FALSE);
                eState    :=  FileWork.CREATE;
                xTitle  :=  TRUE;
            END_IF

        // если файл существует и был успешно открыт, то переходим к шагу записи в файл
            IF fbFileOpen.xDone THEN
                hFile   :=  fbFileOpen.hFile;
                fbFileOpen(xExecute:=FALSE);

                eState    :=  FileWork.WRITE;
            END_IF

    FileWork.CREATE:    // шаг создания файла

            // в созданном файле еще нет записей
            uiArchEntry:=0;


                fbFileOpen(xExecute:=TRUE, sFileName:=sFileName, eFileMode:=FILE.MODE.MWRITE);

            IF fbFileOpen.xDone THEN
                hFile   :=  fbFileOpen.hFile;
                fbFileOpen(xExecute:=FALSE);

                // после создания файла можно перейти к шагу записи данных
                eState    :=  FileWork.WRITE;
            END_IF

            IF fbFileOpen.xError THEN
                // обработка ошибок
            END_IF

    FileWork.WRITE:   // шаг записи в буфер

            // если это первая запись в файле - то перед ней запишем заголовок
            IF xTitle THEN
                sArchEntry   :=  CONCAT(c_sTitle, sArchEntry);

                // после первой записи заголовок записывать уже не нужно
                xTitle      :=  FALSE;
            END_IF

            // запись строки архива в файл
                fbFileWrite(xExecute:=TRUE, hFile:=hFile, pBuffer:=ADR(sArchEntry),
                szSize:=INT_TO_UDINT(LEN(sArchEntry)));

            IF fbFileWrite.xDone THEN
                fbFileWrite(xExecute:=FALSE);

                // после записи число строк в архиве увеличилось на одну
                uiArchEntry:=uiArchEntry+1;

                // теперь можно перейти к шагу сброса буфера в файл
                eState    :=  FileWork.FLUSH;
            END_IF

            IF fbFileWrite.xError THEN
                // обработка ошибок
            END_IF
```

```
    FileWork.FLUSH: // шаг сброса буфера в файл

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

            IF fbFileFlush.xDone THEN
                fbFileFlush(xExecute:=FALSE);

                // теперь можно перейти к шагу закрытия файла
                eState:=FileWork.CLOSE;
            END_IF

            IF fbFileFlush.xError THEN
                // обработка ошибок
            END_IF


    FileWork.CLOSE:  // шаг закрытия файла

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

            IF fbFileClose.xDone THEN
                fbFileClose(xExecute:=FALSE);
                xWBusy  :=  FALSE;

                // теперь можно перейти к шагу определения размера файла
                eState     :=  FileWork.GET_SIZE;
            END_IF


    FileWork.GET_SIZE:  // шаг определения размера файла

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

                // определяем размер файла
                IF fbFileGetSize.xDone THEN
                    udiArchSize:=fbFileGetSize.szSize;
                    fbFileGetSize(xExecute:=FALSE);

        // вернемся на шаг открытия файла для ожидания следующего управляющего сигнала
                    eState     :=  FileWork.OPEN;
                END_IF

            // размер несуществующего файла...
            IF fbFileGetSize.eError=FILE.ERROR.NOT_EXIST THEN

                // очевидно, можно интерпретировать как ноль
                udiArchSize :=  0;
                fbFileGetSize(xExecute:=FALSE);

        // вернемся на шаг открытия файла для ожидания следующего управляющего сигнала
                eState             :=  FileWork.OPEN;
            ELSIF  fbFileGetSize.xError THEN
                fbFileGetSize(xExecute:=FALSE);
                eState             :=  FileWork.OPEN;
            END_IF


END_CASE
```