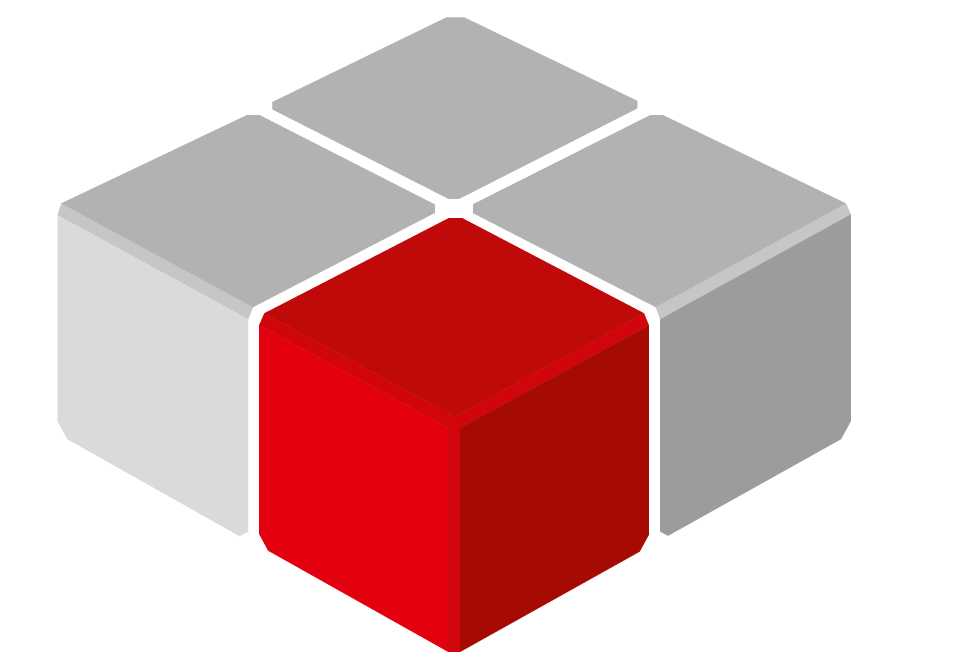


**CODESYS V3.5**

**Архивация**



Руководство пользователя

27.01.2019

версия 2.2

Оглавление

[Глоссарий 3](#_Toc31712886)

[1 Цель и структура документа 3](#_Toc31712887)

[2 Основные сведения о работе с файлами 4](#_Toc31712888)

[2.1 Общие сведения о памяти контроллеров 4](#_Toc31712889)

[2.2 Операции с файлами 5](#_Toc31712890)

[2.3 Требования к подключаемым накопителям (USB/SD) 5](#_Toc31712891)

[2.4 Пути к файлам и накопителям. Монтирование и демонтирование 6](#_Toc31712892)

[2.5 Ограничения на имена файлов и каталогов в ОС Linux 7](#_Toc31712893)

[2.6 Бинарные и текстовые файлы 8](#_Toc31712894)

[2.7 Обработка ошибок и некорректных ситуаций 9](#_Toc31712895)

[2.8 Подключение к файловой системе контроллера 10](#_Toc31712896)

[2.9 Работа с FTP 11](#_Toc31712897)

[3 Компонент OwenArchiver 12](#_Toc31712898)

[3.1 Установка компонента в CODESYS 12](#_Toc31712899)

[3.2 Добавление архиватора в проект 14](#_Toc31712900)

[3.3 Ограничения, связанные с использованием архиватора 18](#_Toc31712901)

[3.4 Пример работы с архиватором 19](#_Toc31712902)

[4 Библиотека CAA File 25](#_Toc31712903)

[4.1 Добавление библиотеки в проект CODESYS 25](#_Toc31712904)

[4.2 Структуры и перечисления 26](#_Toc31712905)

[4.2.1 Структура FILE.FILE\_DIR\_ENTRY 26](#_Toc31712906)

[4.2.2 Перечисление FILE.ERROR 26](#_Toc31712907)

[4.2.3 Перечисление FILE.MODE 27](#_Toc31712908)

[4.3 Пути к каталогам и файлам 27](#_Toc31712909)

[4.4 Ограничения при работе с файлами 27](#_Toc31712910)

[4.5 ФБ работы с каталогами 28](#_Toc31712911)

[4.5.1 ФБ FILE.DirCreate 28](#_Toc31712912)

[4.5.2 ФБ FILE.DirOpen 29](#_Toc31712913)

[4.5.3 ФБ FILE.DirList 30](#_Toc31712914)

[4.5.4 ФБ FILE.DirRemove 31](#_Toc31712915)

[4.5.5 ФБ FILE.DirRename 32](#_Toc31712916)

[4.5.6 ФБ FILE.DirClose 33](#_Toc31712917)

[4.5.7 ФБ FILE.DirCopy 34](#_Toc31712918)

[4.6 ФБ работы с файлами 35](#_Toc31712919)

[4.6.1 ФБ FILE.OPEN 35](#_Toc31712920)

[4.6.2 ФБ FILE.CLOSE 36](#_Toc31712921)

[4.6.3 ФБ FILE.WRITE 37](#_Toc31712922)

[4.6.4 ФБ FILE.READ 38](#_Toc31712923)

[4.6.5 ФБ FILE.RENAME 39](#_Toc31712924)

[4.6.6 ФБ FILE.COPY 40](#_Toc31712925)

[4.6.7 ФБ FILE.DELETE 41](#_Toc31712926)

[4.6.8 ФБ FILE.FLUSH 42](#_Toc31712927)

[4.6.9 ФБ FILE.GetPos 43](#_Toc31712928)

[4.6.10 ФБ FILE.SetPos 44](#_Toc31712929)

[4.6.11 ФБ FILE.EOF 45](#_Toc31712930)

[4.6.12 ФБ FILE.GetSize 46](#_Toc31712931)

[4.6.13 ФБ FILE.GetTime 47](#_Toc31712932)

[5 Пример работы с библиотекой CAA File 48](#_Toc31712933)

[5.1 Краткое описание примера 48](#_Toc31712934)

[5.2 Использованные библиотеки 48](#_Toc31712935)

[5.3 Содержимое примера 49](#_Toc31712936)

[5.4 Получение информации о накопителях (PLC\_PRG, действие act01\_DriveInfo) 50](#_Toc31712937)

[5.4.1 Объявление переменных 50](#_Toc31712938)

[5.4.2 Разработка программы 52](#_Toc31712939)

[5.4.3 Создание визуализации 56](#_Toc31712940)

[5.5 Работа с каталогами (PLC\_PRG, действие act02\_DirExample) 57](#_Toc31712941)

[5.5.1 Объявление переменных 57](#_Toc31712942)

[5.5.2 Разработка программы 58](#_Toc31712943)

[5.5.3 Создание визуализации 60](#_Toc31712944)

[5.5.4 Настройка элемента Комбинированное окно 61](#_Toc31712945)

[5.6 Просмотр содержимого каталогов (PLC\_PRG, действие act03\_DirList) 62](#_Toc31712946)

[5.6.1 Объявление переменных 62](#_Toc31712947)

[5.6.2 Разработка программы 64](#_Toc31712948)

[5.6.3 Создание визуализации 69](#_Toc31712949)

[5.7 Экспорт и импорт бинарных файлов (BinFileExample\_PRG) 70](#_Toc31712950)

[5.7.1 Объявление переменных 70](#_Toc31712951)

[5.7.2 Разработка программы 71](#_Toc31712952)

[5.7.3 Создание визуализации 77](#_Toc31712953)

[5.8 Экспорт текстовых файлов (StringFileExample\_PRG) 78](#_Toc31712954)

[5.8.1 Объявление переменных 78](#_Toc31712955)

[5.8.2 Разработка программы 80](#_Toc31712956)

[5.8.3 Создание визуализации 87](#_Toc31712957)

[5.9 Дополнительные операции с файлами (PLC\_PRG, действие act04\_ActionsWithFiles) 88](#_Toc31712958)

[5.9.1 Объявление переменных 88](#_Toc31712959)

[5.9.2 Разработка программы 88](#_Toc31712960)

[5.9.3 Создание визуализации 90](#_Toc31712961)

[5.10 Работа с примером 91](#_Toc31712962)

[5.11 Рекомендации и замечания 99](#_Toc31712963)

[Приложение А. Листинг примера 100](#_Toc31712964)

[А.1 Структуры и перечисления 100](#_Toc31712965)

[А.1.1. Структура ArchData 100](#_Toc31712966)

[А.1.2. Структура DriveInfo 100](#_Toc31712967)

[А.1.3. Перечисление FileDevice 101](#_Toc31712968)

[А.1.4. Перечисление FileDevice 101](#_Toc31712969)

[А.1.5. Структура VisuDirInfo 101](#_Toc31712970)

[А.2 Структуры и перечисления 102](#_Toc31712971)

[А.2.1. Функция BYTE\_SIZE\_TO\_WSTRING 102](#_Toc31712972)

[А.2.2. Функция DEVICE\_PATH 103](#_Toc31712973)

[А.2.3. ФБ DIR\_INFO 104](#_Toc31712974)

[А.2.4. Функция REAL\_TO\_FSTRING 106](#_Toc31712975)

[А.2.5. Функция REAL\_TO\_FWSTRING 106](#_Toc31712976)

[А.3 Программа PLC\_PRG 107](#_Toc31712977)

[А.3.1. Действие act01\_DriveInfo 109](#_Toc31712978)

[А.3.2. Действие act02\_DirExample 109](#_Toc31712979)

[А.3.3. Действие act03\_DirList 109](#_Toc31712980)

[А.3.4. Действие act04\_ActionsWithFiles 111](#_Toc31712981)

[А.4 Программа BinFileExample 112](#_Toc31712982)

[А.5 Программа StringFileExample 116](#_Toc31712983)

# Глоссарий

**ПЛК** – программируемый логический контроллер.

**ФБ** – функциональный блок.

# 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](#_2._Основные_сведения_1) приведена основная информация о работе с файлами.

В [п. 3](#_3._Библиотека_CAA_1) приведено описание компонента **OwenArchiver.**

В [п. 4](#_3._Библиотека_CAA) приведено описание библиотеки **CAA File.**

В [п. 5](#_4._Пример_опроса) 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. |

# Основные сведения о работе с файлами

## Общие сведения о памяти контроллеров

[Файл](https://ru.wikipedia.org/wiki/%D0%A4%D0%B0%D0%B9%D0%BB) – это именованная область памяти на носителе информации, используемая для хранения данных. Для упрощения работы с файлами используются [каталоги](https://ru.wikipedia.org/wiki/%D0%9A%D0%B0%D1%82%D0%B0%D0%BB%D0%BE%D0%B3_(%D1%84%D0%B0%D0%B9%D0%BB%D0%BE%D0%B2%D0%B0%D1%8F_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0)), которые позволяют разделять файлы по группам.

Способ организации, хранения и именования файлов на конкретном устройстве зависит от его [файловой системы](https://ru.wikipedia.org/wiki/%D0%A4%D0%B0%D0%B9%D0%BB%D0%BE%D0%B2%D0%B0%D1%8F_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0). Файловая система контроллеров ОВЕН ПЛК2хх – [UBIFS](https://ru.wikipedia.org/wiki/UBIFS), контроллеров ОВЕН СПК1хх – [Squashfs](https://ru.wikipedia.org/wiki/Squashfs) (область read-only) и [ext4](https://ru.wikipedia.org/wiki/Ext4) (пользовательская область).

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 **web**-**конфигураторе** и таргет-файле.

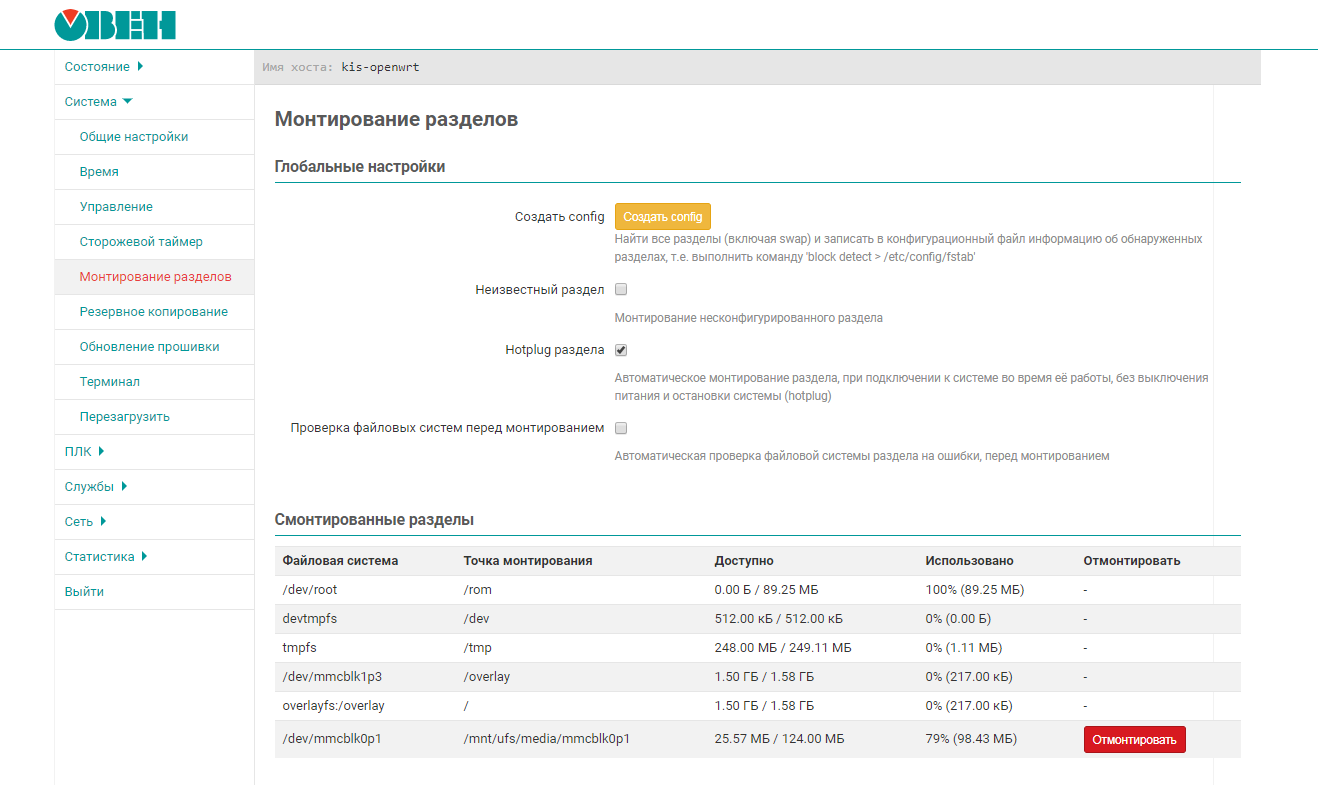


Рисунок 2.1 – Информация о памяти контроллера и накопителей в web-конфигураторе

## 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](https://ru.wikipedia.org/wiki/%D0%90%D1%81%D0%B8%D0%BD%D1%85%D1%80%D0%BE%D0%BD%D0%BD%D1%8B%D0%B9_%D0%B2%D0%B2%D0%BE%D0%B4-%D0%B2%D1%8B%D0%B2%D0%BE%D0%B4) – 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**.

## Требования к подключаемым накопителям (USB/SD)

1. Поддерживаемый стиль разделов – [MBR](https://ru.wikipedia.org/wiki/%D0%93%D0%BB%D0%B0%D0%B2%D0%BD%D0%B0%D1%8F_%D0%B7%D0%B0%D0%B3%D1%80%D1%83%D0%B7%D0%BE%D1%87%D0%BD%D0%B0%D1%8F_%D0%B7%D0%B0%D0%BF%D0%B8%D1%81%D1%8C) ([GPT](https://ru.wikipedia.org/wiki/%D0%A2%D0%B0%D0%B1%D0%BB%D0%B8%D1%86%D0%B0_%D1%80%D0%B0%D0%B7%D0%B4%D0%B5%D0%BB%D0%BE%D0%B2_GUID) не поддерживается). Методика определения стиля разделов доступна по [ссылке](http://remontka.pro/gpt-or-mbr-disk/).
2. Рекомендуется использовать накопители с одним [разделом](https://ru.wikipedia.org/wiki/%D0%A0%D0%B0%D0%B7%D0%B4%D0%B5%D0%BB_%D0%B4%D0%B8%D1%81%D0%BA%D0%B0) – тогда гарантируется монтирование по путям, указанным в [п. 2.4](#_2.4._Пути_к).
3. Поддерживаемые файловые системы – [FAT16/FAT32](https://ru.wikipedia.org/wiki/FAT), [NTFS](https://ru.wikipedia.org/wiki/NTFS) (read only), [ext4](https://ru.wikipedia.org/wiki/Ext4). Обновление прошивки/проекта возможно только при использовании накопителя с файловой системой [FAT16/FAT32](https://ru.wikipedia.org/wiki/FAT).
4. Перед началом работы рекомендуется отформатировать накопитель с помощью утилиты **HP USB Disk Storage Format Tool**.

## 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:

**Таблица 2.1 – Пути монтирования накопителей**

|  |  |  |
| --- | --- | --- |
| **Директория** | **СПК1хх [М01]** | **ПЛК210** |
| Рабочая директория | /home/root/CODESYS\_WRK | /root/CODESYS |
| USB-накопитель | /mnt/ufs/media/sda1 | /mnt/ufs/media/sda1 |
| SD-накопитель | /mnt/ufs/media/mmcblk0p1 | /mnt/ufs/media/mmcblk0p1 |
| Директория FTP-сервера | /mnt/ufs/home/ftp/in | /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:

**C:\ProgramData\CODESYS\CODESYSControlWinV3\<идентификатор\_контроллера>**

При работе с накопителями следует соблюдать два правила:

1. Перед работой с накопителем следует проверить, [смонтирован](https://ru.wikipedia.org/wiki/%D0%9C%D0%BE%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5_%D1%84%D0%B0%D0%B9%D0%BB%D0%BE%D0%B2%D0%BE%D0%B9_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D1%8B) (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.

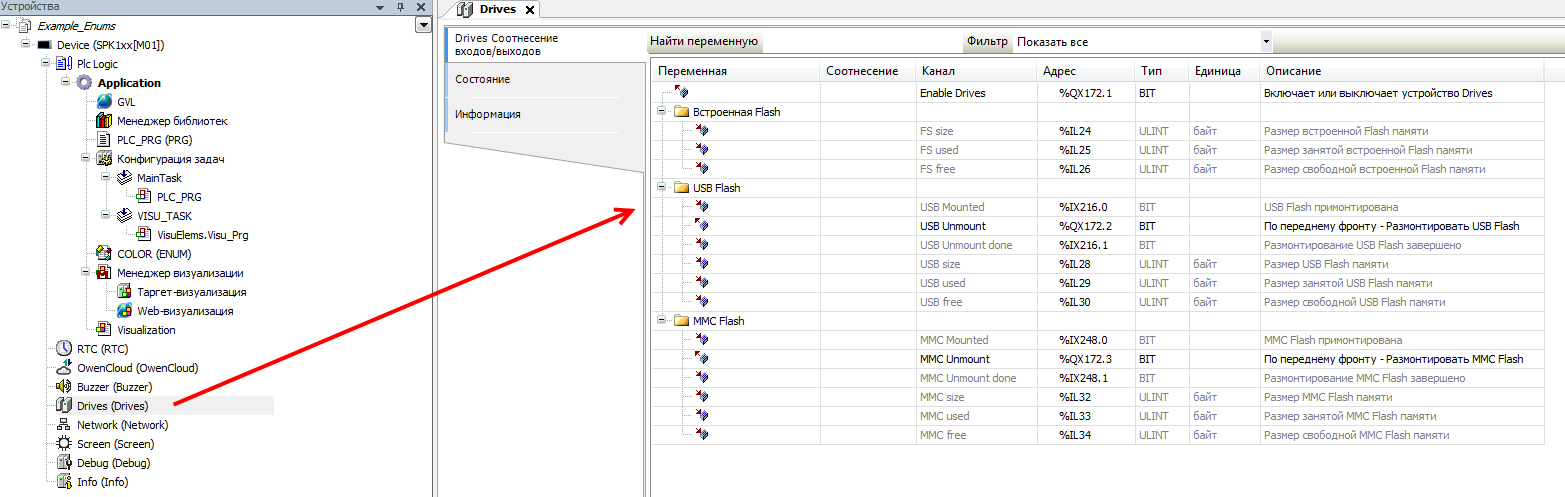


Рисунок 2.3 – Каналы узла Drives

**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]](#footnote-1) |
| FS used | ULINT | Количество занятой Flash-памяти контроллера в байтах1 |
| FS free | ULINT | Количество свободной Flash-памяти контроллера в байтах1 |
| **USB Flash** | | |
| USB Mounted | BOOL | Принимает значение **TRUE** после монтирования USB Flash накопителя, **FALSE** – при демонтировании |
| USB Unmount | BOOL | **TRUE** – демонтирование USB накопителя. Процедура демонтирования завершается в момент появления значения **TRUE** в канале **USB Unmount done**. До этого момента в канале **USB Unmount** должно сохраняться значение **TRUE** |
| USB Unmount done | BOOL | Принимает значение **TRUE** после демонтирования USB накопителя. Принимает значение **FALSE** по заднему фронту в канале **USB Unmount** |
| USB size | ULINT | Объем памяти USB накопителя в байтах |
| USB used | ULINT | Количество занятой памяти USB накопителя в байтах |
| USB free | ULINT | Количество свободной памяти USB накопителя в байтах |
| **MMC Flash** | | |
| MMC Mounted | BOOL | Принимает значение **TRUE** после монтирования MMC накопителя, **FALSE** – при демонтировании |
| MMC Unmount | BOOL | **TRUE** – демонтирование SD накопителя. Процедура демонтирования завершается в момент появления значения **TRUE** в канале **SD Unmount done**. До этого момента в канале **SD Unmount** должно сохраняться значение **TRUE** |
| MMC Unmount done | BOOL | Принимает значение **TRUE** после демонтирования MMC накопителя. Принимает значение **FALSE** по заднему фронту в канале **MMC Unmount** |
| MMC size | ULINT | Объем памяти MMC накопителя в байтах |
| MMC used | ULINT | Количество занятой памяти MMC накопителя в байтах |
| MMC free | ULINT | Количество свободной памяти MMC накопителя в байтах |

## 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

1. Регистр имеет принципиальное значение. **Test**.txt и **test.txt** – это два разных файла.

## 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](https://ru.wikipedia.org/wiki/%D0%9D%D0%B0%D0%B1%D0%BE%D1%80_%D1%81%D0%B8%D0%BC%D0%B2%D0%BE%D0%BB%D0%BE%D0%B2). Среда **CODESYS V3.5** включает два типа переменных, используемых для работы с символами (строками):

* **STRING** – использует 8-битную [ASCII](https://ru.wikipedia.org/wiki/ASCII)-based кодировку, зависящую от конкретного устройства, каждый символ занимает 1 байт;
* **WSTRING** – использует кодировку [Unicode](https://ru.wikipedia.org/wiki/%D0%AE%D0%BD%D0%B8%D0%BA%D0%BE%D0%B4) ([UCS2](http://www.unicode.org/faq/utf_bom.html#utf16-11)), каждый символ занимает 2 байта.

В **CODESYS** строки являются [нуль-терминированными](https://ru.wikipedia.org/wiki/%D0%9D%D1%83%D0%BB%D1%8C-%D1%82%D0%B5%D1%80%D0%BC%D0%B8%D0%BD%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%B0%D1%8F_%D1%81%D1%82%D1%80%D0%BE%D0%BA%D0%B0) – 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](https://ru.wikipedia.org/wiki/%D0%9C%D0%B0%D1%80%D0%BA%D0%B5%D1%80_%D0%BF%D0%BE%D1%81%D0%BB%D0%B5%D0%B4%D0%BE%D0%B2%D0%B0%D1%82%D0%B5%D0%BB%D1%8C%D0%BD%D0%BE%D1%81%D1%82%D0%B8_%D0%B1%D0%B0%D0%B9%D1%82%D0%BE%D0%B2).

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](http://foxtools.ru/ASCII#1252) (только для **STRING**) |

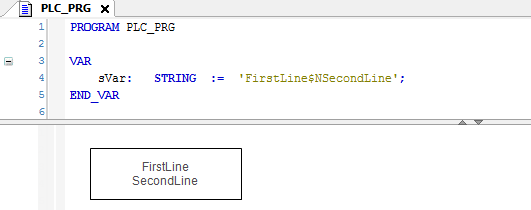


Рисунок 2.4 – Использование управляющих последовательностей

## 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.

## Подключение к файловой системе контроллера

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

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

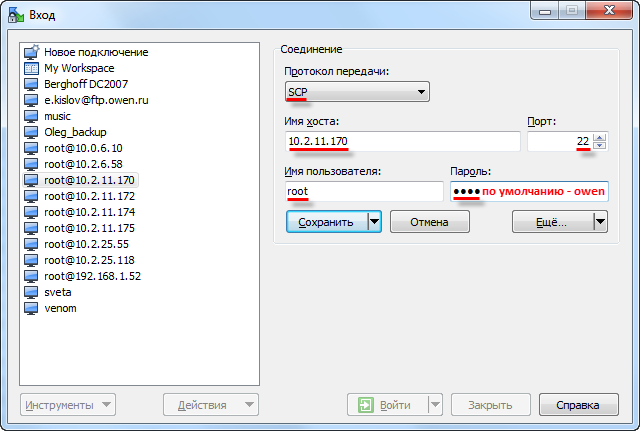


Рисунок 2.5 – Настройки соединения в WinSCP

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

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

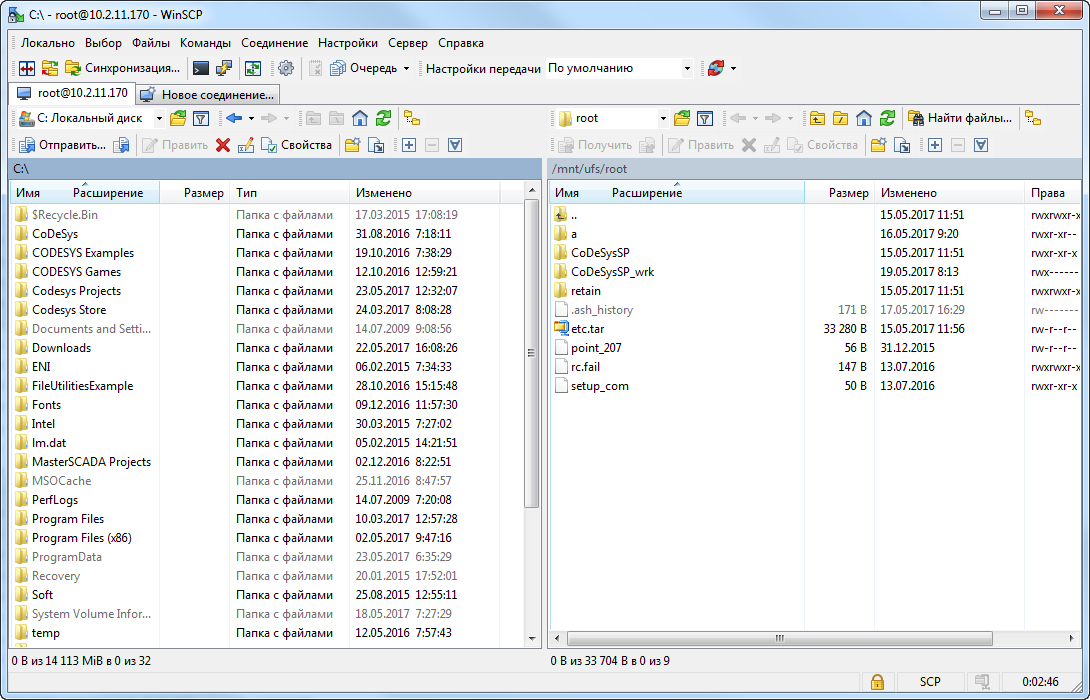


Рисунок 2.6 – Окно файлового менеджера WinSCP

## Работа с FTP

Контроллер может использоваться в режиме FTP-сервера. Логин для доступа: **ftp**, пароль по умолчанию: **ftp** (может быть изменен в web-конфигураторе). См. более подробную информацию в руководстве **CODESYS V3.5. FAQ**.

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

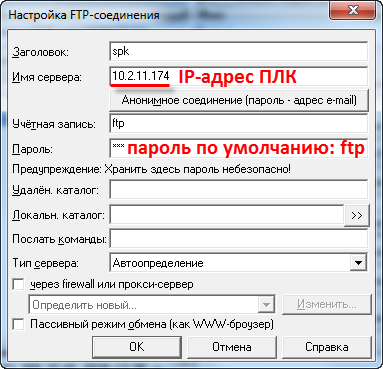


Рисунок 9.7 – Настройки FTP-соединения в файловом менеджере Total Commander

Для работы в режиме FTP-клиента следует использовать утилиту **cURL**. Пример доступен на сайте компании [ОВЕН](http://www.owen.ua/) в разделе **CODESYS V3/Примеры**.

# Компонент OwenArchiver

## 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 сайте компании [ОВЕН](http://www.owen.ua/) 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**.

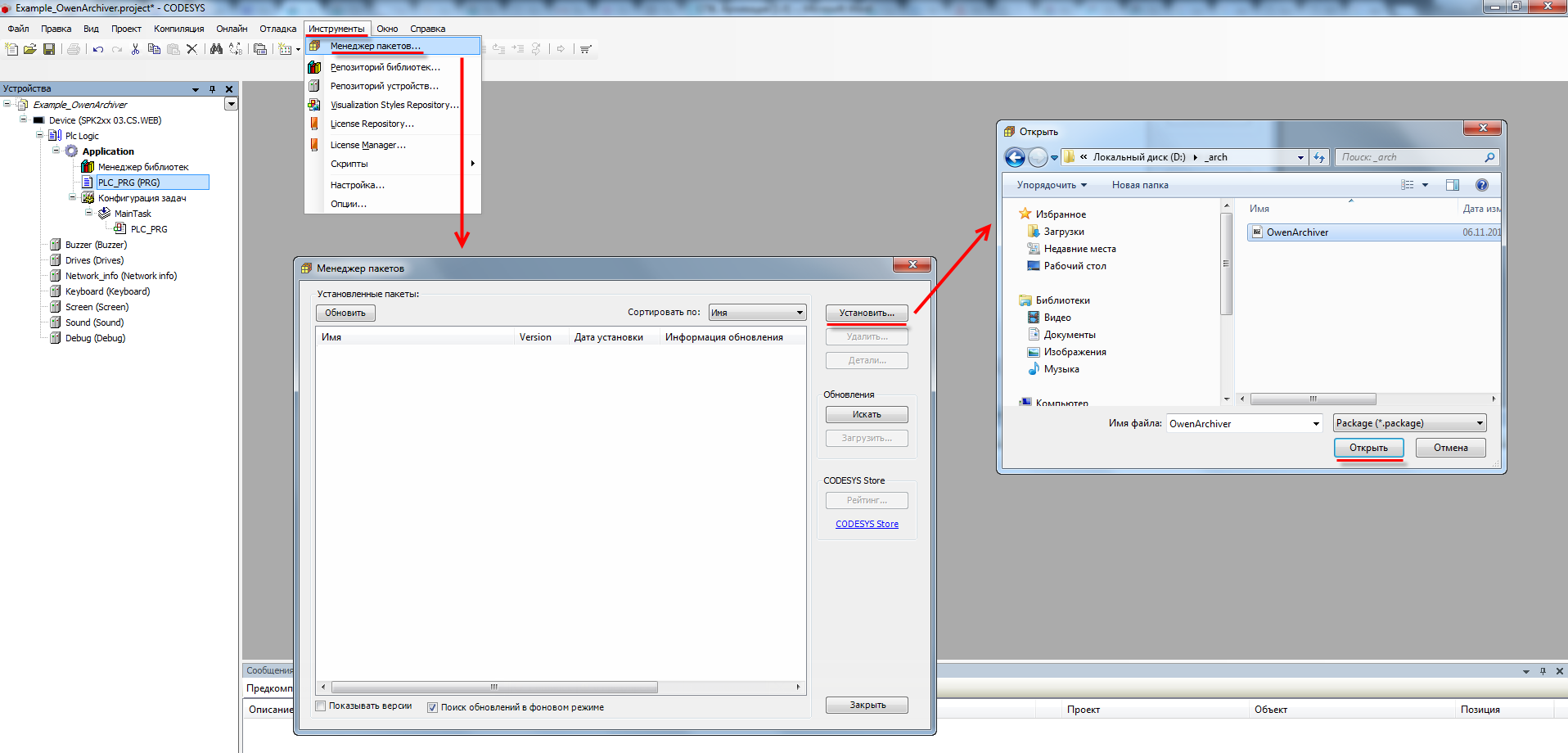


Рисунок 3.1 – Установка пакета OwenArchiver в среду CODESYS

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

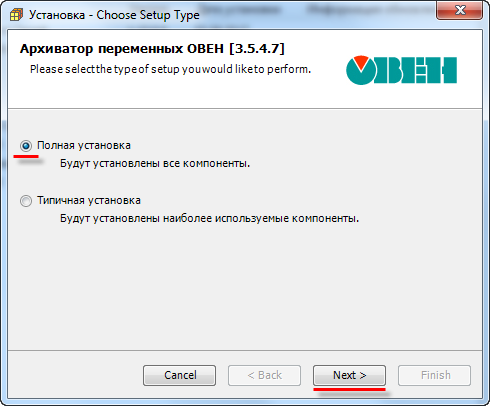


Рисунок 3.2 – Начало установки архиватора

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

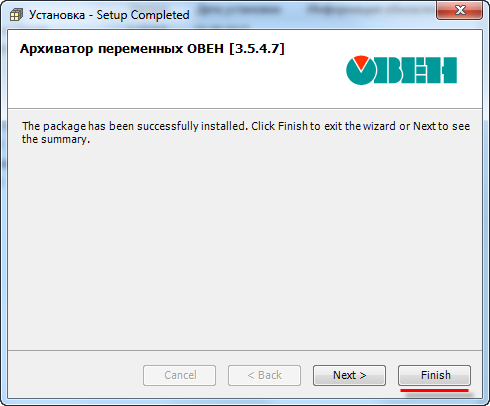


Рисунок 3.3 – Завершение установки архиватора

## 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**):

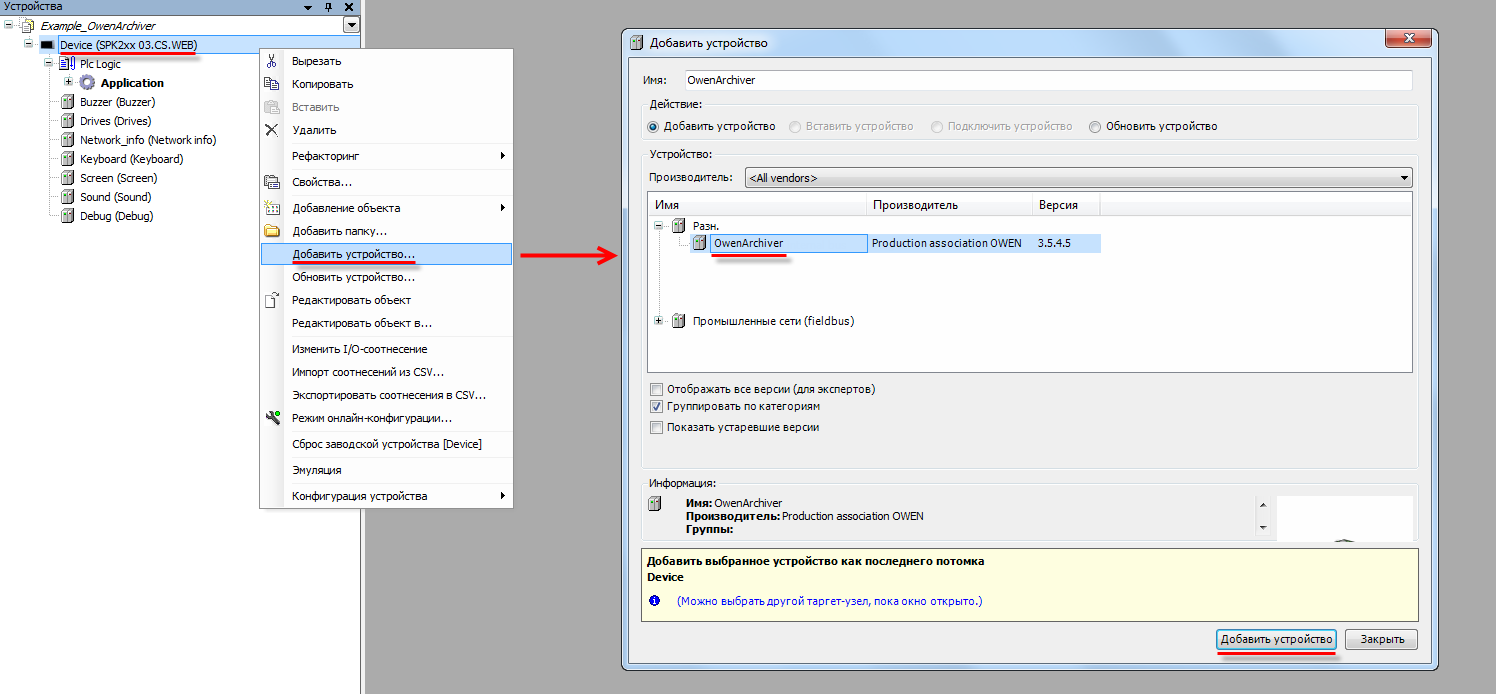


Рисунок 3.4 – Добавление архиватора в проект CODESYS

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

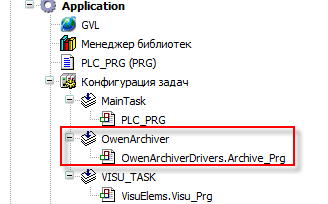


Рисунок 3.5 – Внешний вид дерева проекта после добавления архиватора

1. 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](#_2.3._Пути_к) в папку **PlcLogic/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](#_2.3._Пути_к));

* **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).

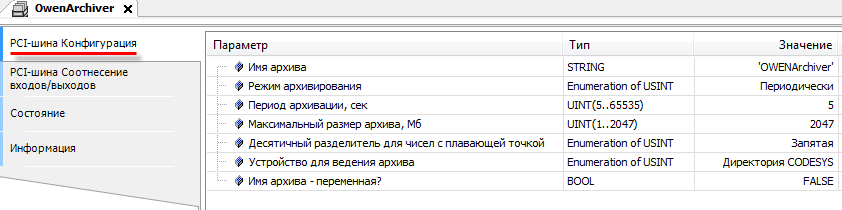


Рисунок 3.6 – Настройки архиватора, вкладка PCI-шина Конфигурация

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

**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](#_2.3._Пути_к), 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 |

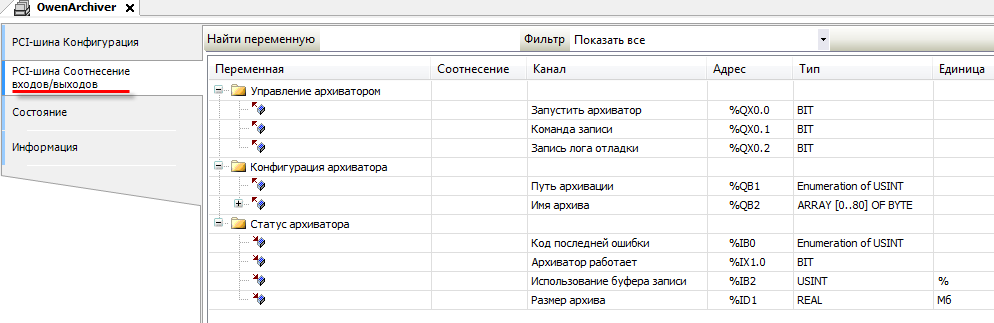


Рисунок 3.7 – Настройки архиватора, вкладка PCI-шина Соотнесение входов/выходов

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

**Непрерывный архив** – все данные будут записываться в один файл. По достижению его максимального размера (см. пп. 3) будет создан новый файл, а по достижению максимального размера нового файла – первый файл будет удален. Таким образом, фактически архив состоит из двух файлов – текущего (в который записываются данные) и предыдущего;

**Год/Месяц/День** – архив за каждые сутки будет записываться в отдельный файл (название – *номер дня*), файлы за каждый месяц будут сохранены в 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

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

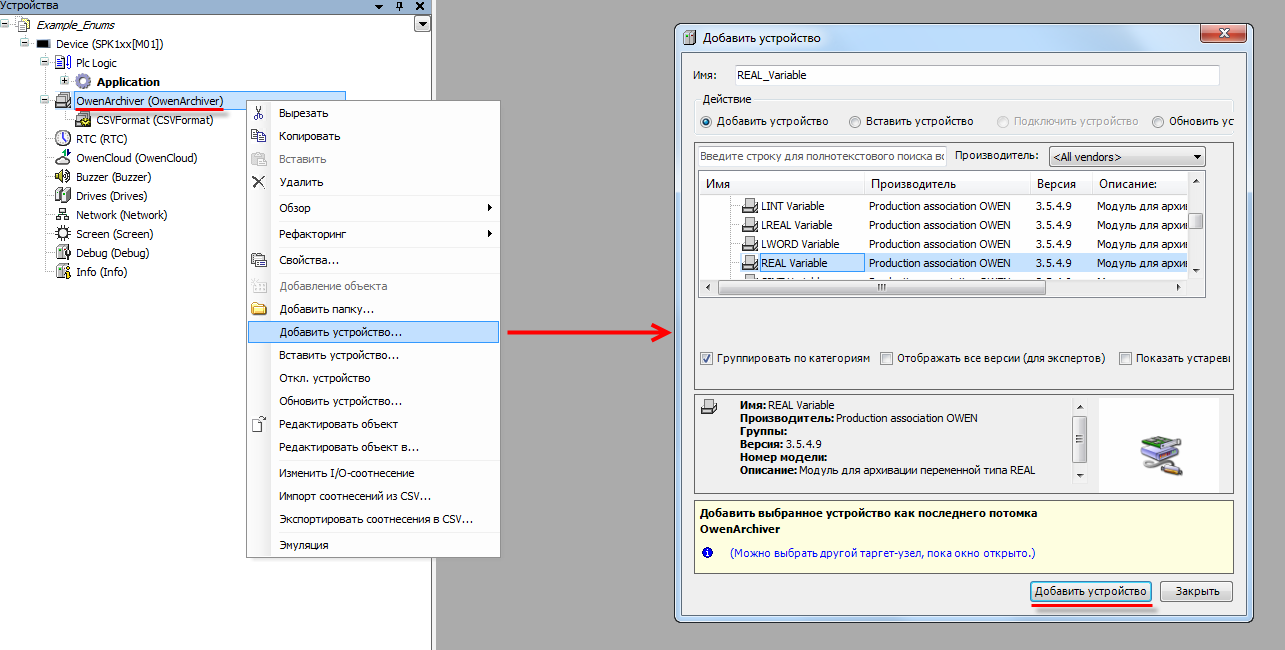


Рисунок 3.9 – Добавление каналов архивируемых переменных

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

**Описание переменной** – название, используемое при формировании заголовка архива;

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

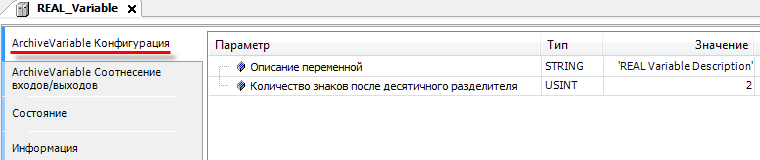


Рисунок 3.10 – Настройки канала архивируемой переменной

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

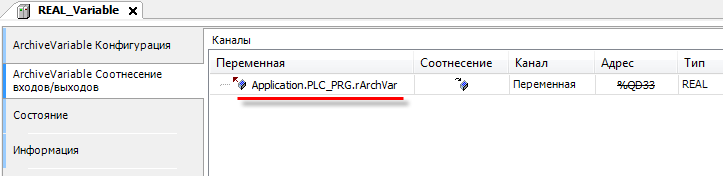


Рисунок 3.11 – Привязка архивируемой переменной к каналу

## 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)**].

## Пример работы с архиватором

Пример создан в среде **CODESYS V3.5 SP11 Patch 5** и подразумевает запуск на **СПК1хх [М01]** 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](https://ftp.owen.ru/CoDeSys3/21_Examples/01_3.5.11.5/05_Archives/Example_OwenArchiver_3511v1.projectarchive)

Расширенная версия примера: [Example\_OwenArchiverExtended.projectarchive](https://ftp.owen.ru/CoDeSys3/21_Examples/01_3.5.11.5/05_Archives/Example_OwenArchiver_Extended_3511v1.projectarchive)

Для работы с архиватором следует:

1. Объявить в программе **PLC\_PRG** следующие переменные:

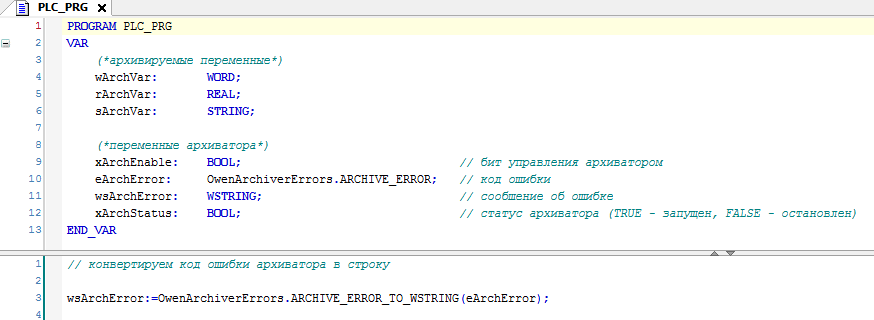


Рисунок 3.12 – Объявление переменных и код программы PLC\_PRG

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

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

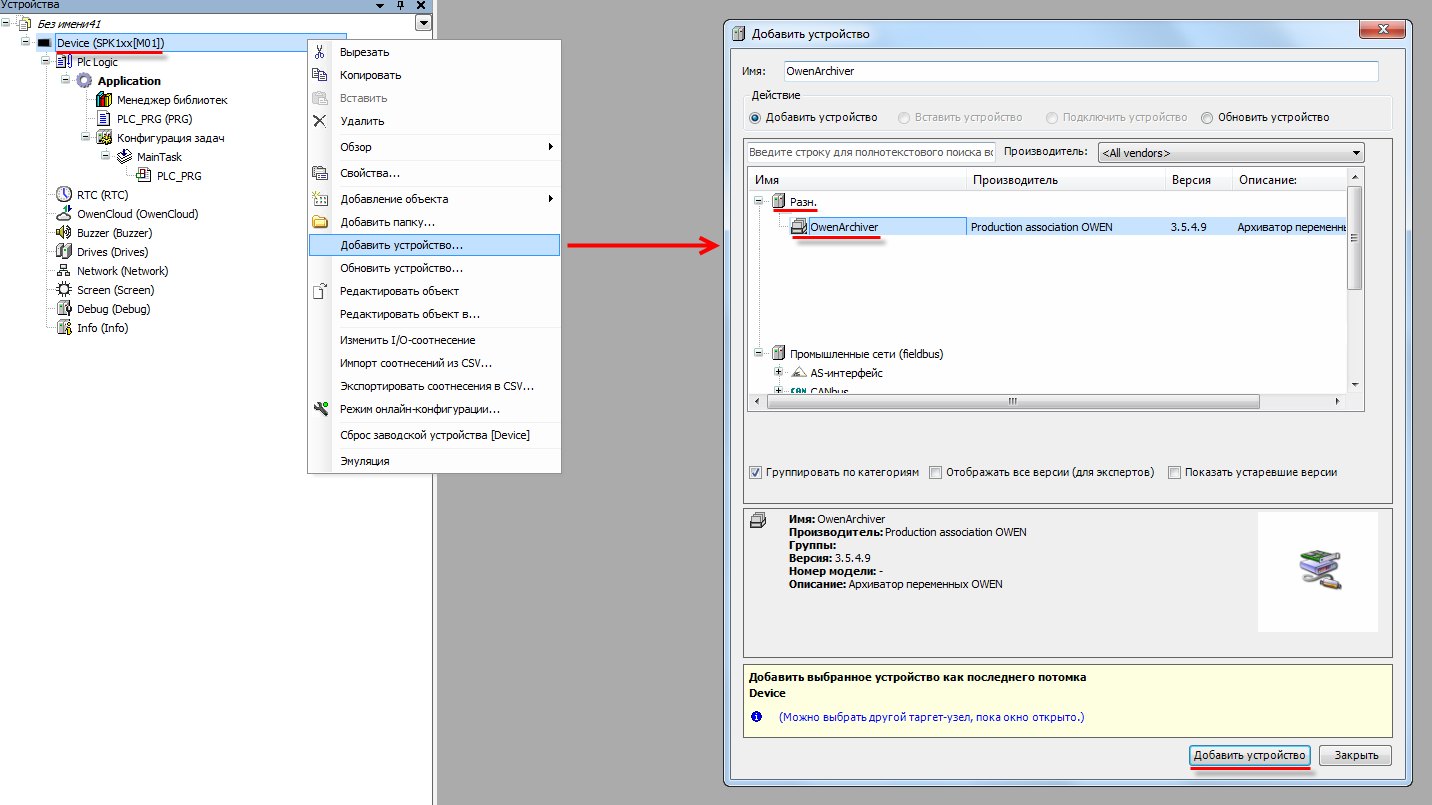


Рисунок 3.13 – Добавление компонента OwenArchiver

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

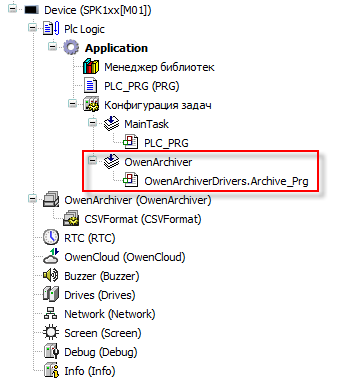


Рисунок 3.14 – Внешний вид дерева проекта после добавления архиватора

1. 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**.

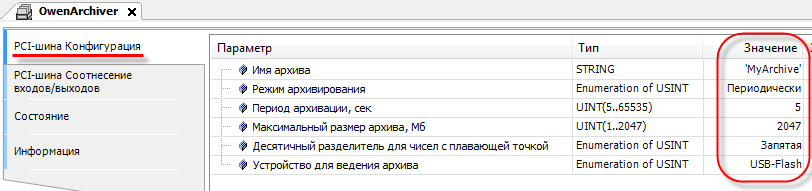


Рисунок 3.15 – Настройка параметров архивации

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

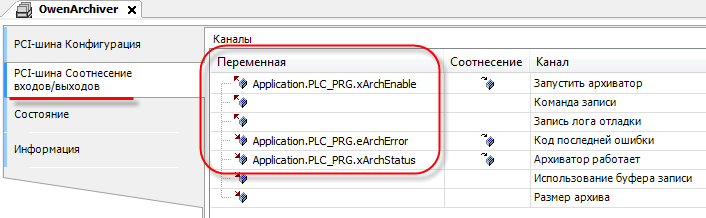


Рисунок 3.16 – Привязка переменных контроля архиватора

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

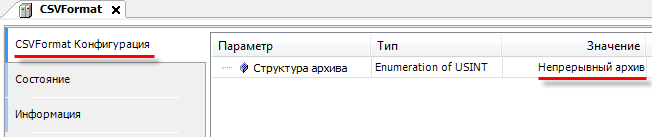


Рисунок 3.17 – Выбор режима архивации

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.

1. 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).

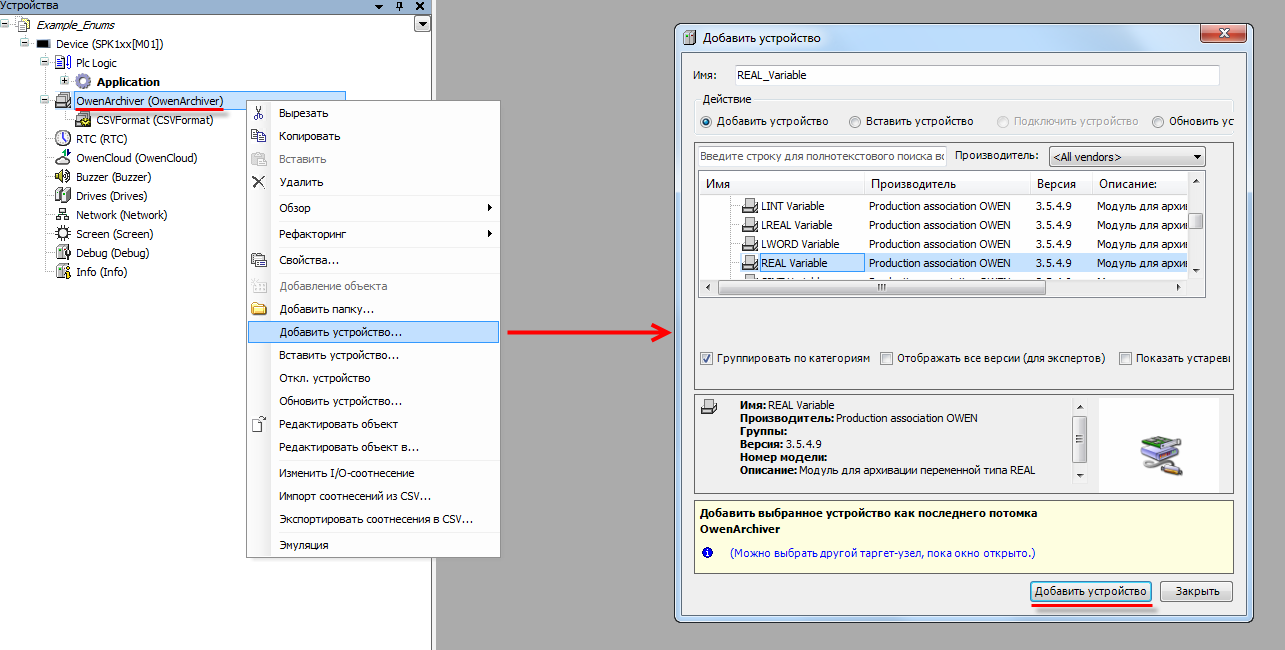


Рисунок 3.18 – Добавление каналов архивации

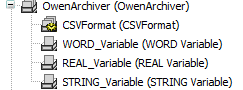


Рисунок 3.19 – Внешний вид дерева проекта после добавления каналов архивации

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.

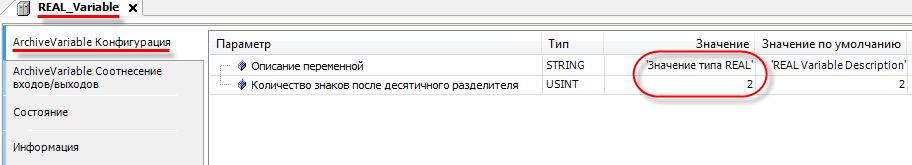


Рисунок 3.20 – Настройка канала архивации

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**.

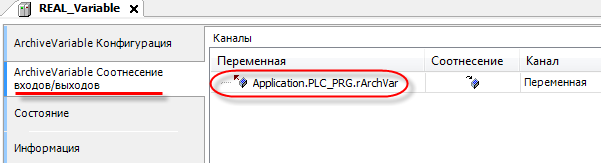


Рисунок 3.21 – Привязка переменных к каналам архивации

1. 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**.

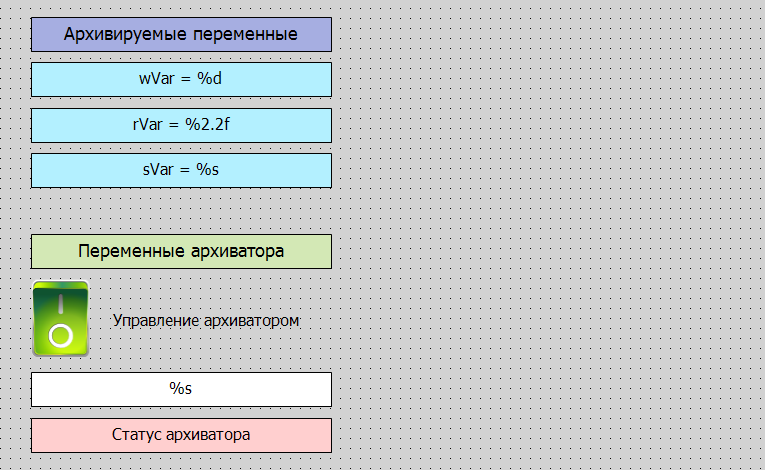


Рисунок 3.22 – Внешний вид экрана визуализации

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

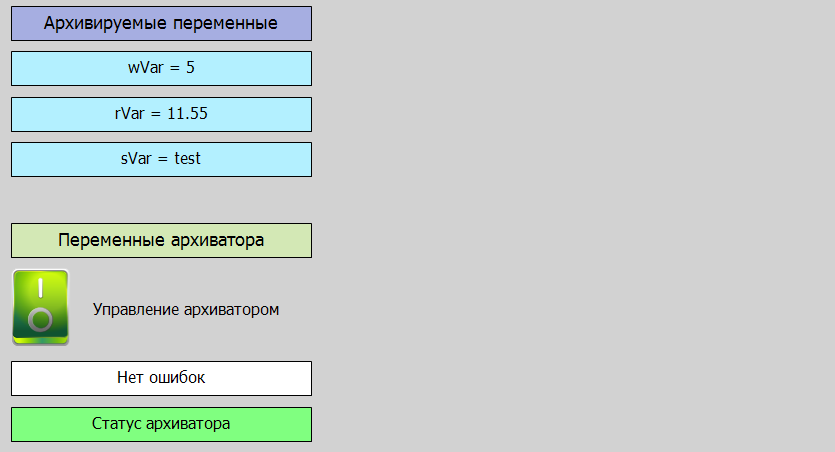


Рисунок 3.23 – Внешний вид экрана визуализации

1. В корневой папке USB-накопителя будет создан файл **MyArchive.csv**.

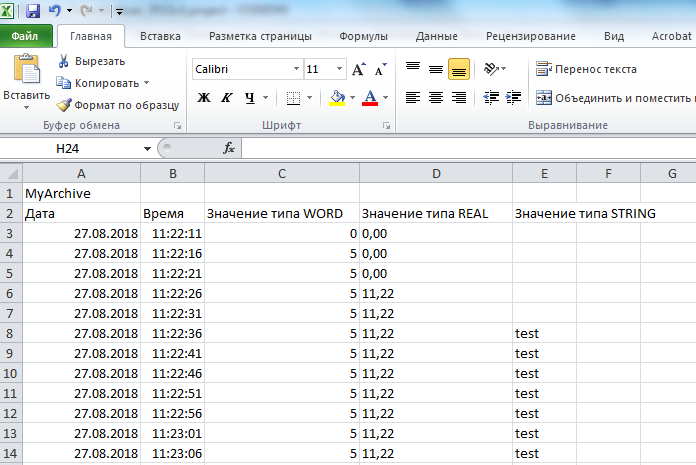


Рисунок 3.24 – Фрагмент архива

1. Рекомендуется ознакомиться с примером [получения информации о накопителях](#_5.4._Получение_информации) – это поможет определять свободный/занятый объем (и в случае необходимости останавливать архивацию), определять статус накопителя (смонтирован/демонтирован), демонтировать его и др.

# Библиотека CAA File

## Добавление библиотеки в проект CODESYS

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

Библиотека реализует [асинхронный доступ](https://ru.wikipedia.org/wiki/%D0%90%D1%81%D0%B8%D0%BD%D1%85%D1%80%D0%BE%D0%BD%D0%BD%D1%8B%D0%B9_%D0%B2%D0%B2%D0%BE%D0%B4-%D0%B2%D1%8B%D0%B2%D0%BE%D0%B4) 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. |

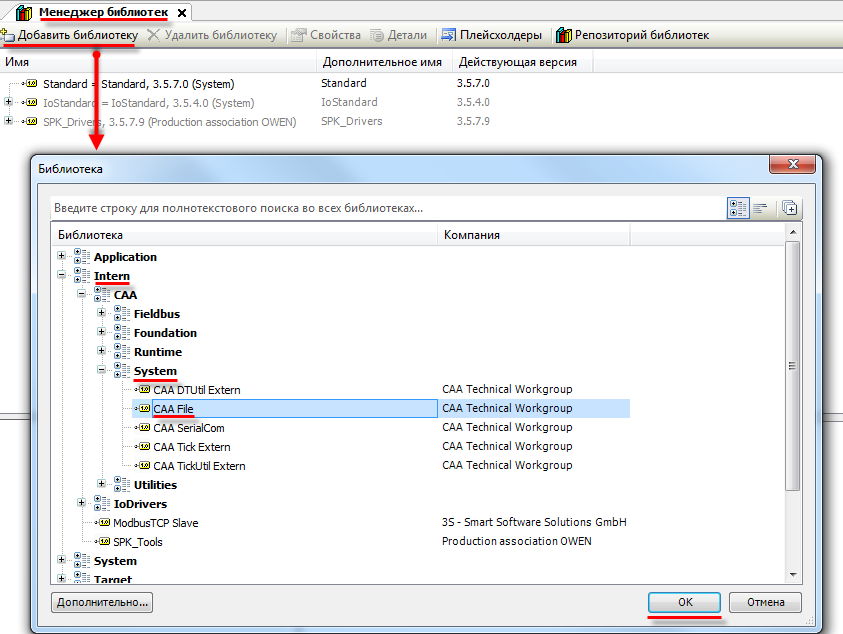


Рисунок 4.1 – Добавление библиотеки CAA File в проект CODESYS

|  |  |
| --- | --- |
|  | **ПРИМЕЧАНИЕ**  При объявлении экземпляров ФБ библиотеки следует перед их названием указывать префикс FILE**.** (пример: **FILE.OPEN**).. |

## Структуры и перечисления

### Структура FILE.FILE\_DIR\_ENTRY

Структура **FILE.FILE\_DIR\_ENTRY** описывает параметры каталога/файла и используется в случае работы с ФБ [FILE.DirList](#_3.4.3._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 **3.5.13.40** и ниже некорректно определяется дата и время последнего изменения каталогов (**CDS-68177**) |

### 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 |

### 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) |
| MREADPLUS[[2]](#footnote-2) | 4 | Чтение/запись c произвольной позиции (позиция определяются с помощью ФБ [FILE.SetPos](#_ФБ_FILE.SetPos)) |

## 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](#_2.3._Пути_к) и [п. 2.5](#_2.5._Ограничения_на) соответственно.

При работе с библиотекой можно указывать как относительные, так и абсолютные пути.

Пример: ФБ [File.DirCreate](#_3.4.1._ФБ_FILE.DirCreate)создает новый каталог по пути **sDirName**.

* Если sDirName=’test1’, то результатом работы ФБ является создание каталога **test1** в [рабочей директории](#_Пути_к_файлам);
* Если sDirName=’/mnt/ufs/home/test2’, то результатом работы ФБ является создание каталога **test2** в каталоге **/mnt/ufs/home/**.

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

## 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](#_4.6.1_ФБ_FILE.Open) возникает ошибка **5802.**

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

## ФБ работы с каталогами

### ФБ FILE.DirCreate

Функциональный блок **FILE.DirCreate** создает новый каталог. Без указания полного пути каталог создается внутри [рабочего каталога](#_2.4_Пути_к).

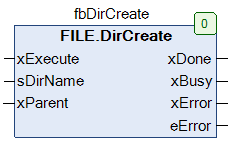


Рисунок 4.2 – Внешний вид ФБ FILE.DirCreate на языке CFC

**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](#_2.3._Пути_к), [п. 2.](#_2.4._Ограничения_на)5 и [п. 4.3](#_3.3._Пути_к) |
| **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](#_3.2.3._Перечисление_FILE.ERROR) | Статус работы ФБ (или имя ошибки) |

### ФБ FILE.DirOpen

Функциональный блок **FILE.DirOpen** открывает каталог и возвращает его дескриптор (**handle**), что требуется для последующего использования ФБ [File.DirList.](#_3.4.3._ФБ_FILE.DirList)

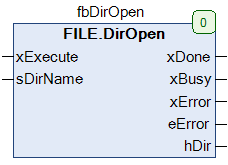


Рисунок 4.3 – Внешний вид ФБ FILE.DirOpen на языке CFC

**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](#_2.3._Пути_к), [п. 2.](#_2.4._Ограничения_на)5 и [п. 4.3](#_3.3._Пути_к) |
| **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](#_3.2.3._Перечисление_FILE.ERROR) | Статус работы ФБ (или имя ошибки) |
| **hDir** | FILE.CAA.HANDLE | Дескриптор открытого каталога |

### 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](#_3.4.2._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**.

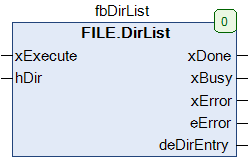


Рисунок 4.4 – Внешний вид ФБ FILE.DirList на языке CFC

**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](#_3.2.3._Перечисление_FILE.ERROR) | Статус работы ФБ (или имя ошибки) |
| **deDirEntry** | [FILE.FILE\_DIR\_ENTRY](#_3.2.1._Структура_FILE.FILE_DIR_ENTR) | Информация о каталоге/файле |

### ФБ FILE.DirRemove

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

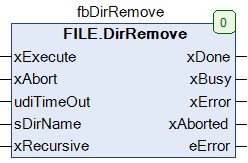


Рисунок 4.5 – Внешний вид ФБ FILE.DirRemove на языке CFC

**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](#_2.3._Пути_к), [п. 2.](#_2.4._Ограничения_на)5 и [п. 4.3](#_3.3._Пути_к) |
| **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](#_3.2.3._Перечисление_FILE.ERROR) | Статус работы ФБ (или имя ошибки) |

### ФБ FILE.DirRename

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

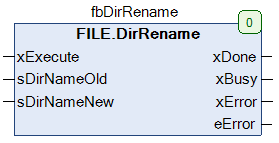


Рисунок 4.6 – Внешний вид ФБ FILE.DirRename на языке CFC

**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](#_2.3._Пути_к), [п. 2.](#_2.4._Ограничения_на)5 и [п. 4.3](#_3.3._Пути_к) |
| **sDirNameNew** | STRING | Новое имя (или полный путь) каталога. См. [п. 2.4](#_2.3._Пути_к), [п. 2.](#_2.4._Ограничения_на)5 и [п. 4.3](#_3.3._Пути_к) |
| **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](#_3.2.3._Перечисление_FILE.ERROR) | Статус работы ФБ (или имя ошибки) |

### ФБ FILE.DirClose

Функциональный блок **FILE.DirClose** закрывает каталог. Данная операция производится после считывания информации о каталоге с помощью [ФБ FILE.DirList](#_3.4.3._ФБ_FILE.DirList).

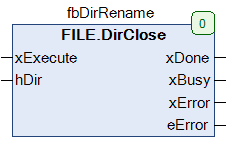


Рисунок 4.7 – Внешний вид ФБ FILE.DirClose на языке CFC

**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](#_3.2.3._Перечисление_FILE.ERROR) | Статус работы ФБ (или имя ошибки) |

### ФБ FILE.DirCopy[[3]](#footnote-3)

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

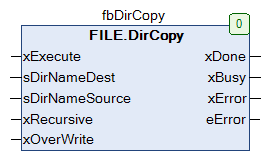


Рисунок 4.8 – Внешний вид ФБ FILE.DirCopy на языке CFC

**Таблица 4.7 – Описание входов и выходов ФБ FILE.DirCopy**

|  |  |  |
| --- | --- | --- |
| **Название** | **Тип данных** | **Описание** |
| **Входные переменные** | | |
| **xExecute** | BOOL | Переменная активации блока. Запуск блока происходит по переднему фронту переменной |
| **sDirNameDest** | STRING | Путь к каталогу назначения. См. [п. 2.4](#_2.3._Пути_к), [п. 2.](#_2.4._Ограничения_на)5 и [п. 4.3](#_3.3._Пути_к) |
| **sDirNameSource** | STRING | Путь к исходному каталогу. См. [п. 2.4](#_2.3._Пути_к), [п. 2.](#_2.4._Ограничения_на)5 и [п. 4.3](#_3.3._Пути_к) |
| **xRecursive** | BOOL | **TRUE** – вложенные каталоги исходного каталога копируются в каталог назначения, **FALSE** – вложенные каталоги не копируются |
| **xOverWrite** | BOOL | **TRUE** – содержимое каталога назначения перезаписывается (в случае совпадения имен файлов), **FALSE** – не перезаписывается |
| **Выходные переменные** | | |
| **xDone** | BOOL | Флаг успешного завершения работы блока |
| **xBusy** | BOOL | Флаг «ФБ в процессе работы» |
| **xError** | BOOL | Флаг ошибки. Принимает значение **TRUE** в случае возникновения ошибки |
| **eError** | [FILE.ERROR](#_3.2.3._Перечисление_FILE.ERROR) | Статус работы ФБ (или имя ошибки) |

## ФБ работы с файлами

### 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](#_3.5.2._ФБ_FILE.Close).

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

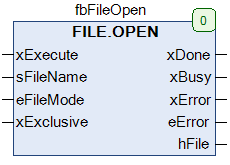


Рисунок 4.9 – Внешний вид ФБ FILE.OPEN на языке CFC

**Таблица 4.8 – 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](#_2.3._Пути_к), [п. 2.](#_2.4._Ограничения_на)5 и [п. 4.3](#_3.3._Пути_к) |
| **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](#_3.2.3._Перечисление_FILE.ERROR) | Статус работы ФБ (или имя ошибки) |
| **hFile** | FILE.CAA.HANDLE | Дескриптор открытого файла |

### 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. |

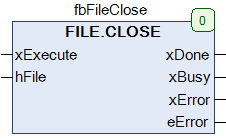


Рисунок 4.10 – Внешний вид ФБ FILE.Сlose на языке CFC

**Таблица 4.9 – 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](#_3.2.3._Перечисление_FILE.ERROR) | Статус работы ФБ (или имя ошибки) |

### ФБ FILE.WRITE

Функциональный блок **FILE.WRITE** используется для записи данных в файл (точнее – в системный буфер, см. также ФБ [FILE.FLUSH](#_3.6.8._ФБ_FILE.Flush)). Предварительно файл должен быть открыт с помощью [ФБ FILE.OPEN](#_3.5.1._ФБ_FILE.Open).

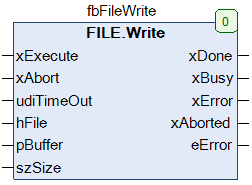


Рисунок 4.11 – Внешний вид ФБ FILE.WRITE на языке CFC

**Таблица 4.10 – 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](#_3.2.3._Перечисление_FILE.ERROR) | Статус работы ФБ (или имя ошибки) |

### ФБ FILE.READ

Функциональный блок **FILE.READ** используется для чтения данных из файла. Предварительно файл должен быть открыт с помощью ФБ [FILE.OPEN](#_3.5.1._ФБ_FILE.Open).

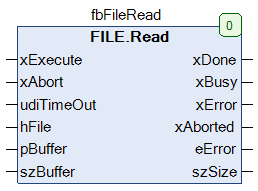


Рисунок 4.12 – Внешний вид ФБ FILE.READ на языке CFC

**Таблица 4.11 – 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](#_3.2.3._Перечисление_FILE.ERROR) | Статус работы ФБ (или имя ошибки) |
| **szSize** | CAA.SIZE | Размер считанных данных в байтах |

### ФБ FILE.RENAME

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

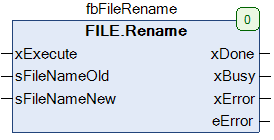


Рисунок 4.13 – Внешний вид ФБ FILE.RENAME на языке CFC

**Таблица 4.12 – 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](#_2.3._Пути_к), [п. 2.](#_2.4._Ограничения_на)5 и [п. 4.3](#_3.3._Пути_к) |
| **sFileNameNew** | STRING | Новое имя (или полный путь) файла. См. [п. 2.4](#_2.3._Пути_к), [п. 2.](#_2.4._Ограничения_на)5 и [п. 4.3](#_3.3._Пути_к) |
| **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](#_3.2.3._Перечисление_FILE.ERROR) | Статус работы ФБ (или имя ошибки) |

### ФБ FILE.COPY

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

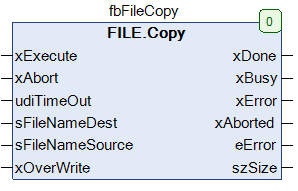


Рисунок 4.14 – Внешний вид ФБ FILE.COPY на языке CFC

**Таблица 4.13 – 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](#_2.3._Пути_к), [п. 2.](#_2.4._Ограничения_на)5 и [п. 4.3](#_3.3._Пути_к) |
| **sFileNameSource** | STRING | Имя (или полный путь) исходного файла. См. [п. 2.4](#_2.3._Пути_к),[п. 2.](#_2.4._Ограничения_на)5 и [п. 4.3](#_3.3._Пути_к) |
| **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](#_3.2.3._Перечисление_FILE.ERROR) | Статус работы ФБ (или имя ошибки) |
| **szSize** | CAA.FILE.SIZE | Размер скопированных данных в байтах |

### ФБ FILE.DELETE

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

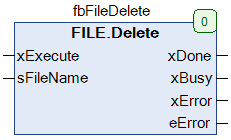


Рисунок 4.15 – Внешний вид ФБ FILE.DELETE на языке CFC

**Таблица 4.14 – 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](#_3.2.3._Перечисление_FILE.ERROR) | Статус работы ФБ (или имя ошибки) |

### ФБ FILE.FLUSH

Функциональный блок **FILE.FLUSH** используется для принудительной записи данных из системного буфера в файл. При работе ФБ [FILE.WRITE](#_3.5.3._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.

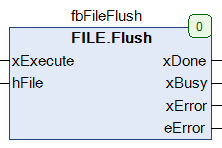


Рисунок 4.16 – Внешний вид ФБ FILE.FLUSH на языке CFC

**Таблица 4.15 – 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](#_3.2.3._Перечисление_FILE.ERROR) | Статус работы ФБ (или имя ошибки) |

### 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.

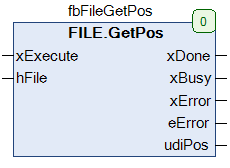


Рисунок 4.17 – Внешний вид ФБ FILE.GetPos на языке CFC

**Таблица 4.16 – 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](#_3.2.3._Перечисление_FILE.ERROR) | Статус работы ФБ (или имя ошибки) |
| **udiPos** | UDINT | Текущая установленная позиция в файле (смещение относительно начала файла в байтах) |

### 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.

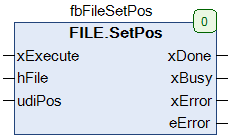


Рисунок 4.18 – Внешний вид ФБ FILE.SetPos на языке CFC

**Таблица 4.17 – 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](#_3.2.3._Перечисление_FILE.ERROR) | Статус работы ФБ (или имя ошибки) |

### 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.

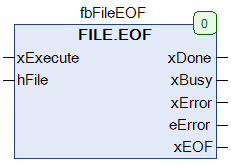


Рисунок 4.19 – Внешний вид ФБ FILE.EOF на языке CFC

**Таблица 4.18 – 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](#_3.2.3._Перечисление_FILE.ERROR) | Статус работы ФБ (или имя ошибки) |
| **xEOF** | BOOL | **TRUE** – достигнут конец файл  **FALSE** – конец файла не достигнут |

### ФБ FILE.GetSize

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

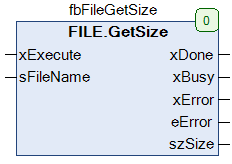


Рисунок 4.20 – Внешний вид ФБ FILE.GetSize на языке CFC

**Таблица 4.19 – 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](#_3.2.3._Перечисление_FILE.ERROR) | Статус работы ФБ (или имя ошибки) |
| **szSize** | FILE.CAA.SIZE | Размер файла в байтах |

### ФБ FILE.GetTime

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

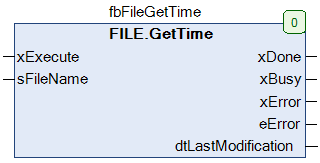


Рисунок 4.21 – Внешний вид ФБ FILE.GetTime на языке CFC

**Таблица 4.20 – 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](#_2.3._Пути_к), [п. 2.](#_2.4._Ограничения_на)5 и [п. 4.3](#_3.3._Пути_к) |
| **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](#_3.2.3._Перечисление_FILE.ERROR) | Статус работы ФБ (или имя ошибки) |
| **dtLastModification** | DT | Дата и время последнего изменения файла |

# Пример работы с библиотекой CAA File

## 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](#_Приложение._Листинг_примера).

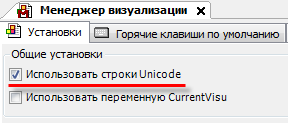
**Таблица 5.1 – Структура примера**

|  |  |  |
| --- | --- | --- |
| **№ 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 SP14 Patch 3** и подразумевает запуск на **СПК1хх [М01]** с таргет-файлом **3.5.14.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\_3514v1.projectarchive](https://ftp.owen.ru/CoDeSys3/21_Examples/01_3.5.11.5/05_Archives/Example_CAA_File_3514v1.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**.



## Использованные библиотеки

Для создания примера были использованы следующие библиотеки:

* **CAA File** (3.5.13.40) – для работы с файлами;
* **Standard64** (3.5.13.0) – для работы со строками типа **WSTRING**;
* **CAA Memory** (3.5.12.0) **–** для работы с памятью;
* **OwenStringUtils** (3.5.4.7) – для конвертации переменных типа **DT** в форматированную строку.

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

## 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 |
| 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 |
| REAL\_TO\_FSTRING | StringFileExample\_PRG | Конвертация REAL в формат. строку |
| REAL\_TO\_FWSTRING | PLC\_PRG (act01) | Конвертация REAL в формат. строку |

## 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](#_2.3._Пути_к).

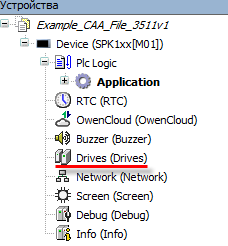


Рисунок 5.4.1 – Узел Drives в дереве проекта

### Объявление переменных

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

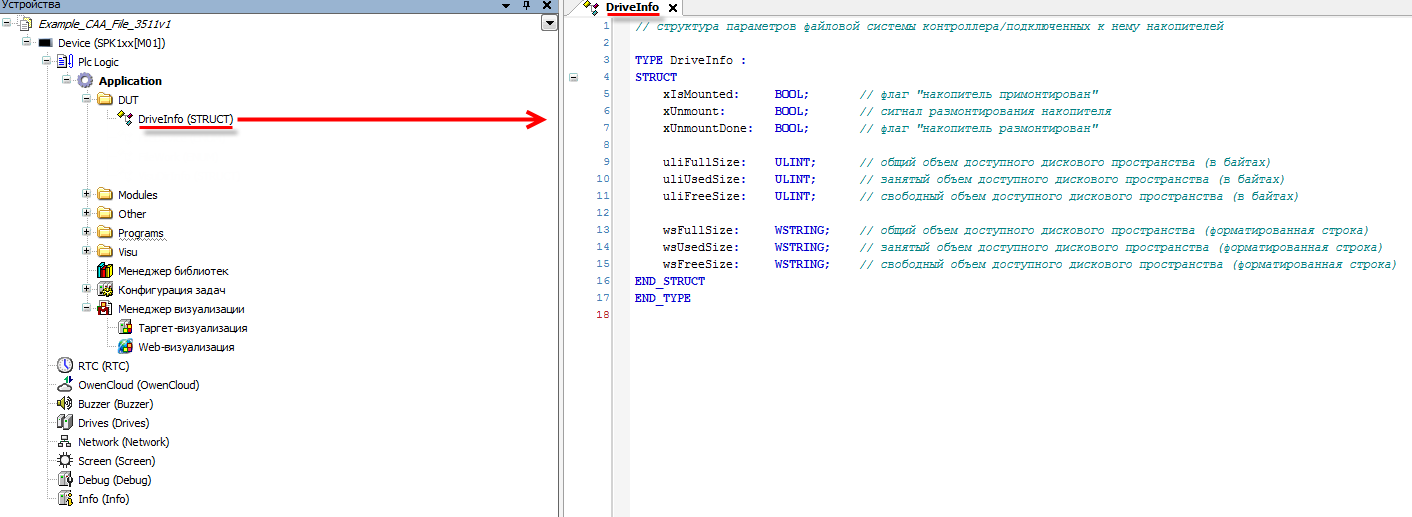


Рисунок 5.4.2 – Объявление структуры DriveInfo

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).

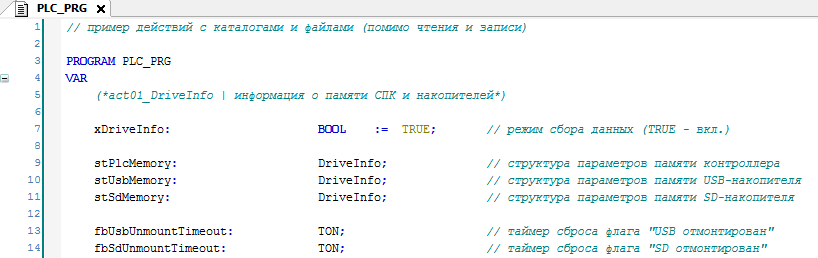


Рисунок 5.4.3 – Объявление переменных в программе PLC\_PRG

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

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

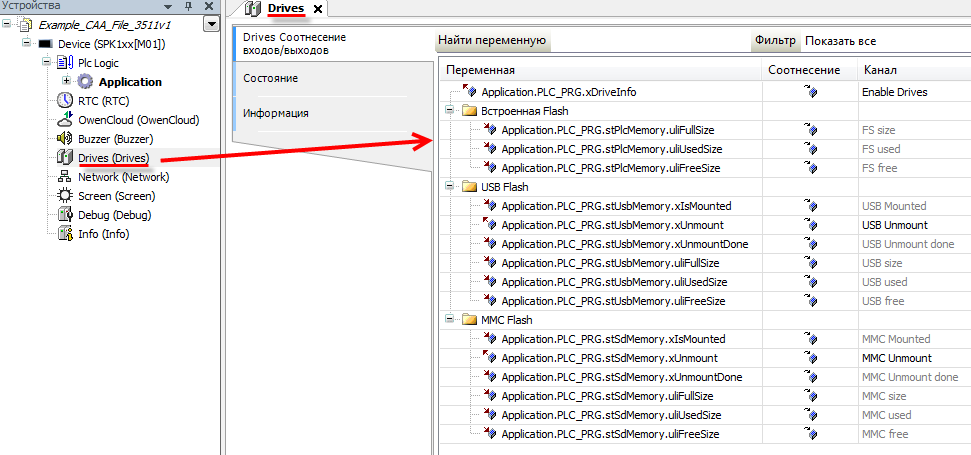


Рисунок 5.4.4 – Привязка переменных к каналам узла Drives

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

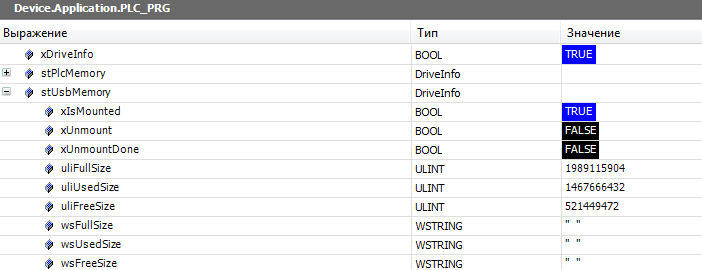


Рисунок 5.4.5 – Значение переменных, привязанных к каналам узла Drives, во время работы проекта

### 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:

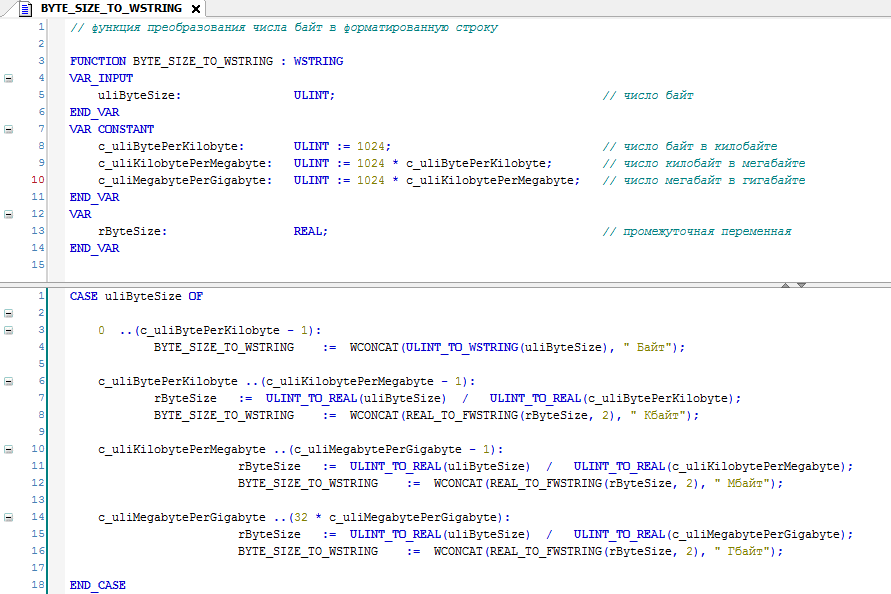


Рисунок 5.4.6 – Код функции BYTE\_SIZE\_TO\_WSTRING

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:

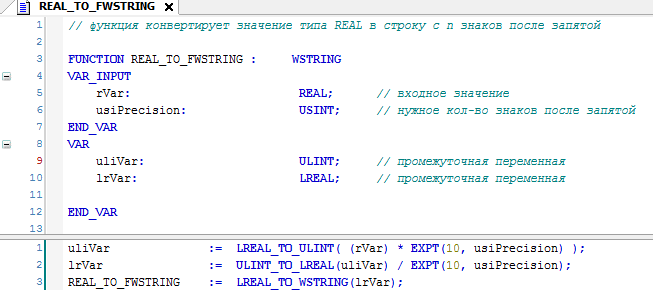


Рисунок 5.4.7 – Код функции REAL\_TO\_FWSTRING

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 **102**), 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 **102**), 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** будет активен в течение нескольких циклов контроллера – накопитель будет успешно демонтирован, но флаг демонтирования примет значение **TRUE** на очень короткое время – 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**:

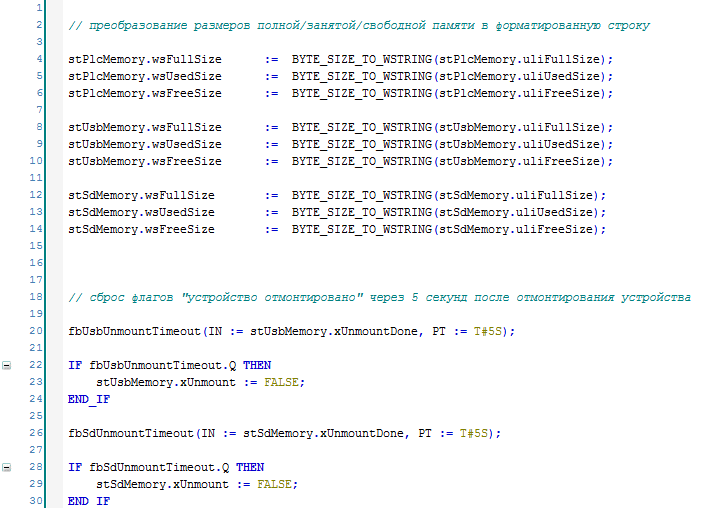


Рисунок 5.4.8 – Код операций с переменными вкладки Drives

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.

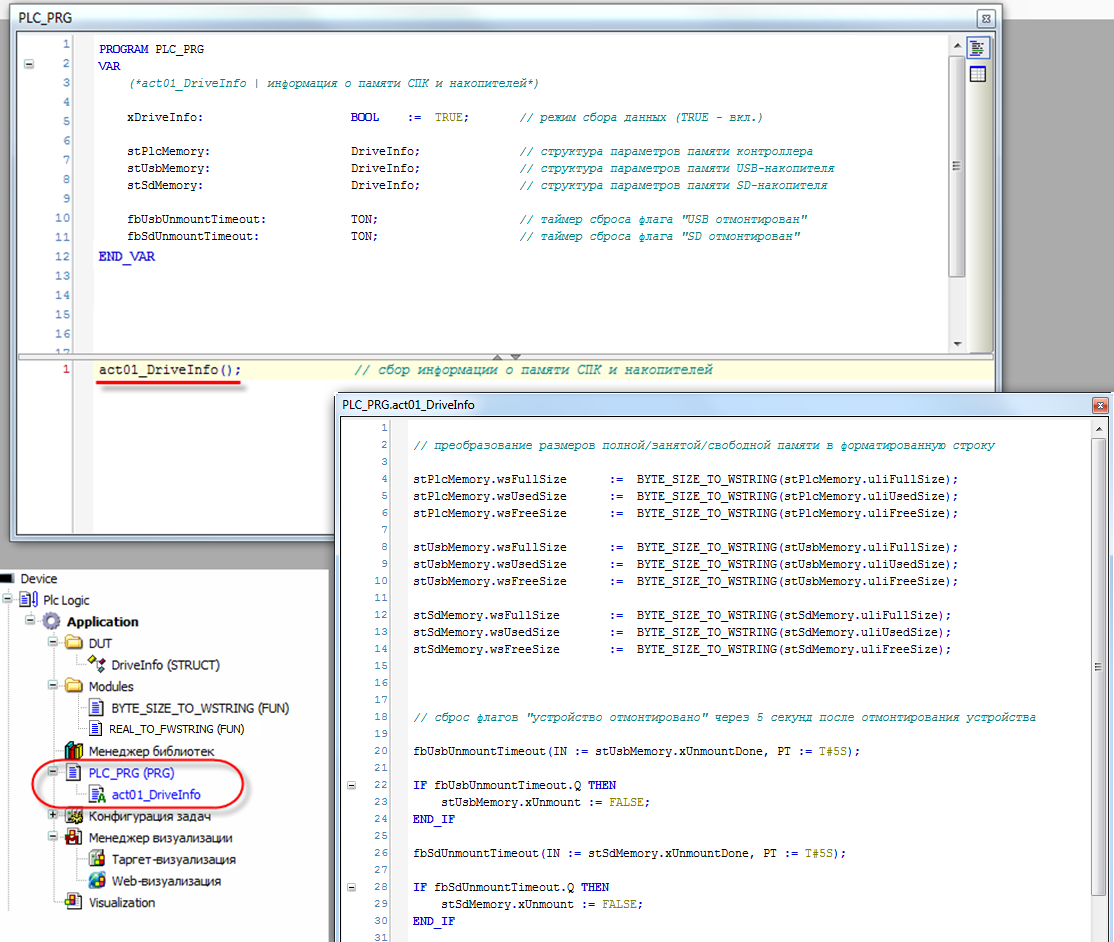


Рисунок 5.4.9 – Код действия act01\_DriveInfo и его вызов в программе PLC\_PRG

### 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](#_4.10._Работа_с).



Рисунок 5.4.10 – Внешний вид экрана Visu01\_DriveInfo

## 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.

### 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**):

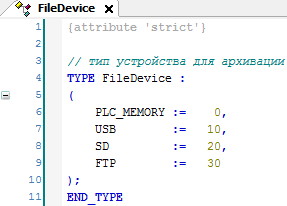


Рисунок 5.5.1 – Объявление перечисления FileDevice

В программе **PLC\_PRG** следует объявить следующие переменные:

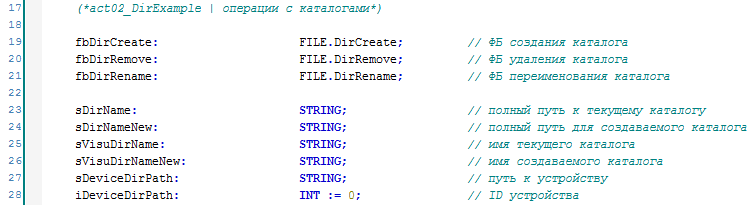


Рисунок 5.5.2 – Объявление переменных в программе PLC\_PRG

### 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](#_2.3._Пути_к). ***Обратите внимание***, что для других ПЛК пути могут отличаться. Для рабочей директории путь можно не указывать.

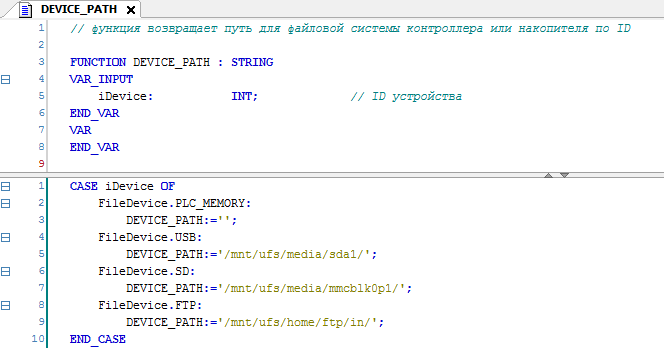


Рисунок 5.5.3 – Код функции DEVICE\_PATH

Оператор должен выбрать ID устройства (например, через элемент **Комбинированное окно/Сombobox**), 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:

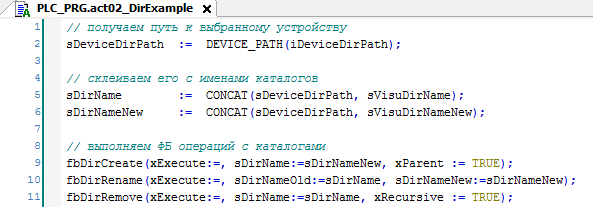


Рисунок 5.5.4 – Код действия act02\_DirExample

В программе **PLC\_PRG** следует добавить вызов данного действия:

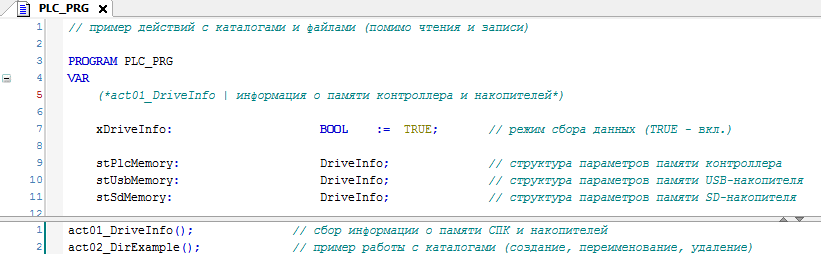


Рисунок 5.5.5 – Вызов действия act02\_DirExample в программе PLC\_PRG

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. |

### 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](#_4.5.4._Настройка_элемента);
* 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**.

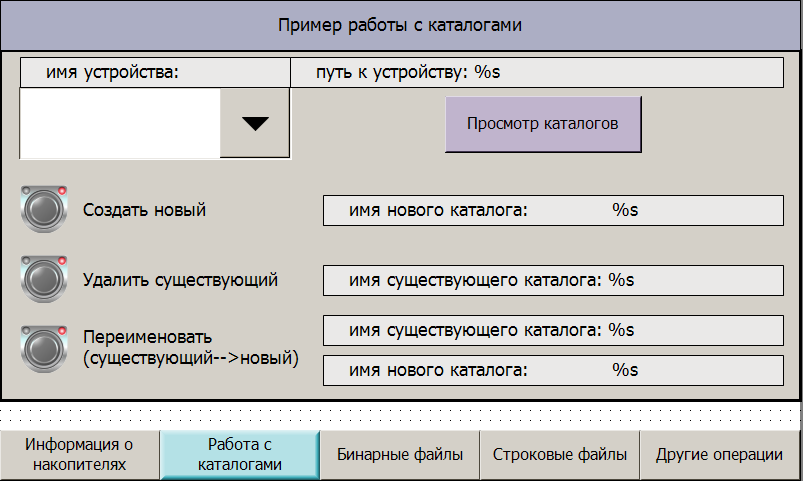


Рисунок 5.5.6. Внешний вид экрана Visu02\_DirExample

Визуализация также содержит кнопки переключения экранов (описание других экранов проекта приведено в соответствующих пунктах). Пример работы с экраном приведен в [п. 5.10](#_4.10._Работа_с).

### 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:

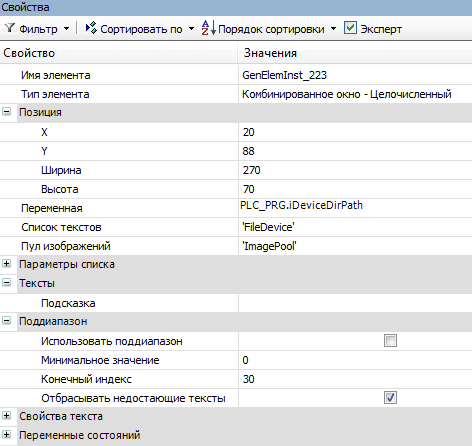


Рисунок 5.5.7 – Настройки элемента Комбинированное окно – Целочисленный

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

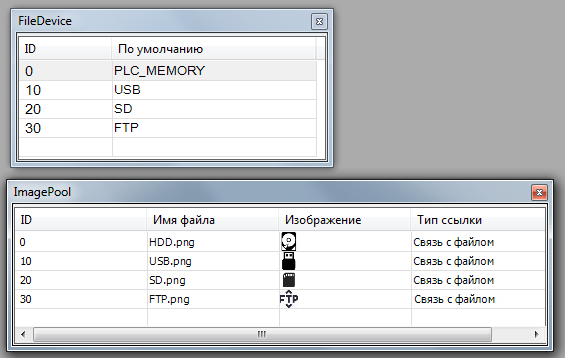


Рисунок 5.5.8 – Содержимое компонентов Список Текстов и Пул изображений

## 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](#_4.5._Работа_с) (создание/удаление/переименование каталогов), это позволит создать простейший файловый менеджер.

### Объявление переменных

Как упоминалось в [п. 2.2](#_2.2._Операции_с), 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](#_4.7._Экспорт_и) и [5.8](#_4.8._Экспорт_текстовых) во время создания архиваторов.

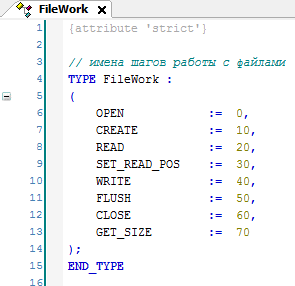


Рисунок 5.6.1 – Объявление перечисления FileWork

Во время просмотра каталогов оператор будет получать о каждом вложенном каталоге/файле в виде экземпляра структуры [FILE.FILE\_DIR\_ENTRY](#_3.2.1._Структура_FILE.FILE_DIR_ENTR). Чтобы отображать данные в визуализации следует привести их к удобному для оператора виду. Для этого объявим структуру **VisuDirInfo**:

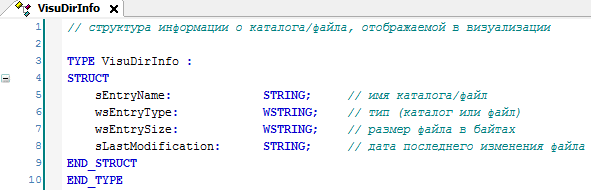


Рисунок 5.6.2 – Объявление структуры VisuDirInfo

Затем следует объявить в программе **PLC\_PRG** следующие переменные:

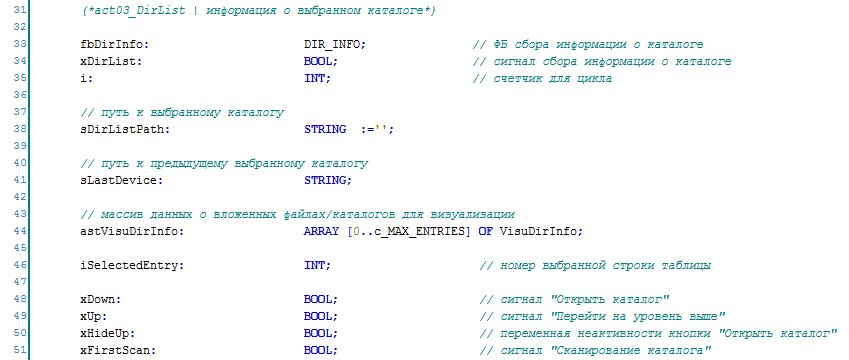


Рисунок 5.6.3 – Объявление переменных в программе PLC\_PRG

Также следует объявить несколько констант:

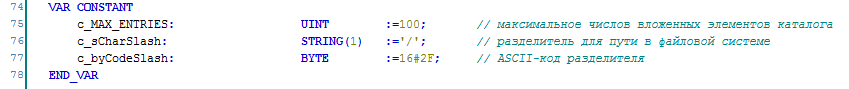


Рисунок 5.6.4 – Объявление констант в программе PLC\_PRG

### Разработка программы

На рисунке 5.6.3 был объявлен экземпляр функционального блока **DIR\_INFO**, но сам блок еще не создан. Он будет использоваться для получения информации о содержимом каталога.

Следует создать ФБ **DIR\_INFO** со следующим интерфейсом:

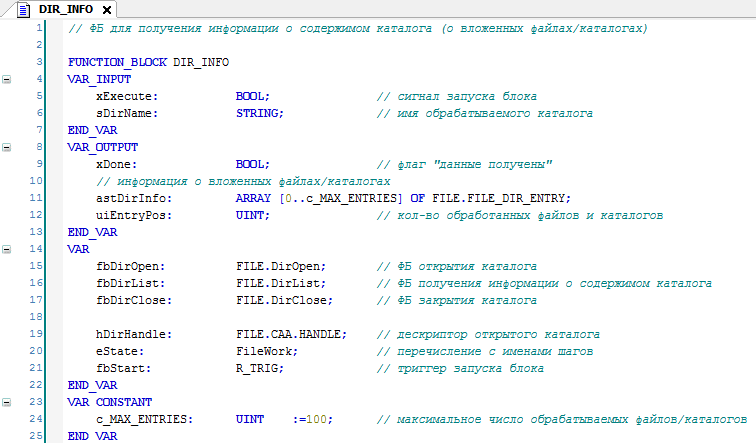


Рисунок 5.6.5 – Объявление переменных ФБ DIR\_INFO

|  |  |
| --- | --- |
|  | **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:

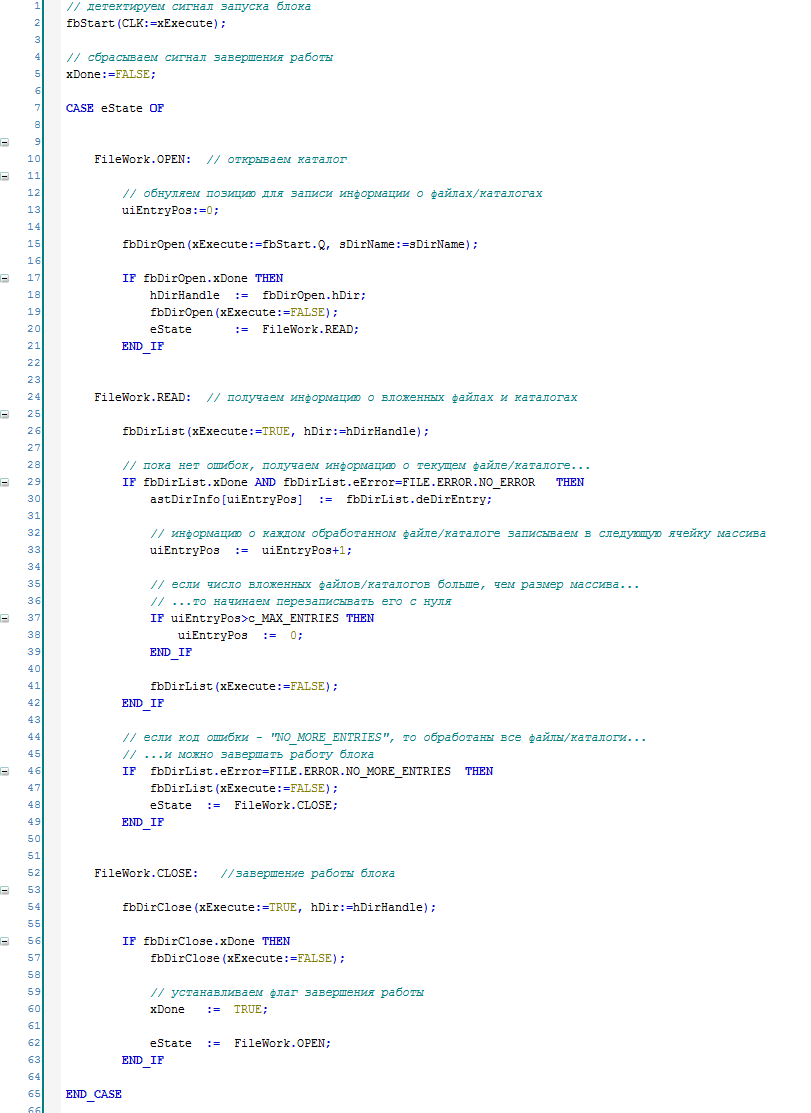


Рисунок 5.6.6 – Код ФБ DIR\_INFO

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](#_3.5.2._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](#_3.5.3._ФБ_FILE.DirList). Полученные данные записываются на выход **astDirInfo**, который представляет собой массив структур типа [FILE.FILE\_DIR\_ENTRY](#_3.2.1._Структура_FILE.FILE_DIR_ENTR). 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**.

В программе **PLC\_PRG** следует создать действие **act03\_DirList** (**PLC\_PRG – Добавление объекта – Действие**) и вынести в него код, приведенный на рисунке 5.6.7.



Рисунок 5.6.7 – 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](#_4.6.3._Создание_визуализации)) – 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.

1. 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.
2. 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.
3. 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](#_4.4.2._Разработка_программы)) и метки времени последнего изменения каталога/файла в форматированную строку с помощью функции **DT\_TO\_STRING\_FORMAT** из библиотеки **OwenStringUtils**.

### Создание визуализации

Затем следует создать интерфейс оператора для просмотра каталогов. На рисунке 5.6.8 приведен внешний вид экрана **Visu06\_DirList**, который включает в себя:

* элемент **Комбинированное окно – целочисленный**, используемый для выбора накопителя, c которым будет работать программа. Настройки элемента описаны в [п. 5.5.4](#_4.5.4._Настройка_элемента). 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;

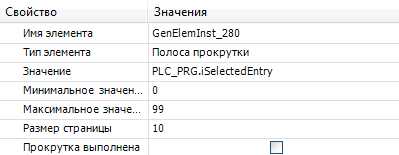


Рисунок 5.6.8 – Настройки элемента Полоса прокрутки

* 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**).

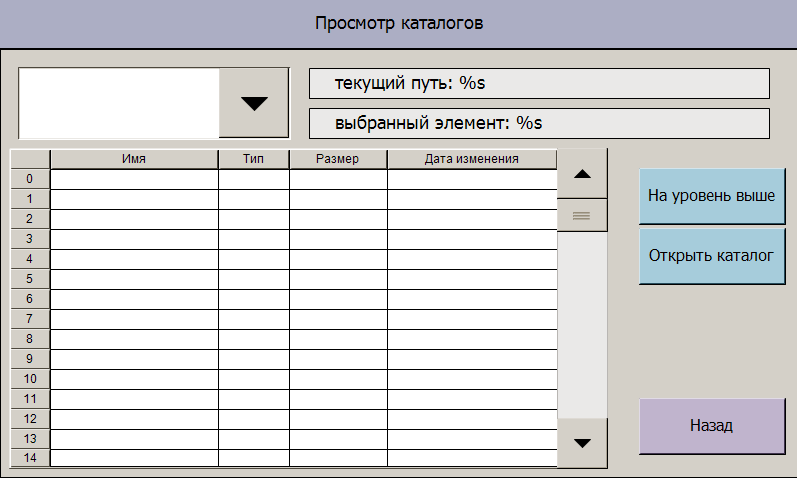


Рисунок 5.6.9 – Внешний вид экрана Visu06\_DirList

Визуализация также содержит кнопки переключения экранов (описание других экранов проекта приведено в соответствующих пунктах). Пример работы с экраном приведен в [п. 5.10](#_4.10._Работа_с).

## Экспорт и импорт бинарных файлов (BinFileExample\_PRG)

Информация о различиях бинарных и текстовых файлов приведена в [п. 2.6](#_2.6._Бинарные_и).

### Объявление переменных

Как упоминалось в [п. 2.2](#_2.2._Операции_с), 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](#_4.6.1._Объявление_переменных):

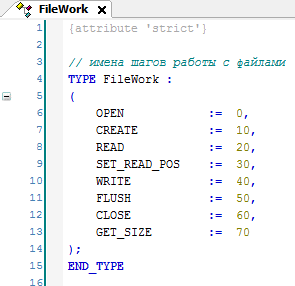


Рисунок 5.7.1 – Объявление перечисления FileWork

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**:

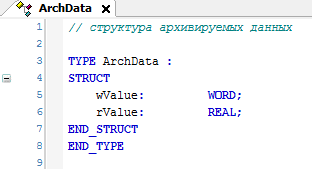


Рисунок 5.7.2 – Объявление структуры ArchData

Следует объявить в программе **BinFileExample\_PRG** следующие переменные:

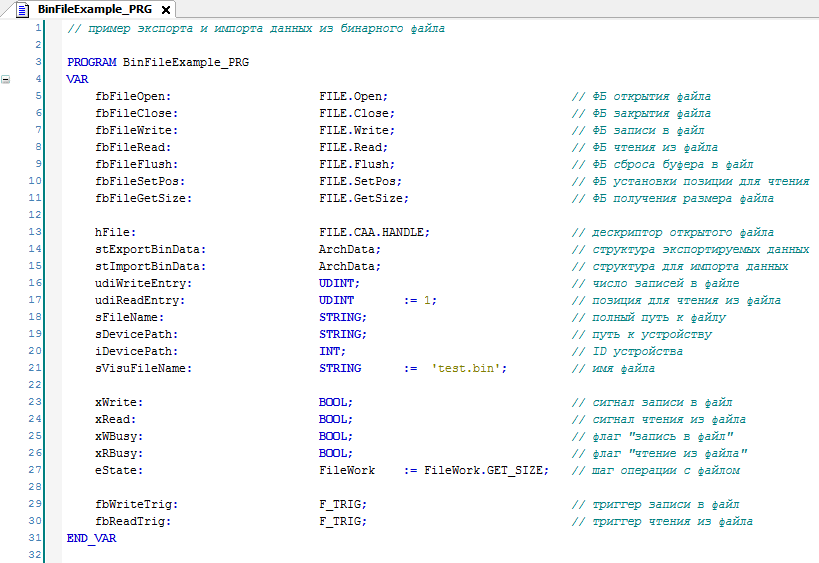


Рисунок 5.7.3 – Объявление переменных в программе BinFileExample\_PRG

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

### 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](#_4.5.2._Разработка_программы)) 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).

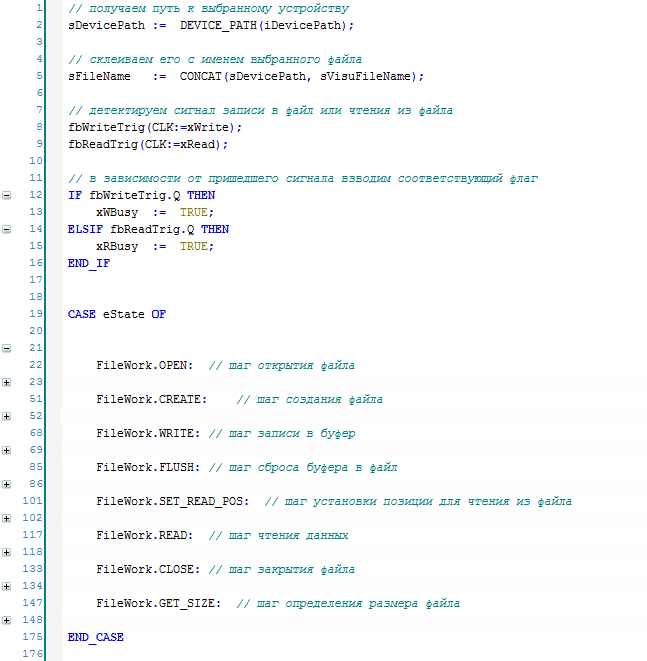


Рисунок 5.7.4 – Структура программы BinFileExample\_PRG

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](#_3.5.1._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).

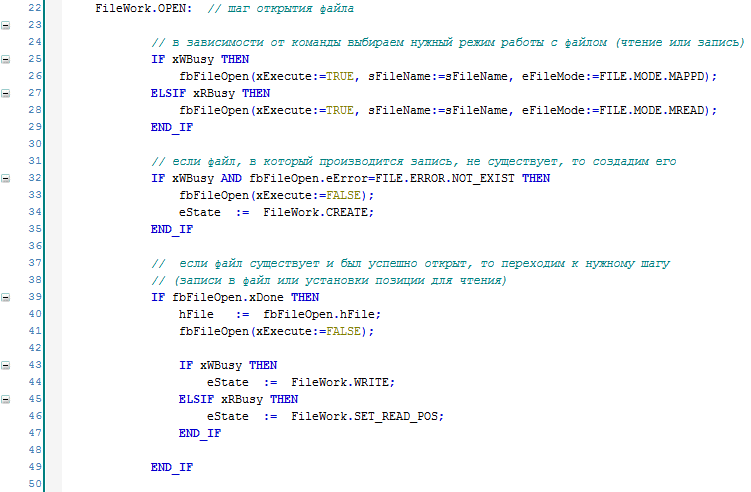
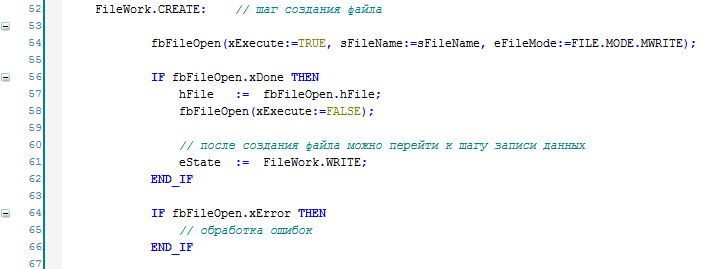


Рисунок 5.7.5 – Код шага OPEN

На шаге **CREATE** происходит создание файла с помощью экземпляра ФБ [FILE.OPEN](#_3.5.1._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.

 Рисунок 5.7.6 – Код шага CREATE

На шаге **WRITE** происходит запись данных структуры **stExportBinData** в системный буфер с помощью экземпляра ФБ [FILE.WRITE](#_3.6.3._ФБ_FILE.Write). После записи осуществляется переход на шаг **FLUSH**. В рамках примера обработка ошибок на данном шаге не производится.

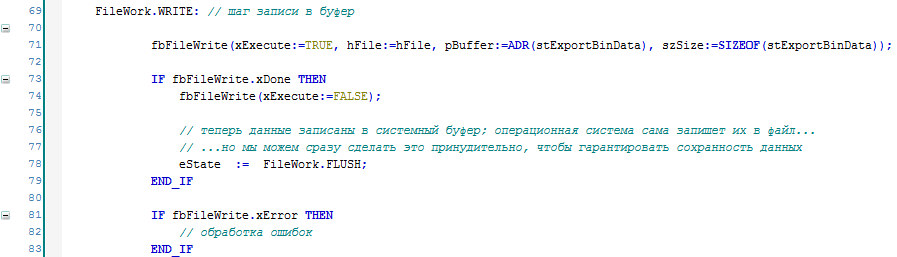


Рисунок 5.7.7 – Код шага WRITE

На шаге **FLUSH** происходит сброс системного буфера в файл с помощью ФБ [FILE.FLUSH](#_3.6.8._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.

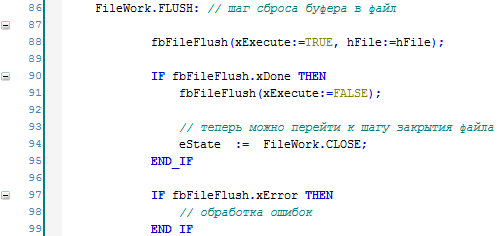


Рисунок 5.7.8 – Код шага FLUSH

На шаге **SET\_READ\_POS** происходит установка позиции для чтения с помощью ФБ [FILE.SetPos](#_4.6.10_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.

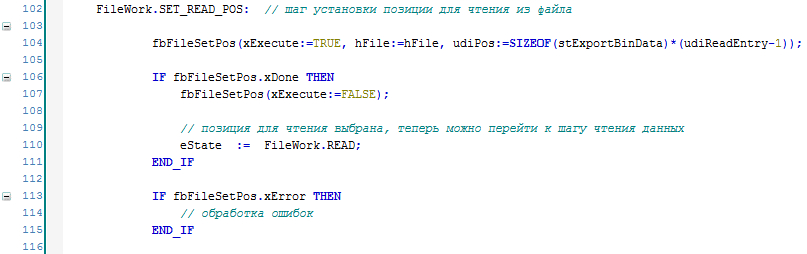


Рисунок 5.7.9 – Код шага SET\_READ\_POS

На шаге **READ** из файла считываются данные с помощью экземпляра ФБ [FILE.READ.](#_4.6.4_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.

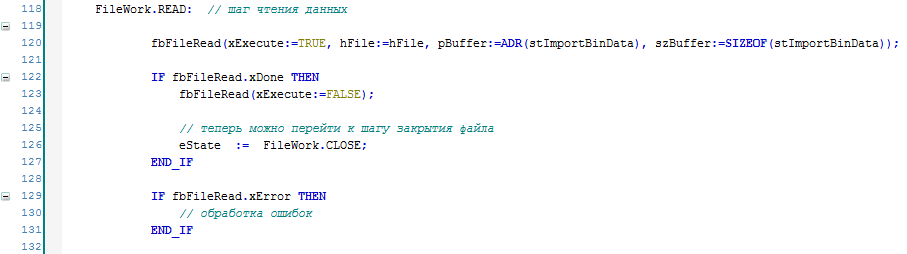


Рисунок 5.7.10 – Код шага READ

На шаге **CLOSE** происходит закрытие файла с помощью экземпляра ФБ [FILE.CLOSE](#_4.6.2_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.

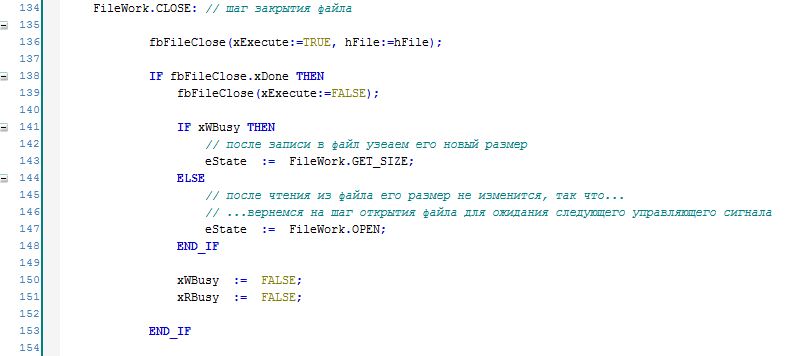


Рисунок 5.7.11 – Код шага CLOSE

На шаге **GET\_SIZE** происходит определение размера файла с помощью экземпляра ФБ [FILE.GetSize](#_3.6.12._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**.

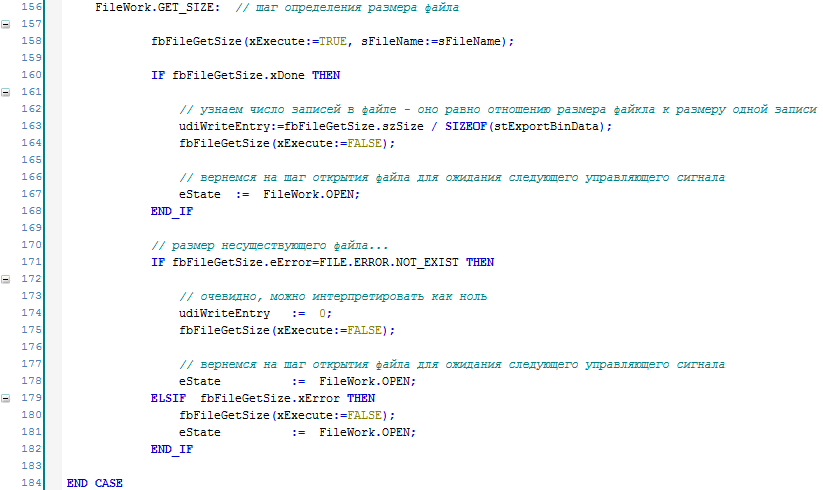


Рисунок 5.7.12 – Код шага GET\_SIZE

### 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, c которым будет работать программа. Настройки элемента описаны в [п. 5.5.4](#_4.5.4._Настройка_элемента). 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.

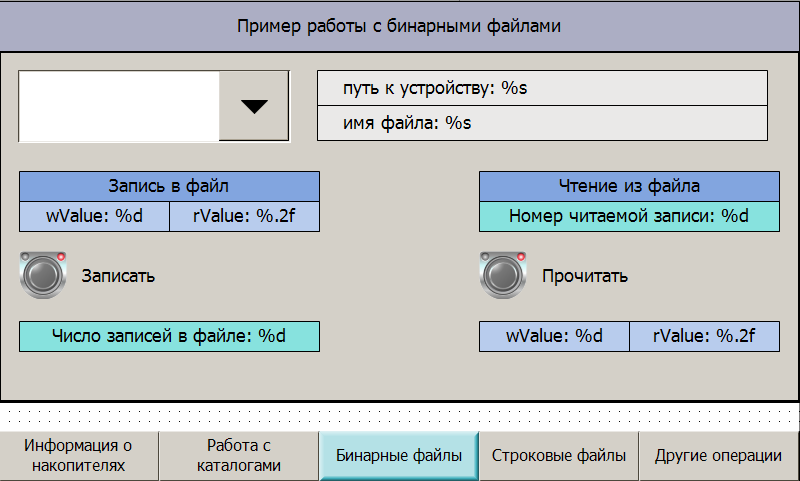


Рисунок 5.7.13 – Внешний вид экрана Visu03\_BinFileExample

## Экспорт текстовых файлов (StringFileExample\_PRG)

Формат [.csv](https://ru.wikipedia.org/wiki/CSV) используется для представления табличных данных и состоит из текстовых записей, разграниченных символом-разделителем. В русскоязычной [локали](https://ru.wikipedia.org/wiki/%D0%9B%D0%BE%D0%BA%D0%B0%D0%BB%D1%8C) таким символом по умолчанию является точка с запятой (**;**). Информация о различиях бинарных и текстовых файлов приведена в [п. 2.6](#_2.6._Бинарные_и).

### Объявление переменных

Как упоминалось в [п. 2.2](#_2.2._Операции_с), 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](#_4.6.1._Объявление_переменных):

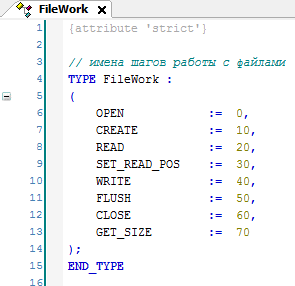


Рисунок 5.8.1 – Объявление перечисления FileWork

Структура архивируемых данных **ArchData**, содержащая одну переменную типа **WORD** и одну переменную типа **REAL**, уже была объявлена в [п. 5.7.1](#_4.7.1._Объявление_переменных):

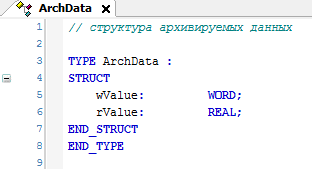


Рисунок 5.8.2 – Объявление структуры ArchData

В программе **StringFileExample\_PRG** следует объявить следующие переменные и константы:

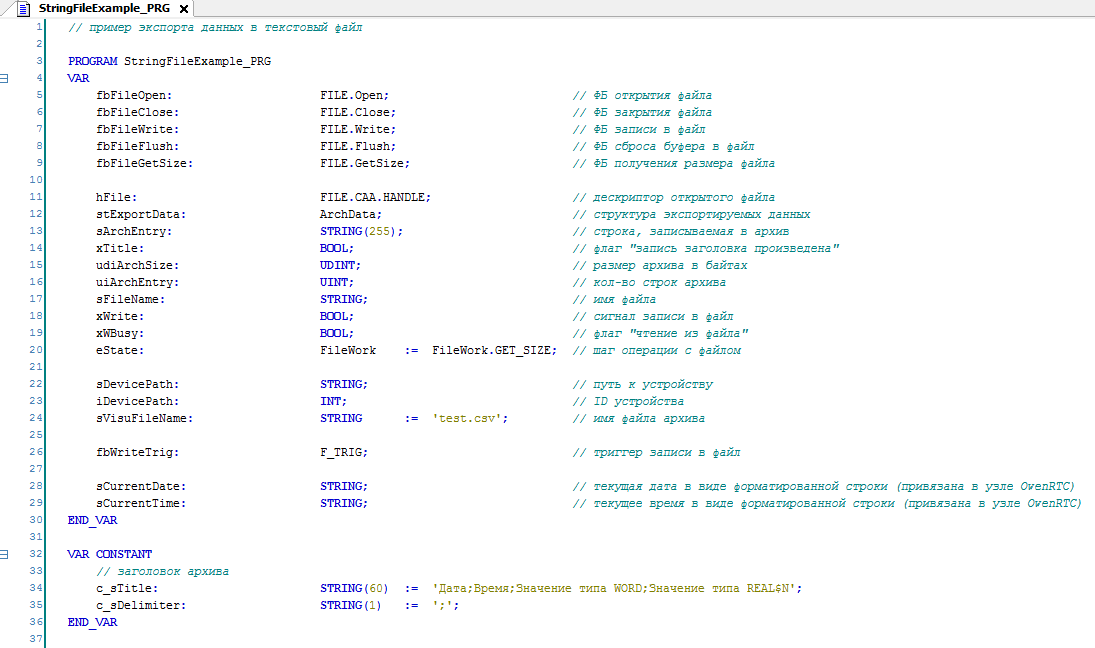


Рисунок 5.8.3 – Объявление переменных в программе StringFileExample\_PRG

Для строковых переменных, значения которых будут заноситься в файл архива, длина указывается в явном виде (например, **STRING(60)**). 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.

### Разработка программы

Структура программы **StringFileExample\_PRG** приведена на рисунке 5.8.4:

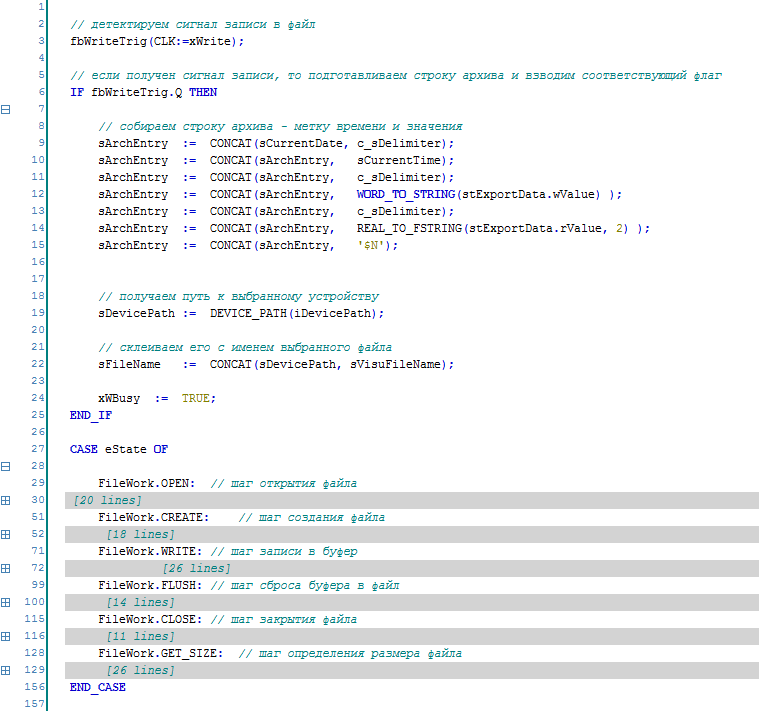
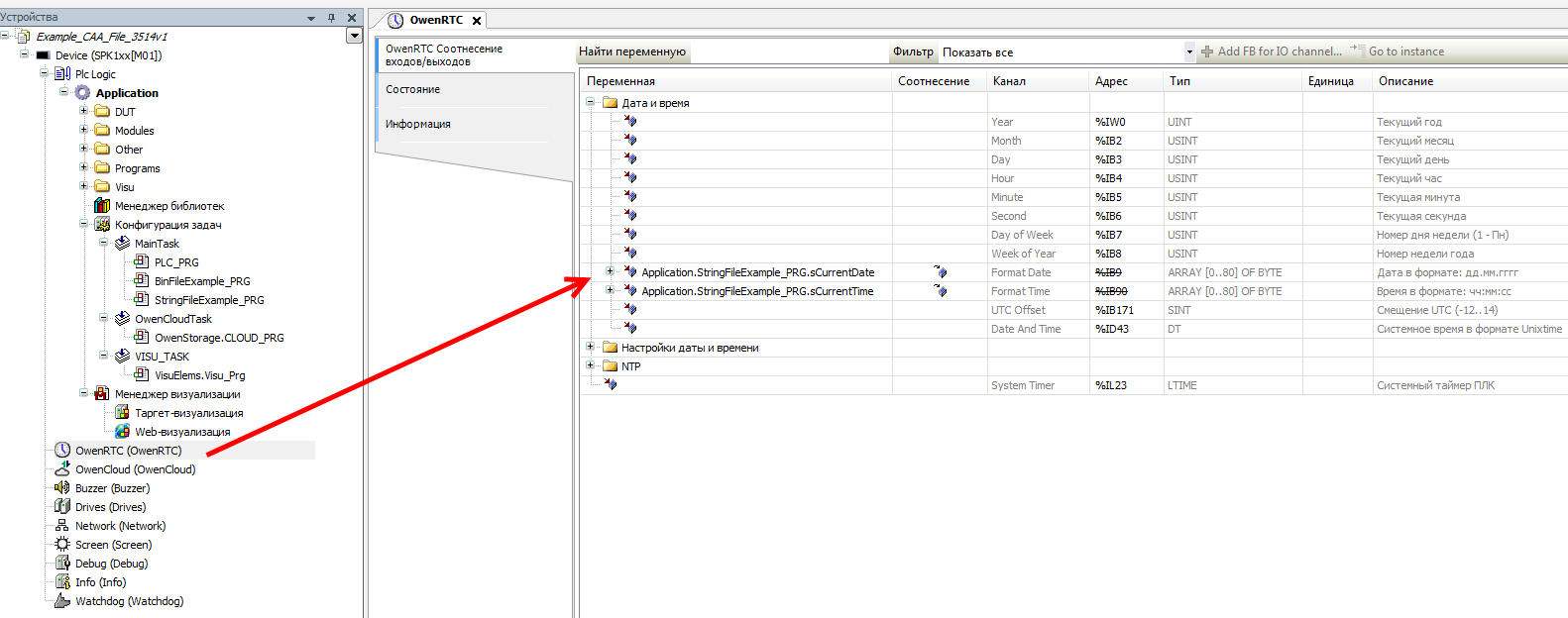


Рисунок 5.8.4 – Структура программы StringFileExample\_PRG

В программе можно выделить два основных сегмента:

1. Детектирование команды записи и подготовка записываемых архивных данных (весь код до оператора **CASE**);
2. Запись в файл (содержимое оператора **CASE**).

Переменные **sCurrentDate** и **sCurentTime** привязаны к соответствующим каналам узла таргет-файла **OwenRTC**.



Метка времени и архивируемые значения склеиваются в одну строку, формируя архивную запись.

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 русскоязычных [локалях](https://ru.wikipedia.org/wiki/%D0%9B%D0%BE%D0%BA%D0%B0%D0%BB%D1%8C)). Код функции выглядит следующим образом:

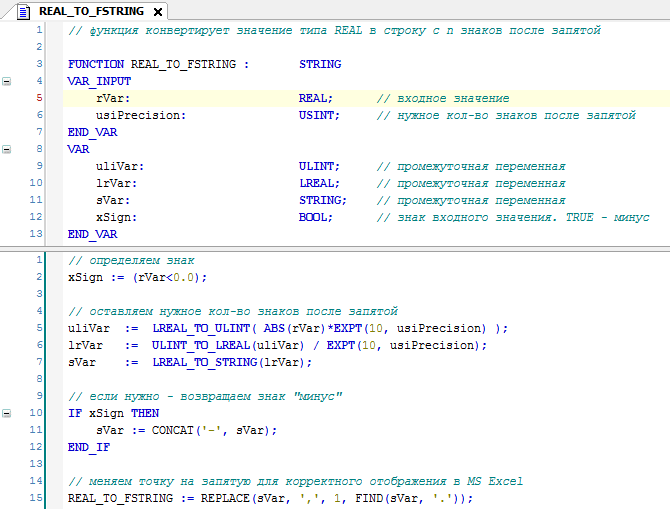


Рисунок 5.8.5 – Код функции REAL\_TO\_FSTRING

Код функции почти полностью совпадает с кодом функции **REAL\_TO\_FWSTRING** из [п. 5.4.2](#_4.4.2._Разработка_программы). Оптимальным решением было бы объединить обе функции в одну, но для упрощения примера рассматривается вариант с двумя отдельными функциями.

Архив сохраняется в формате **.csv**. В русскоязычных [локалях](https://ru.wikipedia.org/wiki/%D0%9B%D0%BE%D0%BA%D0%B0%D0%BB%D1%8C) 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](#_2.6._Бинарные_и)). Соответственно, после детектирования сигнала записи в переменную **sArchEntry** будет записана текущая строка архива. Затем следует записать эту строку в файл.

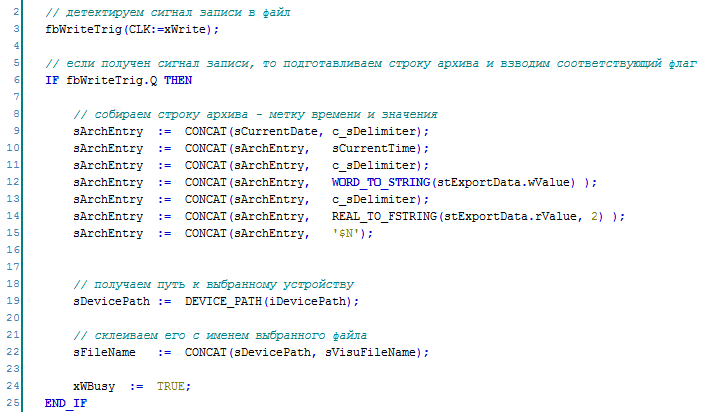


Рисунок 5.8.6 – Фрагмент программы StringFileExample\_PRG

Перед началом работы с файлом программа получает путь к выбранному устройству (с помощью функции **DEVICE\_PATH** из [п. 5.5.2](#_4.5.2._Разработка_программы)), склеивает его с именем файла архива и детектирует задний фронт сигнала записи.

Работа с файлами происходит в управляющем операторе **CASE**. [На рисунке 5.8.4](#_5.8.2._Разработка_программы) 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](#_3.5.1._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**.

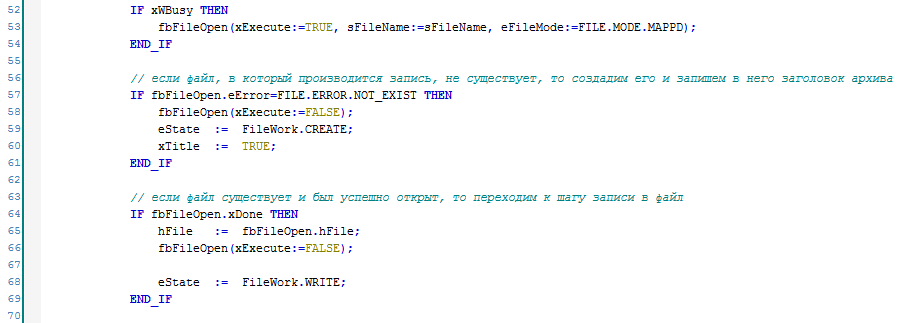
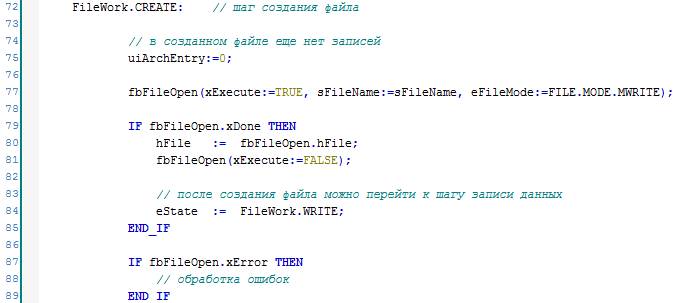


Рисунок 5.8.7 – Код шага OPEN

На шаге **CREATE** происходит создание файла с помощью экземпляра ФБ [FILE.OPEN](#_3.5.1._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.

 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](#_3.5.3._ФБ_FILE.Write). После записи осуществляется переход на шаг **FLUSH**. В рамках примера обработка ошибок на данном шаге не производится.

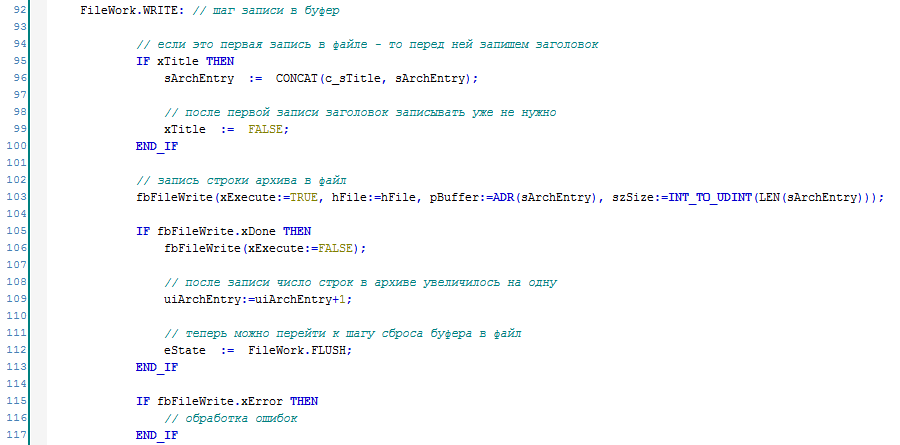


Рисунок 5.8.9 – Код шага WRITE

На шаге **FLUSH** происходит сброс системного буфера в файл с помощью ФБ [FILE.FLUSH](#_3.6.8._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.

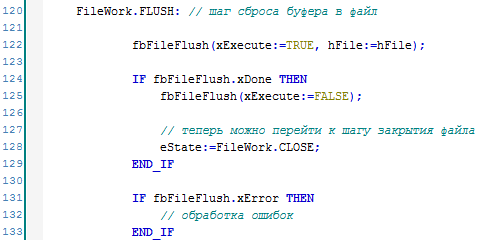


Рисунок 5.8.10 – Код шага FLUSH

На шаге **CLOSE** происходит закрытие файла с помощью ФБ [FILE.CLOSE](#_3.6.2._ФБ_FILE.Close). После закрытия файла происходит переход на шаг **GET\_SIZE**.

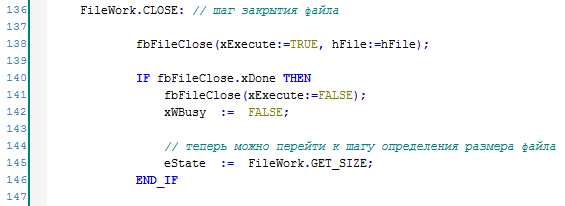


Рисунок 5.8.11 – Код шага CLOSE

На шаге **GET\_SIZE** происходит определение размера файла с помощью ФБ [FILE.GetSize](#_3.6.12._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**.

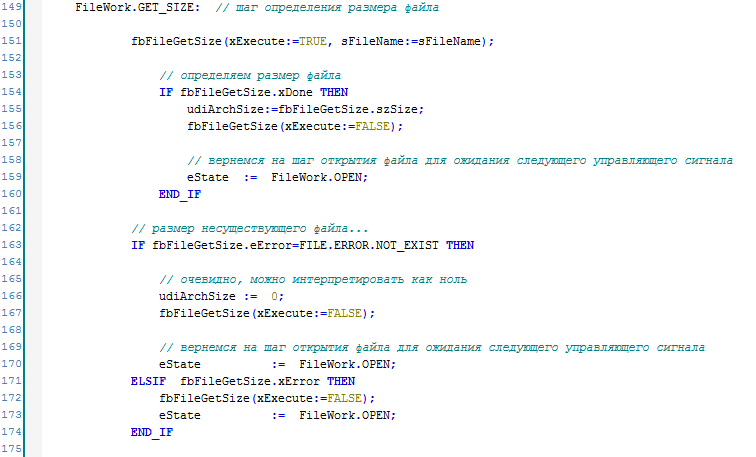


Рисунок 5.8.12 – Код шага GET\_SIZE

### 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, c которым будет работать программа. Настройки элемента описаны в [п. 5.5.4](#_4.5.4._Настройка_элемента). 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**.

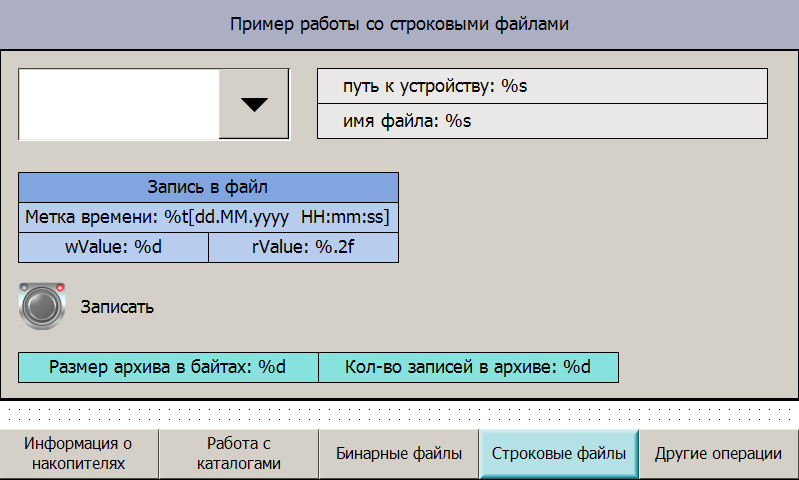


Рисунок 5.8.13 – Внешний вид экрана Visu04\_StringFileExample

## Дополнительные операции с файлами (PLC\_PRG, действие act04\_ActionsWithFiles)

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

### Объявление переменных

В программе **PLC\_PRG** следует объявить следующие переменные:

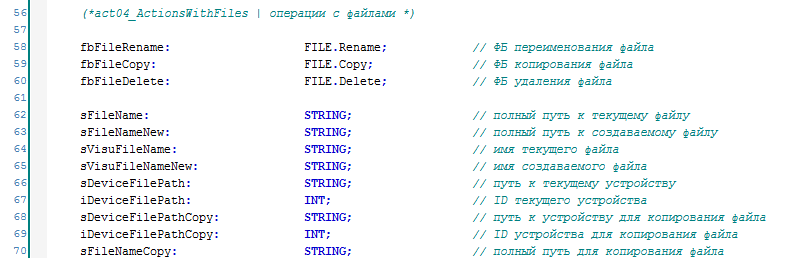


Рисунок 5.9.1 – Объявление переменных в программе PLC\_PRG

### Разработка программы

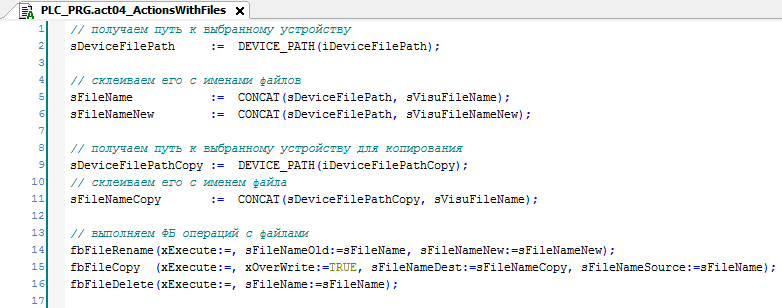
В программе PLC\_PRG следует создать действие act04\_ActionWithFiles (PLC\_PRG – Добавление объекта – Действие) и вынести в него следующий код: 

Рисунок 5.9.2 – Код действия act04\_ActionWithFiles

Действие производит следующие операции:

* + - возвращает путь к выбранному накопителю по его **ID** c помощью функции **DEVICE\_PATH,** созданной в [п. 5.5.2](#_4.5.2._Разработка_программы);
    - 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**:

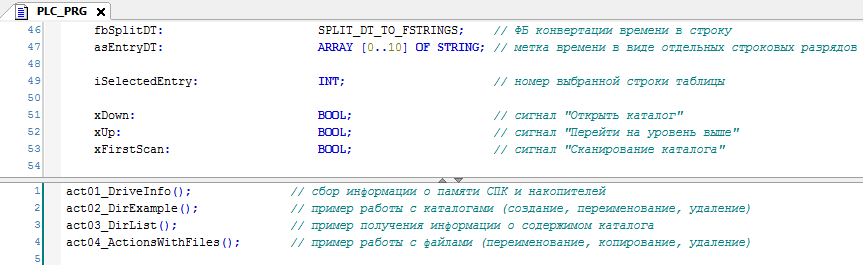


Рисунок 5.9.3 – Вызов действия act04\_ActionsWithFiles в программе PLC\_PRG

### 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](#_4.5.4._Настройка_элемента).
* 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**.

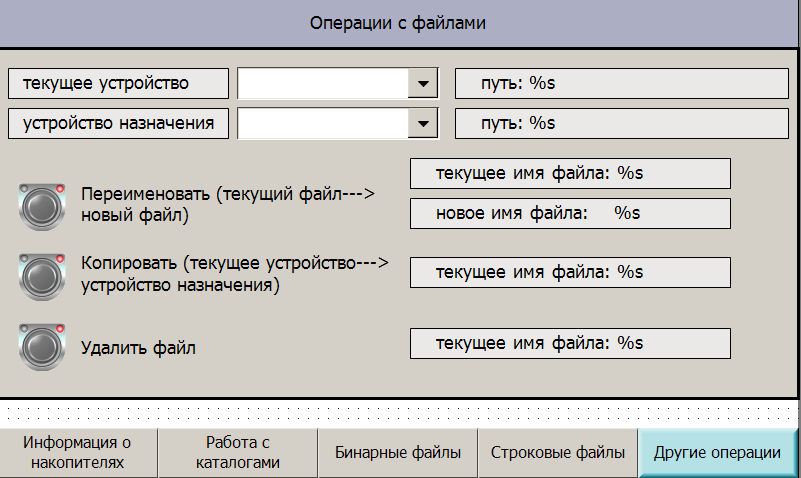


Рисунок 5.9.4 – Внешний вид экрана Visu05\_FilesActionsExample

Визуализация также содержит кнопки переключения экранов (описание других экранов проекта приведено в соответствующих пунктах). Пример работы с экраном приведен в [п. 5.10](#_4.10._Работа_с).

## 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 (**СПК1хх [М01]**), 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.

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

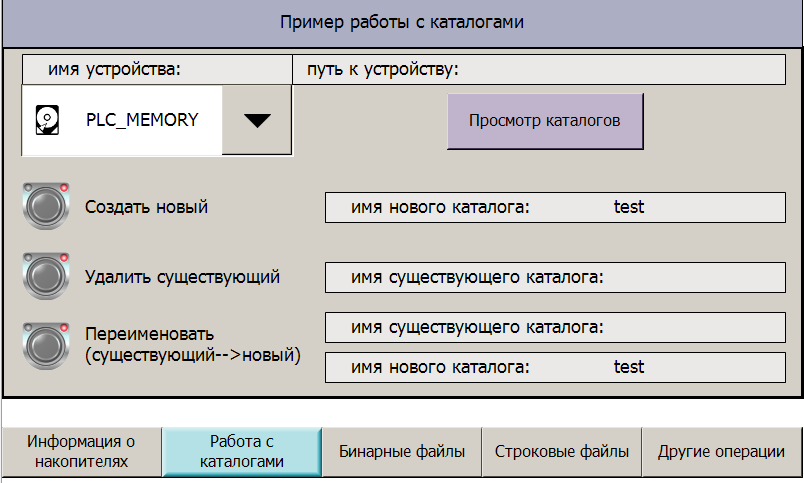


Рисунок 5.10.2 – Внешний вид экрана Работа с каталогами (Visu02\_DirExample)

Затем следует подключиться к контроллеру с помощью утилиты **WinSCP** (см. [п. 2.8](#_2.8._Подключение_к)), перейти по отображаемому пути (на рисунке 5.10.3 это **/home/root/CODESYS\_WRK/PlcLogic**) и убедится, что был создан каталог с названием **test.**

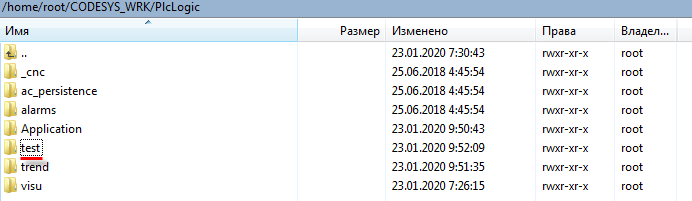
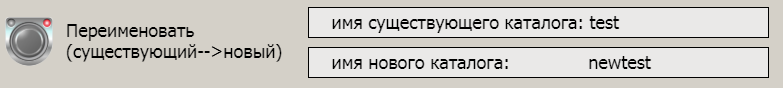


Рисунок 5.10.3 – Проверка создания нового каталога

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



Нажать кнопку **Переименовать**.

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

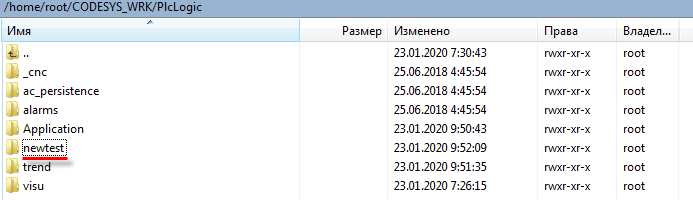


Рисунок 5.10.4 – Проверка переименования каталога

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



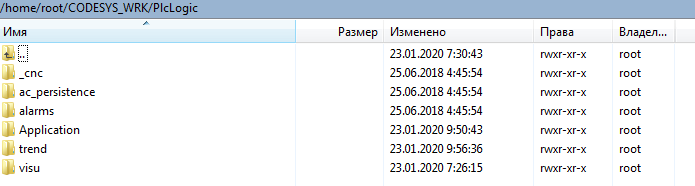


Рисунок 5.10.5 – Проверка удаления каталога

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.

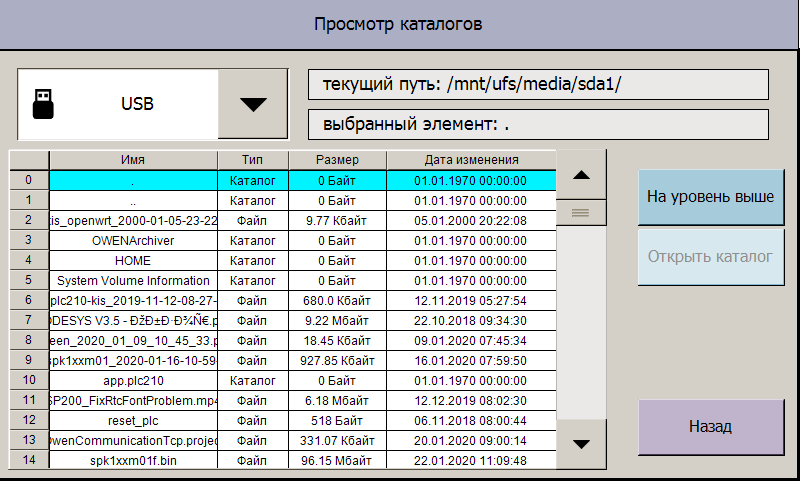


Рисунок 5.10.6 – Внешний вид экрана Просмотр каталогов (Visu06\_DirList)

На рисунке выше размер и дата изменения каталогов отображается некорректно. Также не поддерживается отображения русскоязычных символов. Подробнее см. в [п. 4.2.1](#_3.2.1._Структура_FILE.FILE_DIR_ENTR).

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

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

1. 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.

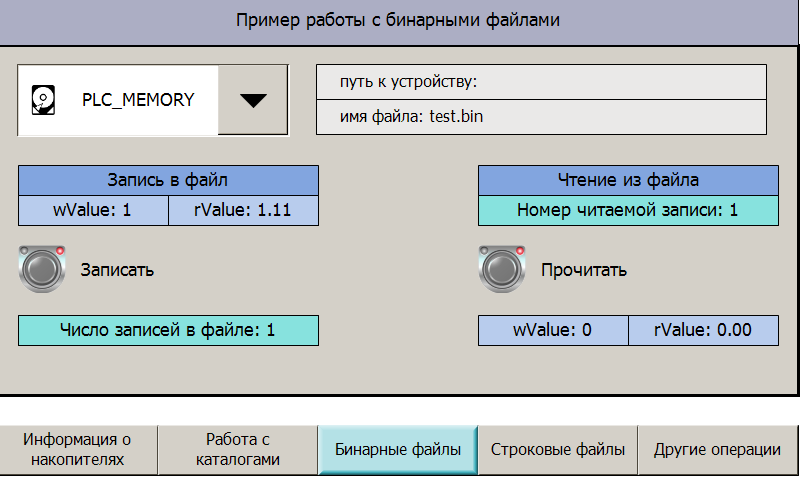


Рисунок 5.10.7 – Внешний вид экрана Бинарные файлы (Visu03\_BinFileExample)

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

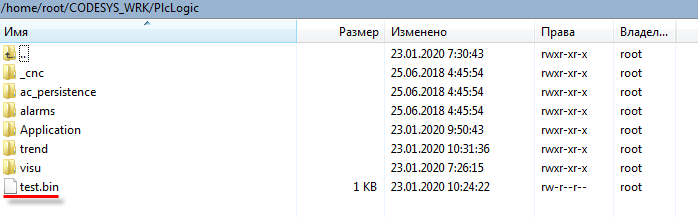


Рисунок 5.10.8 – Проверка создания файла

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

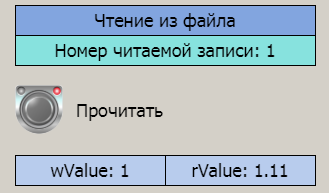


Рисунок 5.10.9 – Чтение данных из файла

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

1. 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.

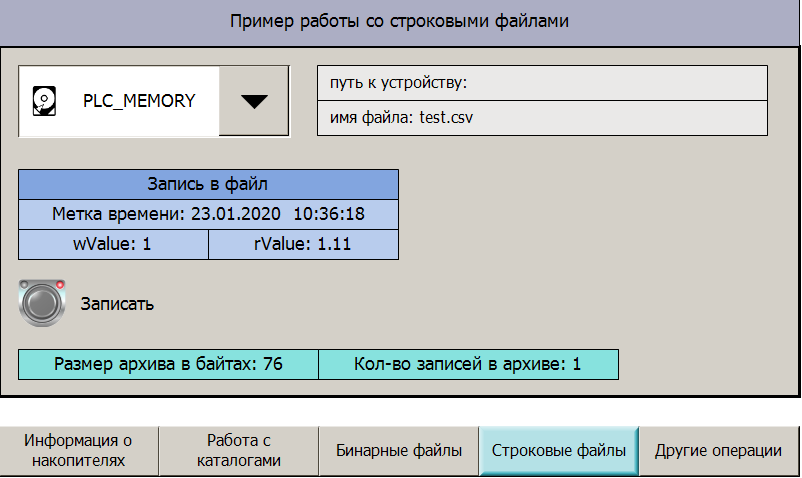


Рисунок 5.10.10 – Внешний вид экрана Строковые файлы (Visu04\_StringFileExample)

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

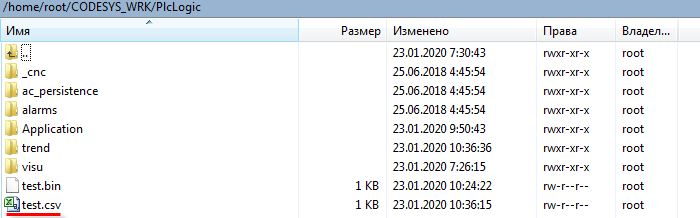


Рисунок 5.10.11 – Проверка создания файла

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

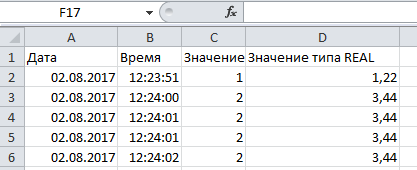


Рисунок 5.10.12 – Пример содержимого файла

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**.

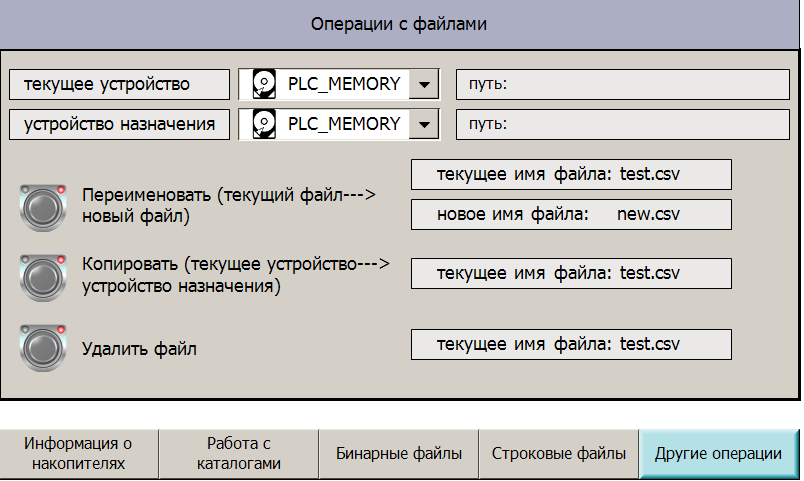


Рисунок 5.10.13 – Внешний вид экрана Другие операции (Visu05\_FileActionsExample)

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

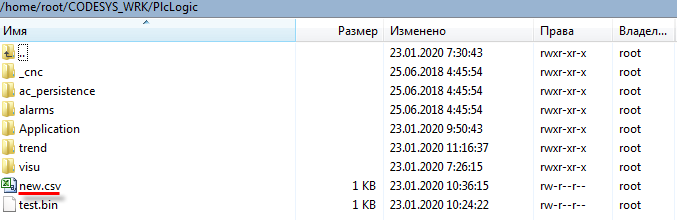


Рисунок 5.10.14 – Проверка переименования файла

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

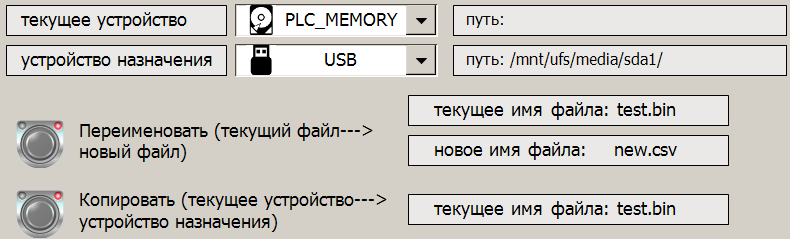


Рисунок 5.10.15 – Копирование файлов

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

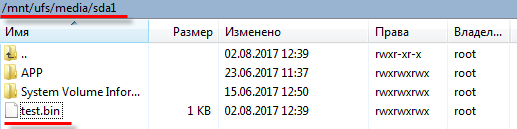


Рисунок 5.10.15 – Проверка копирования файла

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

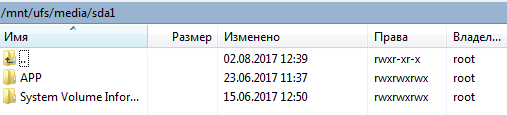


Рисунок 5.10.16 – Проверка удаления файла

## 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](#_3.2.2._Перечисление_FILE.ERROR);
* 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.

# Листинг примера

## Структуры и перечисления

### Структура ArchData

// структура архивируемых данных

TYPE ArchData :

STRUCT

wValue: WORD;

rValue: REAL;

END\_STRUCT

END\_TYPE

### 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

### Перечисление FileDevice

{attribute 'strict'}

// тип устройства для архивации

TYPE FileDevice :

(

PLC\_MEMORY := 0,

USB := 10,

SD := 20,

FTP := 30

);

END\_TYPE

### 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

### 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

## Структуры и перечисления

### 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

### 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:='';

FileDevice.USB:

DEVICE\_PATH:='/mnt/ufs/media/sda1/';

FileDevice.SD:

DEVICE\_PATH:='/mnt/ufs/media/mmcblk0p1/';

FileDevice.FTP:

DEVICE\_PATH:='/mnt/ufs/home/ftp/in/’;

END\_CASE

### 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

### 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, '.'));

### 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);

## 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 :='';

// путь к предыдущему выбранному каталогу

sLastDevice: STRING;

// массив данных о вложенных файлах/каталогов для визуализации

astVisuDirInfo: ARRAY [0..c\_MAX\_ENTRIES] OF VisuDirInfo;

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

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)

### 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

### 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);

### 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

Mem.MemFill(ADR(astVisuDirInfo), SIZEOF(astVisuDirInfo), 0);

// 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

astVisuDirInfo[i].sLastModification := OSU.DT\_TO\_STRING\_FORMAT(fbDirInfo.astDirInfo[i].dtLastModification, '%t[dd.MM.yyyy HH:mm:ss]');

xDirList := FALSE;

END\_FOR

END\_IF

### 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);

## 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

## 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

// текущая дата в виде форматированной строки (привязана в узле OwenRTC)

sCurrentDate: STRING;

// текущее время в виде форматированной строки (привязана в узле OwenRTC)

sCurrentTime: STRING;

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));

// если получен сигнал записи, то подготавливаем строку архива и взводим соответствующий флаг

IF fbWriteTrig.Q THEN

// собираем строку архива - метку времени и значения

sArchEntry := CONCAT(sCurrentDate, c\_sDelimiter);

sArchEntry := CONCAT(sArchEntry, sCurrentTime);

sArchEntry := CONCAT(sArchEntry, c\_sDelimiter);

sArchEntry := CONCAT(sArchEntry, WORD\_TO\_STRING(stExportData.wValue) );

sArchEntry := CONCAT(sArchEntry, c\_sDelimiter);

sArchEntry := CONCAT(sArchEntry, REAL\_TO\_FSTRING(stExportData.rValue, 2) );

sArchEntry := CONCAT(sArchEntry, '$N');

// получаем путь к выбранному устройству

sDevicePath := DEVICE\_PATH(iDevicePath);

// склеиваем его с именем выбранного файла

sFileName := CONCAT(sDevicePath, sVisuFileName);

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

1. Здесь отображается не объем физической памяти, а объем области, выделенный системе исполнения CODESYS [↑](#footnote-ref-1)
2. Данный режим добавлен в версии библиотеки 3.5.13.40. [↑](#footnote-ref-2)
3. Данный ФБ добавлен в версии библиотеки 3.5.13.40. [↑](#footnote-ref-3)