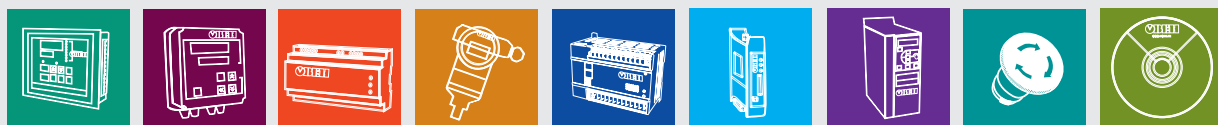


СПК

Архивация

Руководство для начинающих и продвинутых пользователей

Версия: 1.0
Дата: 30.01.2018



Оглавление

1. Цель и структура документа	5
2. Основные сведения о работе с файлами	6
2.1. Общие сведения о памяти СПК	6
2.2. Операции с файлами.....	7
2.3. Требования к подключаемым накопителям (USB/SD)	7
2.4. Пути к файлам и накопителям. Монтирование и размонтирование	8
2.5. Ограничения на имена файлов и каталогов в ОС Linux.....	9
2.6. Бинарные и текстовые файлы	10
2.7. Обработка ошибок и некорректных ситуаций.....	11
2.8. Подключение к файловой системе контроллера	12
3. Компонент OwenArchiver.....	14
3.1. Установка компонента в CODESYS.....	14
3.2. Добавление архиватора в проект	16
3.3. Ограничения, связанные с использованием архиватора	21
3.4. Пример работы с архиватором	22
4. Библиотека CAA File.....	28
4.1. Добавление библиотеки в проект CODESYS.....	28
4.2. Структуры и перечисления	29
4.2.1. Структура FILE.FILE_DIR_ENTRY	29
4.2.2. Перечисление FILE.ERROR.....	29
4.2.3. Перечисление FILE.MODE	30
4.3. Пути к каталогам и файлам.....	30
4.4. Ограничения при работе с файлами.....	30
4.5. ФБ работы с каталогами.....	31
4.5.1. ФБ FILE.DirCreate	31
4.5.2. ФБ FILE.DirOpen	32
4.5.3. ФБ FILE.DirList	33
4.5.4. ФБ FILE.DirRemove	34
4.5.5. ФБ FILE.DirRename	35
4.5.6. ФБ FILE.DirClose	36
4.6. ФБ работы с файлами.....	37
4.6.1. ФБ FILE.Open.....	37

4.6.2. ФБ FILE.Close	38
4.6.3. ФБ FILE.Write	39
4.6.4. ФБ FILE.Read	40
4.6.5. ФБ FILE.Rename	41
4.6.6. ФБ FILE.Copy	42
4.6.7. ФБ FILE.Delete	43
4.6.8. ФБ FILE.Flush	44
4.6.9. ФБ FILE.GetPos	45
4.6.10. ФБ FILE.SetPos	46
4.6.11. ФБ FILE.EOF	47
4.6.12. ФБ FILE.GetSize	48
4.6.13. ФБ FILE.GetTime	49
4.7. Особенности версии библиотеки 3.5.3.0	50
5. Пример работы с библиотекой CAA File	51
5.1. Краткое описание примера	51
5.2. Используемые библиотеки	51
5.3. Содержимое примера	52
5.4. Получение информации о накопителях (PLC_PRG, действие act01_DriveInfo)	53
5.4.1. Объявление переменных	53
5.4.2. Разработка программы	55
5.4.3. Создание визуализации	59
5.5. Работа с каталогами (PLC_PRG, действие act02_DirExample)	60
5.5.1. Объявление переменных	60
5.5.2. Разработка программы	61
5.5.3. Создание визуализации	63
5.5.4. Настройка элемента Комбинированное окно	64
5.6. Просмотр содержимого каталогов (PLC_PRG, действие act03_DirList)	65
5.6.1. Объявление переменных	65
5.6.2. Разработка программы	67
5.6.3. Создание визуализации	74
5.7. Экспорт и импорт бинарных файлов (BinFileExample_PRG)	76
5.7.1. Объявление переменных	76
5.7.2. Разработка программы	78
5.7.3. Создание визуализации	84
5.8. Экспорт текстовых файлов (StringFileExample_PRG)	85

5.8.1. Объявление переменных	85
5.8.2. Разработка программы	87
5.8.3. Создание визуализации.....	94
5.9. Дополнительные операции с файлами (PLC_PRG, действие act04_ActionsWithFiles).....	95
5.9.1. Объявление переменных	95
4.9.2. Разработка программы	95
5.9.3. Создание визуализации.....	97
5.10. Работа с примером.....	98
5.11. Рекомендации и замечания	107
Приложение. Листинг примера.....	108
А. Структуры и перечисления	108
А.1. Структура ArchData.....	108
А.2. Структура DriveInfo	108
А.3. Перечисление FileDevice.....	108
А.4. Перечисление FileDevice.....	109
А.5. Структура VisuDirInfo	109
Б. Структуры и перечисления	110
Б.1. Функция BYTE_SIZE_TO_WSTRING	110
Б.2. Функция CONCAT11	110
Б.3. Функция DEVICE_PATH	111
Б.4. ФБ DIR_INFO	111
Б.5. Функция LEAD_ZERO	113
Б.6. Функция REAL_TO_FSTRING	113
Б.7. Функция REAL_TO_FWSTRING	113
Б.8. ФБ SPLIT_DT_TO_FSTRINGS.....	114
В. Программа PLC_PRG.....	115
В.1. Действие act01_DriveInfo	116
В.2. Действие act02_DirExample	116
В.3. Действие act03_DirList.....	117
В.4. Действие act04_ActionsWithFiles.....	118
Г. Программа BinFileExample	119
Д. Программа StringFileExample	123

1. Цель и структура документа

Одной из типичных задач АСУТП 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 3.5**, способны архивировать данные во внутреннюю память или на внешний носитель (USB- или SD-накопитель), а также считывать данные (например, файлы рецептов, технологические карты и т.д.). Для этого могут использоваться компонент **OwenArchiver** или библиотека **CAA File**, описанные в данном документе.

Особенности компонента **OwenArchiver**:

- рассчитан на начинающих пользователей, не требует навыков программирования;
- настройка через дерево проекта в несколько кликов;
- жестко заданная структура архива и условия архивации.

Особенности библиотеки **CAA File**:

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

В [п. 2](#) приведена основная информация о работе с файлами.

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

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

В [п. 5](#) рассмотрены примеры использования библиотеки.

Необходимо отметить, что разработка ПО для работы с файлами подразумевает высокую квалификацию программиста, а также хорошее знание среды **CODESYS 3.5** и языка ST. Реализация блоков архивации на графических языках (например, CFC) является крайне затруднительной из-за сложности алгоритмов; при этом в программах, написанных на графических языках, можно вызывать готовые блоки, реализованные на языке ST.

Документ рекомендуется читать строго последовательно.

2. Основные сведения о работе с файлами

2.1. Общие сведения о памяти СПК

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

Способ организации, хранения и именования файлов на конкретном устройстве зависит от его **файловой системы**. Файловая система контроллеров СПК – **UBIFS**.

У контроллеров СПК 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. Эта память имеет ограниченный ресурс перезаписи (~50000 перезаписей на блок данных) – поэтому для архивации данных в большинстве случаев рекомендуется использовать внешние накопители (USB, SD). Очевидно, что ресурс перезаписи внешних накопителей также ограничен, но их выход из строя не повлияет на работоспособность контроллера; кроме того, накопители можно оперативно заменить. Информация об общем доступном объеме памяти приведена в руководстве по эксплуатации на соответствующий контроллер. Информация о количестве свободной/занятой памяти доступна в **Конфигураторе СПК** и **таргет-файле** (начиная с версии **3.5.4.25**).



Рис. 2.1. Информация о памяти контроллера и накопителей в **конфигураторе СПК**

2.2. Операции с файлами

При работе 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 при всех остальных операциях с ним.

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

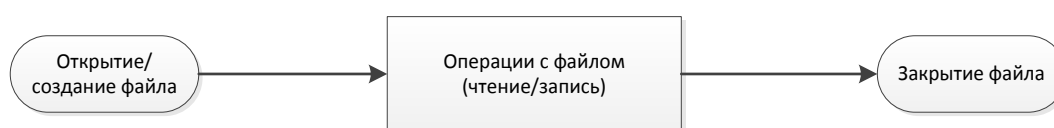


Рис. 2.2. Упрощенная схема работы с файлами

При этом в подавляющем большинстве случаев работа с файлами производится с помощью единичных операций – т.е. файл открывается только на то время, которое нужно, чтобы считать/записать в него требуемые в текущий момент данные. Постоянно держать файл открытым не рекомендуется – в частности, из-за ограничения на максимальное число одновременно открытых файлов.

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

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

2.3. Требования к подключаемым накопителям (USB/SD)

1. Поддерживаемый стиль разделов – [MBR](#) ([GPT](#) не поддерживается). Методика определения стиля разделов доступна по [ссылке](#).
2. Рекомендуется использовать накопители с одним [разделом](#) – в этом случае гарантируется монтирование по путям, указанным в [п. 2.4](#);
3. Поддерживаемые файловые системы – [FAT16/FAT32](#);
4. Максимальный объем накопителей – **32 Гб**;
5. Перед началом работы рекомендуется отформатировать накопитель с помощью утилиты **HP USB Disk Storage Format Tool**.

2.4. Пути к файлам и накопителям. Монтирование и размонтирование

При работе с файлами необходимо знать пути, по которым они расположены.

Контроллеры СПК работают под управлением ОС Linux. Рабочим каталогом CODESYS является `/mnt/ufs/root/CoDeSysSP_wrk`. В большинстве случаев рекомендуется сохранять пользовательские файлы именно в нем, поскольку другие каталоги содержат системные файлы и, например, могут быть защищены от записи.

Пути к накопителям выглядят следующим образом:

- для USB1: `/mnt/ufs/media/sda1`
- для USB2 (для контроллеров с двумя USB): `/mnt/ufs/media/sdb1`
- для SD: `/mnt/ufs/media/mmcblk0p1`
- рабочий каталог контроллера `/mnt/ufs/root/CoDeSysSP_wrk`

В ОС Windows (например, при использовании виртуального контроллера **CODESYS Control Win V3**) пути выглядят очевидным образом: `D:\MyFolder\MyFile.txt`

При работе с накопителями следует соблюдать два правила:

1. перед работой с накопителем необходимо проверить, [примонтирован](#) (подключен) ли он к файловой системе контроллера;
2. перед извлечением накопителя из контроллера необходимо завершить все операции с файлами и размонтировать (отключить) накопитель.

Начиная с версии **3.5.4.25**, таргет-файлы СПК содержат узел **Drives**, с помощью которого можно получить информацию о том, примонтирован ли накопитель, сколько его памяти свободно и занято, а также размонтировать накопитель. Для работы с узлом необходимо привязать переменные к его каналам. Список каналов приведен ниже.

The screenshot shows the CODESYS environment. On the left, the 'Drives' node is highlighted in the project tree. On the right, the 'Channels' table is displayed, listing various drive-related variables and their properties.

Переменная	Соотнесение	Канал	Адрес	Тип	Единица	Описание
		Drives info	%IX0.1	BIT		Включает или выключает сбор информации
		FS size	%QL1	LWORD		Размер дисковой памяти
		FS used	%QL2	LWORD		К-во занятой дисковой памяти
		FS free	%QL3	LWORD		К-во свободной дисковой памяти
		USB1 Mounted	%QX32.0	BIT		USB Flash 1 примонтировано
		USB1 Unmount	%IX0.2	BIT		Размонтировать USB Flash 1
		USB1 Unmount done	%QX32.1	BIT		Размонтирование USB Flash 1 закончено
		USB1 size	%QL5	LWORD		Размер USB Flash 1
		USB1 used	%QL6	LWORD		К-во занятой памяти на USB Flash 1
		USB1 free	%QL7	LWORD		К-во свободной памяти на USB Flash 1
		MMC Mounted	%QX64.0	BIT		MMC-карта примонтирована
		MMC Unmount	%IX0.3	BIT		Размонтировать MMC-карту
		MMC Unmount done	%QX64.1	BIT		Размонтирование MMC-карты закончено
		MMC size	%QL9	LWORD		Размер MMC-карты
		MMC used	%QL10	LWORD		К-во занятой памяти на MMC-карте
		MMC free	%QL11	LWORD		К-во свободной памяти на MMC-карте

Рис. 2.3. Каналы узла Drives

Канал	Тип	Описание
Drives info	BOOL	Бит управления сбором информации о памяти СПК и подключенных носителей. Если переменная имеет значение TRUE , то в остальных каналах каждые 5 секунд обновляется информация. При значении FALSE каналы не содержат информации.
FS size	LWORD	Размер памяти СПК в байтах.
FS used	LWORD	Количество занятой памяти СПК в байтах.
FS free	LWORD	Количество свободной памяти СПК в байтах.
USB1 Mounted	BOOL	Принимает значение TRUE после монтирования USB Flash накопителя, FALSE – при размонтировании.
USB1 Unmount	BOOL	По переднему фронту переменной происходит размонтирование USB накопителя.
USB1 Unmount done	BOOL	Принимает значение TRUE после размонтирования USB накопителя.
USB1 size	LWORD	Размер памяти USB накопителя в байтах.
USB1 used	LWORD	Количество занятой памяти USB накопителя в байтах.
USB1 free	LWORD	Количество свободной памяти USB накопителя в байтах.
MMC Mounted	BOOL	Принимает значение TRUE после монтирования MMC накопителя, FALSE – при размонтировании.
MMC Unmount	BOOL	По переднему фронту переменной происходит размонтирование MMC накопителя.
MMC Unmount done	BOOL	Принимает значение TRUE после размонтирования MMC накопителя.
MMC size	LWORD	Размер памяти MMC накопителя в байтах.
MMC used	LWORD	Количество занятой памяти MMC накопителя в байтах.
MMC free	LWORD	Количество свободной памяти MMC накопителя в байтах.

Если по каким-то причинам использовать таргет-файлы версии **3.5.4.25** (или выше) не представляется возможным, то можно воспользоваться библиотекой **CmpSysExec**, которая позволяет выполнять команды в терминале Linux. См. описание на библиотеку и описание команд Linux:

- [mount](#) (монтирование накопителей);
- [umount](#) (размонтирование накопителей);
- [df](#) (информация о занятом/свободном месте файловых разделов).

2.5. Ограничения на имена файлов и каталогов в ОС Linux

1. Максимальная длина – 255 символов.

2. Могут включать любые символы из кодировки [UTF8](#), кроме `'/'`.

3. Не рекомендуется использовать в названиях следующие символы:

`!@#$%&~%*()[]{}'"\:;><` пробел`

4. Регистр имеет принципиальное значение. `_test.txt` и `test.txt` – это два разных файла.

2.6. Бинарные и текстовые файлы

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

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

При работе с текстовыми файлами следует помнить об их [кодировке](#). Среда **CODESYS 3.5** включает два типа переменных, используемых для работы с символами (строками):

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

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

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

Для обработки строк могут использоваться готовые функции следующих библиотек:

- Standard (базовые функции для работы со **STRING**);
- Standard64 (базовые функции для работы с **WSTRING**);
- String Utils (дополнительные функции работы со строками);
- OwenStringUtils (конвертация строки из ASCII в UNICODE и обратно);
- OSCAT (дополнительные функции работы со строками).

Следует отметить, что контроллеры СПК *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**.

При архивировании строк типа **WSTRING** для корректного отображения архива в текстовом редакторе (или другом ПО) следует использовать [маркер последовательности байт](#).

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

Символ	Результат использования/Отображаемое значение
\$\$	\$ (символ доллара)
\$'	' (апостроф)
\$L	перевод строки
\$N	новая строка
\$R	возврат каретки
\$P	новая страница
\$T	табуляция
\$xx (xx – код символа в HEX)	символ таблицы ASCII (только для STRING)

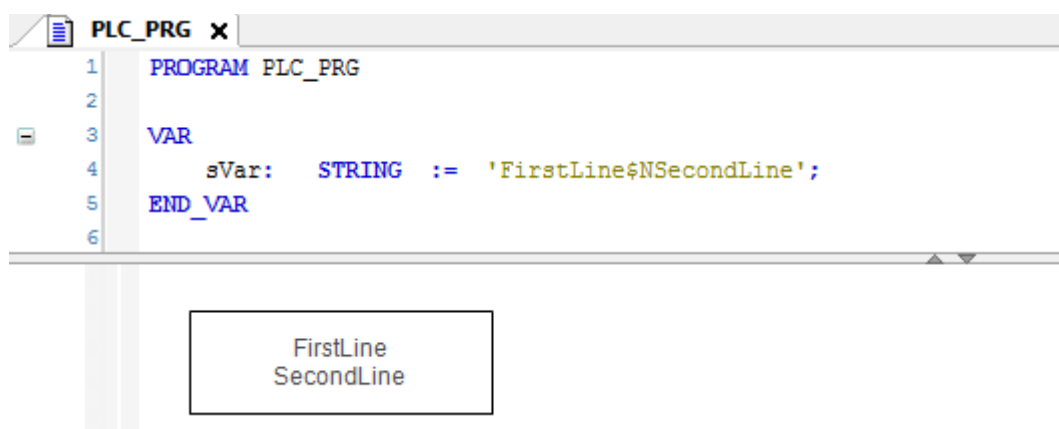


Рис. 2.4. Использование управляющих последовательностей

2.7. Обработка ошибок и некорректных ситуаций

При работе 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 размонтирования накопителя перед извлечением.
6. наличие свободного места для архива на накопителе.

2.8. Подключение к файловой системе контроллера

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

Запустите утилиту и настройте соединение по протоколу **SCP**, указав **IP-адрес** вашего контроллера и имя пользователя – **root**. Поле пароля оставьте пустым (если только ранее он не был задан средствами Linux). Нажмите **Войти**, чтобы подключиться к контроллеру.

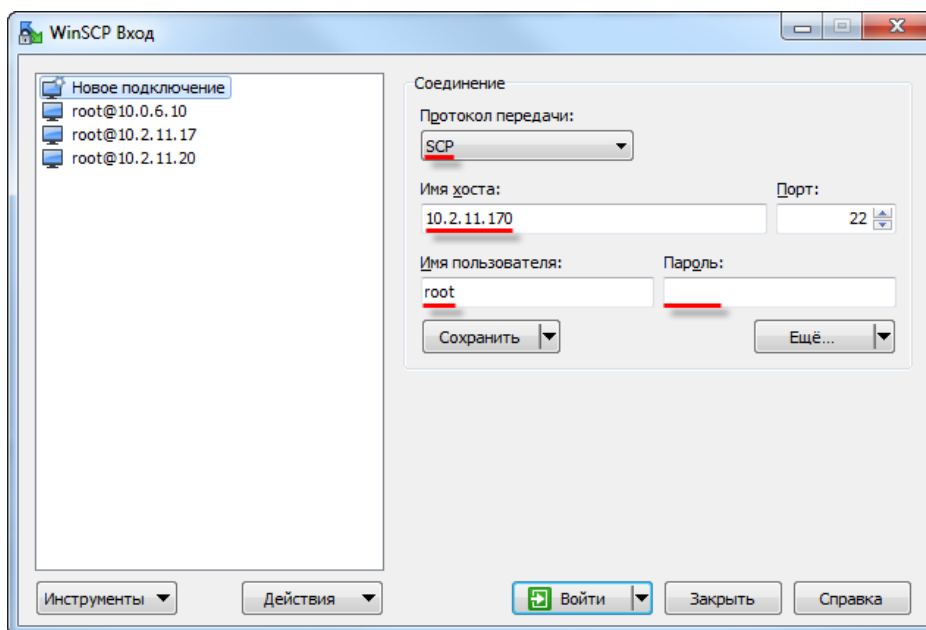


Рис. 2.5. Настройки соединения в **WinSCP**

При возникновении окна аутентификации пользователя нажмите **ОК** (поле **Пароль** следует оставить пустым).

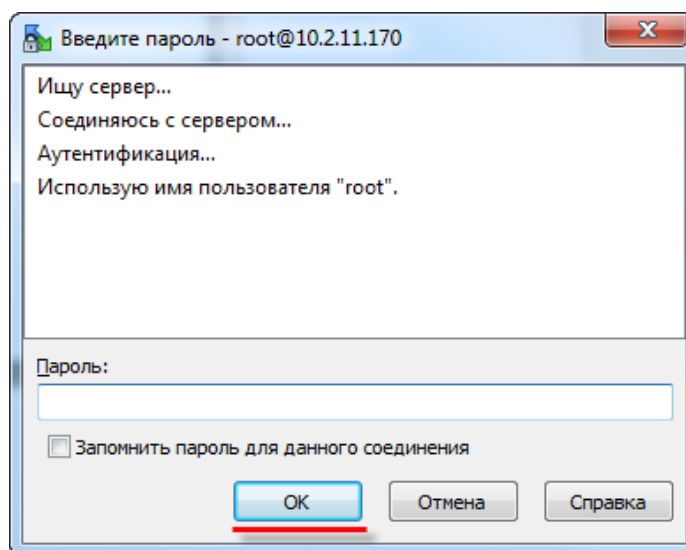


Рис. 2.6. Окно аутентификации в **WinSCP**

При возникновении сообщений типа «Не могу получить имя каталога на сервере» нажмите **ОК**.

После этого будет запущено окно файлового менеджера с интуитивно понятным интерфейсом.

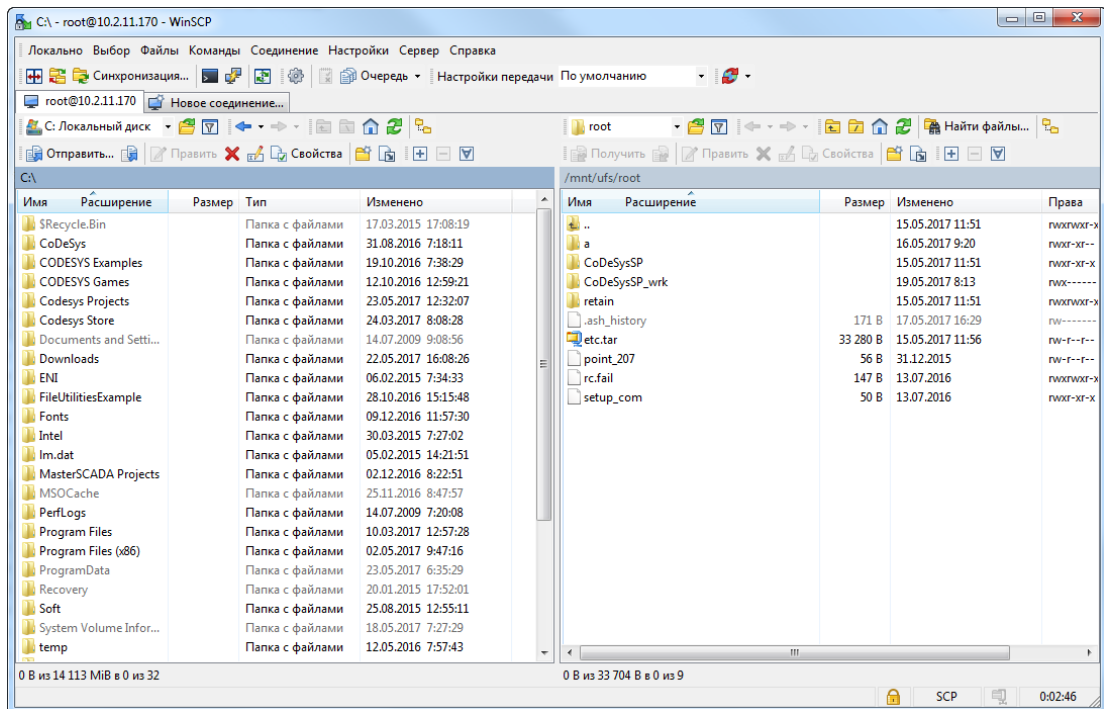


Рис. 2.7. Окно файлового менеджера WinSCP

3. Компонент OwenArchiver

3.1. Установка компонента в CODESYS

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

Для работы с компонентом нужно установить в среду программирования пакет **OwenArchiver_3.5.x.x**. В данном документе описывается работа с компонентом версии **3.5.4.7**.

Архиватор распространяется в виде пакета формата **.package**. Пакет доступен на диске с ПО, входящем в комплект поставки, а также на сайте компании [OBEH](#) в разделе **CODESYS V3/Библиотеки**.

Для установки пакета в **CODESYS** в меню **Инструменты** выберите пункт **Менеджер пакетов**, после чего укажите путь к файлу пакета и нажмите **Установить**.

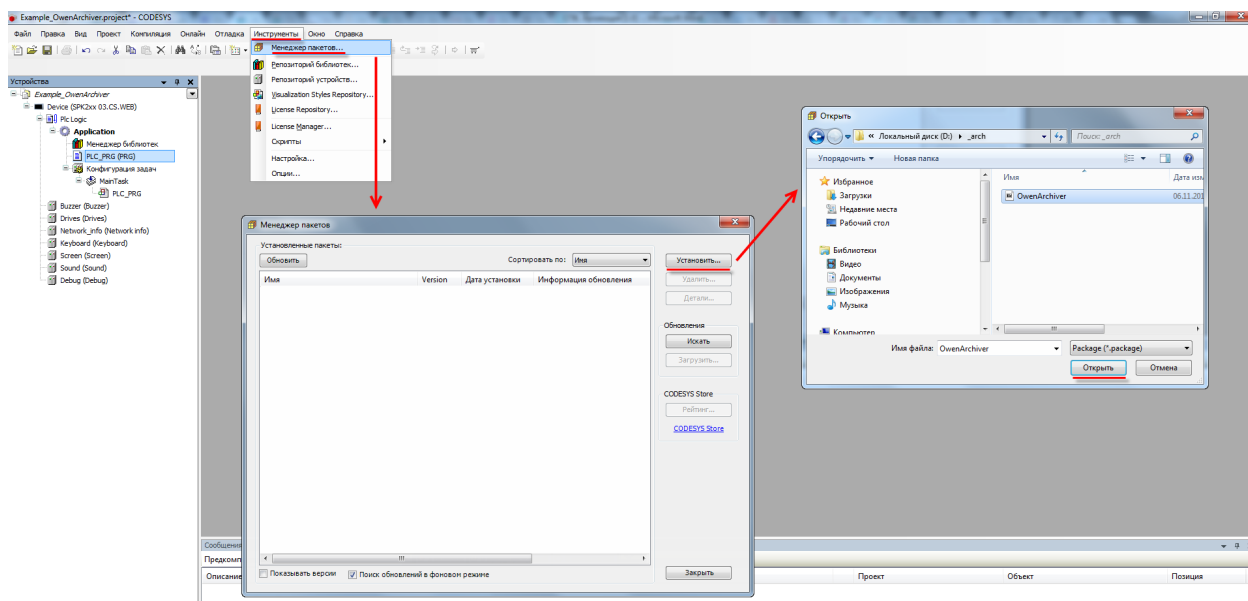


Рис. 3.1. Установка пакета **OwenArchiver** в среде **CODESYS**

В появившемся диалоговом окне выберите пункт **Полная установка**, после чего нажмите кнопку **Next**:

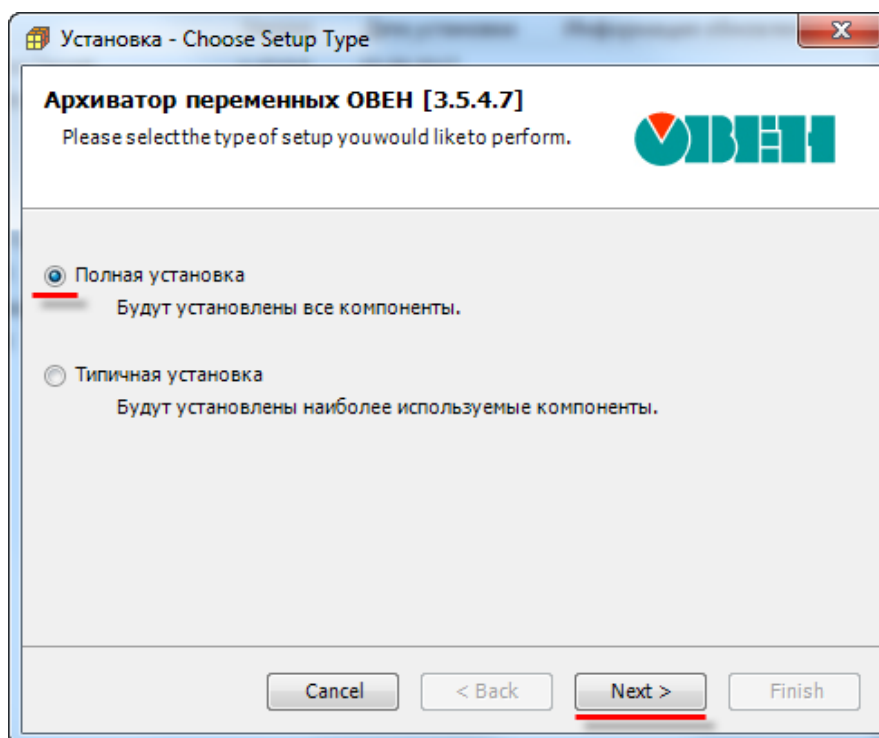


Рис. 3.2. Начало установки архиватора

После завершения установки закройте диалоговое окно с помощью кнопки **Finish**:

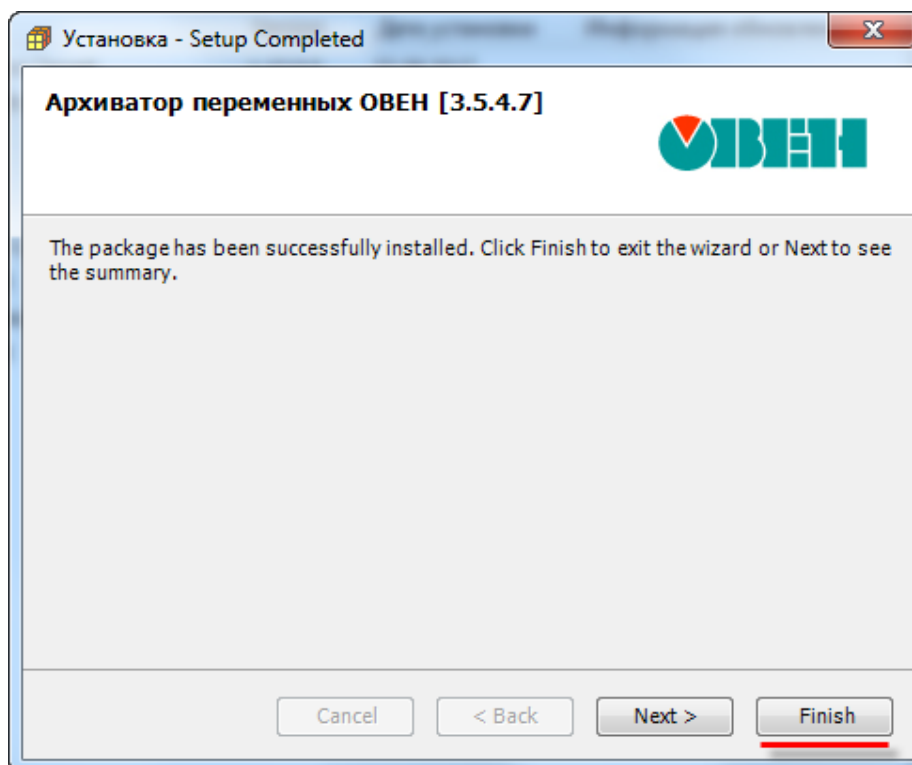


Рис. 3.3. Завершение установки архиватора

3.2. Добавление архиватора в проект

Для того чтобы добавить архиватор в проект **CODESYS**, необходимо выполнить следующие действия:

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

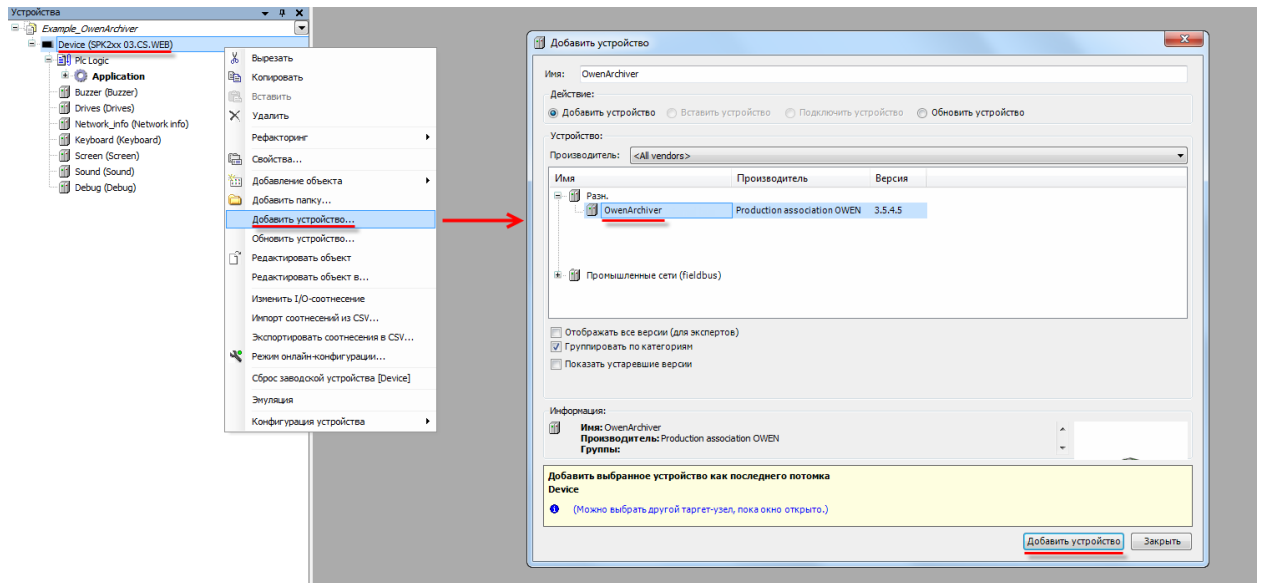


Рис. 3.4. Добавление архиватора в проект CODESYS

Обратите внимание, что при добавлении архиватора в проекте будет автоматически создана задача **OwenArchiver**. Ее не следует удалять или перенастраивать.

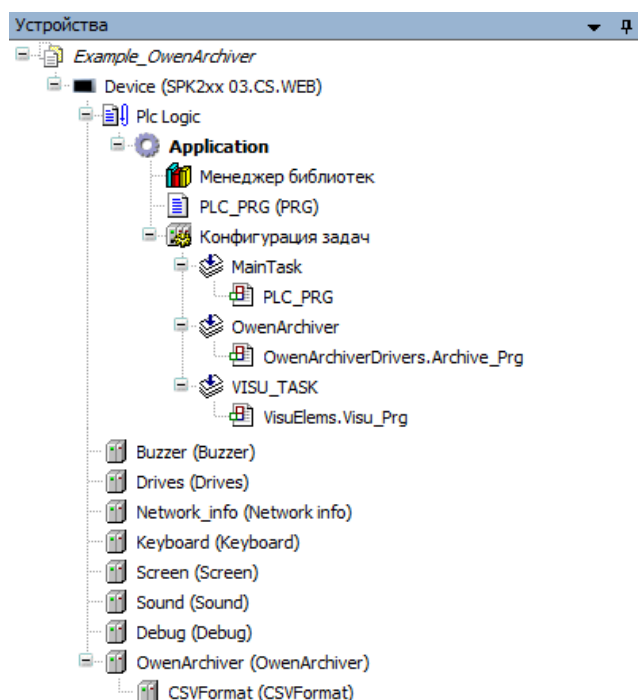


Рис. 3.5. Внешний вид дерева проекта после добавления архиватора

2. В настройках компонента **OwenArchiver** на вкладке **PCI-шина Конфигурация** укажите настройка архива:

Имя архива – должно быть уникальным в рамках проекта;

Режим архивирования – условие добавления записи в архив:

- **Периодически** – записи будут 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.

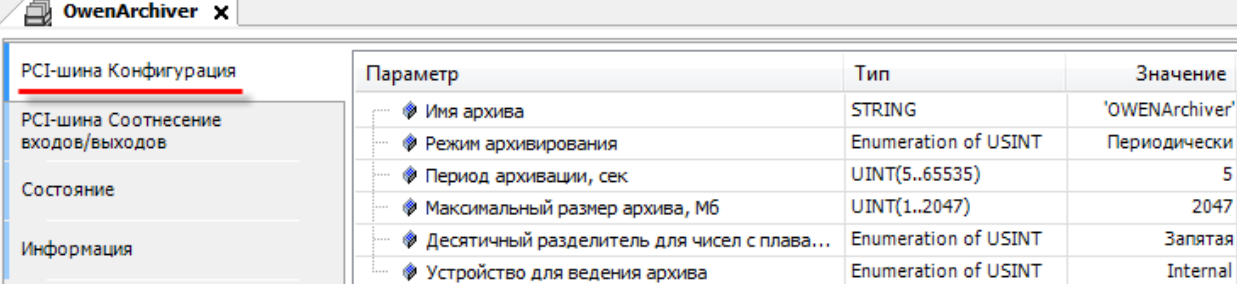
Период архивации, сек – время между двумя операциями записи в архив, минимальное значение – **5 секунд**;

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:

- **Internal** – архив будет вестись во внутреннюю память контроллера (в папку `/mnt/ufs/root/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>`).



Параметр	Тип	Значение
Имя архива	STRING	'OWENArchiver'
Режим архивирования	Enumeration of USINT	Периодически
Период архивации, сек	UINT(5..65535)	5
Максимальный размер архива, Мб	UINT(1..2047)	2047
Десятичный разделитель для чисел с плава...	Enumeration of USINT	Запятая
Устройство для ведения архива	Enumeration of USINT	Internal

Рис. 3.6. Настройки архиватора, вкладка **PCI-шина Конфигурация**

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

Название канала	Тип	Описание
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 происходит добавление новой записи в архив (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 файл /mnt/ufs/root/<имя_архива>.txt). Лог содержит список всех операций, производимых архиватором.
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 при отладке сложных проектах с высокой частотой архивации большого количества данных.
Размер архива	REAL	Размера архив в мегабайтах.

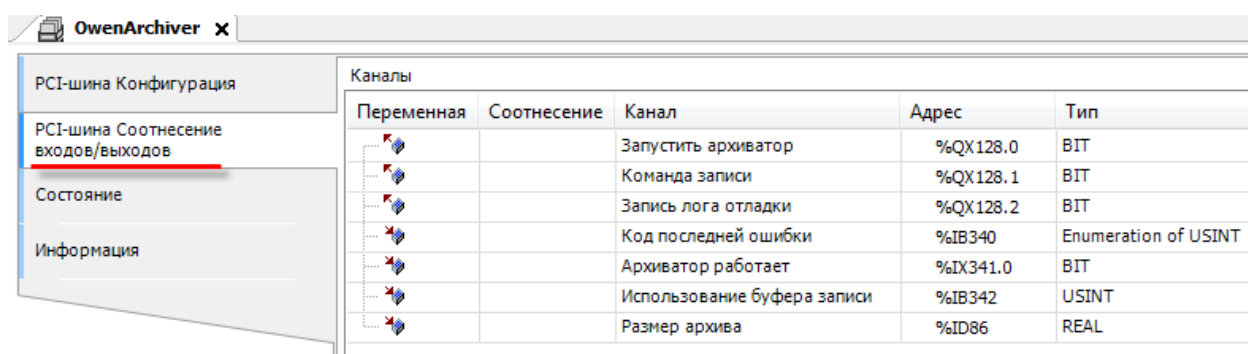


Рис. 3.7. Настройки архиватора, вкладка **PCI-шина Соотнесение входов/выходов**

4. В настройках компонента **CSVFormat** на вкладке **CSVFormat Конфигурация** выберите 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*).

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*).

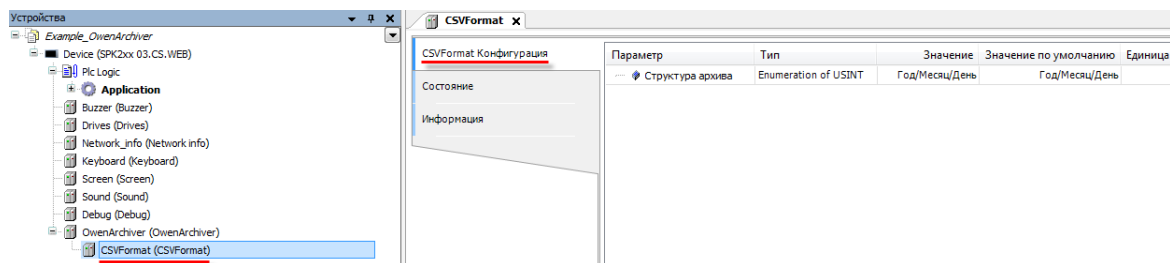


Рис. 3.7. Настройки архиватора, вкладка **CSVFormat Конфигурация**

5. Нажмите **ПКМ** на компонент **OwenArchiver** и добавьте каналы переменных нужных типов. Всего архиватор может содержать до 64-х каналов.

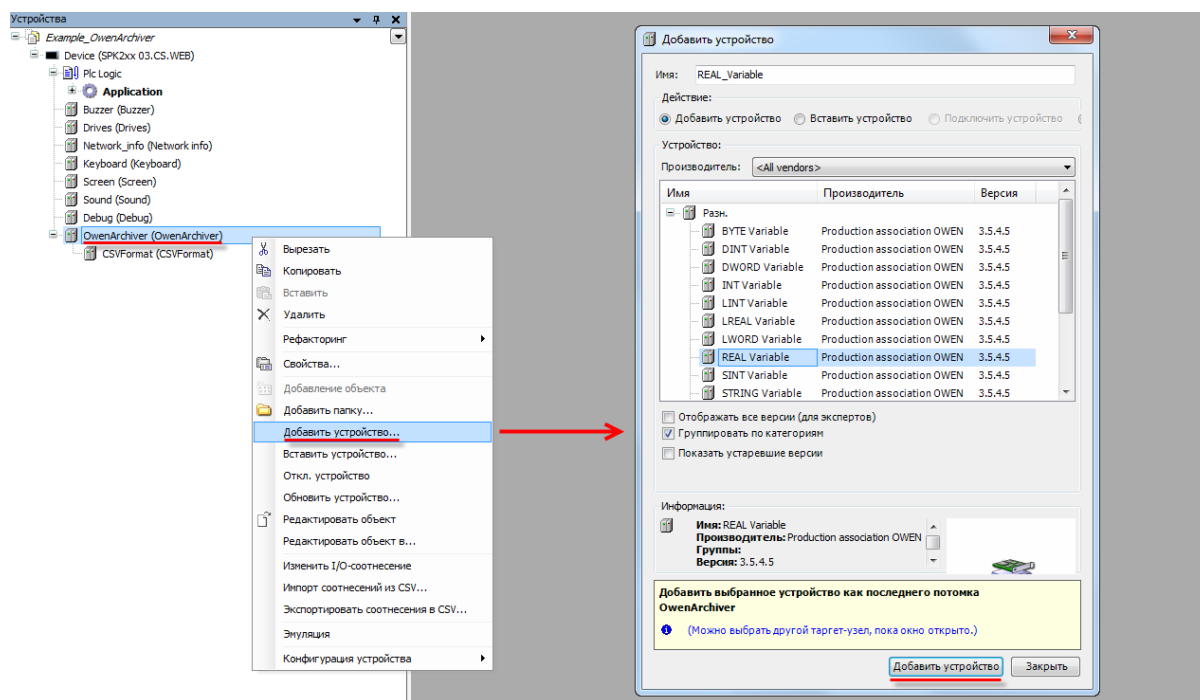


Рис. 3.8. Добавление каналов архивируемых переменных

В настройках модуля на вкладке **ArchiverVariable Конфигурация** укажите:

Описание переменной – используется при формировании заголовка архива;

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

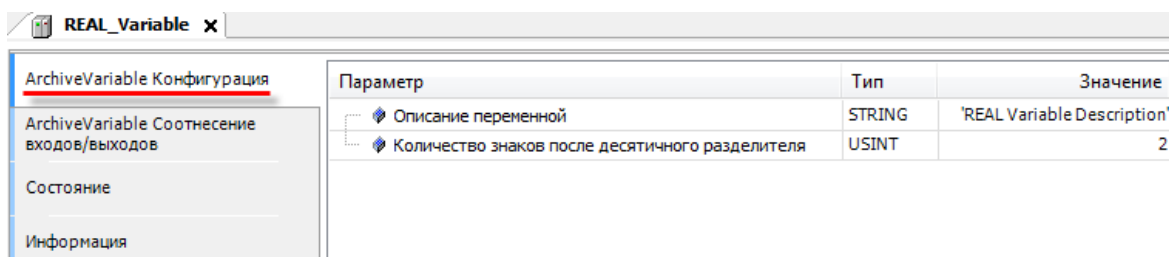


Рис. 3.9. Настройки канала архивируемой переменной

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

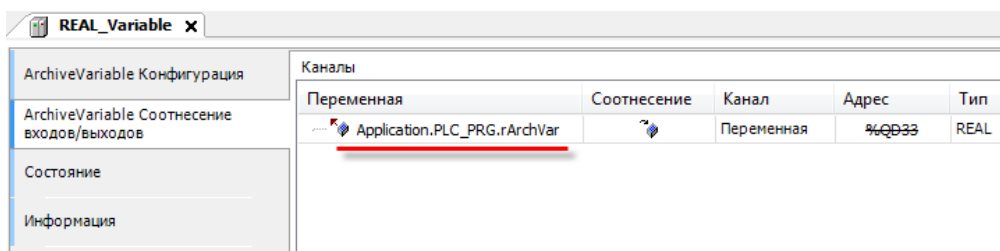


Рис. 3.10. Привязка архивируемой переменной к каналу

3.3. Ограничения, связанные с использованием архиватора

1. Максимальное количество переменных для одного архиватора – **64**.

2. В проекте может использоваться несколько архиваторов, но они должны работать с разными файлами. Максимально допустимое число одновременно работающих архиваторов – **2**. Использование большего количества одновременно запущенных операторов может привести к значительной нагрузке на процессор и высоким затратам оперативной памяти – корректная работа контроллера в данном случае не гарантируется.

3. Архиватор использует **память ввода-вывода CODESYS**. Ее количество ограничено (например, для контроллеров СПК1хх и СП207 ограничение составляет 5000 байт на входы и 5000 байт на выходы). Эта область также используется компонентами Modbus, системными узлами таргет-файла (например, **Buzzer**) и средой CODESYS. При превышении доступного объема памяти при компиляции проекта возникнут соответствующие ошибки.

4. Архиватор не контролирует объем доступной памяти контроллера и подключенных накопителей. Пользователь может реализовать этот функционал самостоятельно (например, остановку архиватора при исчерпании памяти) с помощью каналов системного узла **Drives**.

5. Для архивируемых строковых переменных максимальный размер составляет **80** символов (**STRING(80)**).

3.4. Пример работы с архиватором

Рассмотрим пример работы с архиватором.

Пример создан в среде **CODESYS 3.5 SP7 Patch4** и подразумевает запуск на **СПК207.03.CS(-WEB)**.

Пример доступен для скачивания: [Example_OwenArchiver.projectarchive](#)

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

```
1 PROGRAM PLC_PRG
2 VAR
3   (*архивируемые переменные*)
4   wArchVar:    WORD;
5   rArchVar:    REAL;
6   sArchVar:    STRING;
7
8   (*переменные архиватора*)
9   xArchEnable:  BOOL;           // бит управления архиватором
10  eArchError:   OwenArchiverErrors.ARCHIVE_ERROR; // код ошибки
11  wsArchError:  WSTRING;       // сообщение об ошибке
12  xArchStatus:  BOOL;           // статус архиватора (TRUE - запущен, FALSE - остановлен)
13 END_VAR
14
15 // конвертируем код ошибки архиватора в строку
16
17 wsArchError:=OwenArchiverErrors.ARCHIVE_ERROR_TO_WSTRING(eArchError);
18
```

Рис. 3.11. Объявление переменных и код программы **PLC_PRG**

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

2. Нажмите ПКМ на компонент Device и добавьте компонент OwenArchiver:

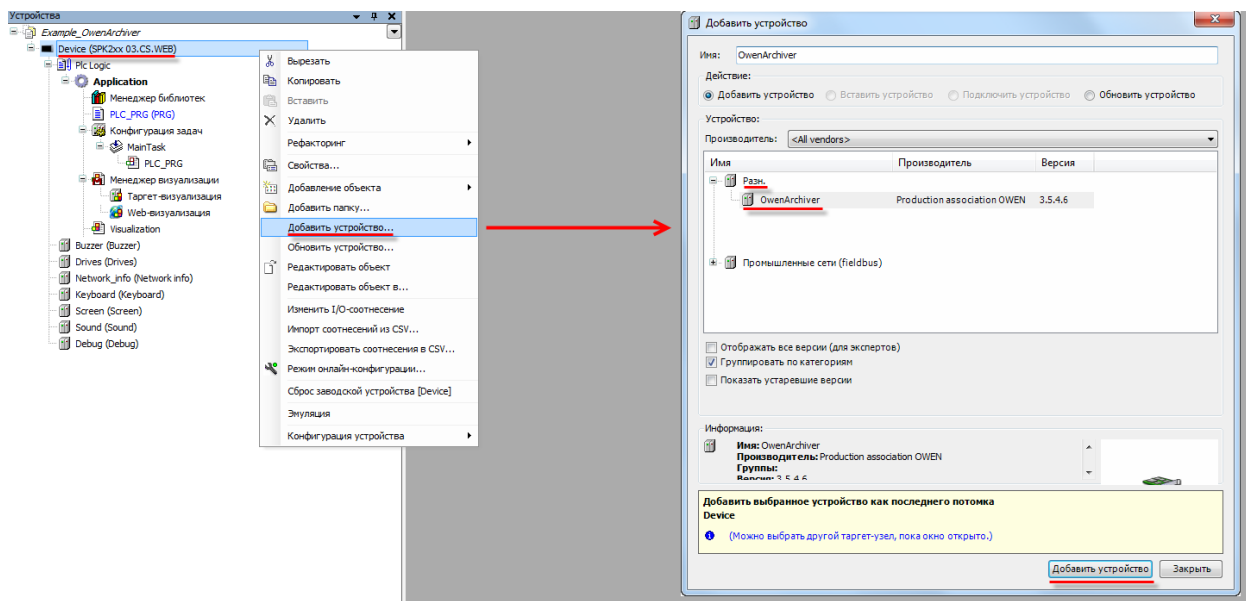


Рис. 3.12. Добавление компонента OwenArchiver

Обратите внимание, что после этого в проект будет автоматически добавлена задача OwenArchiver. Ее не следует удалять или перенастраивать.

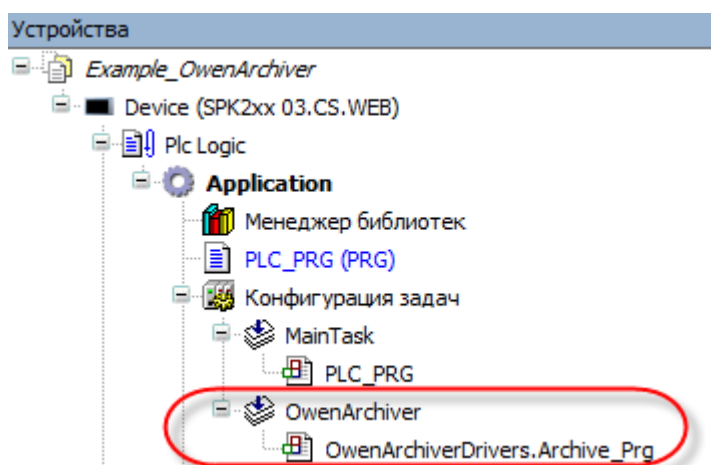


Рис. 3.13. Внешний вид дерева проекта после добавления архиватора

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

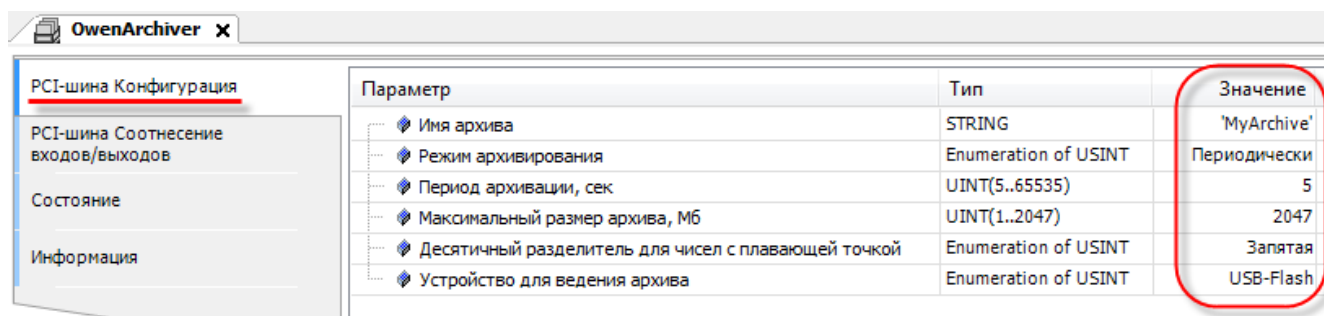


Рис. 3.14. Настройка параметров архивации

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

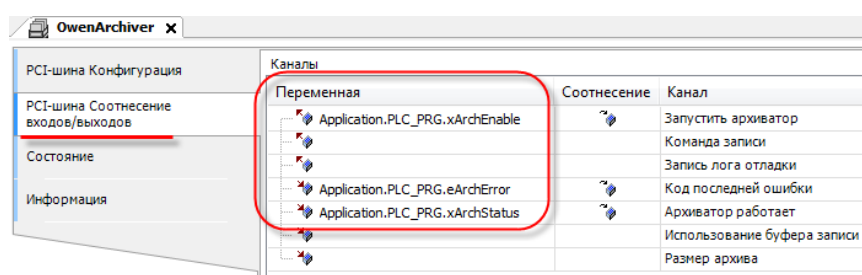


Рис. 3.15. Привязка переменных контроля архиватора

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

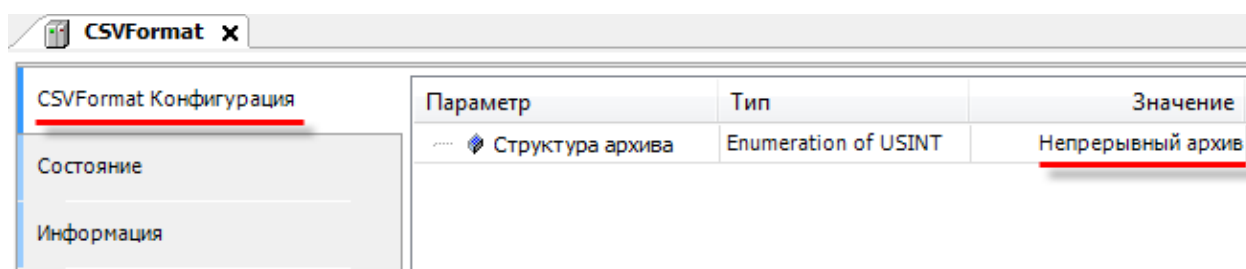


Рис. 3.16. Выбор режима архивации

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. При достижении максимального размера этого файла – файл **<имя_архива_old>** будет удален, текущий файл (**<имя_архива>**) будет переименован в **<имя_архива_old>** и будет создан новый файл (**<имя_архива>**), в который продолжит вестись архивация. Таким образом, в каждый момент времени будет существовать два файла архива – текущий и предыдущий.

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

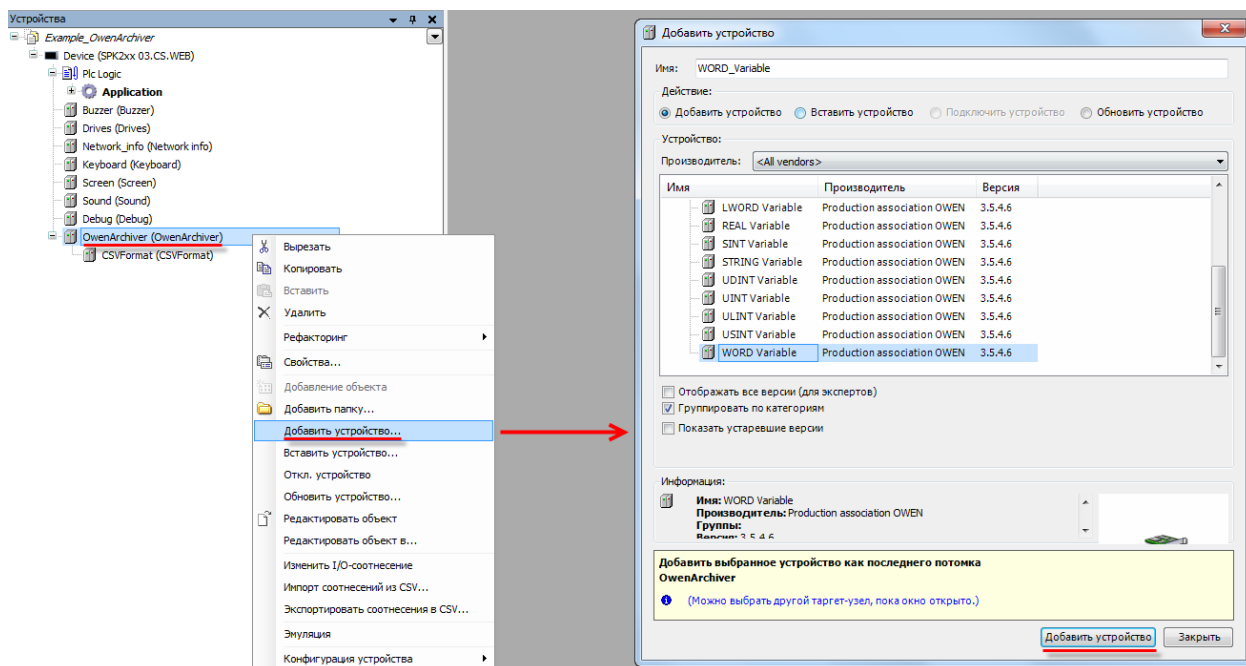


Рис. 3.17. Добавление каналов архивации

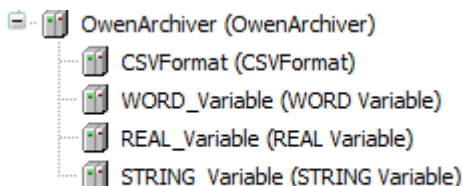


Рис. 3.18. Внешний вид дерева проекта после добавления каналов архивации

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

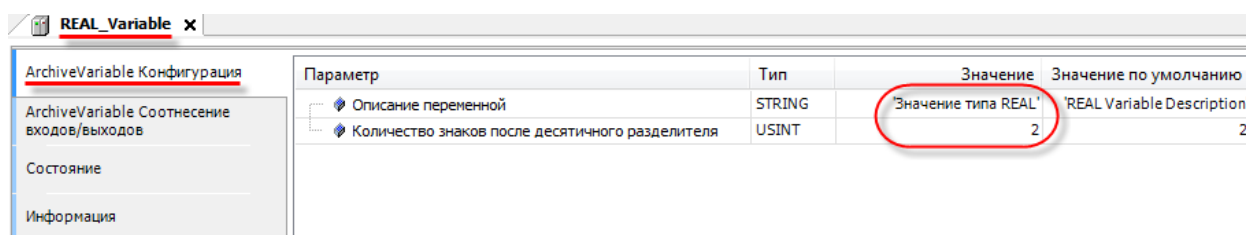


Рис. 3.19. Настройка канала архивации

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

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

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

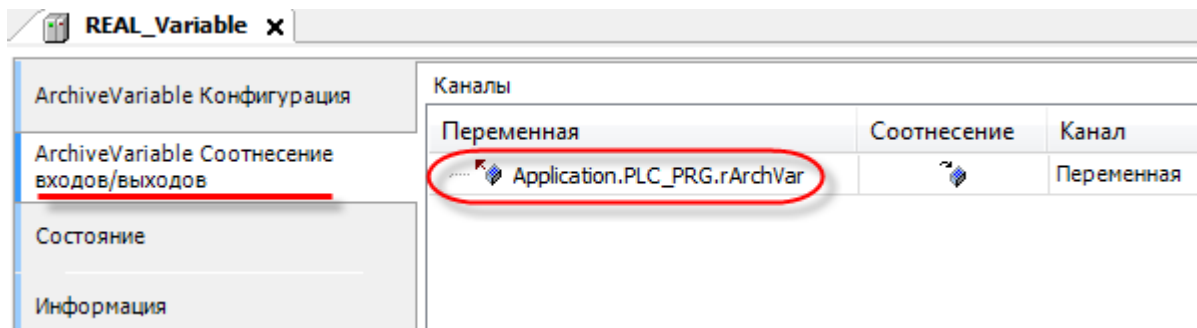


Рис. 3.20. Привязка переменных к каналам архивации

6. Создадим интерфейс оператора. Мы не будем подробно останавливаться на процессе разработки визуализации (вся необходимая информация приведена в документе **СПК. Визуализация**), сделав акцент только на ключевых моментах. На рис. 3.21 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**.



Рис. 3.21. Внешний вид экрана визуализации

7. Загрузите проект в контроллер и запустите его. Нажмите переключатель **Управление архиватором**, чтобы запустить архивацию. Измените значения архивируемых переменных.

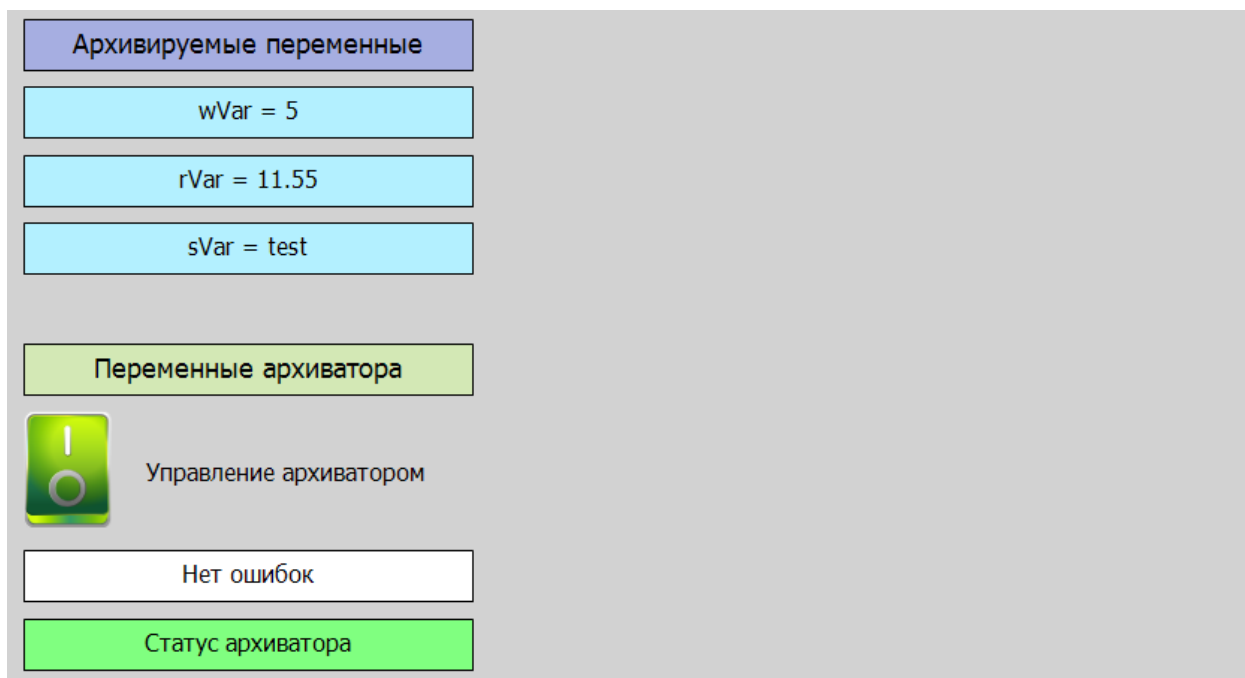


Рис. 3.22. Внешний вид экрана визуализации

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

	A	B	C	D
1	MyArchive			
2	Дата	Значение типа WORD	Значение типа REAL	Значение типа STRING
3	14.12.2017 15:48		0 0,00	
4	14.12.2017 15:48		0 0,00	
5	14.12.2017 15:48		0 0,00	
6	14.12.2017 15:48		0 0,00	
7	14.12.2017 15:48		0 0,00	
8	14.12.2017 15:48		5 0,00	
9	14.12.2017 15:48		5 11,55	
10	14.12.2017 15:48		5 11,55	test
11	14.12.2017 15:48		5 11,55	test
12	14.12.2017 15:49		5 11,55	test
13	14.12.2017 15:49		5 11,55	test

Рис. 3.23. Фрагмент архива

В памяти контроллера в папке **/mnt/ufs/root** будет создан файл **MyArchive.txt**, содержащий лог операций архиватора (запись лога была включена на рис. 3.14).

9. Рекомендуем ознакомиться с примером [получения информации о накопителях](#) – это поможет определять свободный/занятый объем (и при необходимости останавливать архивацию), определять, примонтирован ли накопитель, размонтировать его и др.

4. Библиотека CAA File

4.1. Добавление библиотеки в проект CODESYS

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

Библиотека реализует [асинхронный доступ](#) к файлам – в связи с этим выполнение блоков может занять несколько циклов, но при этом 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** нажмите кнопку **Добавить** и выберите библиотеку **CAA File**, расположенную в папке **Intern/CAA/System**.

Обратите внимание, что версия библиотеки не должна превышать версию таргет-файла контроллера. В противном случае корректная работа контроллера не гарантируется.

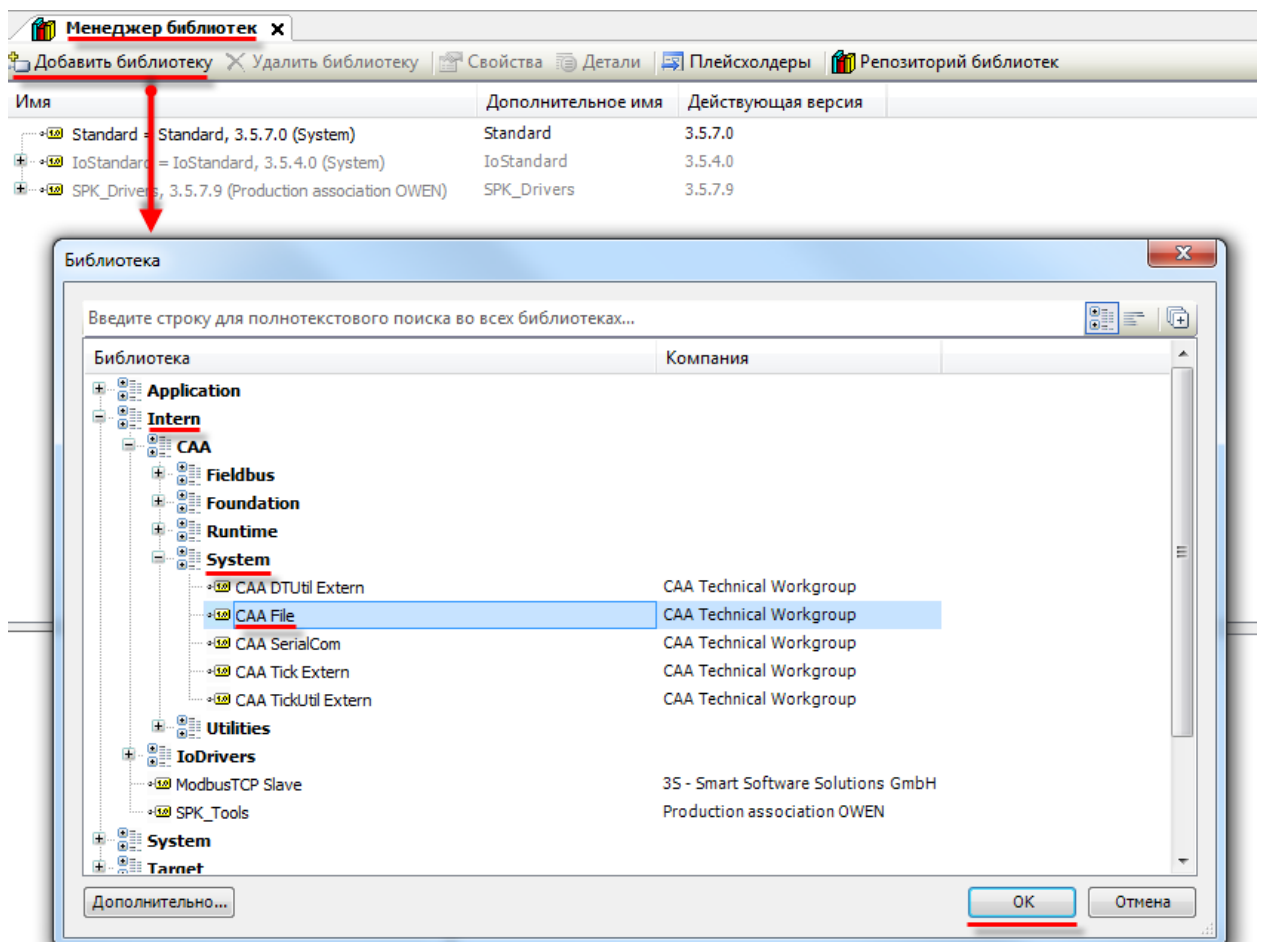


Рис. 4.1. Добавление библиотеки **CAA File** в проект **CODESYS**

Обратите внимание, что при объявлении экземпляров ФБ библиотеки необходимо перед их названием указывать префикс FILE. (пример: **FILE.Open**).

4.2. Структуры и перечисления

4.2.1. Структура FILE.FILE_DIR_ENTRY

Структура **FILE.FILE_DIR_ENTRY** описывает параметры каталога/файла и используется при работе с ФБ [FILE.DirList](#).

Название	Тип данных	Описание
sEntry	CAA.FILENAME	Имя каталога или файла.
szSize	CAA.SIZE	Размер каталога/файла в байтах. <i>В версии библиотеки 3.5.3.0 и ниже некорректно определяется размер каталогов.</i>
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. <i>В версии библиотеки 3.5.3.0 и ниже некорректно определяется дата и время последнего изменения каталогов.</i>

4.2.2. Перечисление FILE.ERROR

Перечисление **FILE.ERROR** описывает ошибки, которые могут возникнуть при использовании ФБ библиотеки.

Название	Значение	Описание
NO_ERROR	0	Нет ошибок.
TIME_OUT	5100	Истек лимит времени для данной операции.
ABORT	5101	Операция была прервана с помощью входа xAbort .
HANDLE_INVALID	5103	Некорректный дескриптор файла.
NOT_EXIST	5104	Каталог или файл не существуют.
EXIST	5105	Каталог или файл уже существуют.
NO_MORE_ENTRIES	5106	Получена информация о всех вложенных элементах.
NOT_EMPTY	5107	Каталог или файл не являются пустыми.
READ_ONLY_CAA	5108	Каталог или файл защищены от записи.
WRONG_PARAMETER	5109	ФБ вызван с неверными аргументами.
WRITE_INCOMPLETE	5111	Запись в файл не была завершена (возможна потеря данных).
NOT_IMPLEMENTED	5112	Операция не поддерживается устройством.

4.2.3. Перечисление FILE.MODE

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

Название	Значение	Описание
MWRITE	0	Запись (файл будет перезаписан или создан).
MREAD	1	Чтение (существующий файл будет открыт для чтения).
MRDWR	2	Чтение/запись (файл будет перезаписан или создан).
MAPPD	3	Дозапись (существующий файл будет открыт в режиме записи, данные будут дописаны в конец файла).

4.3. Пути к каталогам и файлам

При использовании ФБ библиотеки в значительном числе случаев необходимо указывать путь к каталогу или файлу, над которым будет производиться операция. Напомним, что общая информация о путях в Linux и ограничениях для их названий приведена в [п. 2.4](#) и [п. 2.5](#) соответственно.

При работе с библиотекой можно указывать как относительные, так и абсолютные пути. При этом рабочим каталогом CODESYS в СПК является **/mnt/ufs/root/CoDeSysSP_wrk**.

Поясним это на примере. ФБ [File.DirCreate](#) создает новый каталог по пути **sDirName**.

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

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

4.4. Ограничения при работе с файлами

Максимальное количество одновременно выполняемых операций (открытие, чтение, запись и др.) с каталогами и файлами не должно превышать **20-ти** (по возможности рекомендуется в каждый момент времени работать только с одним файлом). Операция считается незавершенной, пока вход **xExecute** имеет значение **TRUE** – поэтому рекомендуется запускать работу блоков с помощью единичных импульсов по переднему фронту.

4.5. ФБ работы с каталогами

4.5.1. ФБ FILE.DirCreate

Функциональный блок **FILE.DirCreate** создает новый каталог. Без указания полного пути каталог создается внутри рабочего. Для контроллеров СПК рабочим является каталог `/mnt/ufs/root/CoDeSysSP_wrk`.

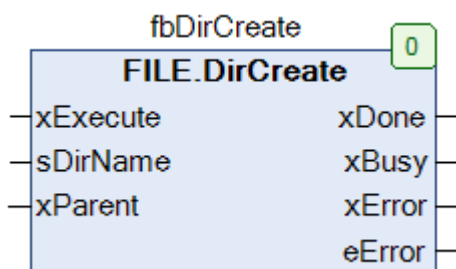


Рис. 4.2. Внешний вид ФБ **FILE.DirCreate** на языке CFC

Название	Тип данных	Описание
Входные переменные		
xExecute	BOOL	Переменная активации блока. Запуск блока происходит по <u>переднему фронту</u> переменной.
sDirName	STRING	Имя (или полный путь) создаваемого каталога. См. п. 2.4 , п. 2.5 и п. 4.3 .
xParent	BOOL	Режим рекурсивного создания каталогов. TRUE – все несуществующие каталоги, указанные в пути, создаются автоматически FALSE – если в пути указано более одного несуществующего каталога, то блок завершает работу с сообщением об ошибке <i>В версии библиотеки 3.5.3.0 и ниже не поддерживается рекурсивное создание каталогов.</i>
Выходные переменные		
xDone	BOOL	Флаг успешного завершения работы блока. Принимает значение TRUE на один цикл.
xBusy	BOOL	Флаг «ФБ в процессе работы».
xError	BOOL	Флаг ошибки. Принимает значение TRUE при возникновении ошибки.
eError	FILE.ERROR	Статус работы ФБ (или имя ошибки).

4.5.2. ФБ FILE.DirOpen

Функциональный блок **FILE.DirOpen** открывает каталог и возвращает его дескриптор (**handle**). Это необходимо для последующего использования ФБ [File.DirList](#).

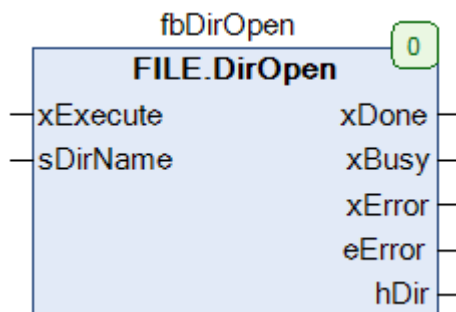


Рис. 4.3. Внешний вид ФБ **FILE.DirOpen** на языке **CFC**

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

4.5.3. ФБ FILE.DirList

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

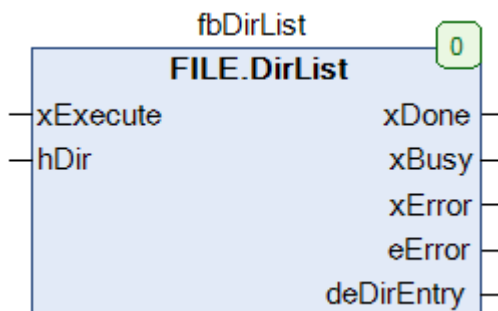


Рис. 4.4. Внешний вид ФБ **FILE.DirList** на языке **CFC**

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

4.5.4. ФБ FILE.DirRemove

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

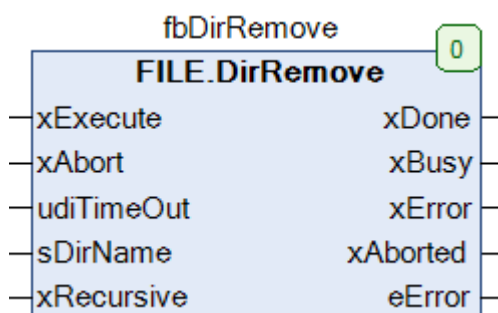


Рис. 4.5. Внешний вид ФБ **FILE.DirRemove** на языке CFC

Название	Тип данных	Описание
Входные переменные		
xExecute	BOOL	Переменная активации блока. Запуск блока происходит по <u>переднему фронту</u> переменной.
xAbort	BOOL	Переменная прерывания работы блока. Прерывание происходит по <u>переднему фронту</u> переменной.
udiTimeOut	UDINT	Допустимое время операции (в мкс). Значение 0 означает, что время выполнения ФБ не ограничивается.
sDirName	STRING	Имя (или полный путь) удаляемого каталога. См. п. 2.4 , п. 2.5 и п. 4.3 .
xRecursive	BOOL	Режим рекурсивного удаления каталогов. TRUE – каталог удаляется вместе со всем содержимым FALSE – каталог удаляется только в том случае, если является пустым, в противном случае ФБ возвращает сообщение об ошибке <i>В версии библиотеки 3.5.3.0 и ниже поддерживается удаление только пустых каталогов.</i>
Выходные переменные		
xDone	BOOL	Флаг успешного завершения работы блока. Принимает значение TRUE на один цикл.
xBusy	BOOL	Флаг «ФБ в процессе работы».
xError	BOOL	Флаг ошибки. Принимает значение TRUE при возникновении ошибки.
xAborted	BOOL	Флаг «прервано пользователем».
eError	FILE.ERROR	Статус работы ФБ (или имя ошибки).

4.5.5. ФБ FILE.DirRename

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

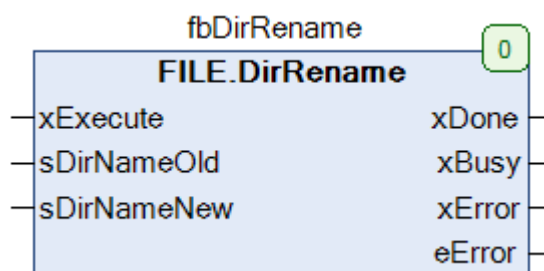


Рис. 4.6. Внешний вид ФБ **FILE.DirRename** на языке CFC

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

4.5.6. ФБ FILE.DirClose

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

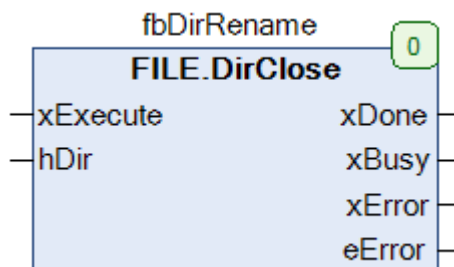


Рис. 4.7. Внешний вид ФБ **FILE.DirClose** на языке **CFC**

Название	Тип данных	Описание
Входные переменные		
xExecute	BOOL	Переменная активации блока. Запуск блока происходит по <u>переднему фронту</u> переменной.
hDir	FILE.CAA.HANDLE	Дескриптор закрываемого каталога.
Выходные переменные		
xDone	BOOL	Флаг успешного завершения работы блока. Принимает значение TRUE на один цикл.
xBusy	BOOL	Флаг «ФБ в процессе работы».
xError	BOOL	Флаг ошибки. Принимает значение TRUE при возникновении ошибки.
eError	FILE.ERROR	Статус работы ФБ (или имя ошибки).

4.6. ФБ работы с файлами

4.6.1. ФБ FILE.Open

Функциональный блок **FILE.Open** открывает файл и возвращает его дескриптор (**handle**), который используется для всех остальных операций с файлом. После окончания работы с файлом необходимо закрыть его с помощью ФБ [FILE.CLOSE](#). **Обратите внимание**, что попытка открытия ранее открытого (и не закрытого) файла может привести к ошибкам в работе контроллера.

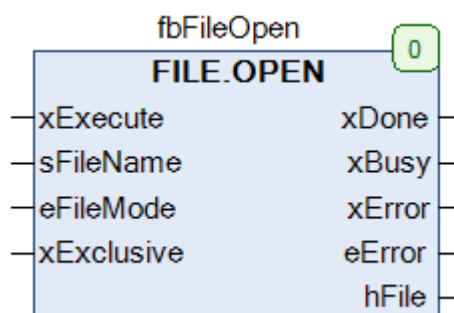


Рис. 4.8. Внешний вид ФБ FILE.OPEN на языке CFC

Название	Тип данных	Описание
Входные переменные		
xExecute	BOOL	Переменная активации блока. Запуск блока происходит по <u>переднему фронту</u> переменной.
sFileName	STRING	Имя (или полный путь) открываемого файла. См. п. 2.4 , п. 2.5 и п. 4.3 .
eFileMode	FILE.MODE	Режим открытия файла.
xExclusive	BOOL	Тип доступа к открываемому файлу. TRUE – монопольный FALSE – многопользовательский <i>В версии библиотеки 3.5.3.0 и ниже не поддерживается открытие файлов в режиме монопольного доступа.</i>
Выходные переменные		
xDone	BOOL	Флаг успешного завершения работы блока. Принимает значение TRUE на один цикл.
xBusy	BOOL	Флаг «ФБ в процессе работы».
xError	BOOL	Флаг ошибки. Принимает значение TRUE при возникновении ошибки.
eError	FILE.ERROR	Статус работы ФБ (или имя ошибки).
hFile	FILE.CAA.HANDLE	Дескриптор открытого файла.

4.6.2. ФБ FILE.Close

Функциональный блок **FILE.Close** используется для закрытия файла после выполнения необходимых операций. **Обратите внимание**, что попытка закрытия ранее закрытого файла (или еще не открытого файла) может привести к ошибкам в работе контроллера.

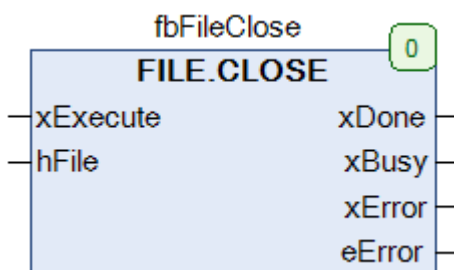


Рис. 4.9. Внешний вид ФБ **FILE.Close** на языке **CFC**

Название	Тип данных	Описание
Входные переменные		
xExecute	BOOL	Переменная активации блока. Запуск блока происходит по <u>переднему фронту</u> переменной.
hFile	FILE.CAA.HANDLE	Дескриптор файла.
Выходные переменные		
xDone	BOOL	Флаг успешного завершения работы блока. Принимает значение TRUE на один цикл.
xBusy	BOOL	Флаг «ФБ в процессе работы».
xError	BOOL	Флаг ошибки. Принимает значение TRUE при возникновении ошибки.
eError	FILE.ERROR	Статус работы ФБ (или имя ошибки).

4.6.3. ФБ FILE.Write

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

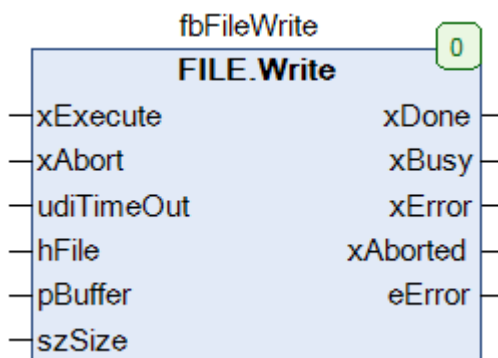


Рис. 4.10. Внешний вид ФБ **FILE.Write** на языке CFC

Название	Тип данных	Описание
Входные переменные		
xExecute	BOOL	Переменная активации блока. Запуск блока происходит по <u>переднему фронту</u> переменной.
xAbort	BOOL	Переменная прерывания работы блока. Прерывание происходит по <u>переднему фронту</u> 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. Принимает значение TRUE на один цикл.
xBusy	BOOL	Флаг «ФБ в процессе работы».
xError	BOOL	Флаг ошибки. Принимает значение TRUE при возникновении ошибки.
xAborted	BOOL	Флаг «прервано пользователем».
eError	FILE.ERROR	Статус работы ФБ (или имя ошибки).

4.6.4. ФБ FILE.Read

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

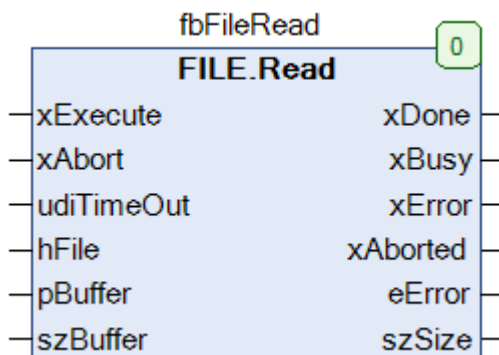


Рис. 4.11. Внешний вид ФБ **FILE.Read** на языке **CFC**

Название	Тип данных	Описание
Входные переменные		
xExecute	BOOL	Переменная активации блока. Запуск блока происходит по <u>переднему фронту</u> переменной.
xAbort	BOOL	Переменная прерывания работы блока. Прерывание происходит по <u>переднему фронту</u> 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. Принимает значение TRUE на один цикл.
xBusy	BOOL	Флаг «ФБ в процессе работы».
xError	BOOL	Флаг ошибки. Принимает значение TRUE при возникновении ошибки.
xAborted	BOOL	Флаг «прервано пользователем».
eError	FILE.ERROR	Статус работы ФБ (или имя ошибки).
szSize	CAA.SIZE	Размер считанных данных в байтах.

4.6.5. ФБ FILE.Rename

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

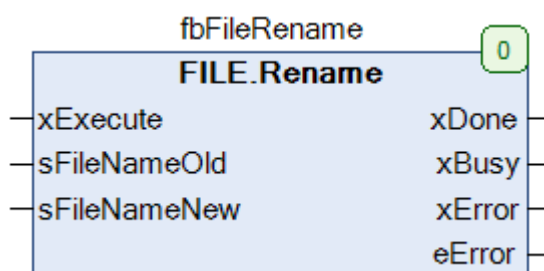


Рис. 4.12. Внешний вид ФБ **FILE.Rename** на языке CFC

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

4.6.6. ФБ FILE.Copy

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

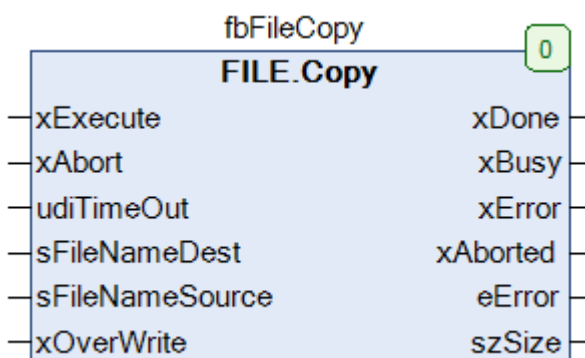


Рис. 4.13. Внешний вид ФБ **FILE.Copy** на языке **CFC**

Название	Тип данных	Описание
Входные переменные		
xExecute	BOOL	Переменная активации блока. Запуск блока происходит по <u>переднему фронту</u> переменной.
xAbort	BOOL	Переменная прерывания работы блока. Прерывание происходит по <u>переднему фронту</u> переменной.
udiTimeOut	UDINT	Допустимое время операции (в мкс). Значение 0 означает, что время выполнения ФБ не ограничивается.
sFileNameDest	STRING	Имя (или полный путь) копии файла. См. п. 2.4 , п. 2.5 и п. 4.3 .
sFileNameSource	STRING	Имя (или полный путь) исходного файла. См. п. 2.4 , п. 2.5 и п. 4.3 .
xOverWrite	BOOL	Обработка ситуации «файл с таким именем уже существует». TRUE – файл будет перезаписан FALSE – файл не будет перезаписан, блок выдаст сообщение об ошибке <i>В версии библиотеки 3.5.3.0 и ниже не поддерживается вызов ФБ с значением FALSE на данном входе.</i>
Выходные переменные		
xDone	BOOL	Флаг успешного завершения работы блока. Принимает значение TRUE на один цикл.
xBusy	BOOL	Флаг «ФБ в процессе работы».
xError	BOOL	Флаг ошибки. Принимает значение TRUE при возникновении ошибки.
xAborted	BOOL	Флаг «прервано пользователем».
eError	FILE.ERROR	Статус работы ФБ (или имя ошибки).
szSize	CAA.FILE.SIZE	Размер скопированных данных в байтах.

4.6.7. ФБ FILE.Delete

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

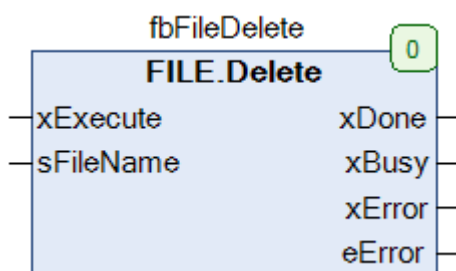


Рис. 4.14. Внешний вид ФБ **FILE.Delete** на языке **CFC**

Название	Тип данных	Описание
<i>Входные переменные</i>		
xExecute	BOOL	Переменная активации блока. Запуск блока происходит по <u>переднему фронту</u> переменной.
sFileName	STRING	Имя удаляемого файла.
<i>Выходные переменные</i>		
xDone	BOOL	Флаг успешного завершения работы блока. Принимает значение TRUE на один цикл.
xBusy	BOOL	Флаг «ФБ в процессе работы».
xError	BOOL	Флаг ошибки. Принимает значение TRUE при возникновении ошибки.
eError	FILE.ERROR	Статус работы ФБ (или имя ошибки).

4.6.8. ФБ FILE.Flush

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

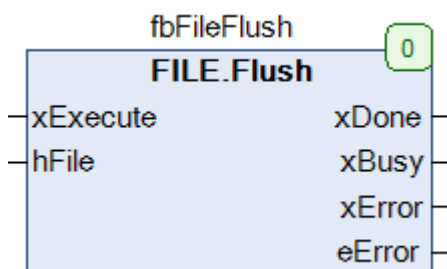


Рис. 4.15. Внешний вид ФБ FILE.Flush на языке CFC

Название	Тип данных	Описание
Входные переменные		
xExecute	BOOL	Переменная активации блока. Запуск блока происходит по <u>переднему фронту</u> переменной.
hFile	FILE.CAA.HANDLE	Дескриптор файла.
Выходные переменные		
xDone	BOOL	Флаг успешного завершения работы блока. Принимает значение TRUE на один цикл.
xBusy	BOOL	Флаг «ФБ в процессе работы».
xError	BOOL	Флаг ошибки. Принимает значение TRUE при возникновении ошибки.
eError	FILE.ERROR	Статус работы ФБ (или имя ошибки).

4.6.9. ФБ FILE.GetPos

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

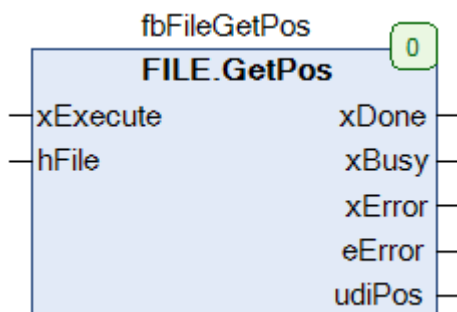


Рис. 4.16. Внешний вид ФБ **FILE.GetPos** на языке **CFC**

Название	Тип данных	Описание
<i>Входные переменные</i>		
xExecute	BOOL	Переменная активации блока. Запуск блока происходит по <u>переднему фронту</u> переменной.
hFile	FILE.CAA.HANDLE	Дескриптор файла.
<i>Выходные переменные</i>		
xDone	BOOL	Флаг успешного завершения работы блока. Принимает значение TRUE на один цикл.
xBusy	BOOL	Флаг «ФБ в процессе работы».
xError	BOOL	Флаг ошибки. Принимает значение TRUE при возникновении ошибки.
eError	FILE.ERROR	Статус работы ФБ (или имя ошибки).
udiPos	UDINT	Текущая установленная позиция в файле (смещение относительно начала файла в байтах).

4.6.10. ФБ FILE.SetPos

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

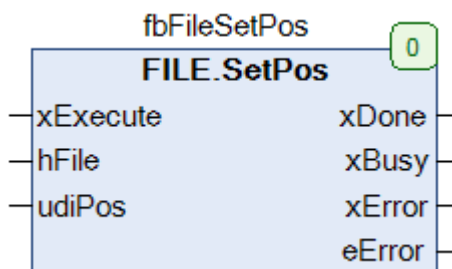


Рис. 4.17. Внешний вид ФБ **FILE.SetPos** на языке **CFC**

Название	Тип данных	Описание
<i>Входные переменные</i>		
xExecute	BOOL	Переменная активации блока. Запуск блока происходит по <u>переднему фронту</u> переменной.
hFile	FILE.CAA.HANDLE	Дескриптор файла.
udiPos	UDINT	Устанавливаемая позиция в файле (смещение относительно начала файла в байтах).
<i>Выходные переменные</i>		
xDone	BOOL	Флаг успешного завершения работы блока. Принимает значение TRUE на один цикл.
xBusy	BOOL	Флаг «ФБ в процессе работы».
xError	BOOL	Флаг ошибки. Принимает значение TRUE при возникновении ошибки.
eError	FILE.ERROR	Статус работы ФБ (или имя ошибки).

4.6.11. ФБ FILE.EOF

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

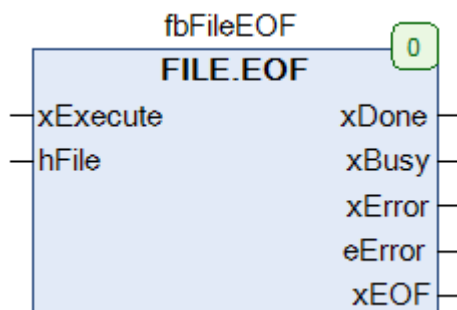


Рис. 4.18. Внешний вид ФБ **FILE.EOF** на языке **CFC**

Название	Тип данных	Описание
<i>Входные переменные</i>		
xExecute	BOOL	Переменная активации блока. Запуск блока происходит по <u>переднему фронту</u> переменной.
hFile	FILE.CAA.HANDLE	Дескриптор файла.
<i>Выходные переменные</i>		
xDone	BOOL	Флаг успешного завершения работы блока. Принимает значение TRUE на один цикл.
xBusy	BOOL	Флаг «ФБ в процессе работы».
xError	BOOL	Флаг ошибки. Принимает значение TRUE при возникновении ошибки.
eError	FILE.ERROR	Статус работы ФБ (или имя ошибки).
xEOF	BOOL	TRUE – достигнут конец файл FALSE – конец файла не достигнут

4.6.12. ФБ FILE.GetSize

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

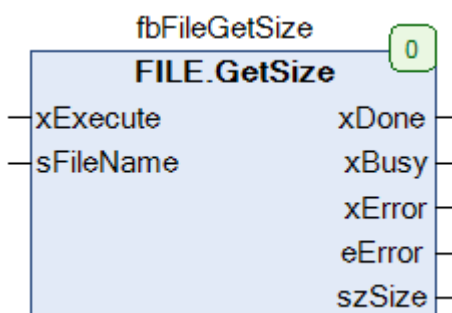


Рис. 4.19. Внешний вид ФБ **FILE.GetSize** на языке CFC

Название	Тип данных	Описание
Входные переменные		
xExecute	BOOL	Переменная активации блока. Запуск блока происходит по <u>переднему фронту</u> переменной.
hFile	FILE.CAA.HANDLE	Дескриптор файла.
Выходные переменные		
xDone	BOOL	Флаг успешного завершения работы блока. Принимает значение TRUE на один цикл.
xBusy	BOOL	Флаг «ФБ в процессе работы».
xError	BOOL	Флаг ошибки. Принимает значение TRUE при возникновении ошибки.
eError	FILE.ERROR	Статус работы ФБ (или имя ошибки).
szSize	FILE.CAA.SIZE	Размер файла в байтах.

4.6.13. ФБ FILE.GetTime

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



Рис. 4.20. Внешний вид ФБ **FILE.GetTime** на языке **CFC**

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

4.7. Особенности версии библиотеки 3.5.3.0

№ п/п	Модуль	Описание	Комментарий
1	FILE.DirCreate	Не поддерживается рекурсивное создание каталогов.	
2	FILE.DirList	Некорректно определяется дата и время последнего изменения каталогов.	
3	FILE.DirRemove	Поддерживается удаление только пустых каталогов.	
4	FILE.Open	Не поддерживается открытие файлов в режиме монопольного доступа.	
5	FILE.Copy	Не поддерживается вызов ФБ с значением FALSE на данном входе.	

5. Пример работы с библиотекой CAA File

5.1. Краткое описание примера

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

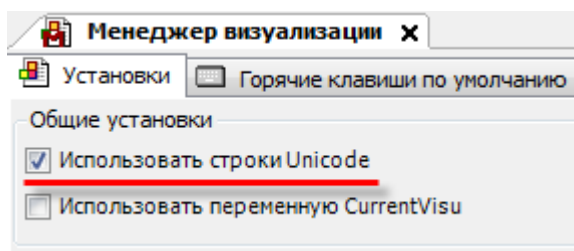
№ п/п	Функционал	Программа
1	получение информации о памяти СПК и подключенных накопителей	PLC_PRG
2	работу с каталогами (создание, удаление, переименование, просмотр содержимого)	PLC_PRG
3	запись и чтение архивов в бинарном формате	BinFileExample_PRG
4	запись архивов в формате .csv	StringFileExample_PRG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

- 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 ROU примера приведен в [Приложении](#).

Пример создан в среде **CODESYS 3.5 SP7 Patch4** и подразумевает запуск на **СПК207.03-CS(-WEB)** с таргет-файлом **3.5.4.26** (или выше).

Пример доступен для скачивания: [Example CAA File v2.projectarchive](#)

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



5.2. Используемые библиотеки

При создании примера были использованы следующие библиотеки:

- **CAA File** (3.5.3.0) – для работы с файлами;
- **CAA DUtil** (3.5.1.0) – для работы с системным временем;
- **Standard64** (3.5.2.0) – для работы со строками типа WSTRING.

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

5.3. Содержимое примера

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		Доп. операции с файлами.
Структуры		
ArchData	BinFileExample_PRG, StringFileExample_PRG	Архивируемые данные.
DriveInfo	PLC_PRG	Информация о накопителе.
VisuDirInfo	PLC_PRG	Информация о содержимом каталога.
Перечисления		
FileDevice	DEVICE_PATH	Названия накопителей.
FileWork	BinFileExample_PRG, StringFileExample_PRG, DIR_INFO	Имена шагов работы с файлами.
Функции и ФБ		
BYTE_SIZE_TO_WSTRING	PLC_PRG (act01, act03)	Конвертация числа байт в формат. строку.
CONCAT11	PLC_PRG (act03), StringFileExample_PRG	Склеивание 11-ти строковых переменных.
DEVICE_PATH	PLC_PRG (act02,act03,act04) StringFileExample_PRG, BinFileExample_PRG	Определение пути к выбранному устройству.
DIR_INFO	PLC_PRG (act03)	Получение информации о содержимом каталога.
LEAD_ZERO	SPLIT_DT_TO_FSTRINGS	Добавление ведущего нуля к числу.
REAL_TO_FSTRING	StringFileExample_PRG	Конвертация REAL в формат. строку.
REAL_TO_FWSTRING	PLC_PRG (act01)	Конвертация REAL в формат. строку.
SPLIT_DT_TO_FSTRINGS	PLC_PRG (act03), StringFileExample_PRG	Выделение из метки времени отдельных разрядов.

5.4. Получение информации о накопителях (PLC_PRG, действие act01_DriveInfo)

Таргет-файлы СПК версии **3.5.4.25** и выше содержат узел **Drives**, который используется для получения информации о памяти СПК и накопителях. Список каналов узла и их описание приведено в [п. 2.4](#).

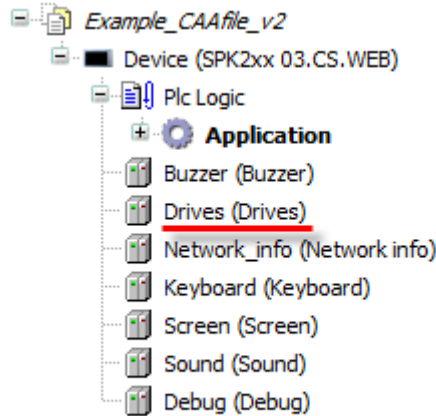


Рис. 5.4.1. Узел **Drives** в дереве проекта

5.4.1. Объявление переменных

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

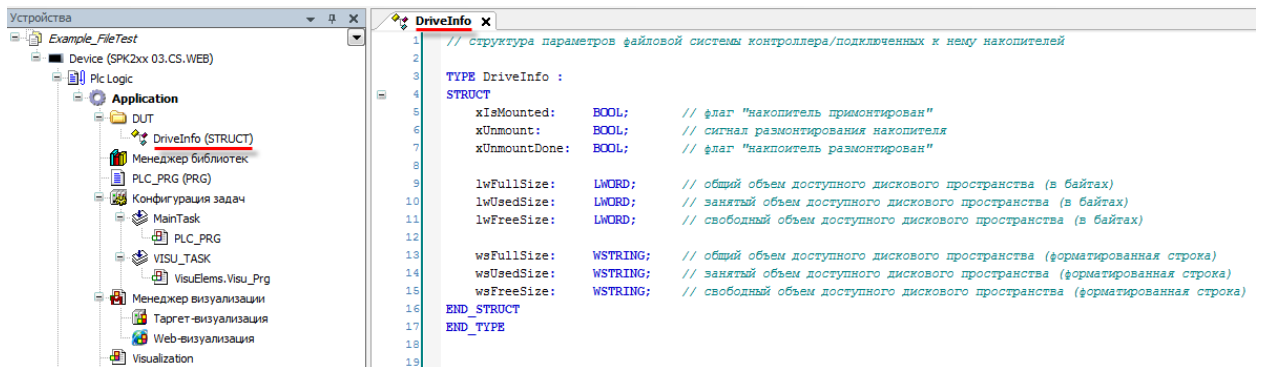


Рис. 5.4.2. Объявление структуры **DriveInfo**

Помимо шести переменных, соответствующих каналам вкладки **Drives**, мы дополнительно объявляем 3 **WSTRING** переменных для отображения общего/занятого/свободного объема накопителей в визуализации – поскольку отображение объема именно в виде числа байт вряд ли будет удобным для оператора.

При работе с файлами в контроллерах СПК можно использовать три разных накопителя:

- Память СПК;
- USB-накопитель;
- SD-накопитель.

Соответственно, объявим в программе **PLC_PRG** три экземпляра структуры **DriveInfo**. Помимо этого 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 поясним чуть позднее).

```

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

```

Рис. 5.4.3. Объявление переменных в программе **PLC_PRG**

Теперь привяжем переменные объявленных нами экземпляров структур к соответствующим каналам узла **Drives**. Как можно заменить, следующие переменные останутся непривязанными:

- в структуре **stSpkMemory** – **xlSMounted**, **xUnmount**, **xUnmountDone** (по очевидным причинам, память СПК нельзя монтировать и размонтировать);
- во всех структурах – переменные типа **WSTRING** (они будут использоваться в визуализации).

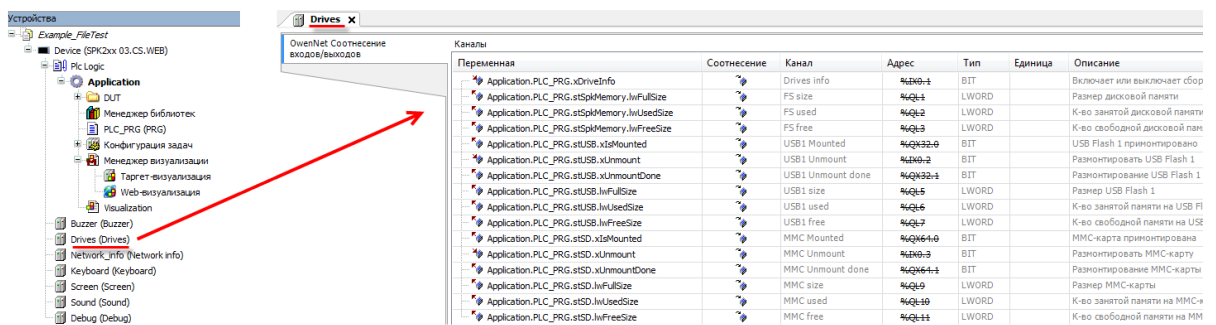


Рис. 5.4.4. Привязка переменных к каналам узла **Drives**

Уже на этом этапе при загрузке проекта в привязанные переменные будет считана информация о накопителях:

stUSB		DriveInfo	
xIsMounted	BOOL		TRUE
xUnmount	BOOL		FALSE
xUnmountDone	BOOL		FALSE
lwFullSize	LWORD		8036253696
lwUsedSize	LWORD		23142400
lwFreeSize	LWORD		8013111296
wsFullSize	WSTRING		" "
wsUsedSize	WSTRING		" "
wsFreeSize	WSTRING		" "

Рис. 5.4.5. Значение переменных, привязанных к каналам узла **Drives**, во время работы проекта

5.4.2. Разработка программы

Как уже упоминалось, отображение объема накопителя в виде числа байт скорее всего будет ненаглядным для оператора. Поэтому преобразуем их в более читабельный формат (например, «11.22 Мбайт»). Для этого создадим 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 рис. 5.4.6:

```

1 // функция преобразования числа байт в форматированную строку
2
3 FUNCTION BYTE_SIZE_TO_WSTRING : WSTRING
4 VAR_INPUT
5     lwByteSize:      LWORD;           // число байт
6 END_VAR
7 VAR_CONSTANT
8     c_KB:            LWORD:=1024;    // число байт в килобайте
9     c_MB:            LWORD:=1024*c_KB; // число килобайт в мегабайте
10    c_GB:            LWORD:=1024*c_MB; // число мегабайт в гигабайте
11 END_VAR
12 VAR
13     rByteSize:      REAL;           // промежуточная переменная
14 END_VAR
15
16 CASE lwByteSize OF
17
18     0 ..(c_KB-1):   BYTE_SIZE_TO_WSTRING := WCONCAT(LWORD_TO_WSTRING(lwByteSize), " Байт");
19
20     c_KB ..(c_MB-1): rByteSize := LWORD_TO_REAL(lwByteSize) / LWORD_TO_REAL(c_KB);
21                     BYTE_SIZE_TO_WSTRING := WCONCAT(REAL_TO_FWSTRING(rByteSize, 3), " Кбайт");
22
23     c_MB ..(c_GB-1): rByteSize := LWORD_TO_REAL(lwByteSize) / LWORD_TO_REAL(c_MB);
24                     BYTE_SIZE_TO_WSTRING := WCONCAT(REAL_TO_FWSTRING(rByteSize, 2), " Мбайт");
25
26     c_GB ..(32*c_GB): rByteSize := LWORD_TO_REAL(lwByteSize) / LWORD_TO_REAL(c_GB);
27                     BYTE_SIZE_TO_WSTRING := WCONCAT(REAL_TO_FWSTRING(rByteSize, 2), " Гбайт");
28
29 END CASE

```

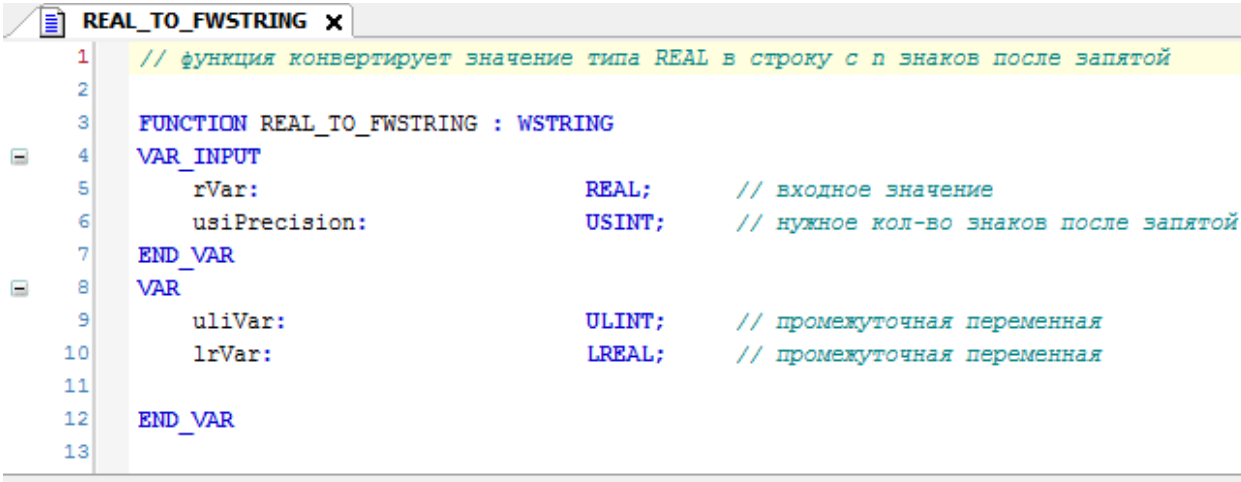
Рис. 5.4.6. Код функции **BYTE_SIZE_TO_WSTRING**

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 рис 4.4.7:



```
1 // функция конвертирует значение типа REAL в строку с n знаков после запятой
2
3 FUNCTION REAL_TO_FWSTRING : WSTRING
4 VAR_INPUT
5     rVar:                REAL;        // входное значение
6     usiPrecision:       USINT;       // нужное кол-во знаков после запятой
7 END_VAR
8 VAR
9     uliVar:              ULINT;      // промежуточная переменная
10    lrVar:                LREAL;     // промежуточная переменная
11
12 END_VAR
13
14 uliVar := LREAL_TO_ULINT( (rVar)*EXPT(10, usiPrecision) );
15 lrVar  := ULINT_TO_LREAL(uliVar) / EXPT(10, usiPrecision);
16 REAL_TO_FWSTRING := LREAL_TO_WSTRING(lrVar);
```

Рис. 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;
- сместим запятую на **две** позиции вправо (с помощью умножения на **10²**), получив число **1122.66**;
- произведем конвертацию в целочисленное значение, получив число **1123**;
- произведем обратную конвертацию в **REAL**, получив число **1123.0**;
- сместим запятую на две позиции влево (с помощью деления на **10²**), получив искомое округленное значение **11.23**.

Итак, мы разработали функции для вывода числа байт в виде форматированной строки. Осталось вызвать их в программе.

Но перед этим рассмотрим еще один момент, который может потребовать оптимизации – индикация размонтирования накопителя. Флаг размонтирования накопителя **xUnmountDone** взводится в **TRUE** на время, пока сигнал размонтирования **xUnmount** имеет значение **TRUE**. Соответственно, если **xUnmount** получит импульс по переднему фронту – накопитель будет успешно размонтирован, но флаг размонтирования примет значение **TRUE** только на один цикл ПЛК – что, очевидно, не будет детектировано человеческим глазом.

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

Внесем код для обеих операций (конвертации объемов накопителей в форматированные строки и сброс сигнала размонтирования) в программу **PLC_PRG**:

```

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

```

Рис. 5.4.8. Код операций с переменными вкладки **Drives**

В следующих пунктах мы будем добавлять в **PLC_PRG** новый код; чтобы разграничить его фрагменты, связанные с разными пунктами документа, мы будем выделять их в действия (**action**). Действие представляет собой изолированный фрагмент кода. Создадим действие (**PLC_PRG – Добавление объекта – Действие**) с названием **act01_DriveInfo** и вынесем в него код из рис. 5.4.8.

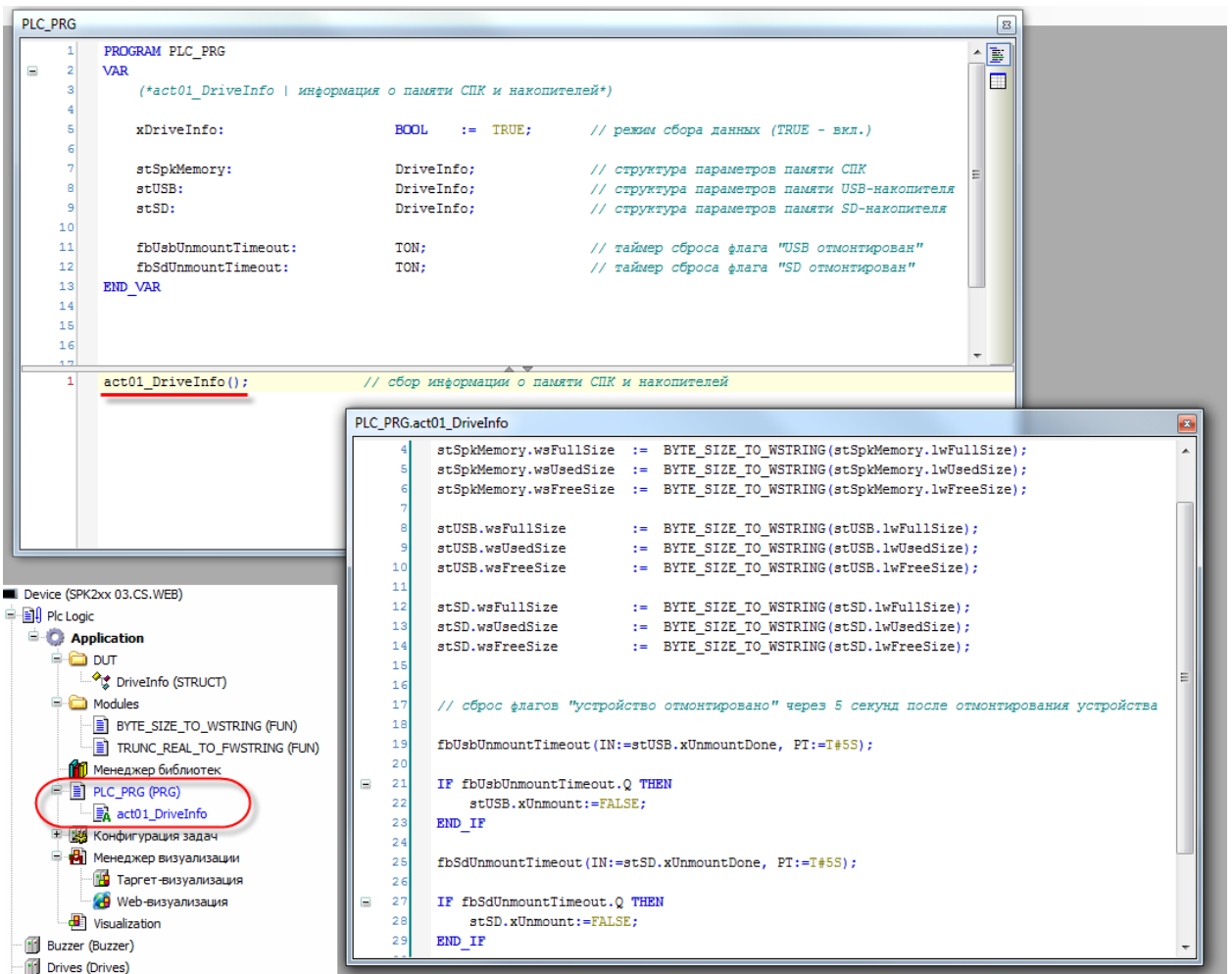


Рис. 5.4.9. Код действия `act01_DriveInfo` и его вызов в программе `PLC_PRG`

5.4.3. Создание визуализации

Теперь создадим интерфейс оператора. Здесь и в следующих пунктах мы не будем подробно останавливаться на процессе разработки визуализации (вся необходимая информация приведена в документе **СПК. Визуализация**), делая акцент только на ключевых моментах. На рис. 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 примонтирования USB- и SD-накопителей (с привязанными переменными **xIsMounted**);
- 2 кнопки для размонтирования накопителей (с привязанными переменными **xUnmount**, поведение - **Переключатель изображения**);
- 2 индикатора, отображающих флаги успешного размонтирования 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](#).

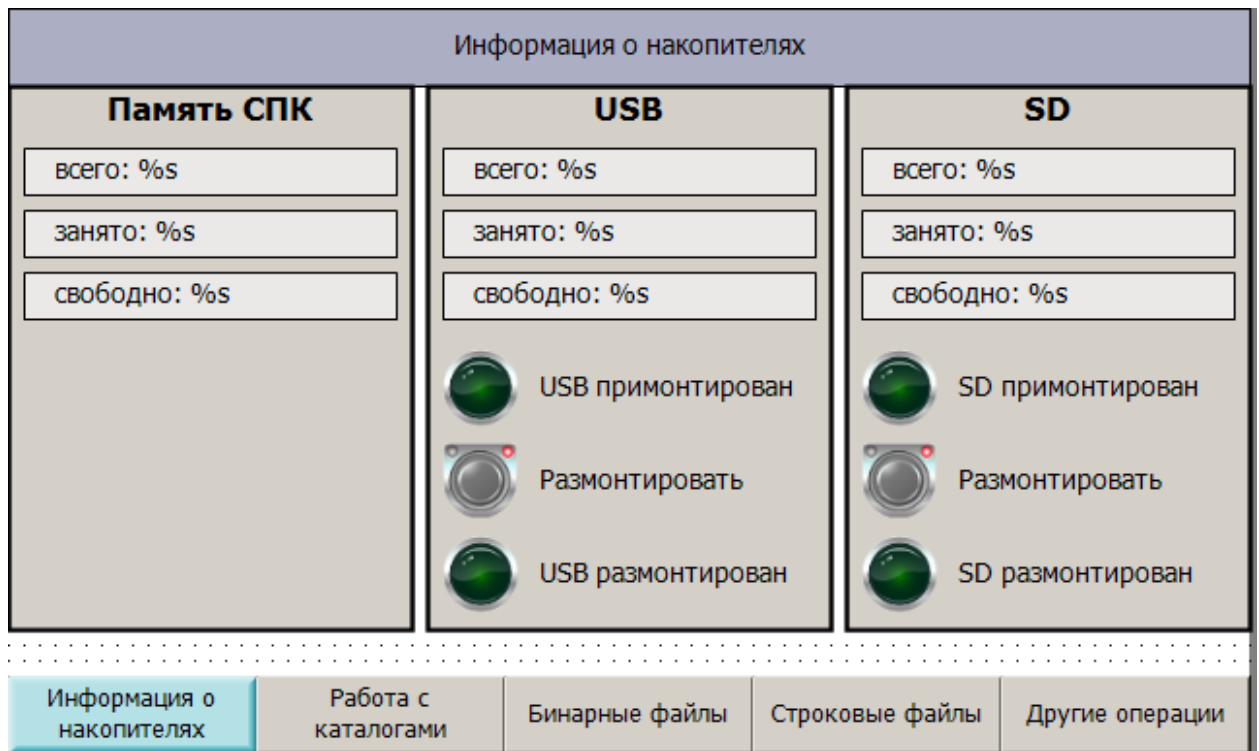


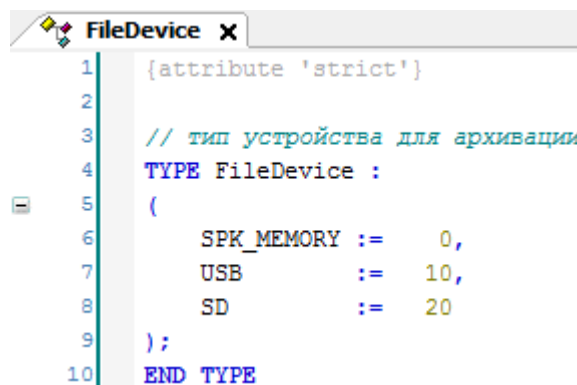
Рис. 5.4.10. Внешний вид экрана **Visu01_DriveInfo**

5.5. Работа с каталогами (PLC_PRG, действие act02_DirExample)

В данном пункте приведен пример работы с каталогами. Каталоги позволяют разделять файлы на группы, что упрощает работу с ними. Каталоги могут создаваться, переименовываться и удаляться. Кроме того, существует возможность получить информацию о содержимом каталоге.

5.5.1. Объявление переменных

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



```
1 {attribute 'strict'}
2
3 // тип устройства для архивации
4 TYPE FileDevice :
5 (
6     SPK_MEMORY := 0,
7     USB        := 10,
8     SD         := 20
9 );
10 END TYPE
```

Рис. 5.5.1. Объявление перечисления **FileDevice**

Объявим в программе **PLC_PRG** следующие переменные:

```
17 (*act02_DirExample | операции с каталогами*)
18
19 fbDirCreate:          FILE.DirCreate;      // фБ создания каталога
20 fbDirRemove:         FILE.DirRemove;      // фБ удаления каталога
21 fbDirRename:         FILE.DirRename;      // фБ переименования каталога
22
23 sDirName:             STRING;              // полный путь к текущему каталогу
24 sDirNameNew:          STRING;              // полный путь для создаваемого каталога
25 sVisuDirName:         STRING;              // имя текущего каталога
26 sVisuDirNameNew:     STRING;              // имя создаваемого каталога
27 sDeviceDirPath:      STRING;              // путь к устройству
28 iDeviceDirPath:      INT;                 // ID устройства
```

Рис. 5.5.2. Объявление переменных в программе **PLC_PRG**

5.5.2. Разработка программы

В прошлом подпункте мы создали перечисление **FileDevice**, содержащее типы накопителей (память контроллера/USB-накопитель/SD-накопитель). Теперь создадим функцию **DEVICE_PATH**, которая в качестве аргумента принимает ID (идентификатор) накопителя и возвращает путь к его файловой системе. Пути приведены в [п. 2.4](#).

```
DEVICE_PATH x
1 // функция возвращает путь для файловой системы СПК или накопителя по ID
2
3 FUNCTION DEVICE_PATH : STRING
4 VAR_INPUT
5     iDevice: INT; // ID устройства
6 END_VAR
7 VAR
8 END_VAR
9
10 CASE iDevice OF
11     FileDevice.SPK_MEMORY:
12         DEVICE_PATH:='/mnt/ufs/root/CoDeSysSP_wrk/';
13     FileDevice.USB:
14         DEVICE_PATH:='/mnt/ufs/media/sda1/';
15     FileDevice.SD:
16         DEVICE_PATH:='/mnt/ufs/media/mmcblk0p1/';
17 END_CASE
```

Рис. 5.5.3. Код функции **DEVICE_PATH**

Таким образом, оператору будет достаточно выбрать ID устройства (например, через элемент **Комбинированное окно/Combobox**), чтобы программа автоматически сформировала путь к нему. В противном случае пришлось бы каждый раз вводить путь с помощью экранной клавиатуры, что, очевидно, нельзя назвать удобным.

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

```
PLC_PRG.act02_DirExample x
1 // получаем путь к выбранному устройству
2 sDeviceDirPath := DEVICE_PATH(iDeviceDirPath);
3
4 // склеиваем его с именами каталогов
5 sDirName := CONCAT(sDeviceDirPath, sVisuDirName);
6 sDirNameNew := CONCAT(sDeviceDirPath, sVisuDirNameNew);
7
8 // выполняем ФБ операций с каталогами
9 fbDirCreate(xExecute:=, sDirName:=sDirNameNew);
10 fbDirRename(xExecute:=, sDirNameOld:=sDirName, sDirNameNew:=sDirNameNew);
11 fbDirRemove(xExecute:=, sDirName:=sDirName);
12
```

Рис. 5.5.4. Код действия **act02_DirExample**

В программе **PLC_PRG** добавим вызов этого действия:

```
PLC_PRG x
1 // пример действий с каталогами и файлами (помимо чтения и записи)
2
3 PROGRAM PLC_PRG
4 VAR
5     (*act01_DriveInfo | информация о памяти СПК и накопителей*)
6
7     xDriveInfo:          BOOL    := TRUE;    // режим сбора данных (TRUE - вкл.)
8
9     stSpkMemory:        DriveInfo;        // структура параметров памяти СПК
10    stUSB:              DriveInfo;        // структура параметров памяти USB-накопителя
11    stSD:              DriveInfo;        // структура параметров памяти SD-накопителя
12
13    fbUsbUnmountTimeout: TON;           // таймер сброса флага "USB отмонтирован"
14    fbSdUnmountTimeout:  TON;           // таймер сброса флага "SD отмонтирован"
15
16
17 act01_DriveInfo ();    // сбор информации о памяти СПК и накопителей
18 act02_DirExample ();   // пример работы с каталогами (создание, переименование, удаление)
19
20
21
22
```

Рис. 5.5.5. Вызов действия **act02_DirExample** в программе **PLC_PRG**

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

- возвращает путь к выбранному накопителю по его ID;
- склеивает путь к накопителю с именами текущего и создаваемого каталога;
- осуществляет вызов функциональных блоков создания, переименования и удаления каталогов.

Необходимо отметить следующее:

1. 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 необходимо реализовать свой алгоритм работы с данными блоками, который позволит решить его конкретную задачу.
2. В рамках примера в качестве строковых аргументов ФБ используются одни и те же переменные. В большинстве практических задач разумно использовать уникальные переменные для каждого ФБ.

5.5.3. Создание визуализации

Создадим интерфейс оператора для работы с каталогами. На рис. 5.5.6 приведен внешний вид экрана **Visu02_DirExample**, который включает в себя:

- элемент **Комбинированное окно – целочисленный**, используемый для выбора накопителя, с каталогами которого будет работать программа. К элементу привязана переменная **iDeviceDirPath**. Настройки элемента описаны в [п. 5.5.4](#).
- прямоугольник **Путь к устройству**, отображающий значение переменной **sDeviceDirPath**;
- 2 прямоугольника **Имя нового каталога** с привязанной переменной **sVisuDirNameNew**. В настройках элементов на вкладке **InputConfiguration** для действия **OnMouseClicked** задана операция **Записать переменную** (тип ввода – диалог **VisuKeypad**).
- 2 прямоугольника **Имя существующего каталога** с привязанной переменной **sVisuDirName**. В настройках элементов на вкладке **InputConfiguration** для действия **OnMouseClicked** задана операция **Записать переменную** (тип ввода – диалог **VisuKeypad**).
- 3 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**.

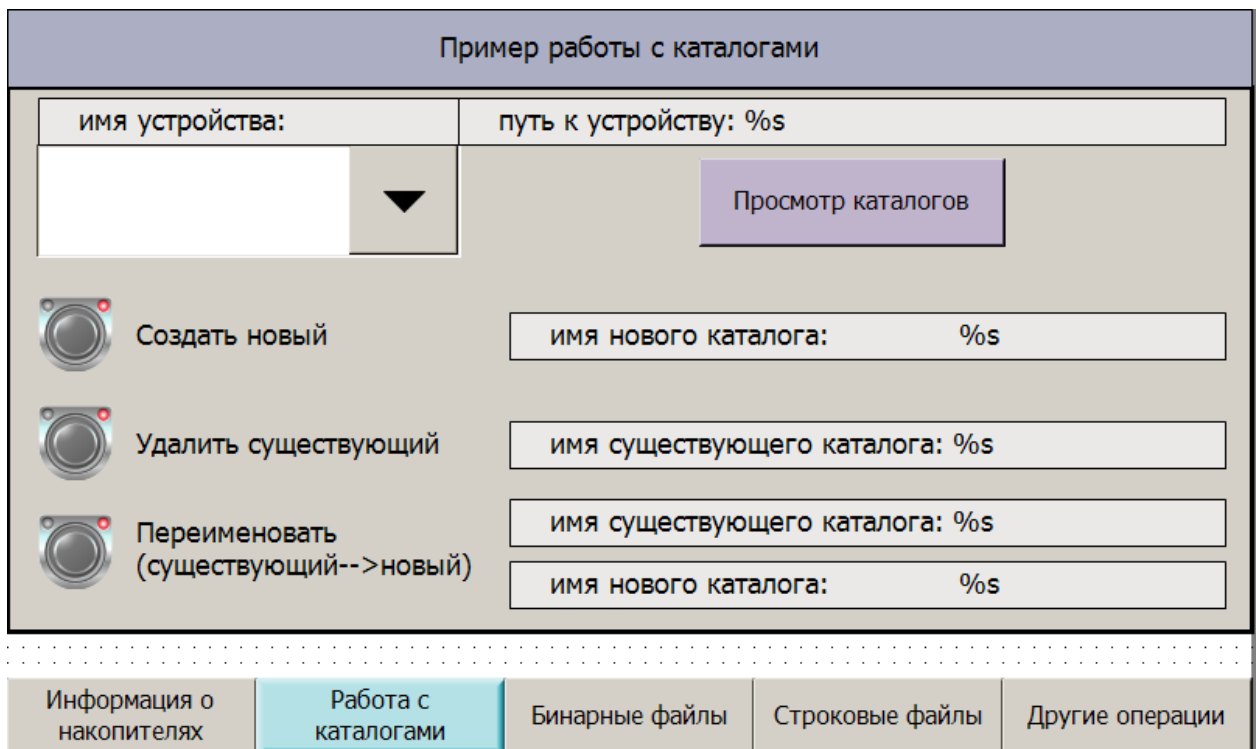


Рис. 5.5.6. Внешний вид экрана **Visu02_DirExample**

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

5.5.4. Настройка элемента Комбинированное окно

В визуализации этого и следующих пунктов мы будем использовать элемент **Комбинированное окно – Целочисленный** для выбора оператором накопителя, с которым необходимо работать. Настройки элемента приведены на рис. 5.5.7:

Свойство	Значения
Имя элемента	GenElemInst_223
Тип элемента	Комбинированное окно - Целочисленный
[-] Позиция	
X	20
Y	88
Ширина	270
Высота	70
Переменная	PLC_PRG.iDeviceDirPath
Список текстов	'FileDevice'
Пул изображений	'ImagePool'
[+] Параметры списка	
[+] Тексты	
[-] Поддиапазон	
Использовать поддиапазон	<input checked="" type="checkbox"/>
Начальный индекс	0
Конечный индекс	20
Отбрасывать недостающие тексты	<input checked="" type="checkbox"/>
[+] Свойства текста	
[+] Переменные состояний	

Рис. 5.5.7. Настройки элемента **Комбинированное окно – Целочисленный**

Как можно заметить, элемент использует компоненты **Список текстов (FileDevice)** и **Пул изображений (ImagePool)**. Их следует добавить в проект (**Application – Добавление объекта**). Содержимое компонентов приведено ниже.

FileDevice x	
ID	По умолчанию
0	SPK_MEMORY
10	USB
20	SD




ImagePool x			
ID	Имя файла	Изображение	Тип ссылки
0	HDD.png		Embedded and link to file
10	USB.png		Embedded and link to file
20	SD.png		Embedded and link to file

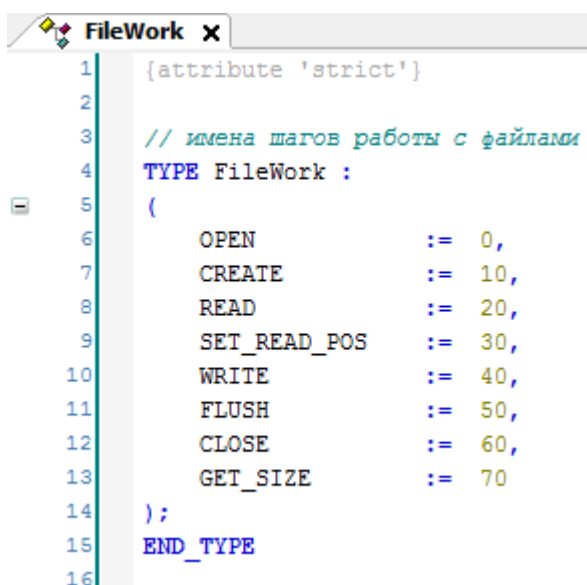
Рис. 5.5.8. Содержимое компонентов **Список Текстов** и **Пул изображений**

5.6. Просмотр содержимого каталогов (PLC_PRG, действие act03_DirList)

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

5.6.1. Объявление переменных

Как упоминалось в [п. 2.2](#), работу с файлами/каталогами можно представить в виде последовательности шагов, выполняемых с помощью оператора **CASE**. В качестве меток оператора **CASE** можно использовать обычные числа (0, 1, 2 и т.д.) – но это затруднит чтение программы. Поэтому объявим перечисление **FileWork (Application – Добавление объекта – DUT – Перечисление)**, в котором свяжем номера шагов с символьными именами. В данном пункте мы воспользуемся лишь несколькими элементами этого перечисления; все остальные будут использованы в [п. 5.7](#) и [5.8](#) при создании архиваторов.



```
1 {attribute 'strict'}
2
3 // имена шагов работы с файлами
4 TYPE FileWork :
5 (
6     OPEN           := 0,
7     CREATE         := 10,
8     READ           := 20,
9     SET_READ_POS   := 30,
10    WRITE          := 40,
11    FLUSH          := 50,
12    CLOSE          := 60,
13    GET_SIZE       := 70
14 );
15 END_TYPE
16
```

Рис. 5.6.1. Объявление перечисления **FileWork**

При просмотре каталогов мы будем получать информацию о каждом вложенном каталоге/файле в виде экземпляра структуры `FILE_FILE_DIR_ENTRY`. Чтобы отображать эти данные в визуализации, необходимо привести их к удобному для оператора виду. Для этого объявим структуру `VisuDirInfo`:

```

1 // структура информации о каталога/файла, отображаемой в визуализации
2
3 TYPE VisuDirInfo :
4 STRUCT
5     sEntryName:          STRING;          // имя каталога/файл
6     wsEntryType:        WSTRING;        // тип (каталог или файл)
7     wsEntrySize:        WSTRING;        // размер файла в байтах
8     sLastModification:  STRING;         // дата последнего изменения файла
9 END_STRUCT
10 END_TYPE

```

Рис. 5.6.2. Объявление структуры `VisuDirInfo`

Объявим в программе `PLC_PRG` следующие переменные:

```

31 (*act03_DirList | информация о выбранном каталоге*)
32
33 fbDirInfo:          DIR_INFO;          // фБ сбора информации о каталоге
34 xDirList:          BOOL;              // сигнал сбора информации о каталоге
35 i:                INT;              // счетчик для цикла
36
37 // путь к выбранному каталогу
38 sDirListPath:      STRING := '/mnt/ufs/root/CoDeSysSP_wrk/';
39
40 // путь к предыдущему выбранному каталогу
41 sLastDevice:       STRING;
42
43 // массив данных о вложенных файлах/каталогах для визуализации
44 astVisuDirInfo:    ARRAY [0..c_MAX_ENTRIES] OF VisuDirInfo;
45
46 fbSplitDT:         SPLIT_DT_TO_FSTRINGS; // фБ конвертации времени в строку
47 asEntryDT:         ARRAY [0..10] OF STRING; // метка времени в виде отдельных строковых разрядов
48
49 iSelectedEntry:    INT;              // номер выбранной строки таблицы
50
51 xDown:             BOOL;             // сигнал "Открыть каталог"
52 xUp:               BOOL;            // сигнал "Перейти на уровень выше"
53 xHideUp:           BOOL;            // переменная неактивности кнопки "Открыть каталог"
54 xFirstScan:        BOOL;            // сигнал "Сканирование каталога"
55

```

Рис. 5.6.3. Объявление переменных в программе `PLC_PRG`

Кроме этого, потребуется объявить несколько констант:

```

74 VAR CONSTANT
75     c_MAX_ENTRIES:  UINT           :=100;          // максимальное число вложенных элементов каталога
76     c_sCharSlash:   STRING(1)     :='/';          // разделитель для пути в файловой системе
77     c_byCodeSlash:  BYTE           :=16#2F;        // ASCII-код разделителя
78
79     // пустая структура для очистки таблицы
80     c_astVisuDirInfoNull: ARRAY [0..c_MAX_ENTRIES] OF VisuDirInfo;
81 END_VAR

```

Рис. 5.6.4. Объявление констант в программе `PLC_PRG`

5.6.2. Разработка программы

Как можно заметить на рис. 5.6.3, мы объявили экземпляры функциональных блоков **DIR_INFO** и **SPLIT_DT_TO_FSTRING**; сами блоки при этом еще не созданы. Первый из них будет использоваться непосредственно для получения информации о содержимом каталога, второй – для преобразования метки времени типа **DT** в строковые представления отдельных разрядов.

Создадим ФБ **DIR_INFO** со следующим интерфейсом:

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

Рис. 5.6.5. Объявление переменных ФБ **DIR_INFO**

Обратите внимание, что ФБ содержит константу **c_MAX_ENTRIES**; одноименная константа уже была объявлена в программе **PLC_PRG** (рис. 5.6.4). Значения обеих констант должны совпадать. Вариант с двумя константами является достаточно простым (поэтому в рамках примера мы остановимся на нем), 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:

```
1 // детектируем сигнал запуска блока
2 fbStart(CLK:=xExecute);
3
4 // сбрасываем сигнал завершения работы
5 xDone:=FALSE;
6
7 CASE eState OF
8
9
10 FileWork.OPEN: // открываем каталог
11
12 // обнуляем позицию для записи информации о файлах/каталогах
13 uiEntryPos:=0;
14
15 fbDirOpen(xExecute:=fbStart.Q, sDirName:=sDirName);
16
17 IF fbDirOpen.xDone THEN
18     hDirHandle := fbDirOpen.hDir;
19     fbDirOpen(xExecute:=FALSE);
20     eState := FileWork.READ;
21 END_IF
22
23
24 FileWork.READ: // получаем информацию о вложенных файлах и каталогах
25
26 fbDirList(xExecute:=TRUE, hDir:=hDirHandle);
27
28 // пока нет ошибок, получаем информацию о текущем файле/каталоге...
29 IF fbDirList.xDone AND fbDirList.eError=FILE.ERROR.NO_ERROR THEN
30     astDirInfo[uiEntryPos] := fbDirList.deDirEntry;
31
32 // информацию о каждом обработанном файле/каталоге записываем в следующую ячейку массива
33 uiEntryPos := uiEntryPos+1;
34
35 // если число вложенных файлов/каталогов больше, чем размер массива...
36 // ...то начинаем перезаписывать его с нуля
37 IF uiEntryPos>c_MAX_ENTRIES THEN
38     uiEntryPos := 0;
39 END_IF
40
41 fbDirList(xExecute:=FALSE);
42 END_IF
43
44 // если код ошибки - "NO_MORE_ENTRIES", то обработаны все файлы/каталоги...
45 // ...и можно завершать работу блока
46 IF fbDirList.eError=FILE.ERROR.NO_MORE_ENTRIES THEN
47     fbDirList(xExecute:=FALSE);
48     eState := FileWork.CLOSE;
49 END_IF
50
51
52 FileWork.CLOSE: //завершение работы блока
53
54 fbDirClose(xExecute:=TRUE, hDir:=hDirHandle);
55
56 IF fbDirClose.xDone THEN
57     fbDirClose(xExecute:=FALSE);
58
59 // устанавливаем флаг завершения работы
60 xDone := TRUE;
61
62 eState := FileWork.OPEN;
63 END_IF
64
65 END_CASE
66
```

Рис. 5.6.6. Код ФБ DIR_INFO

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

- На шаге **OPEN** выполняется открытие каталога с помощью ФБ [FILE.DirOpen](#). Если каталог успешно открыт, то происходит переход на шаг **READ**.
- На шаге **READ** начинается получение информации о вложенных файлах/каталогах с помощью ФБ [FILE.DirList](#). Полученные данные записываются на выход **astDirInfo**, который представляет собой массив структур типа [FILE.FILE_DIR_ENTRY](#). Если число полученных данных превышает размер массива (верхняя граница которого определяется константой **c_MAX_ENTRIES**), то массив перезаписывается начиная с нулевой записи. Иными словами, 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.

Таким образом, для работы с блоком **DIR_INFO** необходимо:

- 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**.

Итак, блок готов; осталось вызвать его в программе **PLC_PRG**. Но перед этим создадим еще один блок, который потребуется нам в этом и следующих пунктах. Блок будет конвертировать значение даты и времени типа **DT** в строковые представления отдельных разрядов с ведущими нулями. Назовем его **SPLIT_DT_TO_FSTRINGS**.

```

SPLIT_DT_TO_FSTRINGS x
1 // ФБ разделяет метку времени типа DT на строковые представления отдельных разрядов с ведущими нулями
2
3 FUNCTION_BLOCK SPLIT_DT_TO_FSTRINGS
4 VAR_INPUT
5   dtDateAndTime:      DT;          // метка времени в формате DT
6 END_VAR
7 VAR_OUTPUT
8   sYear:              STRING;    // разряды времени в строковом представлении
9   sMonth:             STRING;    //
10  sDay:                STRING;    //
11  sHour:               STRING;    //
12  sMinute:            STRING;    //
13  sSecond:            STRING;    //
14 END_VAR
15 VAR
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100

```

Рис. 5.6.7. Объявление переменных и код ФБ **SPLIT_DT_TO_FSTRINGS**

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

```

LEAD_ZERO x
1 // функция преобразует число в строку с ведущим нулем
2
3 FUNCTION LEAD_ZERO : STRING
4 VAR_INPUT
5   uiInput:           UINT;
6 END_VAR
7 VAR
8 END_VAR
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100

```

Рис. 5.6.8. Объявление переменных и код функции **LEAD_ZERO**

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

Рассмотрим пример работы ФБ. Пусть вход **dtDateAndTime** имеет значение

DT#2017-7-27-7:32:5

Тогда выходы блока будут иметь следующие значения:

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

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

```
CONCAT11 x
1 // функция склеивает заданное число строковых переменных, помещенных в массив
2
3 FUNCTION CONCAT11 : STRING(255)
4 VAR_INPUT
5     asSTR:      ARRAY [0..c_MAX_STR] OF STRING;
6 END_VAR
7 VAR
8     sBuffer:    STRING;           // промежуточная переменная
9     i:          INT;             // счетчик для цикла
10 END_VAR
11
12 VAR CONSTANT
13     c_MAX_STR:  INT:=10;         // размер массива строковых переменных
14 END_VAR
15
16 FOR i:=0 TO c_MAX_STR DO
17     sBuffer:=CONCAT(sBuffer, asSTR[i]);
18 END_FOR
19
20 CONCAT11:=sBuffer;
```

Рис. 5.6.9. Объявление переменных и код функции **CONCAT11**

Теперь можно перейти непосредственно к написанию кода программы. Создадим в программе PLC_PRG действие act03_DirList (PLC_PRG – Добавление объекта – Действие) и вынесем в него код, приведенный на рис. 5.6.10.

```

1 // получаем путь к выбранному устройству
2 sDeviceDirPath:=DEVICE_PATH(iDeviceDirPath);
3
4 // при загрузке проекта и при выборе нового устройства сканируем его корневой каталог
5 IF NOT(xFirstScan) OR sDeviceDirPath<>sLastDevice THEN
6     sDirListPath := sDeviceDirPath;
7     sLastDevice := sDeviceDirPath;
8     xDirList := TRUE;
9     xFirstScan := TRUE;
10 END_IF
11
12
13 // если выбранный элемент - файл или символическая ссылка, то скрываем кнопку "Открыть каталог"
14 xHideUp := astVisuDirInfo[iSelectedEntry].sEntryName='..'
15 OR astVisuDirInfo[iSelectedEntry].sEntryName='.' OR astVisuDirInfo[iSelectedEntry].wsEntryType="файл";
16
17
18 // по сигналу переходим в выбранный каталог
19 IF xDown THEN
20
21     sDirListPath := CONCAT(sDirListPath, astVisuDirInfo[iSelectedEntry].sEntryName);
22
23     IF sDirListPath<>sDeviceDirPath THEN
24         sDirListPath := CONCAT(sDirListPath, c_CharSlash);
25     END_IF
26
27     xDown := FALSE;
28     xDirList := TRUE;
29 END_IF
30
31
32 // по сигналу переходим на уровень выше, контролируя, что продолжается работа с прежним устройством
33 IF xUp AND sDirListPath<>sDeviceDirPath THEN
34
35     // удаляем последний символ в текущем пути (это "/")
36     sDirListPath[LEN(sDirListPath)-1] := 0;
37
38     // справа налево стираем символы из пути до тех пор, пока не найдем "/"
39     // таким образом, из текущего пути будет удален самый нижний каталог
40     FOR i:=LEN(sDirListPath)-2 TO 0 BY -1 DO
41
42         IF sDirListPath[i]=c_byCodeSlash THEN
43             EXIT;
44         ELSE
45             sDirListPath[i] := 0;
46         END_IF
47     END_FOR
48
49     xUp := FALSE;
50     xDirList := TRUE;
51 END_IF
52
53 // получаем информацию о содержимом каталога
54 fbDirInfo(xExecute:=xDirList, sDirName:=sDirListPath);
55
56 IF fbDirInfo.xDone THEN
57
58     // стираем информацию о предыдущем открытом каталоге
59     astVisuDirInfo := c_astVisuDirInfoNull;
60     // переходим к верхней строке таблицы
61     iSelectedEntry := 0;
62
63     // заполняем массив структур информацией о содержимом каталога
64     FOR i:=0 TO UINT_TO_INT(fbDirInfo.uiEntryPos-1) DO
65         astVisuDirInfo[i].sEntryName := fbDirInfo.astDirInfo[i].sEntry;
66         astVisuDirInfo[i].wsEntrySize := BYTE_SIZE_TO_WSTRING(fbDirInfo.astDirInfo[i].szSize);
67         astVisuDirInfo[i].wsEntryType := SEL(fbDirInfo.astDirInfo[i].xDirectory, "файл", "Каталог");
68
69         // преобразуем дату и время последнего изменения файла в форматированную строку
70         fbSplitDT(dtDateAndTime:=fbDirInfo.astDirInfo[i].dtLastModification);
71
72         asEntryDT[0] := fbSplitDT.sDay;
73         asEntryDT[1] := '.';
74         asEntryDT[2] := fbSplitDT.sMonth;
75         asEntryDT[3] := '.';
76         asEntryDT[4] := fbSplitDT.sYear;
77         asEntryDT[5] := '.';
78         asEntryDT[6] := fbSplitDT.sHour;
79         asEntryDT[7] := ':';
80         asEntryDT[8] := fbSplitDT.sMinute;
81         asEntryDT[9] := '.';
82         asEntryDT[10] := fbSplitDT.sSecond;
83
84         astVisuDirInfo[i].sLastModification := CONCAT11(asEntryDT);
85
86         xDirList := FALSE;
87     END_FOR
88 END_IF

```

Рис. 5.6.10. Код действия act03_DirList

Код выполняет следующие операции:

1. при загрузке контроллера 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 произведено получение информации о корневом каталоге этого устройства;

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

Если оператор выделил каталог, то он может просмотреть его содержимое, нажав кнопку **Открыть каталог**. Для возвращения в предыдущий каталог необходимо нажать кнопку **На уровень выше**.

Соответственно, при выделении **файла** кнопка **Открыть каталог** должна быть неактивной. В ОС Linux также существуют специальные каталоги «.» и «..», представляющий собой ссылки на текущий и родительский каталог. В рамках примера мы запретим пользователю работать с этими каталогами.

Логическая переменная **xHideUp**, характеризующая неактивность кнопки, будет принимать значение **TRUE** в вышеописанных случаях.

4. 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;

5. 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. При этом у пользователя нет возможности перейти на уровень выше относительно корневого каталога;

6. программа обрабатывает информацию, полученную от блока **fbDirInfo** – в частности, преобразовывает размер вложенных каталогов/файлов в строковый вид с помощью функции **BYTE_SIZE_TO_WSTRING** (которая была создана в [п. 5.4.2](#)) и метки времени последнего изменения каталога/файла в форматированную строку с помощью ФБ **SPLIT_DT_TO_FSTRINGS**.

5.6.3. Создание визуализации

Создадим интерфейс оператора для просмотра каталогов. На рис. 5.6.11 приведен внешний вид экрана **Visu06_DirList**, который включает в себя:

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

Свойство	Значения
Имя элемента	GenElemInst_280
Тип элемента	Полоса прокрутки
Значение	PLC_PRG.iSelectedEntry
Минимальное значен...	0
Максимальное значе...	99
Размер страницы	10
Прокрутка выполнена	<input type="checkbox"/>

- 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**).

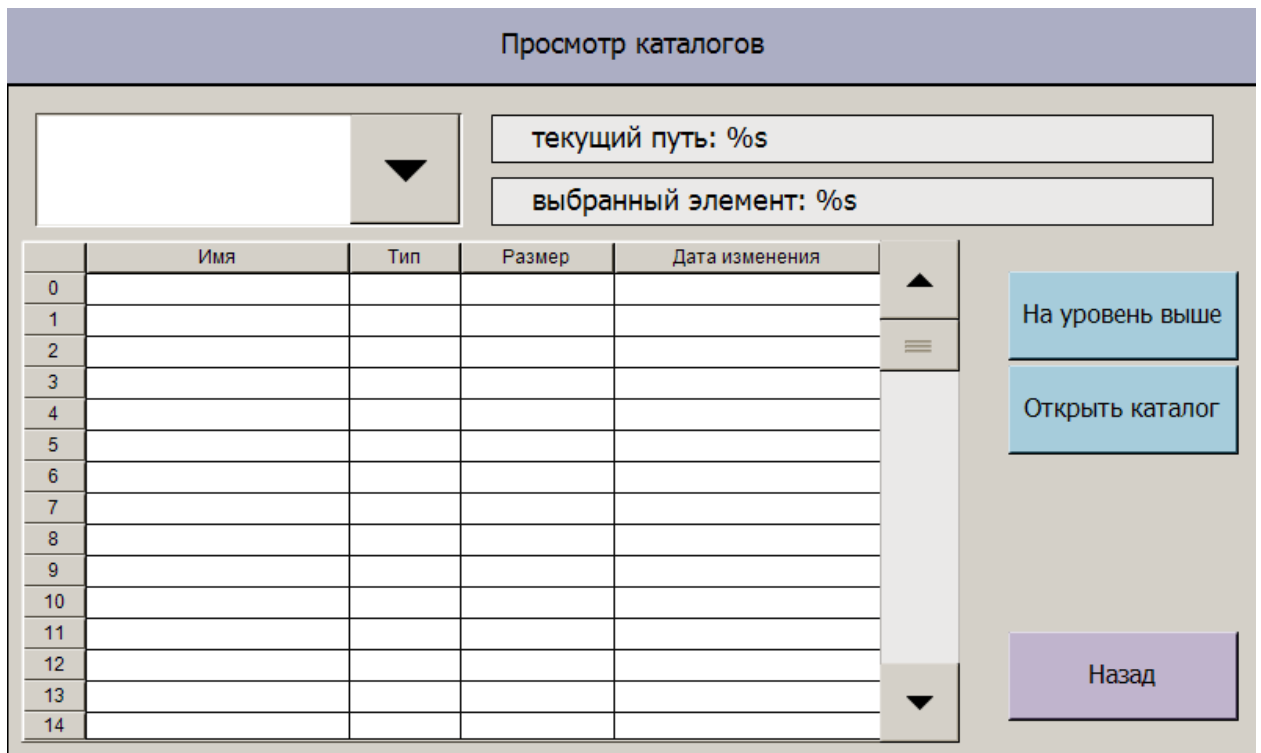


Рис. 5.6.11. Внешний вид экрана **Visu06_DirList**

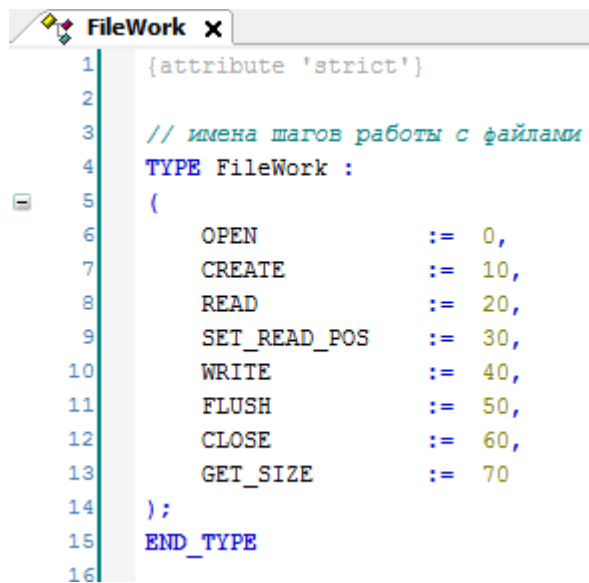
Визуализация также содержит кнопки переключения экранов (описание других экранов проекта приведено в соответствующих пунктах). Пример работы с экраном приведен в [п. 5.10](#).

5.7. Экспорт и импорт бинарных файлов (BinFileExample_PRG)

В данном пункте приведен пример экспорта и импорта бинарных данных. Информация о различиях бинарных и текстовых файлов приведена в [п. 2.6](#).

5.7.1. Объявление переменных

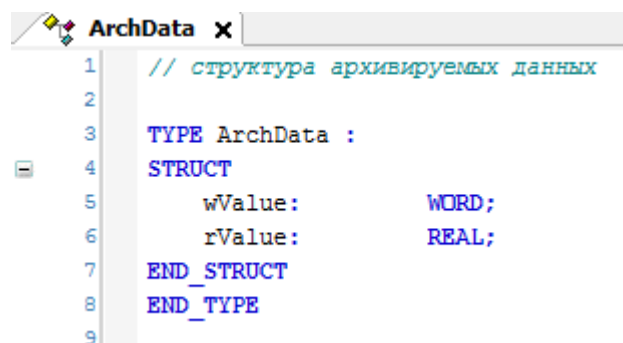
Как упоминалось в [п. 2.2](#), работу с файлами/каталогами можно представить в виде последовательности шагов, выполняемых с помощью оператора **CASE**. В качестве меток оператора **CASE** можно использовать обычные числа (0, 1, 2 и т.д.) – но это затруднит чтение программы. Соответствующее перечисление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



```
1 {attribute 'strict'}
2
3 // имена шагов работы с файлами
4 TYPE FileWork :
5 (
6     OPEN           := 0,
7     CREATE         := 10,
8     READ           := 20,
9     SET_READ_POS   := 30,
10    WRITE          := 40,
11    FLUSH          := 50,
12    CLOSE          := 60,
13    GET_SIZE       := 70
14 );
15 END_TYPE
16
```

Рис. 5.7.1. Объявление перечисления **FileWork**

Объявим структуру данных, которые мы будем записывать в файл и считывать 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**:



```
1 // структура архивируемых данных
2
3 TYPE ArchData :
4 STRUCT
5     wValue:      WORD;
6     rValue:      REAL;
7 END_STRUCT
8 END_TYPE
9
```

Рис. 5.7.2. Объявление структуры **ArchData**

Объявим в программе **BinFileExample_PRG** следующие переменные:

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

Рис. 5.7.3. Объявление переменных в программе **BinFileExample_PRG**

Как можно заметить, мы объявляем два экземпляра структуры **ArchData** – один из них будет содержать данные, записываемые в файл, другой – данные, прочитанные из файла.

5.7.2. Разработка программы

Структура программы **BinFileExample_PRG** приведена рис. 5.7.5. Перед началом работы с файлом программа получает путь к выбранному устройству (с помощью уже знакомой нам по [п. 5.5.2](#) функции **DEVICE_PATH**) 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).

```
1 // получаем путь к выбранному устройству
2 sDevicePath := DEVICE_PATH(iDevicePath);
3
4 // склеиваем его с именем выбранного файла
5 sFileName := CONCAT(sDevicePath, sVisuFileName);
6
7 // детектируем сигнал записи в файл или чтения из файла
8 fbWriteTrig(CLK:=xWrite);
9 fbReadTrig(CLK:=xRead);
10
11 // в зависимости от пришедшего сигнала вводим соответствующий флаг
12 IF fbWriteTrig.Q THEN
13     xWBusy := TRUE;
14 ELSIF fbReadTrig.Q THEN
15     xRBusy := TRUE;
16 END_IF
17
18
19 CASE eState OF
20
21     FileWork.OPEN: // шаг открытия файла
22
23     FileWork.CREATE: // шаг создания файла
24
25     FileWork.WRITE: // шаг записи в буфер
26
27     FileWork.FLUSH: // шаг сброса буфера в файл
28
29     FileWork.SET_READ_POS: // шаг установки позиции для чтения из файла
30
31     FileWork.READ: // шаг чтения данных
32
33     FileWork.CLOSE: // шаг закрытия файла
34
35     FileWork.GET_SIZE: // шаг определения размера файла
36
37 END_CASE
```

Рис. 5.7.4. Структура программы **BinFileExample_PRG**

Сама работа с файлами происходит в управляющем операторе **CASE**. На рис. 5.7.4. приведены только имена шагов без раскрытия их программного кода (он будет приведен ниже). Это наглядно демонстрирует алгоритм работы с файлами:

- перед началом работы файл необходимо 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 необходимо закрыть (шаг **CLOSE**);
- если была произведена запись, то после закрытия файла можно узнать его новый размер (шаг **GET_SIZE**).

Ниже приведен код и комментарии для каждого из шагов.

На шаге **OPEN** происходит открытие файла с помощью экземпляра ФБ [FILE.Open](#). В зависимости от управляющего сигнала (запись или чтение) файл открывается в режиме **MAPPD** (дозапись в конец файла) или **MREAD** (чтение из файла). При обращении 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 мы не будем обрабатывать ситуацию чтения из несуществующего файла. Результатом успешного открытия файла будет получение дескриптора (**hFile**), который будет использоваться при всех следующих действиях с данным файлом. Если файл успешно открыт, то происходит переход на шаг **WRITE** или **SET_READ_POS** (в зависимости от полученного управляющего сигнала).

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

Рис. 5.7.5. Код шага **OPEN**

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

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

Рис. 5.7.6. Код шага **CREATE**

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

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

Рис. 5.7.7. Код шага **WRITE**

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

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

Рис. 5.7.8. Код шага **FLUSH**

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

```
102     FileWork.SET_READ_POS: // шаг установки позиции для чтения из файла
103
104         fbFileSetPos(xExecute:=TRUE, hFile:=hFile, udiPos:=SIZEOF(stExportBinData)*(udiReadEntry-1));
105
106     IF fbFileSetPos.xDone THEN
107         fbFileSetPos(xExecute:=FALSE);
108
109         // позиция для чтения выбрана, теперь можно перейти к шагу чтения данных
110         eState := FileWork.READ;
111     END_IF
112
113     IF fbFileSetPos.xError THEN
114         // обработка ошибок
115     END_IF
116
```

Рис. 5.7.9. Код шага **SET_READ_POS**

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

```
118     FileWork.READ: // шаг чтения данных
119
120         fbFileRead(xExecute:=TRUE, hFile:=hFile, pBuffer:=ADR(stImportBinData), szBuffer:=SIZEOF(stImportBinData));
121
122     IF fbFileRead.xDone THEN
123         fbFileRead(xExecute:=FALSE);
124
125         // теперь можно перейти к шагу закрытия файла
126         eState := FileWork.CLOSE;
127     END_IF
128
129     IF fbFileRead.xError THEN
130         // обработка ошибок
131     END_IF
132
```

Рис. 5.7.10. Код шага **READ**

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

```
134     FileWork.CLOSE: // шаг закрытия файла
135
136         fbFileClose(xExecute:=TRUE, hFile:=hFile);
137
138     IF fbFileClose.xDone THEN
139         fbFileClose(xExecute:=FALSE);
140
141         IF xWBusy THEN
142             // после записи в файл узнаем его новый размер
143             eState := FileWork.GET_SIZE;
144         ELSE
145             // после чтения из файла его размер не изменится, так что...
146             // ...вернемся на шаг открытия файла для ожидания следующего управляющего сигнала
147             eState := FileWork.OPEN;
148         END_IF
149
150         xWBusy := FALSE;
151         xRBusy := FALSE;
152
153     END_IF
154
```

Рис. 5.7.11. Код шага **CLOSE**

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

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

Рис. 5.7.12. Код шага **GET_SIZE**

5.7.3. Создание визуализации

Создадим интерфейс оператора для работы с бинарными файлами. На рис. 5.7.13 приведен внешний вид экрана **Visu03_BinFileExample**, который включает в себя:

- элемент **Комбинированное окно – целочисленный**, используемый для выбора накопителя, с которым будет работать программа. Настройки элемента описаны в [п. 5.5.4](#). К элементу привязана переменная **iDevicePath**;
- прямоугольник **Путь к устройству**, отображающий значение переменной **sDevicePath**;
- прямоугольник **Имя файла** с привязанной переменной **sVisuFileName**. В настройках элемента на вкладке **InputConfiguration** для действия **OnClick** 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 **OnClick** 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 **OnClick** 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.

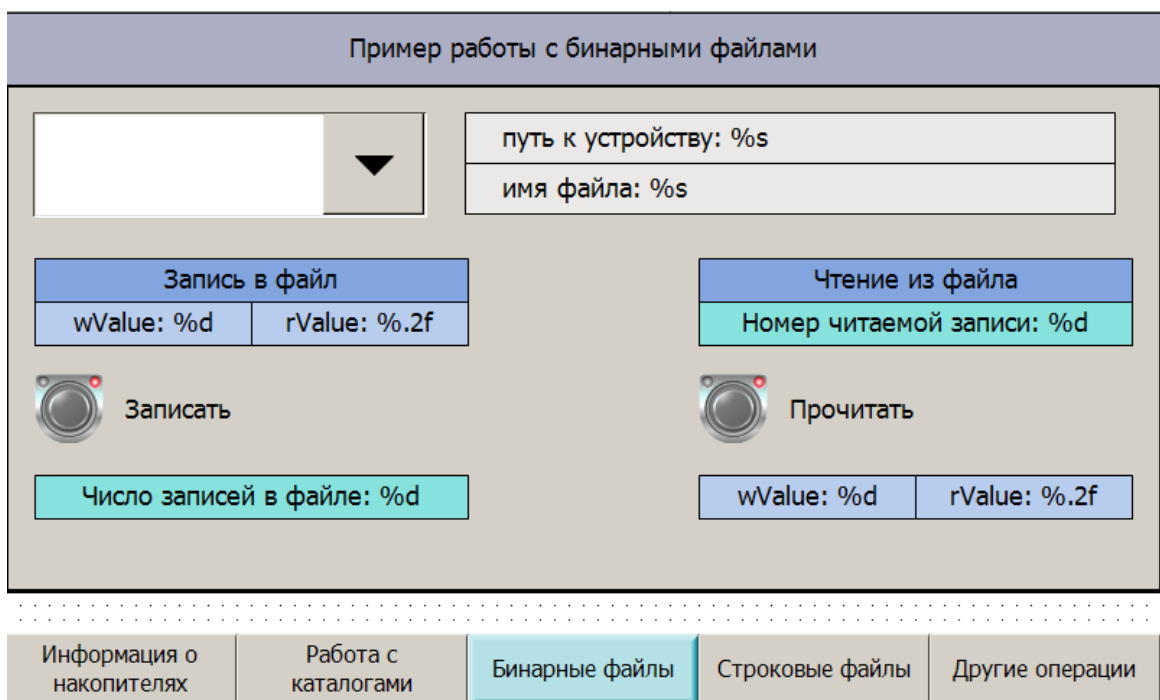


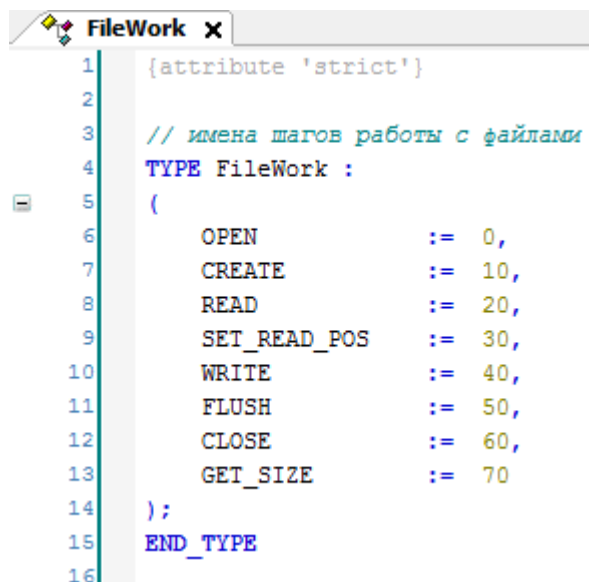
Рис. 5.7.13. Внешний вид экрана **Visu03_BinFileExample**

5.8. Экспорт текстовых файлов (StringFileExample_PRG)

В данном пункте приведен пример экспорта данных в текстовый файл формата [.csv](#). Формат [.csv](#) используется для представления табличных данных и состоит из текстовых записей, разграниченных символом-разделителем. В русской [локали](#) таким символом является точка с запятой (;). Информация о различиях бинарных и текстовых файлов приведена в [п. 2.6](#).

5.8.1. Объявление переменных

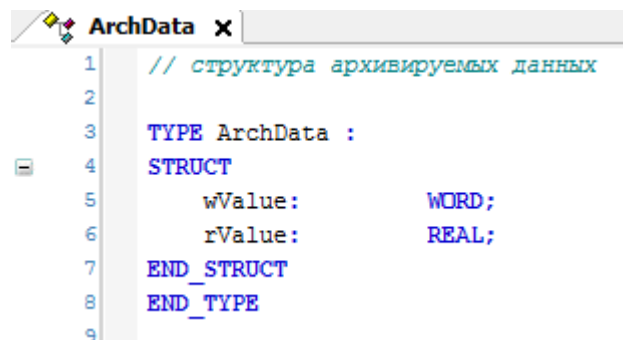
Как упоминалось в [п. 2.2](#), работа с файлами/каталогами можно представить в виде последовательности шагов, выполняемых с помощью оператора **CASE**. В качестве меток оператора **CASE** можно использовать обычные числа (0, 1, 2 и т.д.) – но это затруднит чтение программы. Соответствующее перечисление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



```
1 {attribute 'strict'}
2
3 // имена шагов работы с файлами
4 TYPE FileWork :
5 (
6     OPEN           := 0,
7     CREATE         := 10,
8     READ           := 20,
9     SET_READ_POS   := 30,
10    WRITE          := 40,
11    FLUSH          := 50,
12    CLOSE          := 60,
13    GET_SIZE       := 70
14 );
15 END_TYPE
16
```

Рис. 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](#):



```
1 // структура архивируемых данных
2
3 TYPE ArchData :
4 STRUCT
5     wValue:      WORD;
6     rValue:      REAL;
7 END_STRUCT
8 END_TYPE
9
```

Рис. 5.8.2. Объявление структуры **ArchData**

Объявим в программе **StringFileExample_PRG** следующие переменные и константы:

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

Рис. 5.8.3. Объявление переменных в программе **StringFileExample_PRG**

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

5.8.2. Разработка программы

Структура программы **StringFileExample_PRG** приведена рис. 5.8.4:

```
1 // считываем системное время
2 fbGetCurrentDT(xExecute:=NOT(fbGetCurrentDT.xDone));
3
4 IF fbGetCurrentDT.xDone THEN
5     // вырезаем отдельные разряды времени и конвертируем их в строки
6     fbSplitDT(dtDateAndTime:=fbGetCurrentDT.dtDateAndTime);
7 END_IF
8
9 // подготавливаем метку времени в виде форматированной строки
10 asDateTimeStrings[0] := fbSplitDT.sDay;
11 asDateTimeStrings[1] := '.';
12 asDateTimeStrings[2] := fbSplitDT.sMonth;
13 asDateTimeStrings[3] := '.';
14 asDateTimeStrings[4] := fbSplitDT.sYear;
15 asDateTimeStrings[5] := c_sDelimiter;
16 asDateTimeStrings[6] := fbSplitDT.sHour;
17 asDateTimeStrings[7] := ':';
18 asDateTimeStrings[8] := fbSplitDT.sMinute;
19 asDateTimeStrings[9] := ':';
20 asDateTimeStrings[10] := fbSplitDT.sSecond;
21
22 // собираем строку, которая будет записана в архив
23 asExportStringData[0] := CONCAT11(asDateTimeStrings);
24 asExportStringData[1] := c_sDelimiter;
25 asExportStringData[2] := WORD_TO_STRING(stExportData.wValue);
26 asExportStringData[3] := c_sDelimiter;
27 asExportStringData[4] := REAL_TO_FSTRING(stExportData.rValue,2);
28 asExportStringData[5] := '$N';
29
30 sArchEntry := CONCAT11(asExportStringData);
31
32 // получаем путь к выбранному устройству
33 sDevicePath := DEVICE_PATH(iDevicePath);
34
35 // склеиваем его с именем выбранного файла
36 sFileName := CONCAT(sDevicePath, sVisuFileName);
37
38 // детектируем сигнал записи в файл
39 fbWriteTrig(CLK:=xWrite);
40
41 // если получен сигнал записи, то вводим соответствующий флаг
42 IF fbWriteTrig.Q THEN
43     xWBusy := TRUE;
44 END_IF
45
46
47 CASE eState OF
48
49     FileWork.OPEN: // шаг открытия файла
50     [20 lines]
51     FileWork.CREATE: // шаг создания файла
52     [18 lines]
53     FileWork.WRITE: // шаг записи в буфер
54     [26 lines]
55     FileWork.FLUSH: // шаг сброса буфера в файл
56     [14 lines]
57     FileWork.CLOSE: // шаг закрытия файла
58     [11 lines]
59     FileWork.GET_SIZE: // шаг определения размера файла
60     [26 lines]
61 END_CASE
```

Рис. 5.8.4. Структура программы **StringFileExample_PRG**

В программе можно выделить два основных блока:

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

Обсудим содержимое первого блока. В его начале мы организуем циклическое чтение системного времени с помощью экземпляра ФБ **GetDateAndTime**. Время считывается в формате **DT**; мы вырезаем из него отдельные разряды с помощью ФБ **SPLIT_DT_TO_FSTRINGS** (который уже создан и использован нами в [п. 5.6.2](#)), после чего склеиваем их в форматированную строку с разделителями, получая метку времени для текущей записи архива.

После этого мы склеиваем метку времени и архивируемые значения в одну строку, формируя архивную запись. Для этого мы используем вспомогательные функции **CONCAT11** (она была создана в [п. 5.6.2](#)) и **REAL_TO_FSTRING**. Функция **REAL_TO_FSTRING** используется для преобразования значения с плавающей точкой в строковое представление с заданным количеством знаков после запятой. Кроме того, функция заменяет символ разделителя целой и дробной части с точки на запятую (для корректного отображения значений в **Microsoft Excel** и др. ПО в русских [локалях](#)). Код функции выглядит следующим образом:

```
REAL_TO_FSTRING x
1 // функция конвертирует значение типа REAL в строку с n знаков после запятой
2
3
4 FUNCTION REAL_TO_FSTRING :      STRING
5 VAR_INPUT
6   rVar:      REAL;      // входное значение
7   usiPrecision:  USINT;  // нужное кол-во знаков после запятой
8 END_VAR
9 VAR
10  uliVar:      ULINT;    // промежуточная переменная
11  lrVar:      LREAL;    // промежуточная переменная
12  sVar:      STRING;    // промежуточная переменная
13  xSign:      BOOL;     // знак входного значения. TRUE - минус
14 END_VAR
15
16 // определяем знак
17 xSign := (rVar<0.0);
18
19 // оставляем нужное кол-во знаков после запятой
20 uliVar := LREAL_TO_ULINT( ABS(rVar)*EXPT(10, usiPrecision) );
21 lrVar  := ULINT_TO_LREAL(uliVar) / EXPT(10, usiPrecision);
22 sVar   := LREAL_TO_STRING(lrVar);
23
24 // если нужно - возвращаем знак "минус"
25 IF xSign THEN
26   sVar := CONCAT('-', sVar);
27 END_IF
28
29 // меняем точку на запятую для корректного отображения в MS Excel
30 REAL_TO_FSTRING := REPLACE(sVar, ',', '1', FIND(sVar, '.'));
```

Рис. 5.8.5. Код функции **REAL_TO_FSTRING**

Как можно заметить, код почти полностью совпадает с кодом функции **REAL_TO_FWSTRING** из [п. 5.4.2](#). Оптимальным решением было бы объединить обе функции в одну, но для упрощения примера мы остановимся на варианте с двумя отдельными функциями.

Вспомним, что мы решили сохранять наш архив в формате **.csv**. В русских [локалях](#) разделителем для этого формата является символ **';**, ASCII-код которого содержится в константе **c_sDelimiter**. Последним символом строки архива является спецсимвол **\$N**, который соответствует переходу на новую строку (см. [п. 2.6](#)). Соответственно, после выполнения данного фрагмента кода

```
22 // собираем строку, которая будет записана в архив
23 asExportStringData[0] := CONCAT11(asDateTimeStrings);
24 asExportStringData[1] := c_sDelimiter;
25 asExportStringData[2] := WORD_TO_STRING(stExportData.wValue);
26 asExportStringData[3] := c_sDelimiter;
27 asExportStringData[4] := REAL_TO_FSTRING(stExportData.rValue,2);
28 asExportStringData[5] := c_sDelimiter;
29 asExportStringData[6] := '$N';
30
31 sArchEntry := CONCAT11(asExportStringData);
```

в переменную **sArchEntry** будет записана одна строка архива. Осталось только записать эту строку в файл.

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

```
33 // получаем путь к выбранному устройству
34 sDevicePath := DEVICE_PATH(iDevicePath);
35
36 // склеиваем его с именем выбранного файла
37 sFileName := CONCAT(sDevicePath, sVisuFileName);
38
39 // детектируем сигнал записи в файл
40 fbWriteTrig(CLK:=xWrite);
41
42 // если получен сигнал записи, то взводим соответствующий флаг
43 IF fbWriteTrig.Q THEN
44     xWBusy := TRUE;
45 END IF
```

Рис. 5.8.6. Фрагмент программы **StringFileExample_PRG**

Сама работа с файлами происходит в управляющем операторе **CASE**. На рис. 5.8.4. были приведены только имена шагов без раскрытия их программного кода (он будет приведен ниже). Это наглядно демонстрирует алгоритм работы с файлами:

- перед началом работы файл необходимо 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 необходимо закрыть (шаг **CLOSE**);
- если была произведена запись, то после закрытия файла можно узнать его новый размер (шаг **GET_SIZE**).

Ниже приведен код и комментарии для каждого из шагов.

На шаге **OPEN** происходит открытие файла с помощью экземпляра ФБ [FILE.Open](#). Файл открывается в режиме **MAPPD** (дозапись в конец файла). При обращении 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 при всех следующих действиях с данным файлом. Если файл успешно открыт, то происходит переход на шаг **WRITE**.

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

Рис. 5.8.7. Код шага **OPEN**

На шаге **CREATE** происходит создание файла с помощью экземпляра ФБ [FILE.Open](#). Для создания файла необходимо открыть его в режиме **MWRITE** – в этом случае 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 при всех следующих действиях с данным файлом. После создания файла происходит переход на шаг **WRITE**. В рамках примера обработка ошибок на данном шаге не производится.

```

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

```

Рис. 5.8.8. Код шага **CREATE**

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

```

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

```

Рис. 5.8.9. Код шага **WRITE**

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

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

Рис. 5.8.10. Код шага **FLUSH**

На шаге **CLOSE** происходит закрытие файла с помощью ФБ [FILE.Close](#). После закрытия файла происходит переход на шаг **GET_SIZE**.

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

Рис. 5.8.11. Код шага **CLOSE**

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

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

Рис. 5.8.12. Код шага **GET_SIZE**

5.8.3. Создание визуализации

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

- элемент **Комбинированное окно – целочисленный**, используемый для выбора накопителя, с которым будет работать программа. Настройки элемента описаны в [п. 5.5.4](#). К элементу привязана переменная **iDevicePath**;
- прямоугольник **Путь к устройству**, отображающий значение переменной **sDevicePath**;
- прямоугольник **Имя файла** с привязанной переменной **sVisuFileName**. В настройках элемента на вкладке **InputConfiguration** для действия **OnClick** 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 **OnClick** 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**.

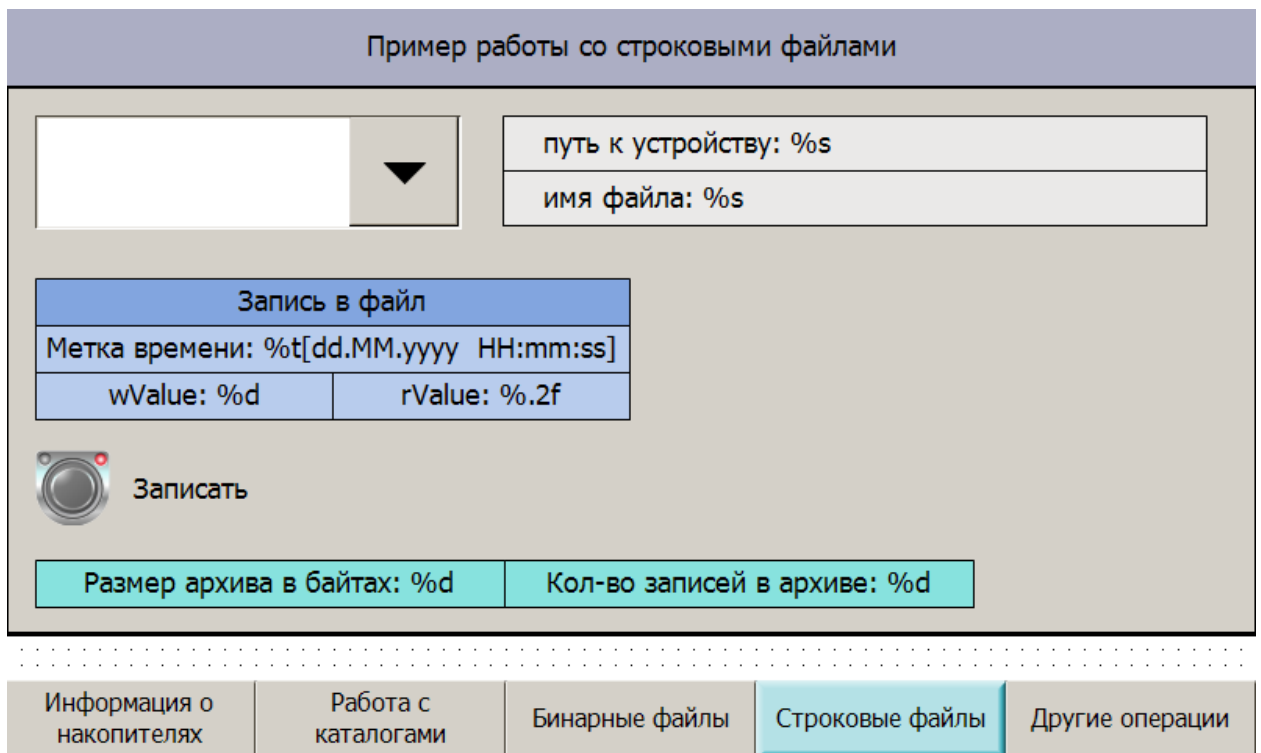


Рис. 5.8.13. Внешний вид экрана **Visu04_StringFileExample**

5.9. Дополнительные операции с файлами (PLC_PRG, действие act04_ActionsWithFiles)

В прошлых пунктах были рассмотрены основные, наиболее часто используемые операции с файлами – запись и чтение. В данном пункте рассмотрены другие доступные операции – переименование, копирование, удаление файлов.

5.9.1. Объявление переменных

Объявим в программе PLC_PRG следующие переменные:

```
56      (*act04_ActionsWithFiles | операции с файлами *)
57
58      fbFileRename:          FILE.Rename;          // ФБ переименования файла
59      fbFileCopy:           FILE.Copy;            // ФБ копирования файла
60      fbFileDelete:        FILE.Delete;          // ФБ удаления файла
61
62      sFileName:            STRING;               // полный путь к текущему файлу
63      sFileNameNew:         STRING;               // полный путь к создаваемому файлу
64      sVisuFileName:        STRING;               // имя текущего файла
65      sVisuFileNameNew:     STRING;               // имя создаваемого файла
66      sDeviceFilePath:      STRING;               // путь к текущему устройству
67      iDeviceFilePath:      INT;                  // ID текущего устройства
68      sDeviceFilePathCopy:  STRING;               // путь к устройству для копирования файла
69      iDeviceFilePathCopy:  INT;                  // ID устройства для копирования файла
70      sFileNameCopy:        STRING;               // полный путь для копирования файла
```

Рис. 5.9.1. Объявление переменных в программе PLC_PRG

4.9.2. Разработка программы

Создадим в программе PLC_PRG действие act04_ActionWithFiles (PLC_PRG – Добавление объекта – Действие) и вынесем в него следующий код:

```
PLC_PRG.act04_ActionsWithFiles x
1 // получаем путь к выбранному устройству
2 sDeviceFilePath := DEVICE_PATH(iDeviceFilePath);
3
4 // склеиваем его с именами файлов
5 sFileName := CONCAT(sDeviceFilePath, sVisuFileName);
6 sFileNameNew := CONCAT(sDeviceFilePath, sVisuFileNameNew);
7
8 // получаем путь к выбранному устройству для копирования
9 sDeviceFilePathCopy := DEVICE_PATH(iDeviceFilePathCopy);
10 // склеиваем его с именем файла
11 sFileNameCopy := CONCAT(sDeviceFilePathCopy, sVisuFileName);
12
13 // выполняем ФБ операций с файлами
14 fbFileRename(xExecute:=, sFileNameOld:=sFileName, sFileNameNew:=sFileNameNew);
15 fbFileCopy (xExecute:=, xOverWrite:=TRUE, sFileNameDest:=sFileNameCopy, sFileNameSource:=sFileName);
16 fbFileDelete(xExecute:=, sFileName:=sFileName);
17
```

Рис. 5.9.2. Код действия act04_ActionWithFiles

Действие производит следующие операции:

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

Необходимо отметить следующее:

1. 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.
2. 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.

Добавим вызов созданного действия в программу **PLC_PRG**:

```
PLC_PRG x
46     FbSplitDT:          SPLIT_DT_TO_FSTRINGS; // ФБ конвертации времени в строку
47     asEntryDT:         ARRAY [0..10] OF STRING; // метка времени в виде отдельных строковых разрядов
48
49     iSelectedEntry:    INT; // номер выбранной строки таблицы
50
51     xDown:             BOOL; // сигнал "Открыть каталог"
52     xUp:               BOOL; // сигнал "Перейти на уровень выше"
53     xFirstScan:       BOOL; // сигнал "Сканирование каталога"
54
-----
1     act01_DriveInfo(); // сбор информации о памяти СПК и накопителей
2     act02_DirExample(); // пример работы с каталогами (создание, переименование, удаление)
3     act03_DirList(); // пример получения информации о содержимом каталога
4     act04_ActionsWithFiles(); // пример работы с файлами (переименование, копирование, удаление)
5
```

Рис. 5.9.3. Вызов действия **act04_ActionsWithFiles** в программе **PLC_PRG**

5.9.3. Создание визуализации

Создадим интерфейс оператора для работы с каталогами. На рис. 5.9.4 приведен внешний вид экрана **Visu05_FilesActionsExample**, который включает в себя:

- 2 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](#).
- 2 прямоугольника **Путь**, отображающие значения переменных **sDeviceFilePath** и **sDeviceFilePathCopy**;
- 3 прямоугольника **Текущее имя файла** с привязанной переменной **sVisuFileName**. В настройках элементов на вкладке **InputConfiguration** для действия **OnMouseClicked** задана операция **Записать переменную** (тип ввода – диалог **VisuKeypad**).
- прямоугольник **Новое имя файла** с привязанной переменной **sVisuFileNameNew**. В настройках элемента на вкладке **InputConfiguration** для действия **OnMouseClicked** задана операция **Записать переменную** (тип ввода – диалог **VisuKeypad**).
- 3 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**.

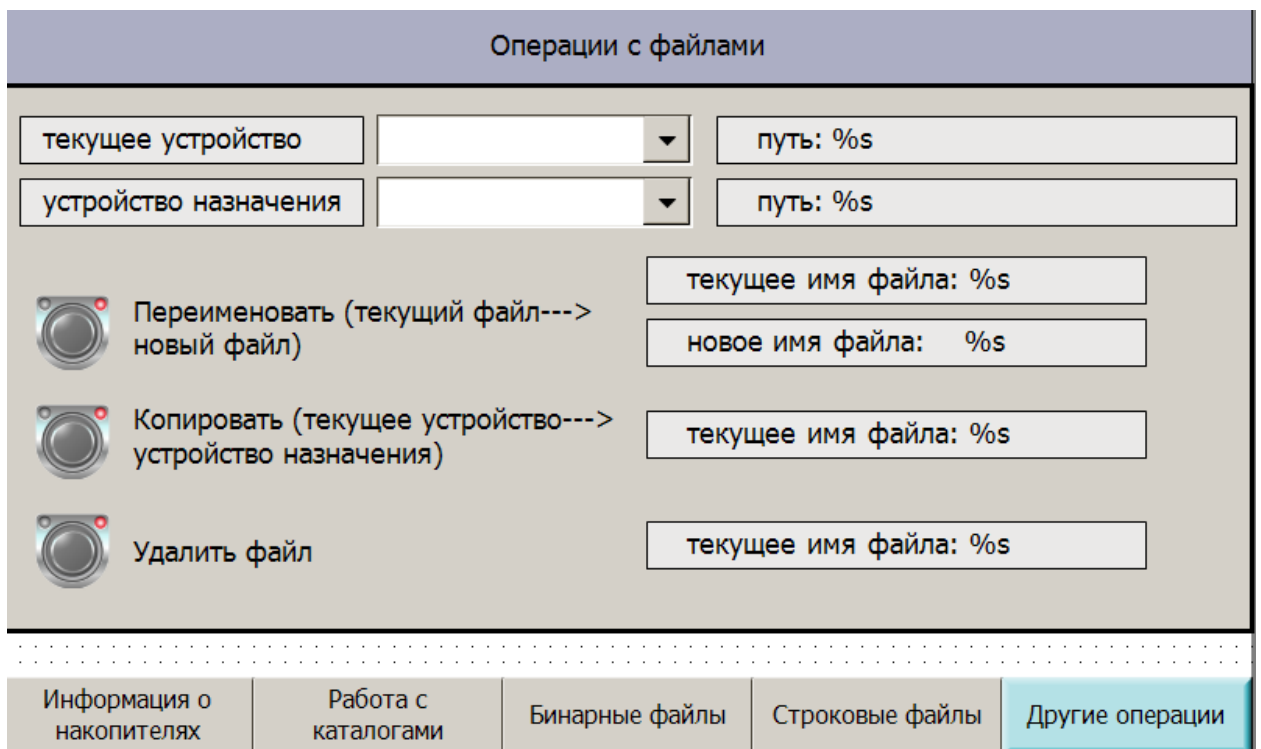


Рис. 5.9.4. Внешний вид экрана **Visu05_FilesActionsExample**

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

5.10. Работа с примером

1. Подключитесь к контроллеру и загрузите в него проект. Если модель вашего контроллера отличается от использованной в примере (**СПК207.03.CS.Web**), то выберите 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.

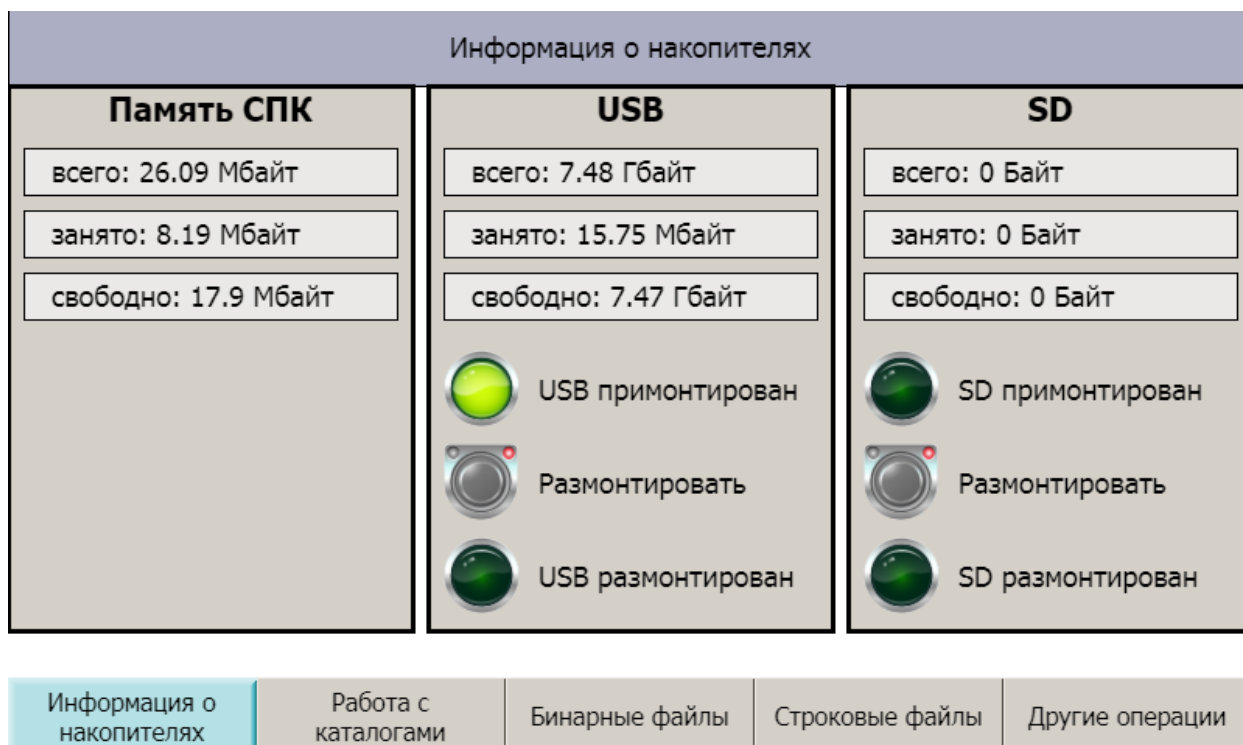


Рис. 5.10.1. Внешний вид экрана **Информация о накопителях (Visu01_DriveInfo)**

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

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

3. На экране **Работа с каталогами** выберите нужное устройство и введите имя нового каталога (например, **test**). Нажмите кнопку **Создать новый**.

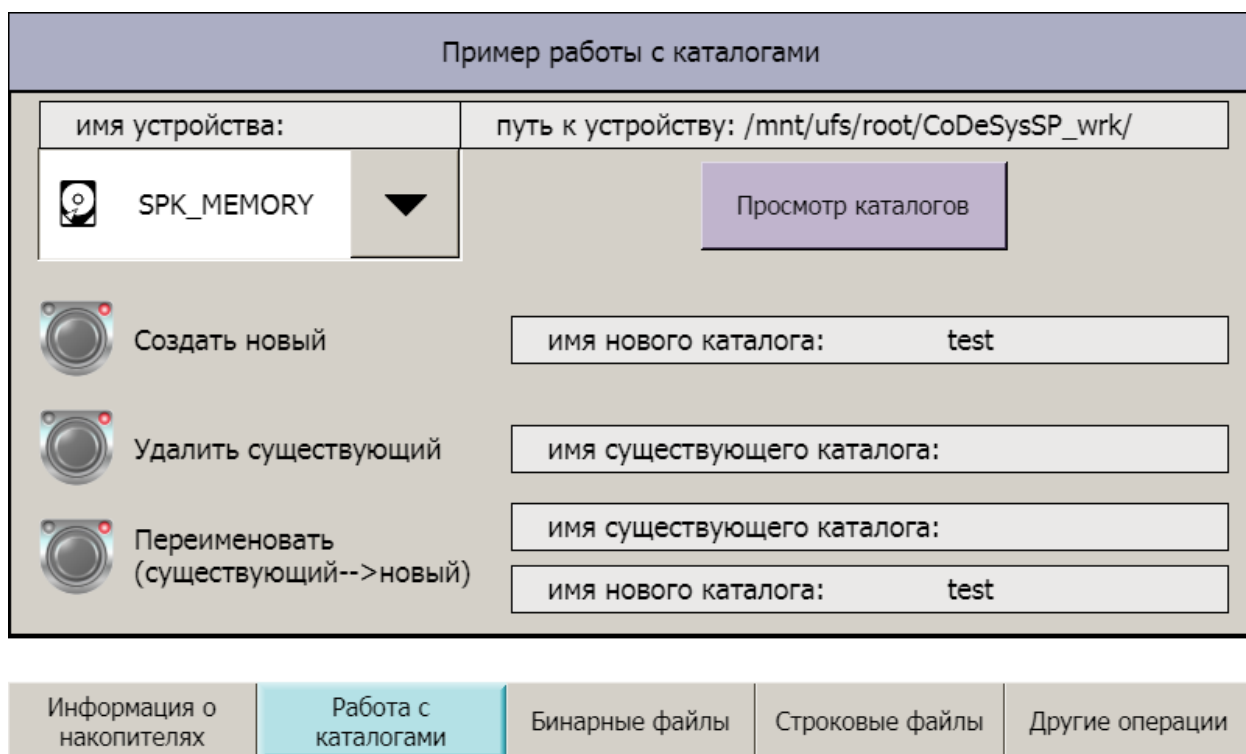


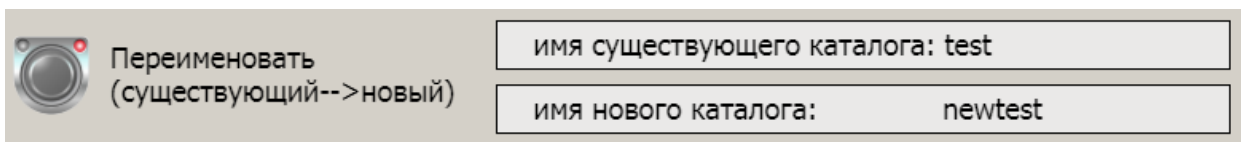
Рис. 5.10.2. Внешний вид экрана **Работа с каталогами (Visu02_DirExample)**

Подключитесь к контроллеру с помощью утилиты **WinSCP** (см. п. 2.8) и перейдите по отображаемому пути (на рис. 5.10.2 это **/mnt/ufs/root/CoDeSysSP_wrk**). Убедитесь, что был создан каталог с названием **test**.

/root/CoDeSysSP_wrk				
Имя	Размер	Изменено	Права	Владел...
..		23.06.2017 8:25	rw-rw-r--	root
test		02.08.2017 10:23	rw-r--r--	root
visu		28.07.2017 10:29	rw-rw-r--	1000
3S.dat	1 KB	22.06.2017 15:02	rw-rw-rw-	root
Application.app	2 081 KB	02.08.2017 9:59	rw-r--r--	root
Application.crc	1 KB	02.08.2017 9:59	rw-r--r--	root
codesysssp	1 KB	22.06.2017 15:02	rw-rw-rw-	root
CoDeSysSP.cfg	6 KB	31.07.2017 10:33	rw-r--r--	root
libCmpLicOemContr...	1 KB	22.06.2017 15:02	rw-rw-rw-	root
libCmpRSmode.so	1 KB	22.06.2017 15:02	rw-rw-rw-	root
libCmpSysExec.so	1 KB	22.06.2017 15:02	rw-rw-rw-	root
libPID_REG3_v32.so	1 KB	22.06.2017 15:02	rw-rw-rw-	root
libSysTargetOEM.so	1 KB	22.06.2017 15:02	rw-rw-rw-	root
libtumbler.so	1 KB	22.06.2017 15:02	rw-rw-rw-	root

Рис. 5.10.3. Проверка создания нового каталога

Теперь введите имя существующего каталога **test** и имя нового каталога **newtest**.

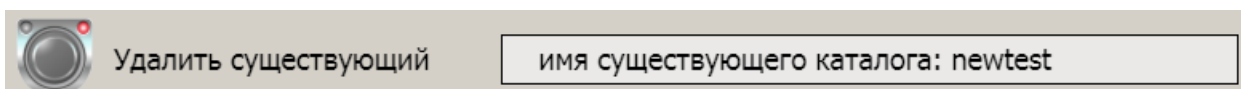


Нажмите кнопку **Переименовать**. Проверьте через **WinSCP**, что каталог был переименован:

Имя	Размер	Изменено	Права	Владел...
..		23.06.2017 8:25	rwxrwxr-x	root
newtest		02.08.2017 10:23	rwxr-xr--	root
visu		28.07.2017 10:29	rwxrwxr-x	1000
3S.dat	1 KB	22.06.2017 15:02	rwxrwxrwx	root
Application.app	2 081 KB	02.08.2017 9:59	rw-r--r--	root
Application.crc	1 KB	02.08.2017 9:59	rw-r--r--	root
codesysssp	1 KB	22.06.2017 15:02	rwxrwxrwx	root
CoDeSysSP.cfg	6 KB	31.07.2017 10:33	rw-r--r--	root
libCmpLicOemContr...	1 KB	22.06.2017 15:02	rwxrwxrwx	root
libCmpRmode.so	1 KB	22.06.2017 15:02	rwxrwxrwx	root
libCmpSysExec.so	1 KB	22.06.2017 15:02	rwxrwxrwx	root
libPID_REG3_v32.so	1 KB	22.06.2017 15:02	rwxrwxrwx	root
libSysTargetOEM.so	1 KB	22.06.2017 15:02	rwxrwxrwx	root
libtumbler.so	1 KB	22.06.2017 15:02	rwxrwxrwx	root

Рис. 5.10.4. Проверка переименования каталога

Введите имя существующего каталога **newtest** и нажмите кнопку **Удалить**. Проверьте через **WinSCP**, что каталог был удален:



Имя	Размер	Изменено	Права	Владел...
..		23.06.2017 8:25	rwxrwxr-x	root
visu		28.07.2017 10:29	rwxrwxr-x	1000
3S.dat	1 KB	22.06.2017 15:02	rwxrwxrwx	root
Application.app	2 081 KB	02.08.2017 9:59	rw-r--r--	root
Application.crc	1 KB	02.08.2017 9:59	rw-r--r--	root
codesysssp	1 KB	22.06.2017 15:02	rwxrwxrwx	root
CoDeSysSP.cfg	6 KB	31.07.2017 10:33	rw-r--r--	root
libCmpLicOemContr...	1 KB	22.06.2017 15:02	rwxrwxrwx	root
libCmpRmode.so	1 KB	22.06.2017 15:02	rwxrwxrwx	root
libCmpSysExec.so	1 KB	22.06.2017 15:02	rwxrwxrwx	root
libPID_REG3_v32.so	1 KB	22.06.2017 15:02	rwxrwxrwx	root
libSysTargetOEM.so	1 KB	22.06.2017 15:02	rwxrwxrwx	root
libtumbler.so	1 KB	22.06.2017 15:02	rwxrwxrwx	root

Рис. 5.10.5. Проверка удаления каталога

Нажмите кнопку **Просмотр каталогов** (см. рис. 5.10.2). Выберите нужный накопитель. Будет автоматически произведено сканирование его корневого каталога. Результаты будут представлены в таблице. Выберите нужный каталог с помощью курсора или элемента **Полоса прокрутки**, после чего нажмите кнопку **Открыть каталог**. Чтобы выйти из каталога, нажмите кнопку на **Уровень выше**. Выйти из рабочего каталога нельзя.

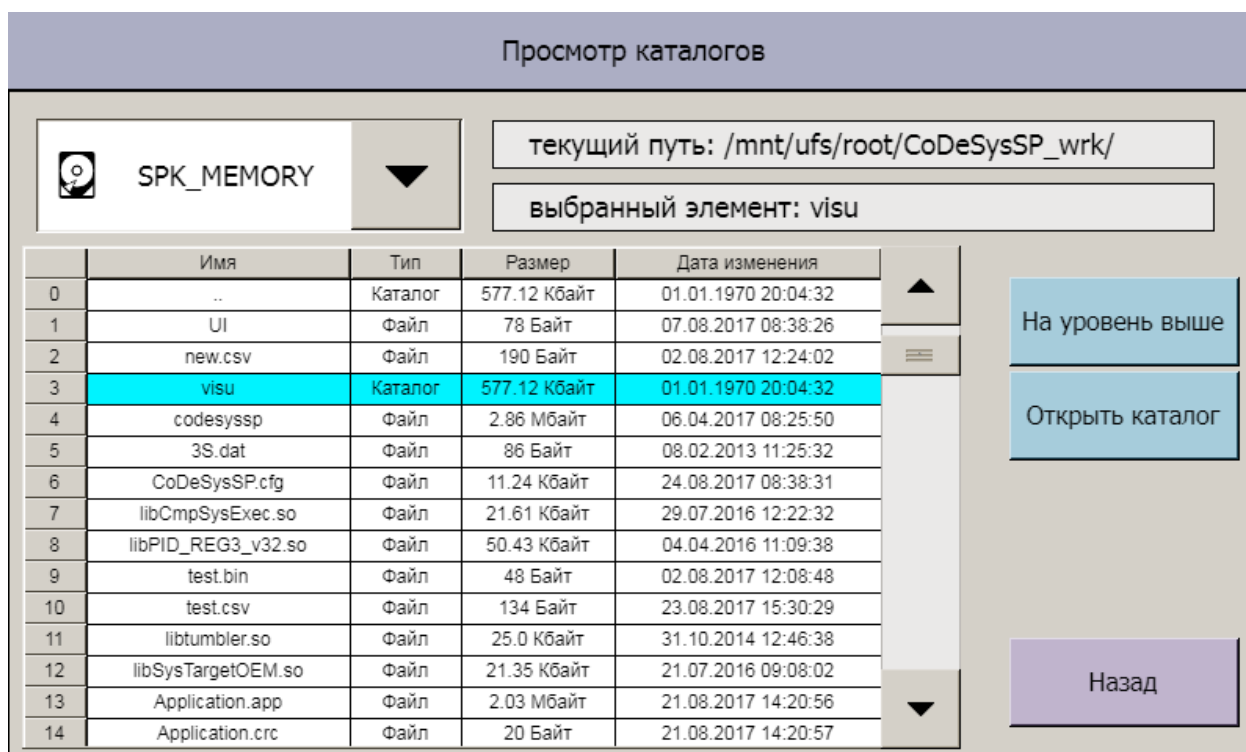


Рис. 5.10.6. Внешний вид экрана **Просмотр каталогов (Visu06_DirList)**

Как можно заметить, размер и дата изменения каталогов отображается некорректно. Также не поддерживается отображения русскоязычных символов. Подробнее см. в [п. 5.2.1](#).

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

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

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

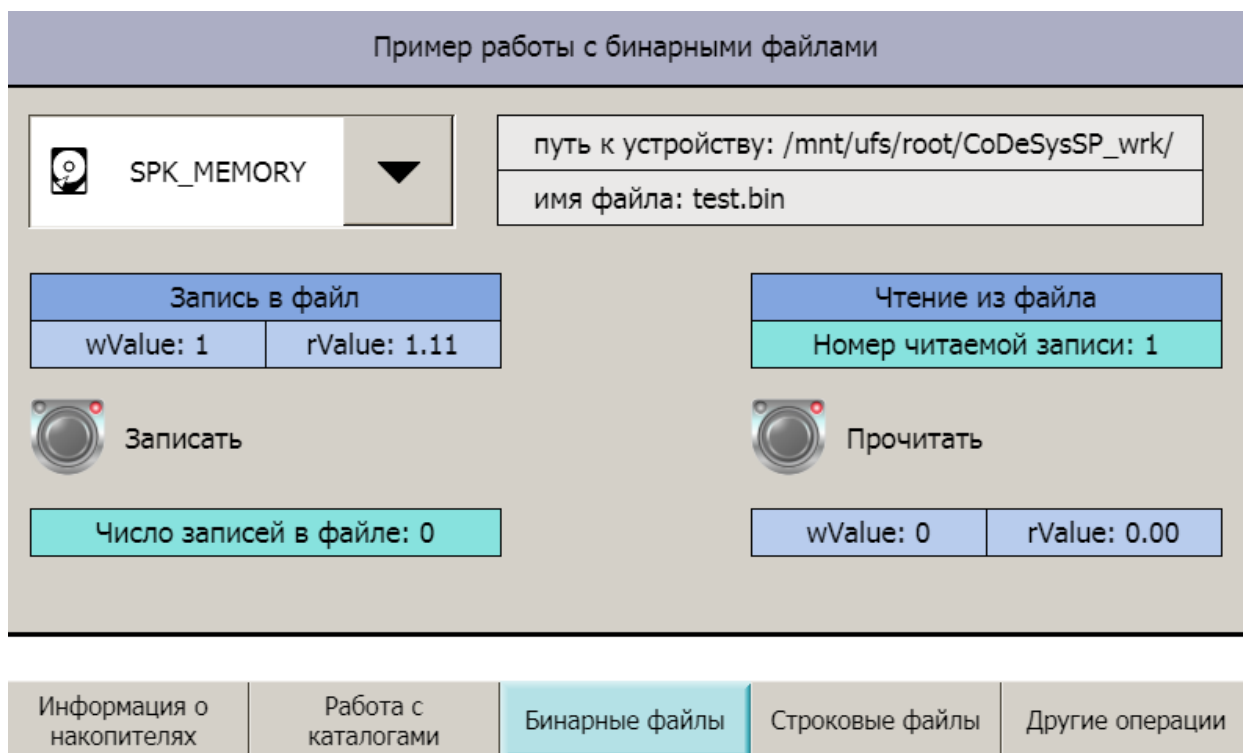


Рис. 5.10.7. Внешний вид экрана **Бинарные файлы** (Visu03_BinFileExample)

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

/root/CoDeSysSP_wrk				
Имя	Размер	Изменено	Права	Владел...
..		23.06.2017 8:25	rw-rw-r-x	root
visu		28.07.2017 10:29	rw-rw-r-x	1000
3S.dat	1 KB	22.06.2017 15:02	rw-rw-rwx	root
Application.app	2 081 KB	02.08.2017 9:59	rw-r--r--	root
Application.crc	1 KB	02.08.2017 9:59	rw-r--r--	root
codesysp	1 KB	22.06.2017 15:02	rw-rw-rwx	root
CoDeSysSP.cfg	6 KB	31.07.2017 10:33	rw-r--r--	root
libCmpLicOemContr...	1 KB	22.06.2017 15:02	rw-rw-rwx	root
libCmpRmode.so	1 KB	22.06.2017 15:02	rw-rw-rwx	root
libCmpSysExec.so	1 KB	22.06.2017 15:02	rw-rw-rwx	root
libPID_REG3_v32.so	1 KB	22.06.2017 15:02	rw-rw-rwx	root
libSysTargetOEM.so	1 KB	22.06.2017 15:02	rw-rw-rwx	root
libtumbler.so	1 KB	22.06.2017 15:02	rw-rw-rwx	root
test.bin	1 KB	02.08.2017 12:08	rw-r--r--	root

Рис. 5.10.8. Проверка создания файла

Выберите номер читаемой записи (по умолчанию – 1) и нажмите кнопку **Прочитать**. Данные из файла будут считаны в переменные **wValue** и **rValue** (*чтение из файла*).

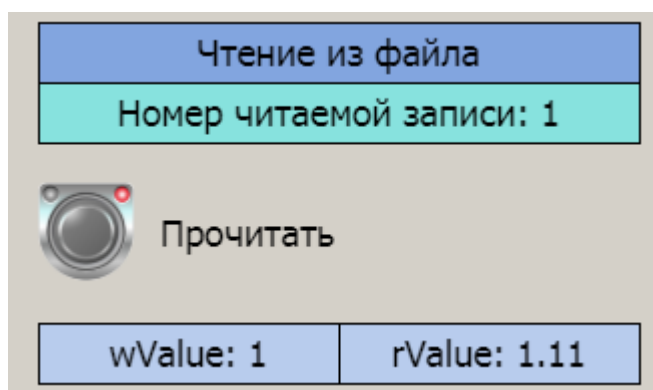


Рис. 5.10.9. Чтение данных из файла

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

5. На экране **Строковые файлы** выберите нужное устройство и введите имя файла (по умолчанию – **test.csv**). Нажмите кнопку **Записать**, чтобы создать файл и записать в него заголовок архива. Задайте значения **wValue** и **rValue** и нажмите кнопку **Записать**. Повторите операцию несколько раз, меняя значения переменных. Счетчик числа записей и размера архива будет увеличиваться после каждой записи.

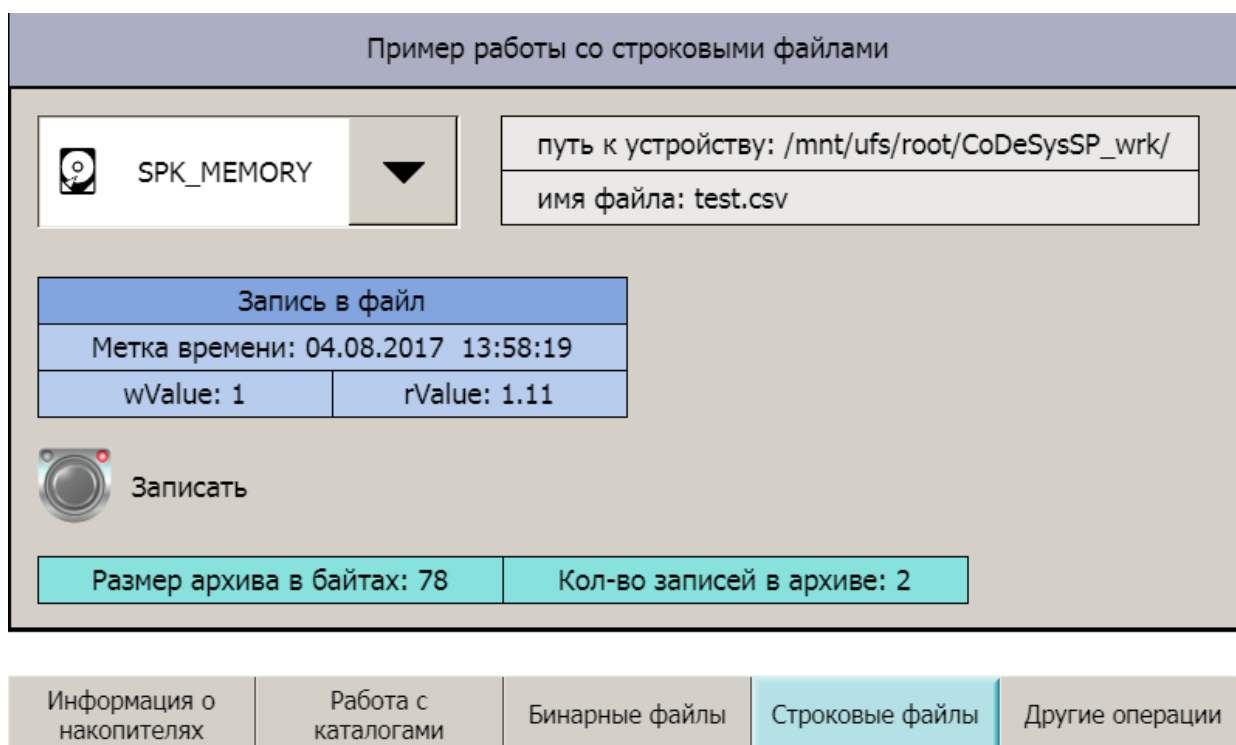


Рис. 5.10.10. Внешний вид экрана **Строковые файлы** (Visu04_StringFileExample)

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

Имя	Размер	Изменено	Права	Владел...
..		23.06.2017 8:25	rw-rw-r-x	root
visu		28.07.2017 10:29	rw-rw-r-x	1000
3S.dat	1 KB	22.06.2017 15:02	rw-rw-rwx	root
Application.app	2 081 KB	02.08.2017 9:59	rw-r--r--	root
Application.crc	1 KB	02.08.2017 9:59	rw-r--r--	root
codesysssp	1 KB	22.06.2017 15:02	rw-rw-rwx	root
CoDeSysSP.cfg	6 KB	31.07.2017 10:33	rw-r--r--	root
libCmpLicOemContr...	1 KB	22.06.2017 15:02	rw-rw-rwx	root
libCmpRSmode.so	1 KB	22.06.2017 15:02	rw-rw-rwx	root
libCmpSysExec.so	1 KB	22.06.2017 15:02	rw-rw-rwx	root
libPID_REG3_v32.so	1 KB	22.06.2017 15:02	rw-rw-rwx	root
libSysTargetOEM.so	1 KB	22.06.2017 15:02	rw-rw-rwx	root
libtumbler.so	1 KB	22.06.2017 15:02	rw-rw-rwx	root
test.bin	1 KB	02.08.2017 12:08	rw-r--r--	root
test.csv	1 KB	02.08.2017 12:24	rw-r--r--	root

Рис. 5.10.11. Проверка создания файла

Скопируйте его на ПК и откройте с помощью **Microsoft Excel** или другого ПО:

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

Рис. 5.10.12. Пример содержимого файла

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

6. На экране **Другие операции** выберите текущее устройство. Введите текущее и новое имя файла (например, **test.csv** и **new.csv**) и нажмите кнопку **Переименовать**.

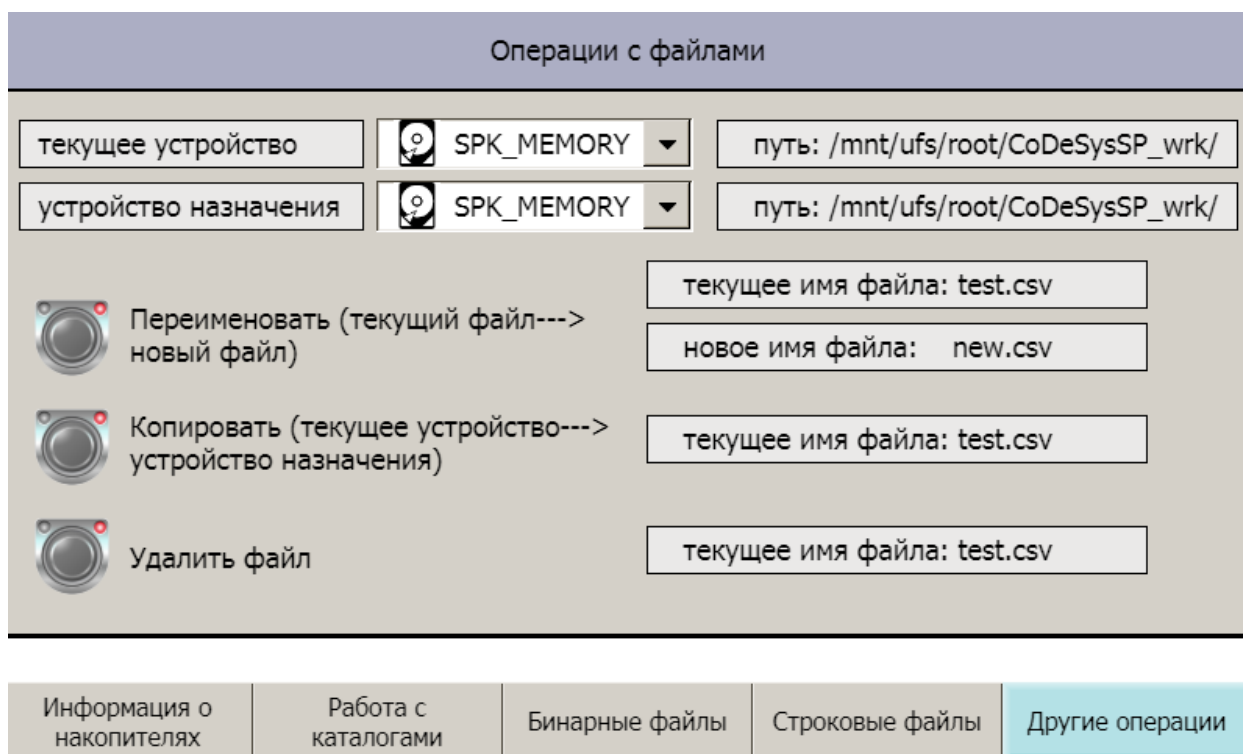


Рис. 5.10.13. Внешний вид экрана **Другие операции** (Visu05_FileActionsExample)

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

/root/CoDeSysSP_wrk				
Имя	Размер	Изменено	Права	Владел...
		23.06.2017 8:25	rw-rwxr-x	root
visu		28.07.2017 10:29	rw-rwxr-x	1000
3S.dat	1 KB	22.06.2017 15:02	rw-rwxrwx	root
Application.app	2 081 KB	02.08.2017 9:59	rw-r--r--	root
Application.crc	1 KB	02.08.2017 9:59	rw-r--r--	root
codesysssp	1 KB	22.06.2017 15:02	rw-rwxrwx	root
CoDeSysSP.cfg	6 KB	31.07.2017 10:33	rw-r--r--	root
libCmpLicOemContr...	1 KB	22.06.2017 15:02	rw-rwxrwx	root
libCmpRSmode.so	1 KB	22.06.2017 15:02	rw-rwxrwx	root
libCmpSysExec.so	1 KB	22.06.2017 15:02	rw-rwxrwx	root
libPID_REG3_v32.so	1 KB	22.06.2017 15:02	rw-rwxrwx	root
libSysTargetOEM.so	1 KB	22.06.2017 15:02	rw-rwxrwx	root
libtumbler.so	1 KB	22.06.2017 15:02	rw-rwxrwx	root
new.csv	1 KB	02.08.2017 12:24	rw-r--r--	root
test.bin	1 KB	02.08.2017 12:08	rw-r--r--	root

Рис. 5.10.14. Проверка переименования файла

Введите текущее имя файла **test.bin** и выберите устройства назначения (например, USB).
Нажмите кнопку **Копировать**.

текущее устройство: SPK_MEMORY путь: /mnt/ufs/root/CoDeSysSP_wrk/

устройство назначения: USB путь: /mnt/ufs/media/sda1/

Переименовать (текущий файл---> новый файл)
 - текущее имя файла: test.bin
 - новое имя файла: new.csv

Копировать (текущее устройство---> устройство назначения)
 - текущее имя файла: test.bin

Рис. 5.10.15. Копирование файлов

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

/mnt/ufs/media/sda1				
Имя	Размер	Изменено	Права	Владел...
..		02.08.2017 12:39	rwXr-xr-x	root
APP		23.06.2017 11:37	rwXrwXrwX	root
System Volume Infor...		15.06.2017 12:50	rwXrwXrwX	root
test.bin	1 KB	02.08.2017 12:39	rwXrwXrwX	root

Рис. 5.10.15. Проверка копирования файла

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

/mnt/ufs/media/sda1				
Имя	Размер	Изменено	Права	Владел...
..		02.08.2017 12:39	rwXr-xr-x	root
APP		23.06.2017 11:37	rwXrwXrwX	root
System Volume Infor...		15.06.2017 12:50	rwXrwXrwX	root

Рис. 5.10.16. Проверка удаления файла

5.11. Рекомендации и замечания

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

- ФБ и программы, работающие с файлами, разбиваются на шаги, которые выполняются через оператор **CASE**.
- Для того 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 – помните о максимальной длине строк (см. рис. 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. Контролируйте выходы **xError** и **eError** соответствующих ФБ. См. описание кодов ошибок в [п. 5.2.2](#).
- Не следует пытаться открыть уже открытый или закрыть уже закрытый файл.
- Перед началом работы с файлом, расположенном на внешнем накопителе, проверьте, примонтирован ли этот накопитель к контроллеру.
- Перед извлечением накопителя его необходимо размонтировать. Позаботьтесь о соответствующих окнах/сообщениях в визуализации, чтобы это было очевидно для оператора.
- Контролируйте доступное свободное место на контроллере/накопителе. Если оно заканчивается, следует остановить архивацию или начать перезаписывать файл.
- Оптимальным решением является в каждый момент времени работать только с одним файлом. Попытка архивировать данные в несколько файлов одновременно может привести к снижению стабильности работы основной программы.

Приложение. Листинг примера

А. Структуры и перечисления

А.1. Структура ArchData

```
// структура архивируемых данных

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

А.2. Структура DriveInfo

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

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

    lwFullSize:      LWORD;          // общий объем накопителя (в байтах)
    lwUsedSize:      LWORD;          // занятый объем накопителя (в байтах)
    lwFreeSize:      LWORD;          // свободный объем накопителя (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
```

А.3. Перечисление FileDevice

```
{attribute 'strict'}

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

A.4. Перечисление FileDevice

```
{attribute 'strict'}

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

A.5. Структура VisuDirInfo

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

Б. Структуры и перечисления

Б.1. Функция BYTE_SIZE_TO_WSTRING

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

FUNCTION BYTE_SIZE_TO_WSTRING : WSTRING
VAR_INPUT
    lwByteSize:          LWORD;          // число байт
END_VAR
VAR CONSTANT
    c_KB:                LWORD:=1024;    // число байт в килобайте
    c_MB:                LWORD:=1024*c_KB; // число килобайт в мегабайте
    c_GB:                LWORD:=1024*c_MB; // число мегабайт в гигабайте
END_VAR
VAR
    rByteSize:          REAL;          // промежуточная переменная
END_VAR

CASE lwByteSize OF
    0 ..(c_KB-1):
        BYTE_SIZE_TO_WSTRING := WCONCAT(LWORD_TO_WSTRING(lwByteSize), " Байт");

    c_KB ..(c_MB-1):
        rByteSize := LWORD_TO_REAL(lwByteSize) / LWORD_TO_REAL(c_KB);
        BYTE_SIZE_TO_WSTRING := WCONCAT(REAL_TO_FWSTRING(rByteSize, 2), " Кбайт");

    c_MB ..(c_GB-1):
        rByteSize := LWORD_TO_REAL(lwByteSize) / LWORD_TO_REAL(c_MB);
        BYTE_SIZE_TO_WSTRING := WCONCAT(REAL_TO_FWSTRING(rByteSize, 2), " Мбайт");

    c_GB ..(32*c_GB):
        rByteSize := LWORD_TO_REAL(lwByteSize) / LWORD_TO_REAL(c_GB);
        BYTE_SIZE_TO_WSTRING := WCONCAT(REAL_TO_FWSTRING(rByteSize, 2), " Гбайт");

END_CASE
```

Б.2. Функция CONCAT11

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

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

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

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

CONCAT11:=sBuffer;
```

Б.3. Функция DEVICE_PATH

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

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

CASE iDevice OF
    FileDevice.SPK_MEMORY:
        DEVICE_PATH:='/mnt/ufs/root/CoDeSysSP_wrk/';
    FileDevice.USB:
        DEVICE_PATH:='/mnt/ufs/media/sdal/';
    FileDevice.SD:
        DEVICE_PATH:='/mnt/ufs/media/mmcblk0p1/';
END_CASE
```

Б.4. ФБ DIR_INFO

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

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

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

```

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

// сбрасываем сигнал завершения работы
xDone:=FALSE;

CASE eState OF

    FileWork.OPEN:          // открываем каталог

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

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

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

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

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

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

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

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

        fbDirList(xExecute:=FALSE);
    END_IF

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

    FileWork.CLOSE:       // завершение работы блока

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

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

            // устанавливаем флаг завершения работы
            xDone :=      TRUE;

            eState :=      FileWork.OPEN;
        END_IF

END_CASE

```


Б.5. Функция LEAD_ZERO

```
// функция преобразует число в строку с ведущим нулем
FUNCTION LEAD_ZERO : STRING
VAR_INPUT
    uiInput:      UINT;
END_VAR
VAR
END_VAR

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

Б.6. Функция REAL_TO_FSTRING

```
// функция конвертирует значение типа REAL в строку с n знаков после запятой
FUNCTION REAL_TO_FSTRING :      STRING
VAR_INPUT
    rVar:      REAL;           // входное значение
    usiPrecision:  USINT;      // нужное кол-во знаков после запятой
END_VAR
VAR
    uliVar:      ULINT;        // промежуточная переменная
    lrVar:      LREAL;        // промежуточная переменная
    sVar:      STRING;        // промежуточная переменная
END_VAR

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

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

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

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

Б.7. Функция REAL_TO_FWSTRING

```
// функция конвертирует значение типа REAL в строку с n знаков после запятой
FUNCTION REAL_TO_FWSTRING :      WSTRING
VAR_INPUT
    rVar:      REAL;           // входное значение
    usiPrecision:  USINT;      // нужное кол-во знаков после запятой
END_VAR
VAR
    uliVar:      ULINT;        // промежуточная переменная
    lrVar:      LREAL;        // промежуточная переменная
END_VAR

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

Б.8. ФБ SPLIT_DT_TO_FSTRINGS

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

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

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

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

В. Программа PLC_PRG

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

PROGRAM PLC_PRG

VAR

(*act01_DriveInfo | информация о памяти СПК и накопителей*)

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

stSpkMemory:         DriveInfo;           // структура параметров памяти СПК
stUSB:               DriveInfo;           // структура параметров памяти USB-накопителя
stSD:               DriveInfo;           // структура параметров памяти SD-накопителя

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

(*act02_DirExample | операции с каталогами*)

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

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

(*act03_DirList | информация о выбранном каталоге*)

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

// путь к выбранному каталогу

```
sDirListPath:        STRING := '/mnt/ufs/root/CoDeSysSP_wrk/';
```

// путь к предыдущему выбранному каталогу

```
sLastDevice:         STRING;
```

// массив данных о вложенных файлах/каталогах для визуализации

```
astVisuDirInfo:      ARRAY [0..c_MAX_ENTRIES] OF VisuDirInfo;
```

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

```
iSelectedEntry:      INT;                 // номер выбранной строки таблицы
```

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

(*act04_ActionsWithFiles | операции с файлами *)

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

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

END_VAR

```

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

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

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

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

// код программы PLC_PRG

act01_DriveInfo();        // сбор информации о памяти СПК 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)

```

B.1. Действие act01_DriveInfo

```

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

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

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

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

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

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

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

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

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

```

B.2. Действие act02_DirExample

```

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

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

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

```

В.3. Действие act03_DirList

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

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

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

// по сигналу переходим в выбранный каталог
IF xDown THEN

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

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

    xDown             :=      FALSE;
    xDirList          :=      TRUE;
END_IF

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

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

    // справа налево стираем символы из пути до тех пор, пока не найдем "/"
    // таким образом, из текущего пути будет удален самый нижний каталог
    FOR i:=LEN(sDirListPath)-2 TO 0 BY -1 DO

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

    xUp               :=      FALSE;
    xDirList          :=      TRUE;
END_IF
```

```

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

IF fbDirInfo.xDone THEN

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

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

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

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

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

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

xDirList := FALSE;

END_FOR
END_IF

```

B.4. Действие act04_ActionsWithFiles

```

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

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

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

// выполняем ФБ операций с файлами
fbFileRename(xExecute:=, sFileNameOld:=sFileName, sFileNameNew:=sFileNameNew);

fbFileCopy (xExecute:=, xOverWrite:=TRUE, sFileNameDest:=sFileNameCopy,
            sFileNameSource:=sFileName);
fbFileDelete(xExecute:=, sFileName:=sFileName);

```

Г. 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

```

Д. Программа StringFileExample

```
// пример экспорта данных в текстовый файл

PROGRAM StringFileExample_PRG
VAR
    fbFileOpen:           FILE.Open;           // ФБ открытия файла
    fbFileClose:          FILE.Close;          // ФБ закрытия файла
    fbFileWrite:          FILE.Write;          // ФБ записи в файл
    fbFileFlush:          FILE.Flush;          // ФБ сброса буфера в файл
    fbFileGetSize:        FILE.GetSize;        // ФБ получения размера файла

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

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

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

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

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

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

    fbWriteTrig:           F_TRIG;             // триггер записи в файл

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

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

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

```

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

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

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

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

sArchEntry := CONCAT11(asExportStringData);

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

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

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

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

CASE eState OF

    FileWork.OPEN: // шаг открытия файла

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

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

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

            eState := FileWork.WRITE;
        END_IF

    FileWork.CREATE: // шаг создания файла

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

```

```

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

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

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

    IF fbFileOpen.xError THEN
        // обработка ошибок
    END_IF

FileWork.WRITE: // шаг записи в буфер

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

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

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

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

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

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

    IF fbFileWrite.xError THEN
        // обработка ошибок
    END_IF

FileWork.FLUSH: // шаг сброса буфера в файл

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

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

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

    IF fbFileFlush.xError THEN
        // обработка ошибок
    END_IF

FileWork.CLOSE: // шаг закрытия файла

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

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

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

FileWork.GET_SIZE: // шаг определения размера файла

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

    // определяем размер файла

```

```

        IF fbFileGetSize.xDone THEN
            udiArchSize:=fbFileGetSize.szSize;
            fbFileGetSize(xExecute:=FALSE);

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

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

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

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

END_CASE

```