Работа с визуализацией из кода программы в CODESYS V3

07.08.2023
версия 1.1

https://oscat.ru/

Оглавление

Введение

Одним из важных компонентов среды CODESYS является визуализация. Редактор визуализации позволяет создать человеко-машинный интерфейс (HMI), с помощью которого оператор сможет получать информацию о технологическом процессе и управлять им. Визуализация может отображаться на одном или нескольких клиентах визуализации. В роли такого клиента может выступать:

- экран панельного контроллера или дисплей, подключенный к контроллеру с видеовыходом (эта технология называется **TargetVisu**);
- web-браузер (**WebVisu**); при этом каждая открытая вкладка считается отдельным клиентом;
- приложение, запускаемое на ПК и представляющее собой специфический вариант виртуального контроллера, предназначенный для работы с визуализацией (**CODESYS HMI**). Является разновидностью таргет-визуализации;
- приложение, запускаемое на ПК и отображающее таргет-визуализацию контроллера (**RemoteTargetVisu**). В этом случае обработка визуализации происходит именно на стороне контроллера, а приложение просто подключается к нему как к удаленному рабочему столу (по принципам технологии VNC).

Набор поддерживаемых конкретным контроллером клиентов визуализации определяется его производителем.

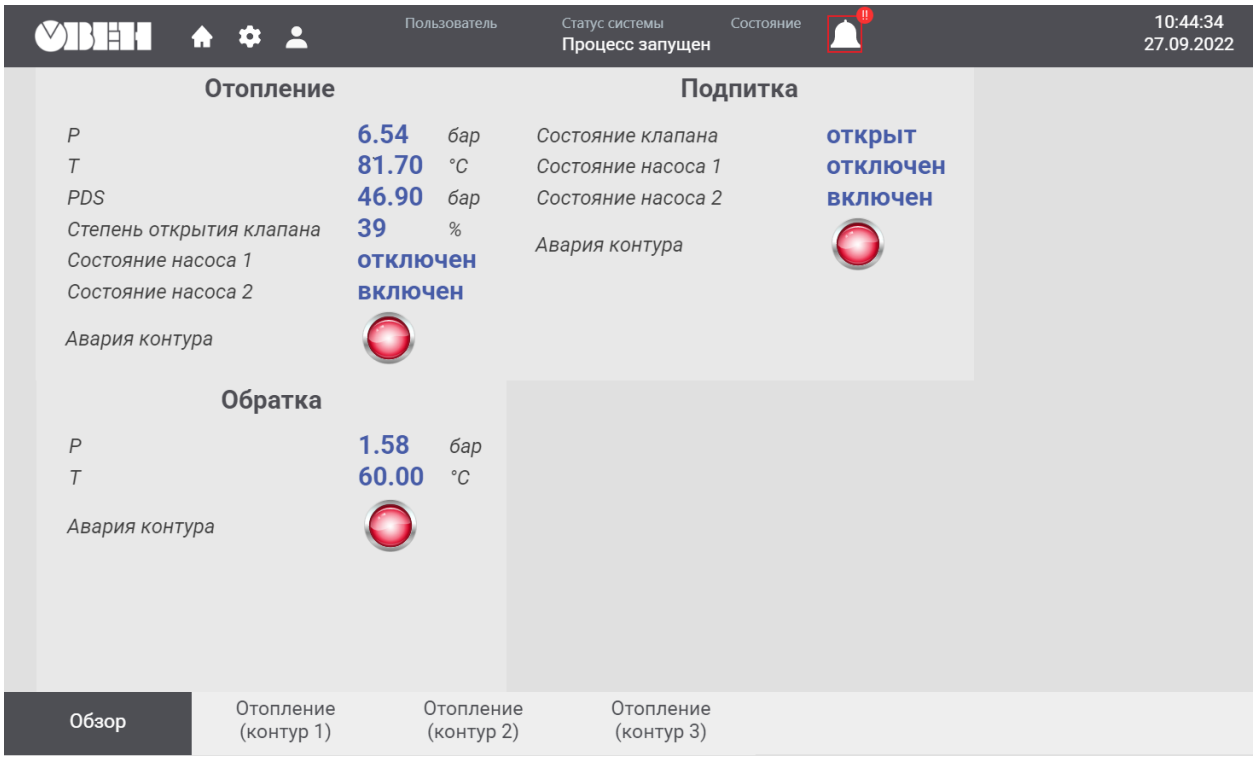


Рисунок 0. Пример визуализации, созданной в редакторе CODESYS

Редактор визуализации CODESYS предоставляет разработчику широкий набор графических элементов – начиная от простых (прямоугольники, индикаторы, кнопки, аналоговые дисплеи, ползунки и т. д.) и заканчивая продвинутыми (таблица тревог, тренд, XY-график, текстовый редактор и т. д.). Практически на каждый элемент можно назначить одно или несколько действий, выполняемых при нажатии на него – переключение экрана визуализации, открытие (или закрытие) диалогового окна, изменение значения привязанной к элементу переменной и т. д. Полный список элементов с описанием их параметров и примерами использования приведен в документе **CODESYS V3.5. Визуализация**, доступном на сайте ОВЕН. Мой опыт, основанный на общении с клиентами и изучении их проектов, показывает, что информации, содержащейся в упомянутом документе, хватает для решения приблизительно 90% возникающих задач.

Тем не менее, иногда у клиентов возникают специфические задачи. Например:

- переключение экранов визуализации из кода программы (например, при появлении аварии, связанной с конкретным экраном);
- открытие и закрытие диалогов из кода программы (например, сообщений о тревогах или о завершении какой-то операции);
- получение информации о подключенных клиентах web-визуализации (например, для определения разрешения экрана и переключения на нужный вариант экрана или для определения IP-адреса, чтобы заблокировать управление технологическим процессом с неизвестных IP-адресов);
- генерация события визуализации из кода программы (например, сымитировать нажатие на поле ввода – чтобы при открытии диалога авторизации пользователя в поле пароля сразу мигал курсор ввода);
- переключение экранов в фрейме для конкретного клиента визуализации;
- авторизация пользователя из кода программы (например, он активировал смарт-карту – и в визуализации произошел вход в систему нужного пользователя);
- и другие.

Все эти задачи связаны с работой с визуализацией из кода программы. Исторически по этому вопросу крайне мало документации от разработчиков CODESYS. С течением времени ими был разработан ряд демо-проектов, но они содержат минимум встроенной документации и их изучение может потребовать существенных усилий – особенно у начинающих пользователей. В этом документе я постараюсь на конкретных примерах показать решение некоторых задач, требующих работы с визуализацией из кода проекта. Я буду ориентироваться на «современные» средства решения подобных задач – в частности, на библиотеку **Visu Utils**, которая появилась в версии **V3.5 SP11**. Варианты, доступные в старых версиях среды (в основном, это касается некоторых внутренностей библиотеки **VisuElems**) в настоящий момент признаны устаревшими и не рекомендуются к использованию – поэтому я не буду рассматривать те из них, которые получили аналоги в **Visu Utils**. Я постараюсь сделать эту статью достаточно детальной и при этом простой для восприятия. Тем не менее, я исхожу из предположения, что вы имеете хорошие навыки программирования на языке ST и существенный опыт работы с визуализацией CODESYS V3.5.

Собственно, начнем мы как раз с рассмотрения возможностей библиотеки **Visu Utils**.

Автор: **Евгений Кислов**

1. Библиотека Visu Utils

1.1. Основная информация

Библиотека Visu Utils (ее пространство имен называется **VU**) содержит функциональные блоки для работы с визуализацией из кода программы. В данной статье рассматривается версия библиотеки **4.3.0.0**, в состав которой входят следующие блоки:

- FbIterateClients – блок получения информации о клиентах визуализации;
- FbChangeVisu – блок переключения экрана визуализации для конкретного набора клиентов визуализации;
- FbOpenDialog, FbOpenDialogExtended – блоки открытия диалога для конкретного набора клиентов визуализации;
- FbCloseDialog – блок закрытия диалога для конкретного набора клиентов визуализации;
- FBFileTransfer – блок передачи файлов между ПЛК и набором конкретных клиентов визуализации.

Эти блоки являются асинхронными и соответствуют модели поведения PLCopen Behaviour Model (также известной, как **CAA Behaviour Model** и Common Behaviour Model). Общий обзор этой модели поведения будет приведен в п. 1.2.4.

В следующих пунктах приводится описание этих блоков и примеры их использования.

1.2. Получение информации о клиентах визуализации (FbIterateClients)

Отличительной особенностью визуализации CODESYS является возможность ее отображения на нескольких клиентах визуализации одновременно. Например, представим себе следующую ситуацию: на производственной линии некоторой фабрики используется панельный контроллер с поддержкой web-визуализации. Оператор производственной линии взаимодействует с экраном контроллера (таргет-визуализацией) – наблюдает за значениями параметров технологического процесса, переключает экраны, вводит значения уставок и т. д. В это же время с web-визуализацией контроллера работают один или даже несколько сотрудников отдела аналитики – формируют отчеты, просматривают историю сообщений и т. д. В этот момент на линии завершается изготовление очередной партии продукции, и программа контроллера должна уведомить об этом оператора – например, отобразив диалоговое окно с соответствующим сообщением. Должно ли «всплыть» это же окно у аналитиков? Ответ, конечно, может отличаться в зависимости от конкретной ситуации – но вообще, вряд ли аналитики заинтересованы в подобного рода оперативной информации; их, скорее, интересуют показатели оптимальности работы оборудования, количество произведенной продукции за определенный интервал времени, длительность периодов простоя линии из-за поломок и т. д. Соответственно, возникает задача не просто «сделать что-то из кода», а «сделать что-то из кода для *конкретного клиента визуализации*». Для этого нужно как-то идентифицировать конкретного клиента – и именно для этого используется функциональный блок FbIterateClients. Получив информацию о клиентах визуализации – можно по каким-то критериям (например, типу визуализации или IP-адресу устройства, на котором открыта web-визуализация) выделить среди них интересующих вас и выполнить для этих какое-то действие с помощью одного из блоков библиотеки (например, **FbOpenDialog**).

Давайте подробнее посмотрим на ФБ **FbIterateClients**:

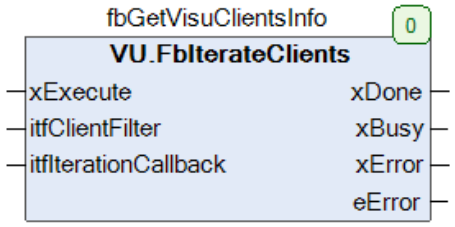


Рисунок 1.2.1 – Внешний вид ФБ **FbIterateClients** на языке CFC

Таблица 1.2.1 – Описание входов и выходов ФБ **FbIterateClients**

Название	Тип	Описание
Входы		
xExecute	BOOL	По переднему фронту запускается процесс однократного получения информации о всех клиентах визуализации, соответствующих фильтру **itfClientFilter**
itfClientFilter	IVisualizationClientFilter	Экземпляр ФБ, реализующего интерфейс **IVisualizationClientFilter**. Используется для определения категории клиентов, о которых будет собрана информация. Пользователь может реализовать этот ФБ самостоятельно или воспользоваться одним из готовых, объявленных в списке глобальных переменных библиотеки (Globals)
itfIterationCallback	IVisualizationClientIteration	Экземпляр ФБ, реализующего интерфейс **IVisualizationClientIteration**. Его метобы будут автоматически (по технологии callback) вызываться для каждого клиента, соответствующего **itfClientFilter**. Именно в этом ФБ можно будет получить доступ к информации о клиенте
Выходы		
xDone	BOOL	Принимает значение **TRUE** после сбора информации о всех заданных клиентах
xBusy	BOOL	Имеет значение **TRUE**, пока блок находится в процессе работы
xError	BOOL	Принимает значение **TRUE** в случае возникновения ошибки в процессе работы блока
eError	VU.ERROR	Код ошибки

Если вы только начинаете знакомиться с библиотекой **Visu Utils** – то, вполне вероятно, приведенное выше описание вызвало у вас смешанные чувства в стиле «очень интересно, но не особо понятно». Всё нормально – это как раз ключевая причина создания данной статьи. Если бы всё было просто, то в её написании не было бы смысла.

Для начала давайте попробуем сформулировать вопросы, которые могут возникнуть после ознакомления с таблицей 1.2.1:

- что подразумевается под словом «фильтр»?
- что значит «экземпляр ФБ, реализующего интерфейс…»? О каком ФБ речь? Что такое интерфейс?
- что значит «пользователь может реализовать этот ФБ самостоятельно или воспользоваться одним из готовых»? В каких случаях следует использовать «готовые» ФБ (и что это за ФБ?), а в каких – реализовывать свой? И как вообще его реализовать?
- что такое «callback»? Что значит «методы экземпляра ФБ будет автоматически вызываться…»? Вызываться кем?
- что значит «именно в этом ФБ можно получить доступ к информации о клиенте»? Как это сделать и какая информация вообще доступна?
- какие ошибки могут возникнуть в процессе работы этого и других блоков библиотеки?

В следующих подпунктах я постараюсь ответить на эти вопросы (не в порядке их перечисления).

1.2.1. Фильтрация клиентов (itfClientFilter)

При выполнении какого-либо действия над клиентами визуализации – нужно определить группу клиентов, для которых это действие будет совершено. Например, может потребоваться открыть диалоговое окно:

- только для клиента таргет-визуализации;
- для клиентов web-визуализации с определенными IP-адресами;
- для клиентов любого типа визуализации, авторизованных в системе и принадлежащих определенной группе пользователей визуализации;
- и т. д.

Для определения группы клиентов используется вход **itfClientFilter**, который присутствует у каждого ФБ библиотеки **Visu Utils**. Этот вход имеет тип интерфейс IVisualizationClientFilter. При вызове экземпляра блока нужно передать на этот вход экземпляр ФБ, реализующего данный интерфейс. Сделать это можно двумя способами:

- если вам нужен «типовой» фильтр клиентов (например, все клиенты web-визуализации; см. таблицу ниже) – то используйте один из готовых экземпляров, объявленных в списке глобальных переменных Globals (см. таблицу ниже);
- если вам нужен «специфический» фильтр клиентов (например, только клиенты web-визуализации с определенными IP-адресами) – то вам потребуется создать функциональный блок, реализующий (**IMPLEMENTS**) интерфейс IVisualizationClientFilter и добавить код в его метод **IsAccepted**.

Таблица 1.2.1.1 – Состав списка глобальных переменных **Globals** (доступные «типовые» фильтры клиентов)

Название	Тип	Описание
AllClients	FBAcceptsAllClients	Фильтр, описывающий всех клиентов визуализации
CurrentClient	FBAcceptsCurrentClient	Фильтр, описывающий текущего клиента визуализации. Может использоваться только в том случае, если вызов экземпляра ФБ производится при нажатии на кнопку в визуализации (потому что в этом случае обработка клиента автоматически производится подсистемой визуализации). Но обычно это не имеет смысла, так как в визуализации можно просто в настройках элемента на вкладке **Конфигурация ввода** задать нужные действия – переключения экрана, открытие диалога и т. д.
OnlyTargetVisu	FBAcceptsTargetVisuOnly	Фильтр, описывающей клиента таргет-визуализации (т. е. оператора, работающего с дисплеем панельного контроллера)

OnlyRemoteTargetVisu	FBAcceptsRemoteTargetVisuOnly	Фильтр, описывающей всех клиентов удаленной таргет-визуализации
OnlyWebVisu	FBAcceptsWebVisuOnly	Фильтр, описывающий всех клиентов web-визуализации

Сейчас мы обсуждаем ФБ **FbIterateClients**. В его контексте фильтрация клиентов является тавтологичной – потому что для определения группы клиентов нужна иметь о них какую-то информацию, а целью блока является как раз сбор этой информации. Поэтому при вызове экземпляра этого блока обычно используется один из готовых фильтров из списка **Globals**. Давайте в рамках примера соберем информацию о всех клиентах визуализации (а создание собственного фильтра рассмотрим в п. 1.3). Напишем первую пару строк кода:

```
PROGRAM PLC_PRG
VAR
   fbGetVisuClientsInfo:          VU.FbIterateClients;
   xExecute:                      BOOL;
END_VAR

fbGetVisuClientsInfo
(
    xExecute             := xExecute,
   // будем собирать информацию о всех клиентах визуализации
   itfClientFilter       := VU.Globals.AllClients,
   // что нужно передать на этот вход – обсудим дальше
   itfIterationCallback  := // ???
);

// информация о всех интересующих клиентах получена
IF fbGetVisuClientsInfo.xDone THEN

   // и здесь можно что-то с ней сделать

END_IF
```

1.2.2. Информация о клиентах (IVisualizationClient)

Итак, мы хотим получить информацию о всех клиентах визуализации. Но для начала неплохо бы разобраться, какую именно информацию мы можем получить. Эта информация определена в интерфейсе IVisualizationClient в виде набора методов и свойств. Каждый из них соответствует одному параметру клиента.

Таблица 1.2.2.1 – Состав интерфейса **IVisualizationClient**

Название	Тип	Описание
Метод **GetClientName**		
Выход: GetClientName	STRING	Имя данного клиента визуализации (если задано). Имена доступны при подключении к web-визуализации – для этого следует указывать их в качестве аргументов URL в стиле http://10.2.11.174:8080/webvisu.htm?ClientName=**My ClientName** (где текст после **ClientName=** является именем клиента, которое будет получено с помощью этого метода)
Выход: xIsValid	BOOL	**TRUE** – у данного клиента есть имя
Метод **GetIPv4Address**		
Выход: GetIPv4Address	STRING	IP-адрес клиента визуализации
Выход: xIsValid	BOOL	**TRUE** – у данного клиента есть IP-адрес
Метод **GetRemoteTargetVisuId**		
Выход: GetRemoteTargetVisuId	STRING	Идентификатор клиента удаленной таргет-визуализации. Поддерживается начиная с версии **3.5.11.0**
Выход: xIsValid	BOOL	**TRUE** – у данного клиента есть идентификатор
Метод **GetRemoteTargetVisuVersion**		
Выход: GetRemoteTargetVisuVersion	__SYSTEM.VERSION	Версия клиента удаленной таргет-визуализации. Поддерживается начиная с версии **3.5.11.0.** **VERSION** – это структура, содержащая 4 поля типа **UINT** (uiMajor, uiMinor, uiServicePack, uiPatch)
Выход: xIsValid	BOOL	**TRUE** – у данного клиента есть версия

Свойства		
ClientId[1]	INT	Глобальный идентификатор клиента. Является уникальным для каждого клиента визуализации. Нумерация – с **0**. Если данных о клиенте нет, то идентификатор имеет значение **-1**. См. чуть подробнее в п. 1.2.7
ClientType	VisuClientType	Тип клиента визуализации
ClientDevicePixelRatio	REAL	Device Pixel Ratio (соотношение логических и физических пикселей) дисплея клиента web-визуализации
ClientWidth	INT	Ширина клиента визуализации в пикселях
ClientHeight	INT	Высота клиента визуализации в пикселях
CurrentVisuName	STRING	Имя текущего экрана визуализации, отображаемого на клиенте
PreviousVisuName	STRING	Имя экрана визуализации, который отображался на клиенте до перехода на текущий экран
NextVisuName	STRING	Имя экрана визуализации, на который будет осуществлен переход при использовании действия **Изменить показанную визуализацию – Следующая визуализация** (см. рис. 1.2.2.1)
UserName	WSTRING	Имя пользователя визуализации (для залогинившегося клиента). См. рис. 1.2.2.2
UserGroupId	DWORD	ID группы пользователей (для залогинившегося клиента). См. рис. 1.2.2.3

Таблица 1.2.2.2 – Состав перечисления **VisuClientType**

Название	Значение	Описание
Unknown	16#0	Не удалость идентифицировать клиента
ProgrammingSystem	16#1	Клиент – среда разработки (сервисная визуализация в редакторе CODESYS при онлайн-подключении к контроллеру)
TargetVisualization	16#4	Клиент работает с таргет-визуализацией
WebVisualization	16#8	Клиент работает с web-визуализацией
RemoteTargetVisualization	16#10	Клиент работает с удаленной таргет-визуализацией
AllRemoteVisualizations	16#1B	Совокупность вариантов ProgrammingSystem, WebVisualization и RemoteTargetVisualization. Вероятно, не может быть возвращена через свойство **ClientType** и используется в коде библиотеки **Visu Utils** для каких-то своих целей

[1] Для работы с этим свойством потребуется добавить в проект библиотеку **VisuGlobalClientManager**

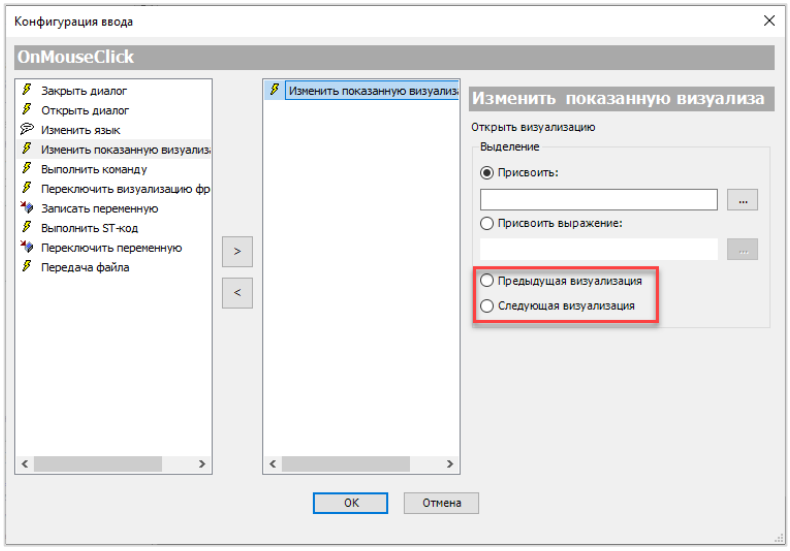


Рисунок 1.2.2.1 – Пояснение к свойствам **PreviousVisuName** и **NextVisuName**

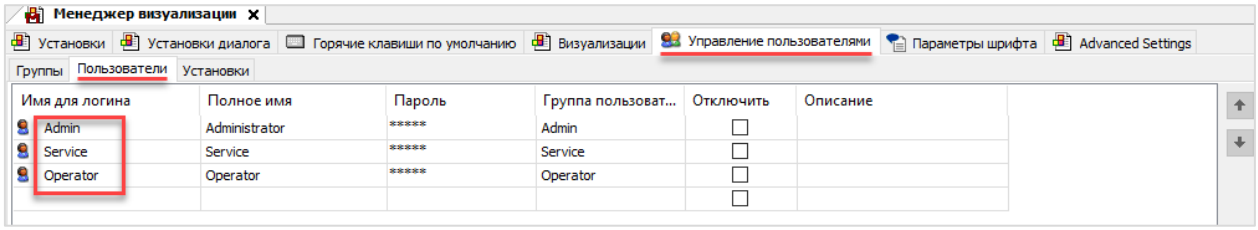


Рисунок 1.2.2.2 – Пояснение к свойству **UserName**

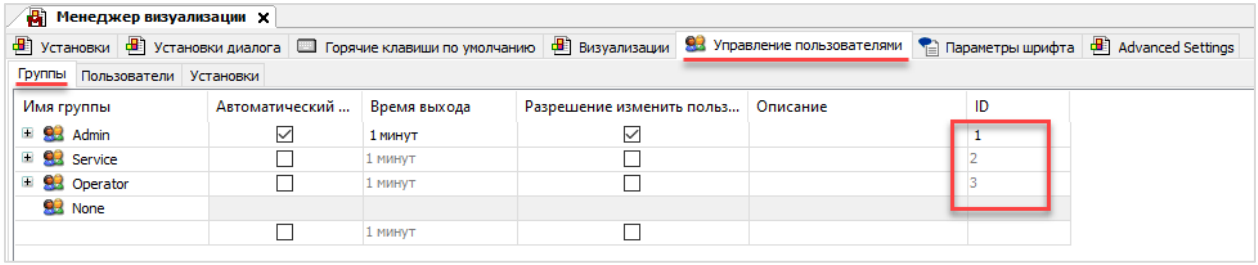


Рисунок 1.2.2.3 – Пояснение к свойству **UserGroupId**

Возможно, у вас возник вопрос – почему некоторые параметры клиента возвращаются через методы, а некоторые – через свойства? Это связано с тем, что ряд параметров имеет флаг «валидности» (**xIsValid**) – а через свойство можно вернуть только одно значение (видимо, разработчики решили, что возвращать через свойство структуру будет сложнее для использования, чем вызов метода).

Теперь мы знаем, какую информацию о клиентах визуализации можем получить. Но на самом деле – это лишь набор параметров, которые разработчики CODESYS решили сделать доступным для программистов в явном виде. Фактически CODESYS собирает гораздо больше данных о клиенте и размещает их в структуре типа **VisuElems.VisuElemBase.VisuStructClientData**. Эта структура представляет собой так называемый **контекст клиента** – полный набор данных, которые нужны подсистеме визуализации для обеспечения работы клиента. В ряде конкретных случаев

(когда вы не находите в **IVisualizationClient** нужных вам параметров) требуется получить доступ к этой структуре в коде программы. Для этого используется интерфейс IVisualizationClientRaw, который наследуется от **IVisualizationClient** и расширяет его еще одним свойством **ClientDataPointer** – указателем на упомянутую выше структуру. Далее мы рассмотрим, как его использовать.

А пока что на основании таблицы 1.2.2.1 сформируем структуру интересующих нас данных о клиентах визуализации (ряд специфичных параметров при этом будет опущен – например, параметры, связанные с идентификацией клиента удаленной таргет-визуализации).

```
// Структура данных о клиенте визуализации
TYPE VISU_CLIENT_DATA :
STRUCT
   // Указатель на структуру данных клиента
   pstClientData:          POINTER TO VisuElems.VisuElemBase.VisuStructClientData;
   // Уникальный ID клиента
   iClientId:              INT;
   // Тип клиента (таргет-визу, веб-визу и т.д.)
   eClientType:            VU.Visu_ClientType;
   // DevicePixelRatio дисплея клиента (https://habr.com/ru/sandbox/128978/)
   rDevicePixelRatio:      REAL;
   // Ширина дисплея клиента в пикселях
   iWidthX:                INT;
   // Высота дисплея клиента в пикселях
   iHeightY:               INT;
   // IP-адрес клиента визуализации
   sIpAddr:                STRING(15);
   // Имя текущего экрана визуализации, отображаемого на клиенте
   sCurrentVisu:           STRING;
   // Имя пользователя визуализации (для залогинившегося клиента)
   wsCurrentUserName:      WSTRING;
   // ID группы пользователей (для залогинившегося клиента)
   dwCurrentUserGroupId:   WORD;
   // Имя группы пользователей (для залогинившегося клиента)
   // Отсутствует в IVisualizationClient
   // (по крайней мере, в версии Visu Utils 4.2.0.0 и более ранних)
   // Это поле добавлено для демонстрации работы с IVisualizationClientRaw
   wsCurrentUserGroupName: WSTRING;
END_STRUCT
END_TYPE
```

Теперь нужно разобраться, как заполнить эту структуру.

1.2.3. Доступ к данным клиентов (itfIterationCallback, IVisualizationClientIteration)

Вспомним первый (и пока что единственный) написанный нами фрагмент кода:

```
PROGRAM PLC_PRG
VAR
   fbGetVisuClientsInfo:          VU.FbIterateClients;
   xExecute:                      BOOL;
END_VAR

fbGetVisuClientsInfo
(
   xExecute             := xExecute,
   // будем собирать информацию о всех клиентах визуализации
   itfClientFilter      := VU.Globals.AllClients,
   // что нужно передать на этот вход – обсудим дальше
   itfIterationCallback := // ???
);


// информация о клиента получена
IF fbGetVisuClientsInfo.xDone THEN

   // и здесь можно что-то с ней сделать

END_IF
```

Пришло время разобраться со входом **itfIterationCallback**. На этот вход нужно передать экземпляр ФБ, внутри которого будет происходить обработка данных клиентов. Этот ФБ должен реализовывать (**IMPLEMENTS**) интерфейс IVisualizationClientIteration. Давайте создадим такой ФБ и назовем его **VisuClientIteration**.

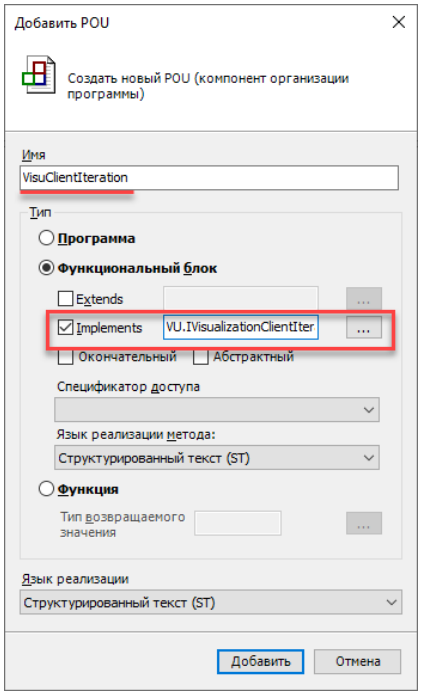


Рисунок 1.2.3.1 – Создание ФБ обработки информации от клиентов. Обратите внимание, что имя реализуемого интерфейса не влезло в скриншот, но записать его надо полностью – **VU.IVisualizationClientIteration**

У этого ФБ автоматически будут созданы три метода:

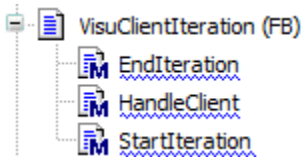


Рисунок 1.2.3.2 – Методы, полученные от интерфейса IVisualizationClientIteration

Сразу объявим экземпляр этого ФБ в нашей программе и передадим его в качестве аргумента экземпляру рассматриваемого нами в данном пункте блока **FbIterateClients** (здесь и далее – добавляемый код выделен **жирным**):

```
PROGRAM PLC_PRG
VAR
   fbGetVisuClientsInfo:          VU.FbIterateClients;
   xExecute:                      BOOL;
   fbClientIterationCallback:     VisuClientIteration;
END_VAR

fbGetVisuClientsInfo
(
   xExecute             := xExecute,
   // будем собирать информацию о всех клиентах визуализации
   itfClientFilter      := Vu.Globals.AllClients,
   // экземпляр ФБ, автоматически вызываемого для обработки клиентов
   itfIterationCallback := fbClientIterationCallback
);


// информация о всех интересующих клиентах получена
IF fbGetVisuClientsInfo.xDone THEN

   // и здесь можно что-то с ней сделать

END_IF
```

Итак, объявленный нами экземпляр ФБ передается в качестве аргумента экземпляру ФБ **FbIterateClients**. Вызов методов этого экземпляра будет производить сам **fbGetVisuClientsInfo** (по технологии callback) – вам не нужно вызывать их в вашем коде. Нужно только создать реализацию этих методов – то есть написать их код.

Для начала обзорно рассмотрим эти методы:

- **StartIteration** – однократно выполняется перед началом обработки клиентов;
- **HandleClient** – для каждого клиента визуализации, соответствующего фильтру **itfClientFilter**, осуществляется однократный вызов этого метода;
- **EndIteration** – однократно выполняется после окончания обработки клиентов.

Методы **StartIteration** и **EndIteration** являются, по большому счету, опциональными – например, в методе **StartIteration** можно произвести инициализацию переменных блока. Наибольший интерес представляет **HandleClient** – именно в нем происходит получение информации о клиентах.

Происходит это следующим образом: у метода есть вход **itfClient** типа IVisualizationClient. В каждом вызове метода (напомним, этот вызов происходит автоматически без вашего участия) на этот вход передается экземпляр интерфейса очередного клиента. Вы можете обратиться к методам и свойствам этого объекта и получить доступ к данным клиента.

VisuClientIteration.HandleClient

```
1   {warning 'добавить реализацию метода'}
2   (* This method will be called for each client that is currently within the lis
3    of active visualization clients.
4
5    .. note:: Please remark that this method will be called from VISU_TASK*)
6   METHOD HandleClient
7   VAR_INPUT
8       (* The object representing the according client. Will not be 0.*)
9       itfClient   : VU.IVisualizationClient;
10  END_VAR
11
```

Рисунок 1.2.3.3 – Область объявления переменных метода **HandleClient**

Но перед этим давайте определимся – где мы планируем хранить информацию о клиентах? Вероятно, разумнее всего делать это в программе, в которой мы вызываем **fbGetVisuClientsInfo** – объявив в ней массив структур VISU_CLIENT_DATA (эту структуру мы создали в конце п. 1.2.2). Значит, нужно как-то передать данные из автоматически вызываемого метода в переменную программы. Это можно сделать как минимум тремя различными способами:

- через глобальные переменные. В методе мы можем записывать данные в переменные из списка глобальных переменных, а в программе копировать их значение в переменные программы (можно отказаться от копирования и использовать в коде программы глобальные переменные). Такой подход регулярно используется в примерах от разработчиков CODESYS, но вообще его сложно назвать оптимальным – наличие глобальных переменных затрудняет понимание логики передачи данных внутри приложения. Впрочем, вы можете справедливо возразить, что информация о клиентах может потребоваться в разных программах (а не одной) и что их запись (т. е. запись глобальной переменной) будет производиться только из этого метода – поэтому это не сильно повлияет на сложность приложения;
- через выход (**VAR_OUTPUT**) экземпляра **VisuClientIteration**. Действительно, в нашей программе мы можем обратиться к выходам экземпляра блока и скопировать их значения в переменные программу. Недостатком этого варианта (как, кстати, и предыдущего) является сам факт этого копирования – поскольку массив с информацией о клиентах может занять довольно много байт памяти контроллера. Впрочем, вы можете справедливо возразить, что для вашего конкретного приложения эти дополнительные накладные расходы на копирование ничтожны и не стоит их вообще обсуждать;
- по указателю через вход (**VAR_INPUT**) экземпляра **VisuClientIteration**. Это несколько усложнит наш код, но зато позволит избежать дополнительного копирования данных. Далее мы рассмотрим именно этот вариант (как раз потому, что первые два варианта проще в реализации и, вероятно, у вас не возникнет вопросов, если вы решите использовать их).

Итак, переходим к кодированию. Сначала объявим в программе массив структур VISU_CLIENT_DATA для хранения информации о клиентах. Для этого нам нужно определить размерность этого массива – то есть мы должны понять, сколько клиентов могут одновременно работать с визуализацией нашего контроллера. Это можно настроить в менеджере визуализации:

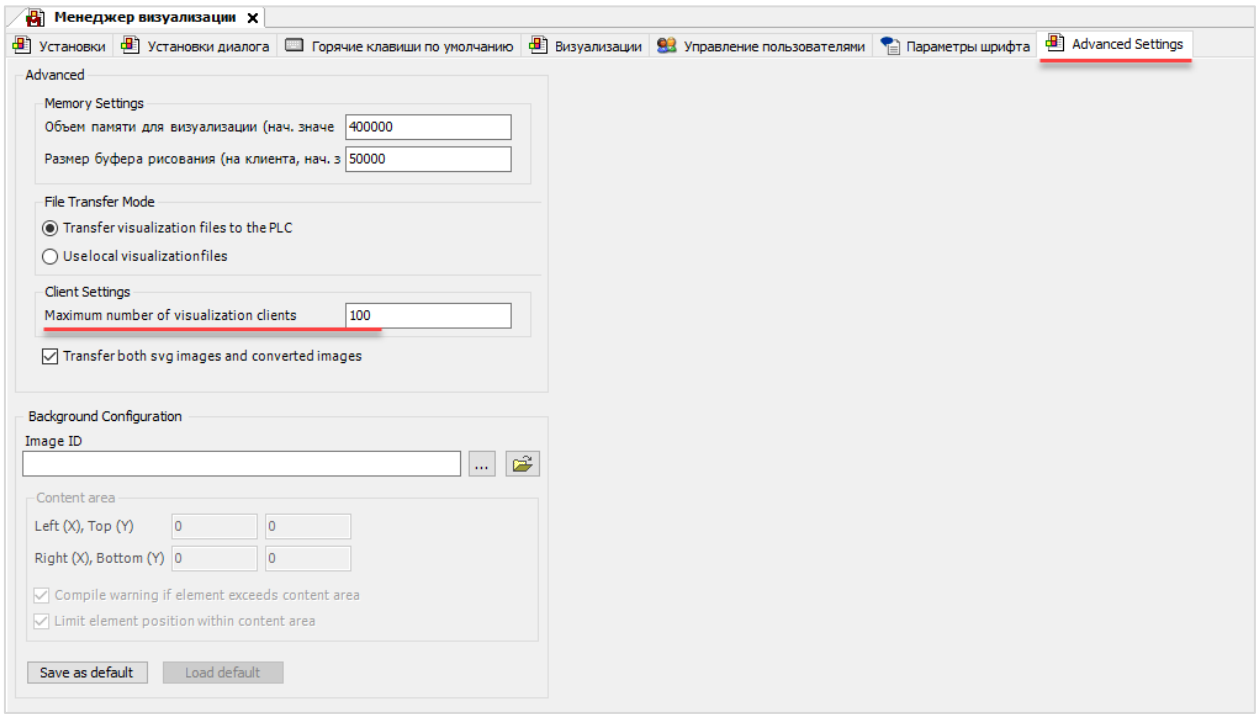


Рисунок 1.2.3.4 – Настройка ограничения максимального числа клиентов в менеджере визуализации

В коде программы мы можем «дотянуться» до этого значения, использовав системную константу **Visu_Superglobal_Constants.VISU_MAX_NUMBER_OF_CLIENTS**. Кстати, есть и константа **Visu_Superglobal_Constants.VISU_MIN_NUMBER_OF_CLIENTS** со значением **-1** – но я не понимаю, как именно предполагается ее использовать. В качестве нижней границы массива я буду использовать значение **1** – то есть первый обработанный клиент визуализации будет иметь индекс «1» (мне кажется это логичным).

```
PROGRAM PLC_PRG
VAR
   fbGetVisuClientsInfo:          VU.FbIterateClients;
   xExecute:                      BOOL;
   fbClientIterationCallback:     VisuClientIteration;
   astVisuClientInfo:             ARRAY
       [1..Visu_Superglobal_Constants.VISU_MAX_NUMBER_OF_CLIENTS]
         OF VISU_CLIENT_DATA;
END_VAR
```

Теперь вернемся к ФБ **VisuClientIteration** и объявим у него вход типа **POINTER TO VISU_CLIENT_DATA** (через этот вход по указателю мы будем возвращать в программу информацию об очередном клиенте) и выход типа **INT**, который будет характеризовать текущее число обработанных клиентов.

```
FUNCTION_BLOCK VisuClientIteration IMPLEMENTS VU.IVisualizationClientIteration
VAR_INPUT
   pstVisuClientData: POINTER TO VISU_CLIENT_DATA;
END_VAR
VAR_OUTPUT
   iCurrentClientCount: INT;
END_VAR
VAR
END_VAR
```

В программе инициализируем этот вход экземпляра нашего блока адресом объявленного нами массива структур (вообще, это можно сделать однократно, но в рамках примера я поленился добавлять оператор IF для этого):

```
PROGRAM PLC_PRG
VAR
   fbGetVisuClientsInfo:          VU.FbIterateClients;
   xExecute:                      BOOL;
   fbClientIterationCallback:     VisuClientIteration;
   astVisuClientInfo:             ARRAY
       [1..Visu_Superglobal_Constants.VISU_MAX_NUMBER_OF_CLIENTS]
         OF VISU_CLIENT_DATA;
END_VAR


fbClientIterationCallback.pstVisuClientData := ADR(astVisuClientInfo);

fbGetVisuClientsInfo
(
   xExecute             := xExecute,
   // будем собирать информацию о всех клиентах визуализации
   itfClientFilter      := Vu.Globals.AllClients,
   // экземпляр ФБ, автоматически вызываемого для обработки клиентов
   itfIterationCallback := fbClientIterationCallback
);

// информация о всех интересующих клиентах получена
IF fbGetVisuClientsInfo.xDone THEN

   // и здесь можно что-то с ней сделать

END_IF
```

Теперь осталось добавить код в методы.

В методе **StartIteration** будем очищать данные клиентов, полученные в прошлом вызове блока (чтобы, например, в массиве не сохранялась информация об отключившихся между вызовами экземпляра блока клиентах web-визуализации) и обнулять счетчик обработанных клиентов. Очистка выполняется с помощью функции **MemFill** из библиотеки **CAA Memory**.

```
{warning 'добавить реализацию метода'}
(* This method will be called at the start of an iteration over all clients.

 .. note:: Please remark that this method will be called from VISU_TASK.*)
METHOD StartIteration

// код метода

MEM.MemFill(pstVisuClientData, SIZEOF(VISU_CLIENT_DATA) *
   TO_UINT(Visu_Superglobal_Constants.VISU_MAX_NUMBER_OF_CLIENTS), 0);

iCurrentClientCount := 0;
```

В коде **HandleClient** организуем наполнение массива структур, переданного по указателю **pstClientData**, информацией о клиентах. Сначала я приведу полный листинг метода, а потом прокомментирую его.

```
{warning 'добавить реализацию метода'}
(* This method will be called for each client that is currently within the list
 of active visualization clients.

 .. note:: Please remark that this method will be called from VISU_TASK*)
METHOD HandleClient
VAR_INPUT
        (* The object representing the according client. Will not be 0.*)
   itfClient    : VU.IVisualizationClient;
END_VAR
VAR
   itfClientRaw : VU.IVisualizationClientRaw;
END_VAR

// код метода

IF iCurrentClientCount < Visu_Superglobal_Constants.VISU_MAX_NUMBER_OF_CLIENTS THEN

   pstVisuClientData[iCurrentClientCount].iClientId := itfClient.ClientId;
   pstVisuClientData[iCurrentClientCount].eClientType := itfClient.ClientType;
   pstVisuClientData[iCurrentClientCount].rDevicePixelRatio :=
         itfClient.ClientDevicePixelRatio;
   pstVisuClientData[iCurrentClientCount].iWidthX := itfClient.ClientWidth;
   pstVisuClientData[iCurrentClientCount].iHeightY := itfClient.ClientHeight;
   pstVisuClientData[iCurrentClientCount].sIpAddr := itfClient.GetIPv4Address();
   pstVisuClientData[iCurrentClientCount].sCurrentVisu := itfClient.CurrentVisuName;
   pstVisuClientData[iCurrentClientCount].wsCurrentUserName := itfClient.UserName;
   pstVisuClientData[iCurrentClientCount].dwCurrentUserGroupId :=
      itfClient.UserGroupId;

   IF __QUERYINTERFACE(itfClient, itfClientRaw) THEN

         pstVisuClientData[iCurrentClientCount].pstClientData :=
                  itfClientRaw.ClientDataPointer;
         pstVisuClientData[iCurrentClientCount].wsCurrentUserGroupName :=
pstVisuClientData[iCurrentClientCount].pstClientData^.GlobalData.CurrentUserGroupName;

   END_IF

END_IF
iCurrentClientCount := iCurrentClientCount + 1;
```

Напоминаю – при callback-вызове метода на его вход **itfClient** подставляется экземпляр интерфейса очередного клиента. Доступ к этому экземпляру обернут в проверку на то, что номер текущего обрабатываемого клиента не превышает ограничения на максимальное число клиентов (определяемое нашей «суперглобальной» константой из менеджера визуализации). Это наша страховка при работе с указателем **pstVisuClientData** – даже в том случае, если вызов метода по каким-то причинам произойдет больше предполагаемого нами числа раз (это число совпадает с ограничением на максимальное число клиентов, заданное в менеджере визуализации), то при записи по указателю мы не выйдем за границы нашего массива **astVisuClientInfo** и не перезапишем данные других переменных программы.

Если эта проверка завершается успешно – то мы обращаемся к массиву, используя индексный доступ по указателю. Возможно, при первом прочтении эта фраза показалась вам не особо понятной, даже если вы уже имели опыт работы с указателями. Итак:

- у нашего ФБ **VisuClientIteration** есть вход **pstVisuClientData** типа **POINTER TO VISU_CLIENT_DATA**;
- в коде программы мы передаем на этот вход адрес нашего массива **astVisuClientInfo**;
- всё это для того, чтобы заполнять из кода ФБ этот массив;
- данные каждого очередного клиента должны быть размещены в каждом последующем элементе массива (т. е. данные первого клиента – в первом элементе и т. д.);
- мы используем индексный доступ к указателю – т. е. обращение **pstVisuClientData[iCurrentClientCount]** эквивалентно обращению к элементу массиву с индексом **(iCurrentClientCount + 1),** потому что индексация указателей ведется с нуля, а нижняя граница нашего массива **astVisuClientInfo** равна «**1**»;
- именно поэтому инкремент **iCurrentClientCount** происходит в конце метода, а не в начале – нам как раз нужно, чтобы первое обращение к указателю произошло с индексом **0**;
- глаза вас не обманывают – оператор разыменования («^») при индексном доступе не нужен;
- если вы программировали на Си – то да, индексный доступ к указателям в CODESYS реализован по тем же принципам, что и в этом языке.

Сам код выглядит обыденно – просто копирование значений свойств (и результата вызова одного метода) экземпляра интерфейса **itfClient** в поля структуры VISU_CLIENT_DATA.

Но в конце будет чуть интереснее – помните, мы добавили в структуре поле **wsCurrentUserGroupName**? Его мы извлечь из **itfClient** не сможем – такого свойства у него просто нет. Поэтому мы конвертируем экземпляр нашего интерфейса IVisualizationClient в экземпляр интерфейса IVisualizationClientRaw с помощью системного оператора __QUERYINTERFACE (экземпляр расширенного интерфейса был заблаговременно объявлен нами в локальных переменных метода). Далее мы обращаемся к свойству **ClientDataPointer** – это указатель на контекст клиента (простыми словами – на экземпляр огромной структуры данных с кучей информации о клиенте; чуть больше информации о ней я приведу в следующем пункте). Разыменовываем этот указатель и обращаемся нужному полю этой огромной структуры, получая название группы пользователей.

И последнее, что делаем – инкрементируем значение выхода **iCurrentClientCount**, чтобы в следующем вызове метода записать информацию об очередном клиенте в следующий элемент массива **astVisuClientInfo**. Таким образом – после завершения работы **fbGetVisuClientsInfo** (т. е. в момент, когда **xDone** станет равным **TRUE**) значение выхода **iCurrentClientCount** будет соответствовать текущему числу клиентов визуализации. А дальше, например, можно обработать клиентов в цикле **FOR**, используя **iCurrentClientCount** как верхнюю граница счетчика цикла.

Вероятно, спустя столько страниц рассказов всего лишь об одном блоке вы устали и требуете каких-то реальных примеров его работы. Справедливо – давайте запустим созданный нами проект и посмотрим, как он работает.

1.2.4. Проверка работы примера

Перед запуском внесем в пример последнее изменение – создадим в менеджере визуализации управление пользователями (с настройками по умолчанию) и зададим пользователям пароли. Я использую следующие пароли:

- для пользователя **Admin** пароль «**1**»;
- для пользователя **Service** пароль «**2**»;
- для пользователя **Operator** пароль «**3**».

На экран визуализации добавим кнопку с открытием диалога **Логин**.

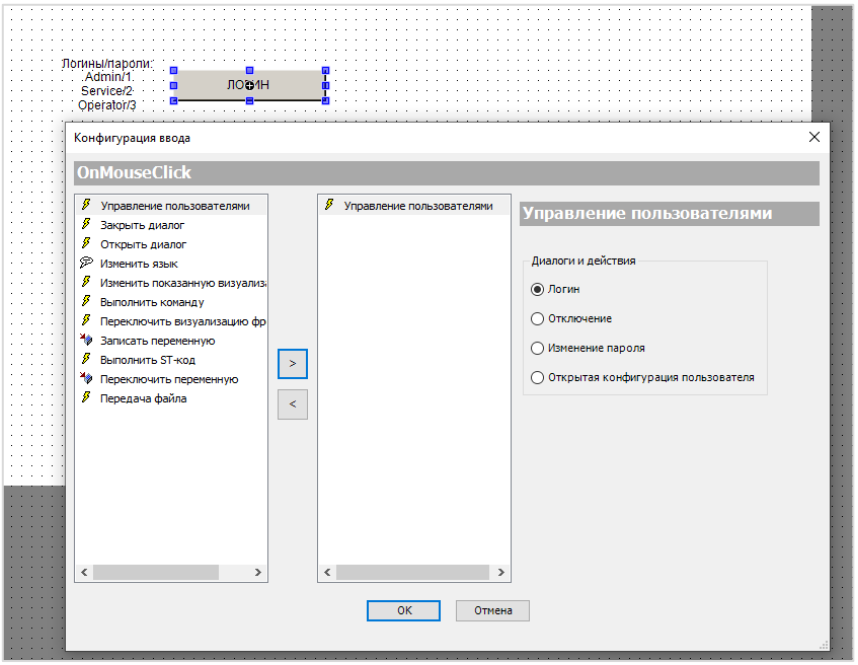


Рисунок 1.2.4.1 – Добавление кнопки логина пользователя

Вот ссылка на архив готового примера. Он создан в среде **CODESYS V3.5 SP17 Patch 3** с установленным плагином CODESYS Visualization 4.3.0.0:

скачать

Загрузим этот пример в контроллер. Я использую панельный контроллер ОВЕН СПК1xx [М01] – у него есть собственный дисплей и, соответственно, таргет-визуализация. Откроем экран **Visualization** в редакторе CODESYS, а также подключимся к web-визуализации контроллера. Произведем авторизацию пользователя – в таргет-визуализации залогинимся как **Admin**, в сервисной визуализации CODESYS – как **Service**, в web-визуализации – как **Operator**.

После этого в программе **PLC_PRG** присвоим значение **TRUE** переменной **xExecute** (*уточню, что так мы поступаем в рамках примера; в реальных проектах вы, естественно, будете вызывать экземпляры блоков по какому-то событию или, если речь о периодическом сборе информации о клиентах, циклически с заданным интервалом*).

Теперь посмотрим на значения различных переменных в программе **PLC_PRG**.

Начнем с **fbGetVisuClientsInfo**. Его выход **xDone** принял значение **TRUE** – значит, блок успешно (без каких-либо ошибок) завершил свою работу. Если бы в процессе работы блока возникла ошибка – то выход **xDone** имел бы значение **FALSE**, выход **xError** имел бы значение **TRUE**, а выход **eError** содержал бы код текущей ошибки.

Изменение значений выходов происходит согласно модели поведения PLCopen Behaviour Model (в рамках CODESYS она также известна как **CAA Behavior Model** и Common Behavior Model):

- блок запускается в работу по переднему фронту на входе **xExecute**. При этом необязательно удерживать **xExecute** в состоянии **TRUE** в процессе работы блока – можно передать на этот вход единичный импульс;
- после запуска блока выход **xBusy** принимает значение **TRUE** и сохраняет его на всё время работы блока;
- в момент завершения работы блока выход **xBusy** принимает значение **FALSE**;
- если к моменту завершения работы блока на входе **xExecute** остается значение **TRUE**, то выход **xDone** (в случае успешного завершения работы) или выход **xError** (в случае завершения из-за ошибки) принимает значение **TRUE** и сохраняет их до того момента, пока вход **xExecute** не получит значение **FALSE**. Пока выход **xError** имеет значение **TRUE** – на выходе **eError** отображается код возникшей ошибки;
- если к моменту завершения работы блока вход **xExecute** имеет значение **FALSE** (например, если блок был запущен в работу единичным импульсом), то выход **xDone** (в случае успешного завершения работы) или выход **xError** (в случае завершения из-за ошибки) принимает значение **TRUE** на один цикл задачи контроллера, в которой происходит вызов данного экземпляра ФБ. Пока выход **xError** имеет значение **TRUE** – на выходе **eError** отображается код возникшей ошибки.

Выражение	Тип	Значение
fbGetVisuClientsInfo	VU.FbIterateClients	
SUPER^	FbExecuteSingleVisuActionBase	
SUPER^	ETrig	
SUPER^	EdgeTriggeredBehaviourModelBase	
xExecute	BOOL	TRUE
xDone	BOOL	TRUE
xBusy	BOOL	FALSE
xError	BOOL	FALSE
itfClientFilter	IVisualizationClientFilter	16#AF54F814
eError	ERROR	NO_ERROR
_xActionDone	BOOL	TRUE
_pCurrentClientData	POINTER TO VisuStructClientData	16#00000000
_iOnlineChangeStartValue	INT	0
itfIterationCallback	IVisualizationClientIteration	16#AF698D48

Рисунок 1.2.4.2 – Значение **TRUE** на выходе **xDone** сигнализирует об успешном завершении работы блока

У **fbClientIterationCallback** выход **iCurrentClientCount** имеет значение **3** – это логично, ведь у нашей визуализации три клиента (таргет-визуализация, сервисная визуализация в редакторе CODESYS и один клиент web-визуализации).

fbClientIterationCallback	VisuClientIteration	
pstVisuClientData	POINTER TO VISU_CLIENT_DATA	16#AF815358
iCurrentClientCount	INT	3

Рисунок 1.2.4.3 – Отображение числа подключенных клиентов визуализации в переменной **iCurrentClientCount**

Теперь раскроем наш массив с информацией о клиентах:

astVisuClientInfo	ARRAY [1..Visu_Superglobal_Constants.VISU_MAX_NUMBE...	
astVisuClientInfo[1]	VISU_CLIENT_DATA	
pstClientData	POINTER TO VisuElems.VisuStructClientData	16#AF633BF8
iClientId	INT	0
eClientType	VISU_CLIENTTYPE	ProgrammingSystem
rDevicePixelRatio	REAL	1
iWidthX	INT	1575
iHeightY	INT	846
sIpAddr	STRING(15)	''
sCurrentVisu	STRING	'Visualization'
wsCurrentUserName	WSTRING	"Service"
dwCurrentUserGroupId	DWORD	2
wsCurrentUserGroupName	WSTRING	"Service"
astVisuClientInfo[2]	VISU_CLIENT_DATA	
pstClientData	POINTER TO VisuElems.VisuStructClientData	16#AF647038
iClientId	INT	1
eClientType	VISU_CLIENTTYPE	TargetVisualization
rDevicePixelRatio	REAL	1
iWidthX	INT	799
iHeightY	INT	479
sIpAddr	STRING(15)	''
sCurrentVisu	STRING	'Visualization'
wsCurrentUserName	WSTRING	"Admin"
dwCurrentUserGroupId	DWORD	1
wsCurrentUserGroupName	WSTRING	"Admin"
astVisuClientInfo[3]	VISU_CLIENT_DATA	
pstClientData	POINTER TO VisuElems.VisuStructClientData	16#AF65A7D0
iClientId	INT	2
eClientType	VISU_CLIENTTYPE	WebVisualization
rDevicePixelRatio	REAL	1
iWidthX	INT	1919
iHeightY	INT	936
sIpAddr	STRING(15)	'10.2.8.133'
sCurrentVisu	STRING	'Visualization'
wsCurrentUserName	WSTRING	"Operator"
dwCurrentUserGroupId	DWORD	3
wsCurrentUserGroupName	WSTRING	"Operator"

Рисунок 1.2.4.4 – Информация о клиентах в массиве **astVisuClientInfo**

Значения большинства параметров выглядят очевидно, но стоит дать несколько комментариев:

- обратите внимание, что **iClientId** нумеруются с **0**;
- касательно **iWidthX** / **iHeightY** – разрешение дисплея контроллера **СПК107 [M01]** составляет **800x480**. Действительно, в полученных данных есть ошибка на 1 пиксель. Что касается разрешений сервисной визуализации и web-визуализации – то ***обратите внимание***, что это разрешение не монитора (у меня оно **1920x1080**), а разрешение приложения – то есть области экрана, на которой отображается визуализация. Например, в моем web-браузере отображается панель вкладок, адресная строка и т. д. – и поэтому непосредственно высота отображаемой web-страницы визуализации составляет **936** пикселей. То же касается и сервисной визуализации. Кстати, обратите внимание, что в этих значениях тоже есть «ошибка на единицу» (это заметно по ширине **1919** для web-визуализации);
- для всех клиентов значение **sCurrentVisu** совпадают. Действительно, в моем примере только один экран визуализации с названием **Visualization**. Если бы в проекте было несколько экранов, и разные клиенты работали бы с разными экранами – то, соответственно, значения этой переменной были бы для них различными.

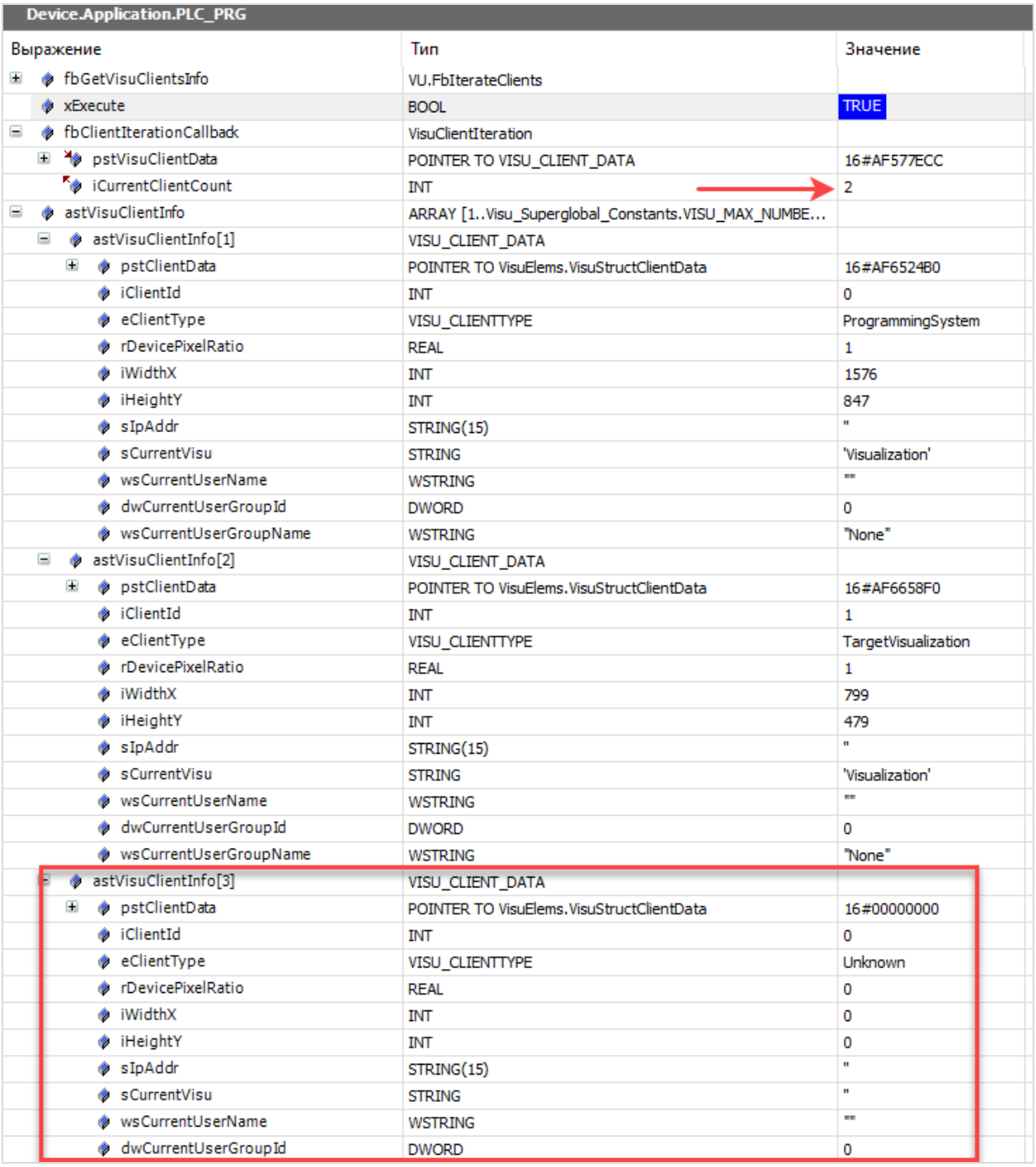
Итак, клиентов визуализации у нас всего 3. Четвертый и последующие элементы массива выглядят вот так:

⊟ astVisuClientInfo[4]	VISU_CLIENT_DATA	
⊞ pstClientData	POINTER TO VisuElems.VisuStructClientData	16#00000000
iClientId	INT	0
eClientType	VISU_CLIENTTYPE	Unknown
rDevicePixelRatio	REAL	0
iWidthX	INT	0
iHeightY	INT	0
sIpAddr	STRING(15)	''
sCurrentVisu	STRING	''
wsCurrentUserName	WSTRING	""
dwCurrentUserGroupId	DWORD	0
wsCurrentUserGroupName	WSTRING	""

Рисунок 1.2.4.5 – Содержимое незаполненного элемента массива для отсутствующего клиента визуализации

Давайте закроем вкладку web-визуализации и заново запустим **fbGetVisuClientsInfo** – то есть подадим на вход **xExecute** значение **FALSE**, а потом снова **TRUE** (напомню, что запуск происходит по переднему фронту).

Теперь мы получим информацию только о двух оставшихся клиентах – третий элемент массива **astVisuClientInfo** будет «пустым».



Device.Application.PLC_PRG

Выражение	Тип	Значение
fbGetVisuClientsInfo	VU.FbIterateClients	
xExecute	BOOL	TRUE
fbClientIterationCallback	VisuClientIteration	
pstVisuClientData	POINTER TO VISU_CLIENT_DATA	16#AF577ECC
iCurrentClientCount	INT	2
astVisuClientInfo	ARRAY [1..Visu_Superglobal_Constants.VISU_MAX_NUMBE...	
astVisuClientInfo[1]	VISU_CLIENT_DATA	
pstClientData	POINTER TO VisuElems.VisuStructClientData	16#AF6524B0
iClientId	INT	0
eClientType	VISU_CLIENTTYPE	ProgrammingSystem
rDevicePixelRatio	REAL	1
iWidthX	INT	1576
iHeightY	INT	847
sIpAddr	STRING(15)	''
sCurrentVisu	STRING	'Visualization'
wsCurrentUserName	WSTRING	""
dwCurrentUserGroupId	DWORD	0
wsCurrentUserGroupName	WSTRING	"None"
astVisuClientInfo[2]	VISU_CLIENT_DATA	
pstClientData	POINTER TO VisuElems.VisuStructClientData	16#AF6658F0
iClientId	INT	1
eClientType	VISU_CLIENTTYPE	TargetVisualization
rDevicePixelRatio	REAL	1
iWidthX	INT	799
iHeightY	INT	479
sIpAddr	STRING(15)	''
sCurrentVisu	STRING	'Visualization'
wsCurrentUserName	WSTRING	""
dwCurrentUserGroupId	DWORD	0
wsCurrentUserGroupName	WSTRING	"None"
astVisuClientInfo[3]	VISU_CLIENT_DATA	
pstClientData	POINTER TO VisuElems.VisuStructClientData	16#00000000
iClientId	INT	0
eClientType	VISU_CLIENTTYPE	Unknown
rDevicePixelRatio	REAL	0
iWidthX	INT	0
iHeightY	INT	0
sIpAddr	STRING(15)	''
sCurrentVisu	STRING	''
wsCurrentUserName	WSTRING	""
dwCurrentUserGroupId	DWORD	0

Рисунок 1.2.4.6 – Информация об отключившемся клиенте будет уделана (см. код в методе StartIteration в ФБ **VisuClientIteration**)

Не обращайте внимания на то, что информация о пользователях (**wsCurrentUserName** и т. д.) больше не отображается – просто для снятия этого скриншота я уже перезагрузил проект и поленился логиниться заново.

Опять подключитесь к web-визуализации контроллера и обновите информацию о клиентах. В третьем элементе массива снова отобразится информация о клиенте web-визуализации, и при этом значение **iClientId**, как и в прошлый раз, будет равно **2**. Если открыть в браузере еще одну вкладку web-визуализации – то второй web-клиент получит **iClientId = 3**.

При обновлении страницы в браузере происходит «разлогинивание» пользователя и открытие экрана визуализации, заданного в качестве стартовой визуализации в узле **WebVisualization**.

Помните, как мы получили значение **wsCurrentUserGroupName**, используя **IVisualizationClientRaw**? Если уже нет – то перечитайте вторую половину п. 1.2.2. Коротко напомню, что для этого мы использовали контекст клиента, который я описал как «огромную структуру данных». Так вот, **pstClientData** на скриншотах выше и является указателем на эту структуру. Выберите в массиве элемент любого клиента и с помощью кнопки «+» раскройте список полей этой структуры:

astVisuClientInfo	ARRAY [1..Visu_Superglobal_Constants.VISU_MAX_NUMBER_OF_CLIENTS] OF...	
astVisuClientInfo[1]	VISU_CLIENT_DATA	
pstClientData	POINTER TO VisuElems.VisuStructClientData	16#AF6524B0
pstClientData^	VisuElems.VisuStructClientData	
GlobalData	VisuStructGlobalClientData	
ElementsData	VisuStructElementsClientData	
AdditionalElementsData	VisuStructAdditionalElementClientData	
AdditionalElementsData2	VisuStructAdditionalElementClientData2	
PaintBuffer	VisuFbCommandBuffer	
pPaintBuffer	POINTER TO VisuFbCommandBuffer	16#00000000
pVisuSpecificData	POINTER TO BYTE	16#AF651A70
Inputdata	Visu_InputData	
Flags	BYTE	16
pDataToDelete	POINTER TO BYTE	16#AF656760
bBestFit	BOOL	FALSE
bBestFitForDialogs	BOOL	FALSE
bScaleTypeIsotropic	BOOL	FALSE
rClientRect	VisuStructSimpleRectangle	
rClientDisplayRect	VisuStructSimpleRectangle	
rDevicePixelRatio	REAL	1
bWasDemo	BOOL	FALSE
itfPaintBufferCreator	IPaintBufferCreator	16#AF5686DC
TemporaryRenderLocationManager	VisuFbTemporaryRenderLocationManager	
BufferLogMessage	VisuFbClientLogger	
FileTransferManager	VisuFbFileTransferManager	

Рисунок 1.2.4.7 – Содержимое структуры **VisuElems.VisuStructClientPoint**

И это только верхний уровень! А ведь практически каждый объект является структурой, ФБ или интерфейсом – то есть их тоже можно раскрыть, и уровней вложенности будет очень много. Некоторые из полей структуры имеют комментарий (обычно он не превышает одной-двух фраз), но об назначении большинства из них остается только догадываться и строить гипотезы. С другой стороны, в некоторых случаях по названию переменной можно догадаться о ее назначении – как, например, мы сделали с **CurrentUserGroupName** – ее не видно на скриншоте, но если вы раскроете **GlobalData**, то у вас не составит труда ее найти.

В старых версиях CODESYS работа с визуализацией из кода требовала взаимодействия с этой структурой – поэтому вполне понятно решение разработчиков CODESYS оградить нас от этой сложности и предоставить «чистое API» в виде библиотеки **Visu Utils**.

Итак, мы разобрались, как использовать **FbIterateClients** и получать информацию о клиентах визуализации. В конце п. 1.2 (его «корневой» части) я сформулировал ряд вопросов – и к этому моменту ответил на большинство из них. Но два вопроса я еще не рассмотрел:

- что такое интерфейсы?
- какие ошибки могут возникнуть в процессе работы блоков библиотеки?

Осталось разобраться с ними – и можно переходить к описанию следующего блока (поскольку в этом разделе мы уже рассмотрели все общие вопросы, то дальше дело должно пойти гораздо бодрее).

1.2.5. Пара слов об интерфейсах

К этому моменту я уже около двух десятков раз использовал термин «интерфейс», никак его не комментируя и как будто бы подразумевая таким же очевидным, как, например, «функциональный блок». Вообще, интерфейсы напрямую не связаны с темой статьи, но поскольку в примерах всё равно приходится их использовать – то я думаю, что стоит сказать о них пару слов.

Если вы программировали на enterprise-языках с использованием объектно-ориентированного подхода (C#, Java и т. д.) – то этот термин вам интуитивно понятен. Если у вас такого опыта нет, то я постараюсь привести несколько тезисов, чтобы создать общее впечатление:

- интерфейс представляет собой набор **прототипов** методов и свойств. Если вам не очевидно, что такое «метод» и «свойство» – то воспринимайте метод как функцию, встраиваемую в ФБ, а свойство – как аналог переменной ФБ, позволяющий при доступе на чтение/запись выполнить фрагмент кода (например, отмасштабировать значение или провалидировать его). Термин «прототип» означает, что на уровне интерфейса методы и свойства не содержат никакого кода – только область объявления переменных;
- интерфейс добавляется в дерево проекта следующим образом: **ПКМ** на узел **Application – Добавление объекта – Интерфейс**;
- интерфейс может наследовать (**EXTENDS**) другой интерфейс, получая его методы и свойства и расширяя их своими;
- ФБ может реализовывать (**IMPLEMENTS**) интерфейс. В этом случае при добавлении ФБ в дереве проекта он автоматически получит методы и свойства этого интерфейса (см. самое начало в п. 1.2.3) – и разработчику останется только добавить их реализацию, написав соответствующий код;
- можно объявить экземпляр интерфейса (то есть переменную, типом которой является интерфейс). Фактически эта переменная будет представлять собой ссылку (неявный указатель) на экземпляр функционального блока, реализующего этот интерфейс. Это обеспечивает определенную гибкость – потому что для разных задач вы можете присваивать экземпляру интерфейса экземпляры разных ФБ – главное, чтобы все они реализовывали общий интерфейс. Вспомните табл. 1.2.1, в которой описаны входы ФБ **FbIterateClients**, среди которых есть **itfClientFilter** и **itfIterationCallback** – они и являются экземплярами интерфейсов. Например, на вход **itfClientFilter** можно передать любой из экземпляров блоков из табл. 1.2.1.1 – **AllClients**, **OnlyTargetVisu** и т. д. Это обеспечивает универсальность – с помощью всего одного входа можно определить группу клиентов, с которой будет работать блок. Вы можете справедливо заметить, что реализовать подобное поведение можно было бы и другими способами – но разработчики CODESYS предпочли именно такой вариант.

На этом мы заканчиваем обсуждать интерфейсы. Если вам хотелось бы получить больше информации об использовании объектно-ориентированного подхода в CODESYS V3.5 – то вы можете ознакомиться со списком литературы из этой статьи.

1.2.6. Коды ошибок (ERROR)

В процессе работы блоков библиотеки **Visu Utils** могут возникнуть ошибки. В этом случае выход **xError** экземпляра блока принимает значение **TRUE**, а на выходе **eError** отображается код ошибки. Подробнее об изменении значений выходов см. в пояснении перед рис. 1.2.4.2.

Выход **eError** представляет собой перечисление типа ERROR. Ниже описаны содержащиеся в нем коды ошибок:

Таблица 1.2.6.1 – Элементы перечисления **ERROR** (коды ошибок)

Название	Описание
NO_ERROR	Отсутствие ошибок. Выход **eError** имеет это значение, если **xError = FALSE**
TIME_OUT	Ошибка таймаута (блок не успел выполниться за заданное время). У блоков библиотеки нет входа **udiTimeOut**, определяющего время таймаута, так что неясно, в каких случаях может быть возвращена эта ошибка
NOT_CALLED_FROM_VISU	ФБ вызван не из визуализации. Вероятно, эта ошибка возвращается в том случае, если при вызове блока в коде программы в качестве фильтра используется **CurrentClient** (см. табл. 1.2.1.1)
NOT_SUPPORTED_FOR_INTEGRATED_VISU	Данный ФБ не поддержан для выполнения в сервисной визуализации CODESYS. В частности, это касается ФБ FBFileTransfer
ERROR	Недокументированная ошибка
TRANSFER_CANCELLED	Передача файла была прервана (ФБ FBFileTransfer)
TRANSER_IN_PROGRESS	Передача файла в процессе выполнения (ФБ FBFileTransfer)
TRANSFER_FAILED	Передача файла не была успешно завершена (ФБ FBFileTransfer)
FILE_READ_ERROR	Не удалось прочитать передаваемый файл (ФБ FBFileTransfer)
VISU_NOT_FOUND	Не удалось переключить экран визуализации, так как в проекте нет экрана с таким именем (ФБ FbChangeVisu)
NO_CLIENT_FILTER	Не удалось выполнить операцию, так как не задано значение входа **itfClientFilter**
NO_ITERATION_CALLBACK	Не удалось выполнить операцию, так как не задано значение входа **itfIterationCallback** (ФБ FbIterateClients)
FILE_TRANSFER_SERVICES_NOT_SUPPORTED	Функционал передачи файлов не поддерживается данным контроллером (ФБ FBFileTransfer)
BLOCKED_DUE_TO_ONLINECHANGE	Выполнение ФБ было заблокировано на время выполнения горячего обновления кода (online change)
CANCELLED_DUE_TO_ONLINE_CHANGE	Выполнение ФБ было отменено из-за выполнения горячего обновления кода (online change)

В будущем разработчики CODESYS планируют детализировать описание ошибок (VIS-2704).

1.2.7. Пара слов об идентификаторах клиентов

Я написал этот пункт только для того, чтобы потом суметь вспомнить всю изложенную ниже специфику без повторных экспериментов. Скорее всего, он будет вам не интересен – это нормально.

В состав интерфейса IVisualizationClient входит свойство **ClientId** (типа **INT**) – уникальный идентификатор клиента. Его польза интуитивно понятна – с помощью него можно однозначно отличить одного клиента от другого (все остальные свойства у клиентов могут совпадать – например, если открыть одну и ту же вкладку web-визуализации на одном ПК).

Важно отметить, что для работы с с этим свойством потребуется добавить в проект библиотеку **VisuGlobalClientManager** – иначе для всех клиентов оно будет иметь значение **-1**.

В документации на библиотеку **ClientId** описывается так:

«*The global client ID, which is set, if the according setting in the visumanager is active. See VisuElemBase.VisuStructGlobalClientData.GlobalClientID*»

Стоит сказать вот что:

- я не понимаю, какая именно «according setting» имеется в виду;
- если вы объявите в проекте переменную типа **VisuElems.VisuElemBase.VisuStructGlobalClientData** и подключитесь к контроллеру – то увидите, что содержимое этой структуры фактически повторяет контекст клиента (см. рис. 1.2.4.7). При этом все поля этой структуры будут «пустыми»;
- я могу предположить, что использовать эту структуру можно только в контексте визуализации, чтобы получить информацию о текущем клиенте. Но при попытке привязать к прямоугольнику переменную **VisuElems.VisuElemBase.VisuStructGlobalClientData.GlobalClientID** возникает ошибка компиляции:

C0407: Instance required instead of type name in address-of expression 'VisuElems.VisuElemBase.VisuStructGlobalClientData.GlobalClientId'

Но мы не сдаемся. Мы читали примеры из FAQ CODESYS и даже злобным «dsgsdg» нас не смутишь. Привязываем к элементу визуализации **GlobalClientId** текущего клиента (**VisuElems.CurrentVisuClient^.GlobalData.GlobalClientId**):

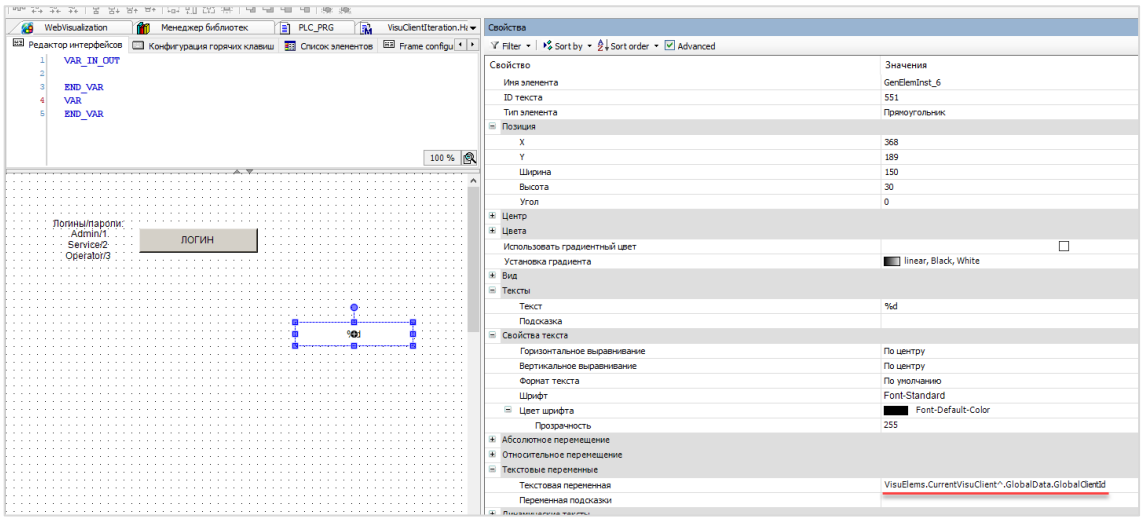


Рисунок 1.2.7.1 – Отображение идентификатора текущего клиента визуализации

Он будет совпадать со значением **ClientId** из интерфейса IVisualizationClient. Теперь мы знаем, что это одно и то же. Стала ли от этого наша жизнь легче? Навряд ли.

Но это еще не все. Мы читали про **CURRENTCLIENTID** (например, в документе CODESYS V3.5. Визуализация). Добавляем в менеджер библиотек библиотеку **VisuGlobalClientManager** (это необходимо, чтобы вопользоваться нужной нам системной переменной) и привязываем переменную **CURRENTCLIENTID** к элементу визуализации:

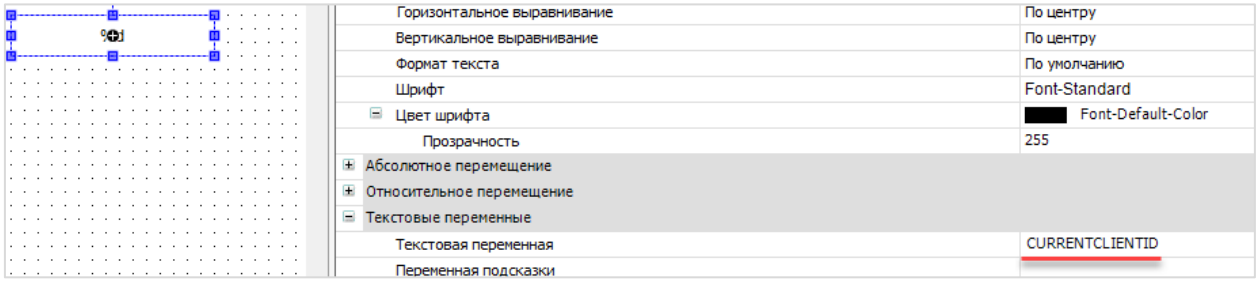


Рисунок 1.2.7.2 – Привязка системной переменной **CURRENTCLIENTID** к элементу визуализации

Отображаемое значение будет совпадать с **ClientId** и **VisuElems.CurrentVisuClient^.GlobalData.GlobalClientId**. Я вижу это собственными глазами (но вам не стоит мне доверять – проведите описанный эксперимент самостоятельно), и поэтому не могу согласиться с теми, кто утверждает обратное.

Ценный комментарий дает сотрудник **CODESYS Group** по имени **Marcel Prestel** (я привожу его сразу в переводе):

«*Вот некоторые пояснения*:

- *GlobalClientID – внутренний идентификатор клиента;*
- *CurrentClientID – всегда обновляется для конкретного клиента до его GlobalClientID. Его значение не определено, если обращение к нему производиться не из контекста визуализации;*
- *GetClientID – другой диапазон значений (не имеет ничего общего с GlobalClientID).»*

Осталось только добавить, что **GetClientID** – это метод менеджера клиентов (**VisuElems.VisuElemBase.Visu_Globals.g_ClientManager**), о котором я еще упомяну в п. 3.2.

Если вы внимательно прочитали этот пункт до конца – то, видимо, вы действительно интересуетесь нюансами визуализации CODESYS.

1.2.8. ФБ библиотеки Visu Utils и задача VISU_TASK

В документации ФБ **FbIterateClients** есть интересное примечание: «*As the iteration takes place within **VISU_TASK** please make sure that the given object is alive long enough!*». Похожее примечание есть в объявлении методов интерфейса IVisualizationClientIteration (см. рис. 1.2.3.3): «Please remark that this method will be called from **VISU_TASK**».

Надо сказать, что вызывать экземпляры ФБ из **Visu Utils** можно в любой задаче (необязательно в **VISU_TASK**), но их выполнение все равно произойдет в контексте задачи **VISU_TASK** (пруф от разработчиков CODESYS). В этом заключается одно из их существенных отличий от «старого» способа работы с визуализацией через объекты библиотеки **VisuElemBase** – для корректной работы требовалось вызывать их именно в **VISU_TASK**.

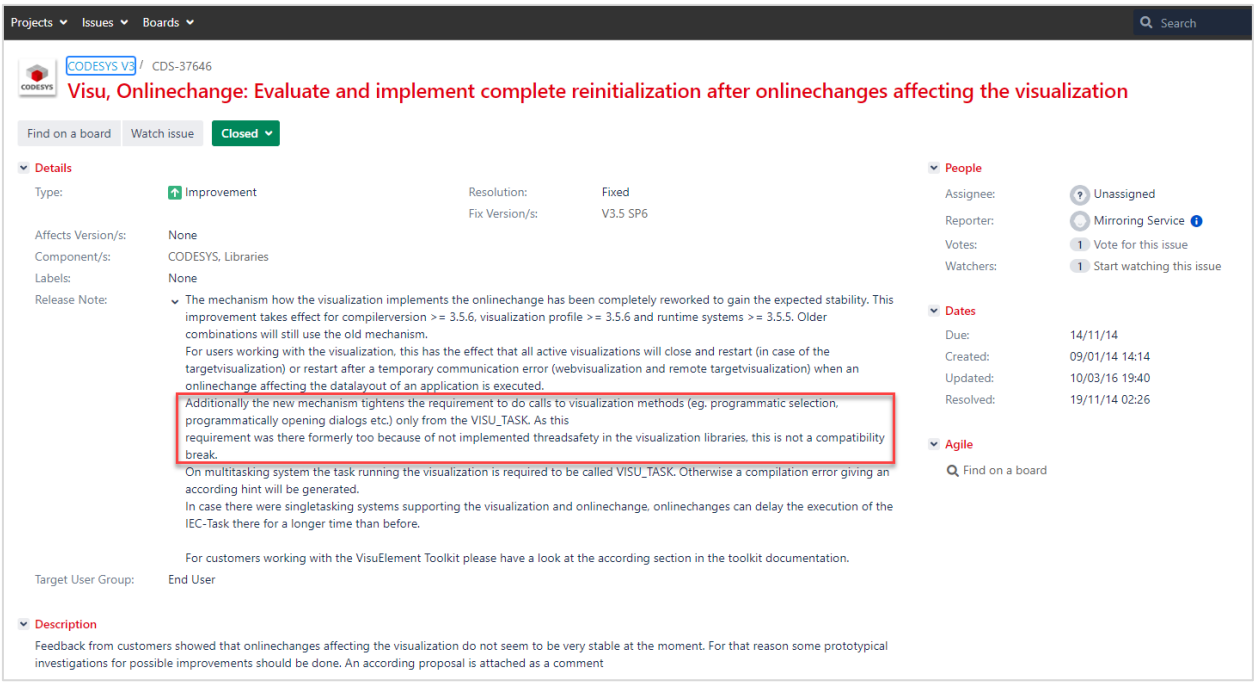


Рисунок 1.2.8.1 – Примечание в баг-трекере CODESYS о том, что «старые» способы работы с визуализацией из кода требовали вызовов объектов именно в контексте задачи **VISU_TASK**

Сложно сказать, какой именно вывод должен сделать разработчик из этих примечаний. Возможно, речь о том, что если заданный интервал задачи, в которой происходит вызов экземпляра **FbIterateClients**, существенно превышает период вызова **VISU_TASK** – то к моменту получения информации она может перестать быть актуальной (например, один из клиентов уже отключится).

1.2.9. Клиенты и пользователи визуализации

В этом и следующих пунктах я использую термины «клиенты визуализации» и «пользователи визуализации».

Под «клиентом» я подразумеваю устройство, на котором отображается визуализация CODESYS – например, дисплей панельного контроллера (тип этого клиента – таргет-визуализация), дисплей смартфона (тип клиента – web-визуализация; при этом каждая открытая вкладка считается отдельным клиентом), монитор компьютера (тип клиента – web-визуализация, сервисная визуализация среды CODESYS, удаленная таргет-визуализация или приложение CODESYS HMI) и т. д.

«Пользователем» я буду называть клиента, который авторизуется в визуализации, используя логин и пароль (для этого в проекте CODESYS должно быть настроено управление пользователями – созданы группы пользователей, добавлены пользователи, настроены их права и т. д.).

1.3. Переключение экрана для клиентов визуализации (FbChangeVisu)

Функциональный блок FbChangeVisu предназначен для переключения экрана визуализации для заданной группы клиентов. Как и остальные блоки библиотеки **Visu Utils** – данный блок соответствует модели поведения PLCopen Behaviour Model.

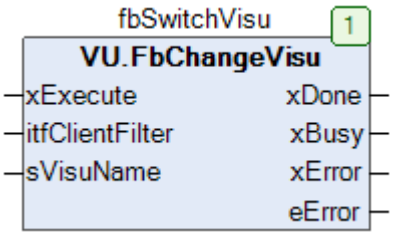


Рисунок 1.3.1 – Внешний вид ФБ **FbChangeVisu** на языке CFC

Таблица 1.3.1 – Описание входов и выходов ФБ **FbChangeVisu**

Название	Тип	Описание
Входы		
xExecute	BOOL	По переднему фронту происходит переключение экрана визуализации для всех клиентов визуализации, соответствующих фильтру **itfClientFilter**
itfClientFilter	IVisualizationClientFilter	Экземпляр ФБ, реализующего интерфейс **IVisualizationClientFilter**. Используется для определения категории клиентов, для которых будет переключен экран. Пользователь может реализовать этот ФБ самостоятельно или воспользоваться одним из готовых, объявленных в списке глобальных переменных библиотеки (Globals)
sVisuName	STRING	Имя экрана визуализации, на который произойдет переключение
Выходы		
xDone	BOOL	Принимает значение **TRUE** после переключения экрана визуализации
xBusy	BOOL	Имеет значение **TRUE**, пока блок находится в процессе работы
xError	BOOL	Принимает значение **TRUE** в случае возникновения ошибки в процессе работы блока
eError	VU.ERROR	Код ошибки

В прошлом пункте в качестве фильтра клиентов мы использовали готовый фильтр **AllClients** из списка глобальных переменных Globals. В этом пункте давайте разберемся, как реализовать свой собственный фильтр. Для это продолжим работать с примером из предыдущего пункта и добавим в него следующий функционал – пусть при авторизации пользователя **Admin** именно для этого пользователя произойдет переключение на экран **VisuAdmin**. Такого экрана еще нет в нашем проекте – так что сразу его добавим:

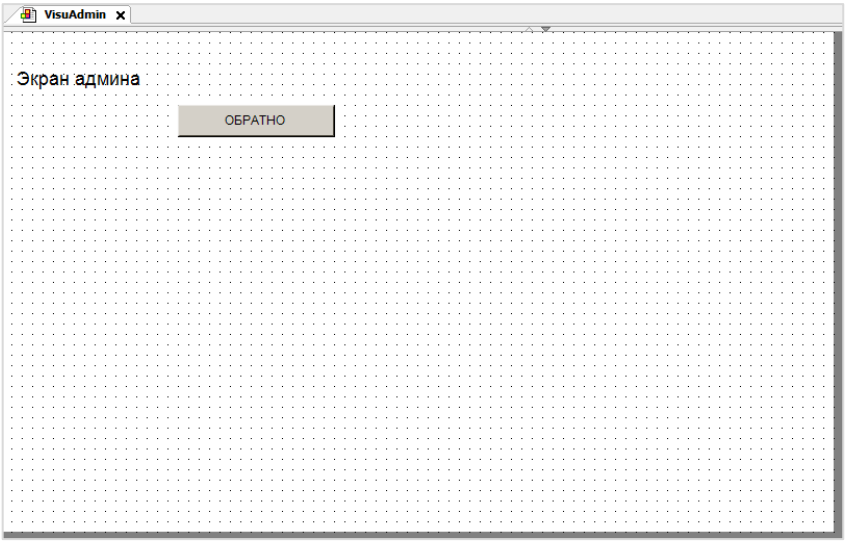


Рисунок 1.3.2 – Внешний вид экрана **VisuAdmin**

В действиях кнопки **ОБРАТНО** настроено возвращение на предыдущий экран визуализации.

Теперь приступаем к созданию фильтра клиентов. Создаем ФБ с названием **VisuClientFilter**, который реализует (IMPLEMENTS) интерфейс **IVisualizationClientFilter**.

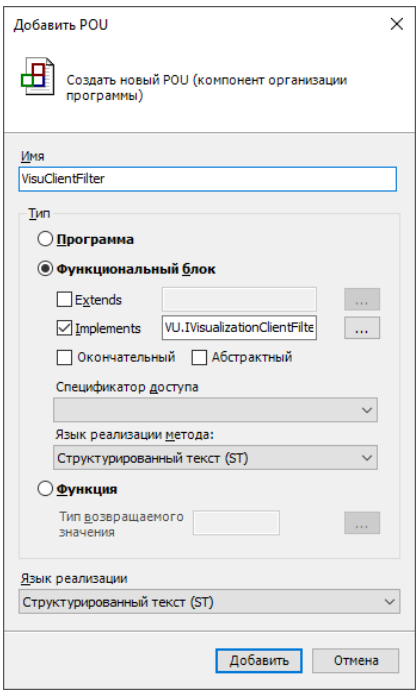


Рисунок 1.3.4 – Создание ФБ, реализующего интерфейс **IVisualizationClientFilter**

ФБ автоматически получит единственный метод, имеющийся у интерфейса – **IsAccepted**. Именно этот метод используется для определения фильтра клиентов. Рассмотрим, как всё это работает:

- у нас есть экземпляр ФБ **FbChangeVisu**, который имеет вход **itfClientFilter**;
- при вызове этого экземпляра мы передаем на данный вход экземпляр ФБ, реализующего интерфейс **IVisualizationClientFilter** – т. е. экземпляр **VisuClientFilter** (который мы еще не объявили, но скоро это сделаем);
- при вызове экземпляра **FbChangeVisu** происходит вызов экземпляра **VisuClientFilter** по технологии callback (помните, что мы уже рассматривали этот подход в п. 1.2.3?) и его метода **IsAccepted**;
- метод **IsAccepted** однократно вызывается для каждого клиента визуализации, и при этом на его вход **itfClient** передается экземпляр интерфейса IVisualizationClient очередного клиента;
- разработчик должен реализовать код метода **IsAccepted** – используя информацию о клиенте определить, нужен ли этот клиент для его фильтра, и, если нужен, – присвоить выходу метода значение **TRUE**.

Методы и свойства интерфейса **IVisualizationClient** мы уже рассмотрели в п. 1.2.2, так что решить озвученную выше задачу будет несложно:

VisuClientFilter.IsAccepted

```
{warning 'добавить реализацию метода'}
(* For every client can be desided, if it is accepted.

 ``TRUE``: Client is accepted*)
METHOD IsAccepted : BOOL
VAR_INPUT
    (* The client, to check*)
    itfClient   : VU.IVisualizationClient;
END_VAR

IsAccepted := ( itfClient.UserName = "Admin" );
```

Рисунок 1.3.5 – Код метода **IsAccepted**

Вы можете справедливо заметить, что название ФБ (**VisuClientFilter**) подразумевает универсальность, но реализация нашего метода предельно конкретна и даже использует хардкод ("Admin"). Действительно, в реальном проекте, вероятно, будет удобнее сделать блок универсальным – добавить у него вход типа перечисление, определяющее конкретный фильтр, и в коде метода в зависимости от значения этого входа через оператор **CASE** выбирать нужных клиентов. Соответственно, перед передачей экземпляра ФБ на вход **itfClientFilter** потребуется присвоить этому входу нужное значение.

Но для нашего простого примера вполне подойдет и показанный выше вариант.

Осталось объявить в программе экземпляры **VisuClientFilter** и **FbChangeVisu**, и организовать вызов последнего из них:

```
PROGRAM PLC_PRG
VAR
   fbGetVisuClientsInfo:          VU.FbIterateClients;
   xExecute:                      BOOL;
   fbClientIterationCallback:     VisuClientIteration;
   astVisuClientInfo:             ARRAY
       [1..Visu_Superglobal_Constants.VISU_MAX_NUMBER_OF_CLIENTS]
         OF VISU_CLIENT_DATA;

   fbClientFilter:                VisuClientFilter;
   fbChangeVisu:                  VU.FbChangeVisu;
END_VAR


fbClientIterationCallback.pstVisuClientData := ADR(astVisuClientInfo);

fbGetVisuClientsInfo
(
   xExecute             := xExecute,
   // будем собирать информацию о всех клиентах визуализации
   itfClientFilter      := Vu.Globals.AllClients,
   // экземпляр ФБ, автоматически вызываемого для обработки клиентов
   itfIterationCallback := fbClientIterationCallback
);

// информация о всех интересующих клиентах получена
IF fbGetVisuClientsInfo.xDone THEN

   // и здесь можно что-то с ней сделать

END_IF

// переключаем экран визуализации для пользователя Admin
fbChangeVisu
(
   xExecute        := fbGetVisuClientsInfo.xDone,
   itfClientFilter := fbClientFilter,
   sVisuName       := 'VisuAdmin'
);
```

На вход **xExecute** мы передаем выход **xDone** экземпляра ФБ **VU.FbIterateClients** – то есть сразу после сбора информации о клиентах будет выполнено переключение экрана визуализации для пользователя **Admin**. Загрузите проект в контроллер и проверьте, как он работает:

- авторизуйтесь под пользователем **Admin** (пароль – «**1**»);
- присвойте переменной программы **xExecute** значение **TRUE**;
- наблюдайте, как произойдет переключение экрана визуализации.

Вот ссылка на готовый пример: скачать

На самом деле, у нашего примера есть существенный недостаток. Вызов экземпляра **FbChangeVisu** происходит при каждом очередном получении информации о клиентах. То есть возможна такая ситуация:

- клиент авторизовался как пользователь **Admin**;
- произошел очередной сбор информации о клиентах;
- пользователя переключило на экран **VisuAdmin**;
- пользователь перешел на какой-то другой экран;
- произошел очередной сбор информации о клиентах;
- …и пользователя опять переключило на экран **VisuAdmin**, хотя вряд ли он этого хотел.

Решить эту проблему можно разными способами, но главное – четко сформулировать, что нам нужно. А нужно нам, чтобы переключение экрана происходило однократно после авторизации пользователя «**Admin**».

На мой взгляд, сделать это средствами **Visu Utils** можно, но потребует существенных усилий. Предлагаю вам самим реализовать такой пример. Я же покажу простое действенное решение (я являюсь сторонником именно таких решений), основанное не на **Visu Utils**, а использующее возможности низкоуровневых системных библиотек визуализации.

Для начала объявим в программе массив переменных типа **WSTRING** с именем «предыдущего» пользователя (то есть пользователя, который был авторизован в системе ранее), массив переменных типа **INT** и массив переменных типа **BOOL** (*и да, их можно было бы объединить в структуру, но я поленился это сделать*). Сразу объявим счетчик для цикла **FOR**. Далее покажу готовый код, а потом прокомментирую его.

Вот ссылка на готовый пример: скачать

```
PROGRAM PLC_PRG
VAR
   fbGetVisuClientsInfo:          VU.FbIterateClients;
   xExecute:                      BOOL;
   fbClientIterationCallback:     VisuClientIteration;
   astVisuClientInfo:             ARRAY
       [1..Visu_Superglobal_Constants.VISU_MAX_NUMBER_OF_CLIENTS]
         OF VISU_CLIENT_DATA;

   fbClientFilter:                VisuClientFilter;
   fbChangeVisu:                  VU.FbChangeVisu;

   awsPrevUserName:               ARRAY
      [1..Visu_Superglobal_Constants.VISU_MAX_NUMBER_OF_CLIENTS] OF WSTRING;
   auiCompareUserNameResult:      ARRAY
      [1..Visu_Superglobal_Constants.VISU_MAX_NUMBER_OF_CLIENTS] OF UINT;
   axNewClientFlag:               ARRAY
      [1..Visu_Superglobal_Constants.VISU_MAX_NUMBER_OF_CLIENTS] OF BOOL;
   i:                             INT;
END_VAR
```

```
fbClientIterationCallback.pstVisuClientData := ADR(astVisuClientInfo);

fbGetVisuClientsInfo
(
   xExecute             := xExecute,
   // будем собирать информацию о всех клиентах визуализации
   itfClientFilter      := VU.Globals.AllClients,
   // экземпляр ФБ, автоматически вызываемого для обработки клиентов
   itfIterationCallback := fbClientIterationCallback
);

// информация о всех интересующих клиентах получена
IF fbGetVisuClientsInfo.xDone THEN

   // проходимся по всем клиентам
   FOR i := 1 TO Visu_Superglobal_Constants.VISU_MAX_NUMBER_OF_CLIENTS DO

      // проверяем, что произошла авторизация нового пользователя
      // (т.е. имя пользователя изменилось)
      auiCompareUserNameResult[i] :=
      MEM.Compare(ADR(astVisuClientInfo[i].wsCurrentUserName),
         ADR(awsPrevUserName[i]), TO_UINT(Standard64.WLEN(awsPrevUserName[i]) ) );

      // если новый пользователь - админ, и он только что зашел...
      IF ( auiCompareUserNameResult[i] = 0 AND
         astVisuClientInfo[i].wsCurrentUserName = "Admin" AND
            NOT(axNewClientFlag[i]) ) THEN

         // то переключаем его на экран VisuAdmin

        VisuElems.g_VisuManager.SetMainVisu(astVisuClientInfo[i].pstClientData,
           'VisuAdmin');

         // помечаем, что он авторизовался,
          // чтобы переключение экрана произошло однократно
         axNewClientFlag[i] := TRUE;
         awsPrevUserName[i] := astVisuClientInfo[i].wsCurrentUserName;

      // если имя пользователя изменилось с Admin на какое-то другое -
      // то клиент зашел под другим пользователем или отключился
      ELSIF awsPrevUserName[i] = "Admin" AND (auiCompareUserNameResult[i] <> 0 OR
         astVisuClientInfo[i].wsCurrentUserName = "") THEN

         // очищаем данные о старом клиенте
        MEM.MemFill(ADR(awsPrevUserName[i]), SIZEOF(awsPrevUserName[i]), 0);
        axNewClientFlag[i] := FALSE;

      END_IF

   END_FOR

END_IF
```

Итак, переключение экрана должно происходить однократно после авторизации пользователя «**Admin**». Соответственно, нам нужно определить, что в систему вошел новый пользователь. Это можно сделать с помощью библиотеки **VisuElemBase**, о которой мы пока что не знаем (узнаем в п. 2.9). Поэтому используем подход, доступный нам сейчас – сравним текущее имя пользователя с именем пользователя, которое было у этого клиента визуализации в прошлом цикле задачи контроллера. Для хранения имен из прошлого цикла используем массив **awsPrevUserName**.

Поэтому проходимся по всем клиентам в цикле **FOR**. Для каждого клиента с помощью функции **Compare** из библиотеки **CAA Memory** проверяем – не изменилось ли имя пользователя?

Мы должны переключить экран для клиента, если:

- его имя изменилось;
- новое имя – **Admin**;
- он только что вошел (чтобы переключение экрана было однократным).

Именно эти три условия проверяются в первом **IF** внутри **FOR**. Если все они выполняются – то происходит переключение экрана визуализации для конкретного клиента с помощью метода **VisuElems.g_VisuManager.SetMainVisu**. Первым аргументом метода является указатель на контекст клиента, для которого будет произведено переключение экрана, а вторым – имя этого экрана.

После переключения мы устанавливаем флаг **axNewClientFlag** и записываем в **awsPrevUserName** имя вошедшего пользователя (**Admin**). Это нужно, чтобы предотвратить «циклическое» переключение экрана для вошедшего админа – ведь иначе он не сможет перейти на какой-нибудь другой экран. Теперь наше условие в **IF** не выполняется.

Что может произойти дальше? Вот что:

- клиент может «разлогиниться» (по своей инициативе, по таймауту неативности или из-за обновления страницы в web-браузере);
- клиент может перелогиниться под другим пользователем (например, «**Service**»).

Но результат один – текущее имя авторизованного пользователя изменится (на другое или на «пустое» – в случае разлогинивания). Поэтому в ветке **ELSIF** мы проверяем, что если клиент был админом, а стал кем-то другим – то сбрасываем наш флаг **axNewClientFlag** и очищаем **awsPrevUserName** с помощью функции **MemFill** из библиотеки **CAA Memory** (очистку строк следует производить именно так; процедура присвоения пустой строки только записывает значение 0 в первый символ строки, но остальные данные строки сохраняются. Если, например, имя нового пользователя будет короче предыдущего – то в этом случае к нему могут «приклеиться» остатки имени предыдущего пользователя).

Вы можете справедливо заметить, что приведенный выше код не оптимален – он многословен, требует явной итерации по всему массиву клиентов и т. д. Даже компилятор выдает предупреждение, что используемый нами системный метод **SetMainVisu** является устаревшим и предлагает «использовать вместо него Visu Utils». В ответ я еще раз предлагаю вам сделать это в качестве самостоятельного упражнения. У приведенного выше примера есть одно конкретное преимущество – он работает уже здесь и сейчас.

Вполне возможно, что в одной из следующих версий **Visu Utils** добавят функционал, позволяющий определить момент авторизации (или разлогинивания) нового клиента и выполнить в этот момент свой код – например, с помощью нового ФБ, реализующего технологию callback (по аналогии с **itfIterationCallback** у FbIterateClients). Соответствующее пожелание есть в баг-трекере CODESYS (см. рис. 1.3.6 на следующей странице).

Мы еще поговорим о работе с пользователями визуализации из кода в п. 2.

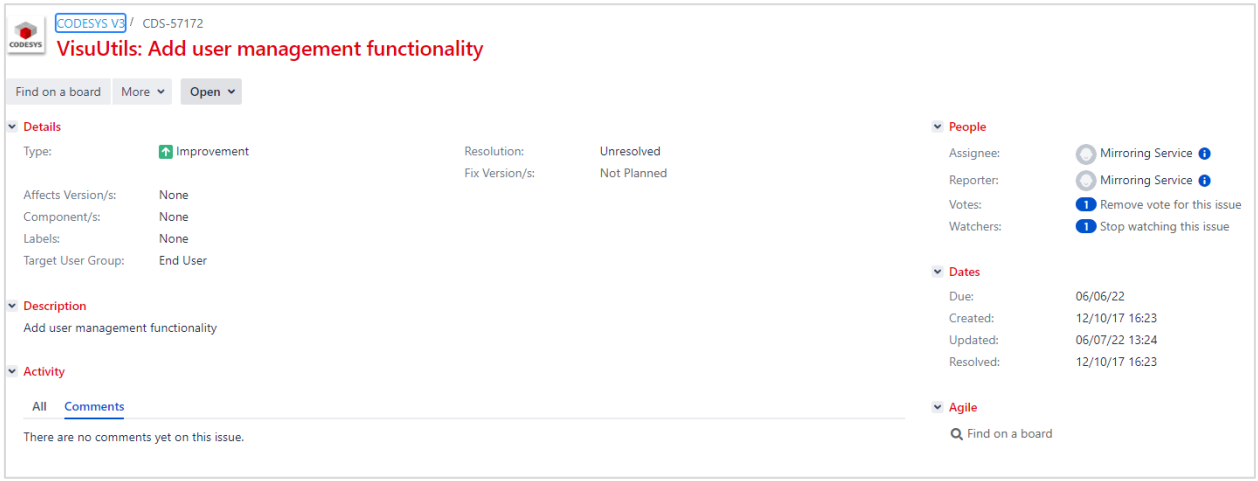


Рисунок 1.3.6 – Пожелание в баг-трекере CODESYS о добавлении в библиотеку **Visu Utils** функционала по работе с пользователями визуализации (**VisuUserManagement**)

1.4. Открытие диалога для клиентов визуализации (FbOpenDialog, FbOpenDialog Extended)

Функциональные блоки FbOpenDialog и FbOpenDialogExtended предназначены для открытия диалогов для заданной группы клиентов. Блок **FbOpenDialogExtended** является «расширенной версией» блока **FbOpenDialog** и позволяет не только открыть диалог, но и передать в него нужные данные, а также получить данные, введенные пользователем в диалоге, при его закрытии. Пример:

- если вам нужно открыть окно с сообщением о тревоге, содержащее только статический текст – используйте ФБ **FbOpenDialog**;
- если вам нужно открыть диалог с произвольным сообщением (и передать текст этого сообщения как аргумент при вызове диалоге) и получить в программе информацию, которую пользователь введет в диалоге – то используйте ФБ **FbOpenDialogExtended**.

Закрыть диалог можно с помощью кнопки (или другого элемента визуализации) с настроенным во вкладке **Конфигурация ввода** действием **Закрыть диалог** или же с помощью ФБ FbCloseDialog.

Как и остальные блоки библиотеки **Visu Utils** – данный блок соответствует модели поведения PLCopen Behaviour Model.

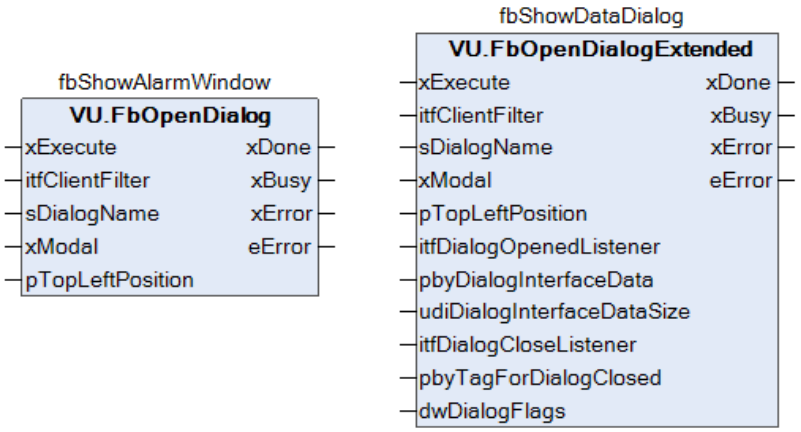


Рисунок 1.4.1 – Внешний вид ФБ **FbOpenDialog** и **FbOpenDialogExtended** на языке CFC

Таблица 1.4.1 – Описание входов и выходов ФБ **FbOpenDialog**

Название	Тип	Описание
Входы		
xExecute	BOOL	По переднему фронту происходит открытие диалога для всех клиентов визуализации, соответствующих фильтру **itfClientFilter**
itfClientFilter	IVisualizationClientFilter	Экземпляр ФБ, реализующего интерфейс **IVisualizationClientFilter**. Используется для определения категории клиентов, для которых будет открыт диалог. Пользователь может реализовать этот ФБ самостоятельно или воспользоваться одним из готовых, объявленных в списке глобальных переменных библиотеки (Globals)
sDialogName	STRING	Имя открываемого диалога
xModal	BOOL	**TRUE** – диалог открывается в модальном режиме. В этом случае элементы, расположенные на экране визуализации, поверх которого открыт диалог, не будут реагировать на нажатия. **FALSE** – элементы, расположенные на экране визуализации, поверх которого открыт диалог, будут реагировать на нажатия.
pTopLeftPostion	POINTER TO VisuElems.CmpVisuHandler.VisuStructPoint	Указатель на структуру координат открытия диалога. Структура содержит два поля (**iX** и **iY**), которые определяют координаты верхней левой точки диалога на экране. Если указатель не инициализирован – то диалог будет открыт по центру экрана
Выходы		
xDone	BOOL	Принимает значение **TRUE** после открытия диалога
xBusy	BOOL	Имеет значение **TRUE**, пока блок находится в процессе работы
xError	BOOL	Принимает значение **TRUE** в случае возникновения ошибки в процессе работы блока
eError	VU.ERROR	Код ошибки

Принцип фильтрации клиентов мы уже рассмотрели в п. 1.3.

1.4.1. Специфические входы ФБ FbOpenDialogExtended

По сравнению с ФБ FbOpenDialog, рассмотренном выше, блок **FbOpenDialogExtended** имеет следующие специфические входы:

Таблица 1.4.2 – Описание специфических (по сравнению с **FbOpenDialog**) входов ФБ **FbOpenDialogExtended**

Название	Тип	Описание
Специфические входы		
itfDialogOpenedListener	VisuElems.VisuElemBase.IDialogOpenedListener	Экземпляр ФБ, реализующего интерфейс **IDialogOpenedListener**. Автоматически вызывается после открытия диалога для каждого клиента, соответствующего фильтру **itfClientFilter** (см. табл. 1.4.1)
pbyDialogInterfaceData	POINTER TO BYTE	Указатель на структуру данных диалога. См. подробности в тексте под таблицей
udiDialogInterfaceDataSize	UDINT	Размер структуры данных диалога в байтах
itfDialogCloseListener	VisuElems.VisuElemBase.IDialogCloseListener	Экземпляр ФБ, реализующего интерфейс **IDialogCloseListener**. Автоматически вызывается после закрытия диалога для каждого клиента, соответствующего фильтру **itfClientFilter** (см. табл. 1.4.1)
pbyTagForDialogClosed	POINTER TO BYTE	Указатель на данные диалога, передаваемые в экземпляр ФБ, который присвоен на вход **itfDialogCloseListener**. Позволяет обработать информацию, введенную пользователем в диалоге
dwDialogFlags	DWORD	Битовая маска флагов открытия диалога. См. подробности в п. 1.4.7

Эти специфические входы позволяют нам:

- выполнить какой-то код при открытии диалога (**itfDialogOpenedListener**);
- инициализировать структуру данных диалога (**pbyDialogInterfaceData**, **udiDialogInterfaceDataSize**);
- выполнить какой-то код при закрытии диалога (**itfDialogCloseListener**);
- получить структуру данных при закрытии диалога – то есть узнать значения переменных диалога, измененных пользователем;
- установить флаги диалога.

Рассмотрим всё это на конкретном примере. Как обычно, я сразу привожу ссылку на готовый проект: скачать

1.4.2. Описание диалога из рассматриваемого примера

В примере мы будем работать с диалогом **dlgFeedbackDialog**. Он имеет одну входную переменную (**VAR_INPUT**) и одну переменную класса «вход-выход» (**VAR_IN_OUT**).

Вход **wsMessage** – это текстовое сообщение, которое мы будем формировать в программе и отображать в диалоге с помощью элемента **Прямоугольник**.

Вход-выход **usiQualityOfService** – это число, которое характеризует качество обслуживания (то есть степень удовлетворенности клиента). Начальное значение этой переменной мы будем задавать в программе. В диалоге клиент сможет изменить ее значение с помощью элемента **Радио-кнопка**.

К кнопкам **ОК** и **ОТМЕНА** привязано действие **Закрыть диалог** с результатом **ОК** и **Отмена** соответственно.

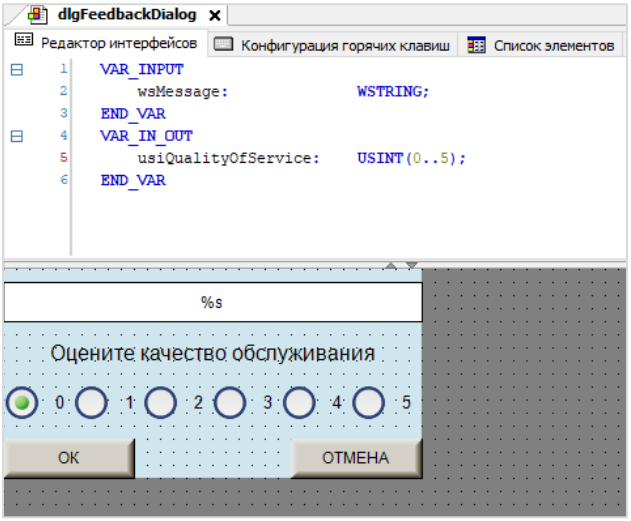


Рисунок 1.4.2.1 – Внешний и переменные диалога **dlgFeedbackDialog**

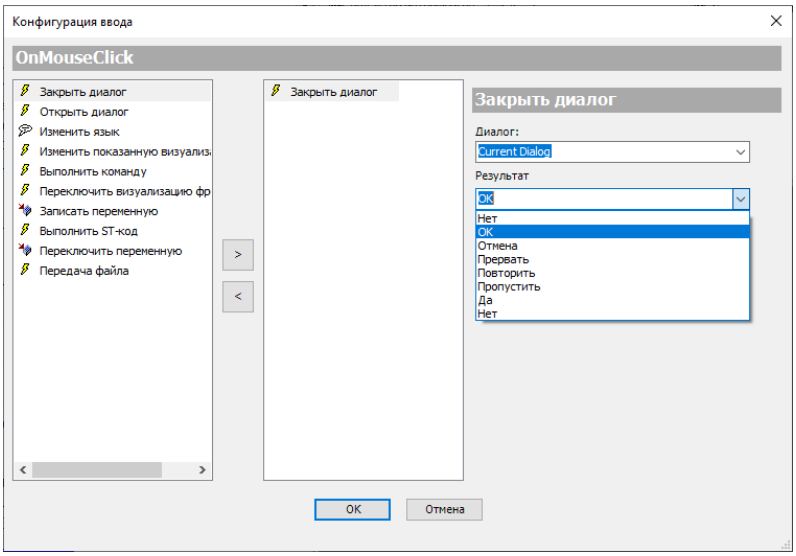


Рисунок 1.4.2.2 – Настройки конфигурации ввода кнопки **ОК**

Для каждого диалога автоматически и неявно создается структура, содержащая его поля (в нашем случае – переменные **wsMessage** и **usiQualityOfService**). Название этой структуры выглядит следующим образом: имя_диалога_VISU_STRUCT (т. е. в нашем случае – **dlgFeedbackDialog_VISU_STRUCT**).

Давайте начнем объявлять переменные нашей программы и напишем первые строки кода.

```
PROGRAM PLC_PRG
VAR
   fbOpenFeedbackDialog:             VU.FbOpenDialogExtended;
   xExecute:                         BOOL;

   stOpenFeedbackDialog:             dlgFeedbackDialog_VISU_STRUCT;
   stCloseFeedbackDialog:            dlgFeedbackDialog_VISU_STRUCT;

   xInit:                            BOOL;
END_VAR


IF NOT(xInit) THEN

   stOpenFeedbackDialog.wsMessage := "Какое-то сообщение";
   stOpenFeedbackDialog.usiQualityOfService := 3;

   xInit := TRUE;

END_IF

// Вызов fbOpenFeedbackDialog добавим в п. 1.4.4
```

Я буду использовать два экземпляра структуры – с помощью первого из них будут задаваться начальные значения переменных, которые клиент будет видеть при открытии диалога, а второй будет содержать значение **usiQualityOfService**, введенное клиентом (значение поля **wsMessage** в обоих экземплярах будет совпадать, так как оно не может быть изменено клиентом). В ваших проектах мы можете обойтись одним экземпляром структуры, если перезапись начальных значений диалога теми значениями, которые введет клиент, приемлема в рамках вашей задачи.

1.4.3. Обработка открытия диалога (itfDialogOpenedListener)

Обработка открытия и закрытия диалога производится по технологии обратного вызова (callback), о которой мы уже упоминали в п. 1.2.

Поэтому действуем уже известным нам способом – создаем функциональный блок, реализующий интерфейс **VisuElems.VisuElemBase.IDialogOpenedListener**. Назовем его **VisuCallbackOpenDialog**.

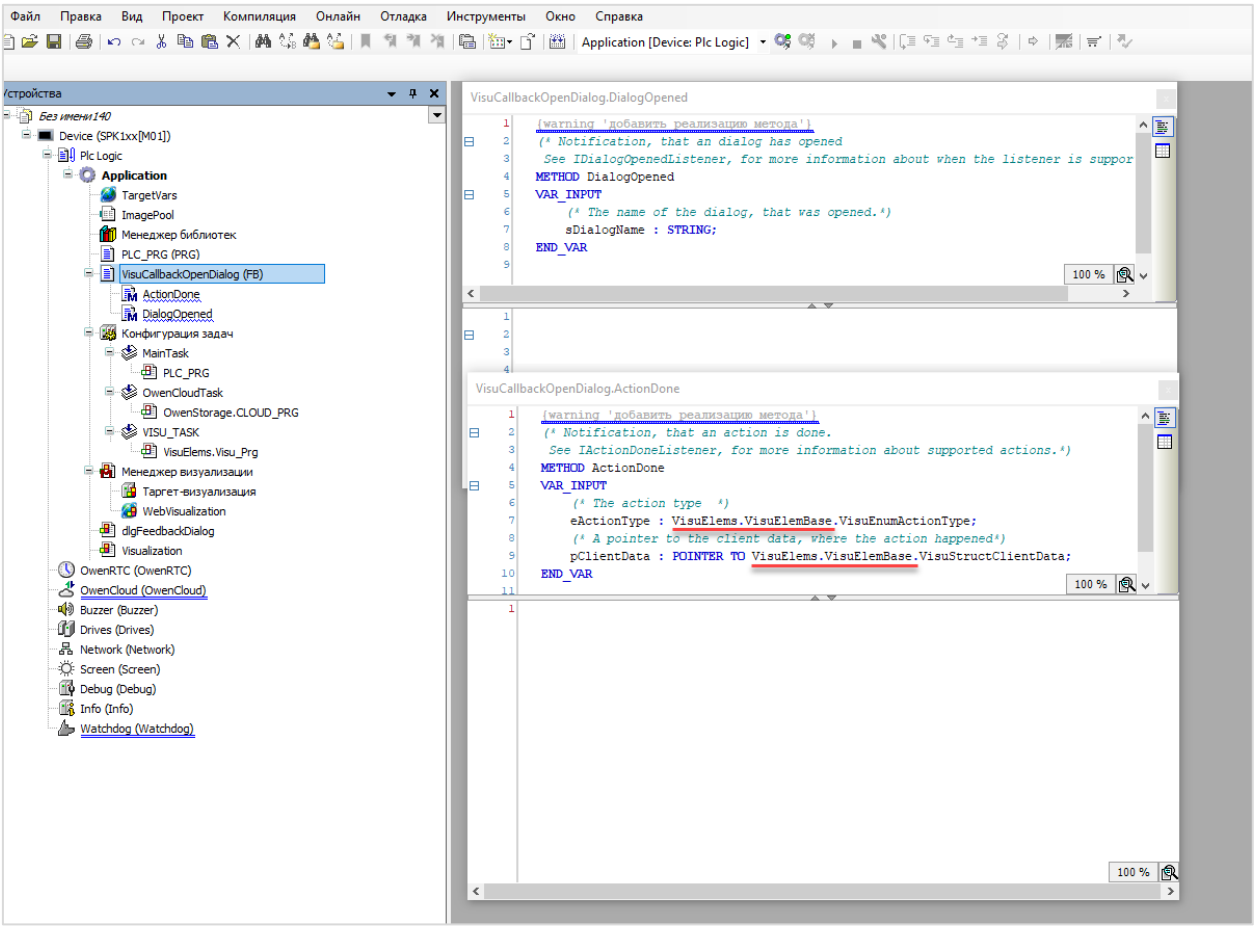


Рисунок 1.4.3.1 – Методы, полученные от интерфейса **itfDialogOpenedListener**

Блок автоматически получит от интерфейса **itfDialogOpenedListener** два метода – **DialogOpened** и **ActionDone**. Если у вас возникнут ошибки компиляции – то допишите к типам переменных **eActionType** и **pClientData** пространство имен **VisuElems.VisuElemsBase** (так называется библиотека, в состав которой входят эти типы). Обратите внимание на троллинг от разработчиков CODESYS – комментарии к методам содержат ссылки («see IDialogOpenedListener for more information», «see IActionDoneListener for more information»), но в действительности эти интерфейсы являются скрытыми (по крайней мере, в версии библиотеки **4.3.0.0** и более ранних) – вы не увидите их в менеджере библиотек и, соответственно, не сможете прочитать их документацию.

Оба метода автоматически (по технологии callback) вызываются при открытии любого из диалогов проекта для каждого клиента, соответствующего фильтру **itfClientFilter** (судя по документации, отображаемой в их заголовке – после открытия диалога, но точной уверенности нет, равно как и нет информации о порядке их вызова). Аргументом метода **DialogOpened** является имя открытого диалога (**sDialogName**). Аргументами метода **ActionDone** являются тип действия **eActionType** (в версии **4.3.0.0** библиотеки **VisuElemBase** описаны только два действия – **Undefined** и **OpenDialog**) и указатель на контекст клиента **pClientData** (контекст клиента мы уже обсуждали в п. 1.2).

Контекст позволяет определить, какой именно клиент визуализации открыл диалог (например, клиент таргет-визуализации или один из клиентов web-визуализации). Но эта информация доступна только в том случае, если открытие произошло путем нажатия на кнопку в визуализации. В нашем примере мы будем открывать диалог из кода программы – так что контекст клиента в данном случае будет отсутствовать и этот метод не будет нам интересен. Отмечу только, что, на мой взгляд, было бы более удобным наличием одного метода, объединяющего в себе все три входа.

Давайте реализуем простейшую обработку открытия нашего диалога **dlgFeedbackDialog**. Добавим счетчик открытия диалога. Воспользуемся способом, уже рассмотренном нами в п. 1.2.3 – будем возвращать значение счетчика через вход блока по указателю.

Добавим в ФБ **VisuCallbackOpenDialog** вход с типом «указатель на UINT»:

```
FUNCTION_BLOCK VisuCallbackOpenDialog IMPLEMENTS
   VisuElems.VisuElemBase.IDialogOpenedListener
VAR_INPUT
   puiFeedbackDialogOpenedCounter: POINTER TO UINT;
END_VAR
VAR_OUTPUT
END_VAR
VAR
END_VAR
```

В методе **DialogOpened** будем проверять вход **sDialogName** – и если будет открыт именно наш диалог, то мы инкрементируем значение счетчика, переданного на вход экземпляра блока по указателю.

```
(* Notification, that an dialog has opened
See IDialogOpenedListener, for more information about when the listener is
supported.*)
METHOD DialogOpened
VAR_INPUT
   (* The name of the dialog, that was opened.*)
   sDialogName   : STRING;
END_VAR


IF sDialogName = 'dlgFeedbackDialog' THEN

   puiFeedbackDialogOpenedCounter^ := puiFeedbackDialogOpenedCounter^ + 1;

END_IF
```

Еще раз уточню – метод будет вызываться для каждого клиента визуализации, соответствующего фильтру **itfClientFilter**. Далее в примере мы используем фильтр **VU.Globals.AllClients** – то есть после открытия диалога счетчик увеличится на текущее число клиентов, подключенных в визуализации.

Теперь объявим в нашей программе экземпляр блока **VisuCallbackOpenDialog** и переменную-счетчик. При инициализации переменных программы передадим на вход блок адрес переменной счетчика.

```
PROGRAM PLC_PRG
VAR
   fbOpenFeedbackDialog:             VU.FbOpenDialogExtended;
   xExecute:                         BOOL;

   stOpenFeedbackDialog:             dlgFeedbackDialog_VISU_STRUCT;
   stCloseFeedbackDialog:            dlgFeedbackDialog_VISU_STRUCT;

   fbVisuCallbackOpenDialog:         VisuCallbackOpenDialog;

   uiFeedbackDialogOpenedCounter:    UINT;

   xInit:                            BOOL;
END_VAR


IF NOT(xInit) THEN

   stOpenFeedbackDialog.wsMessage := "Какое-то сообщение";
   stOpenFeedbackDialog.usiQualityOfService := 3;

   fbVisuCallbackOpenDialog.puiFeedbackDialogOpenedCounter :=
      ADR(uiFeedbackDialogOpenedCounter);

   xInit := TRUE;

END_IF

// Вызов fbOpenFeedbackDialog добавим в п. 1.4.4
```

С обработкой открытия диалога разобрались. Теперь перейдем к обработке закрытия.

1.4.4. Обработка закрытия диалога (itfDialogCloseListener)

Как и в случае открытия диалога – обработка закрытия диалога производится по технологии обратного вызова (callback).

Документация на блок **FbOpenDialogExtended** (см. табл. 1.4.2) указывает, что на вход **itfDialogCloseListener** нужно передать экземпляр ФБ, реализующего интерфейс VisuElems.VisuElemBase.IDialogCloseListener. Но если мы хотим не просто контролировать факт закрытия диалога, но и получать результат его закрытия – то нужно использовать расширенную версию этого интерфейса (т. е. интерфейс-наследник) с названием IDialogCloseListenerWithTag.

Поэтому создаем ФБ **VisuCallbackCloseDialog**, реализующий (**IMPLEMENTS**) интерфейс **VisuElems.VisuElemBase.IDialogCloseListenerWithTag**.

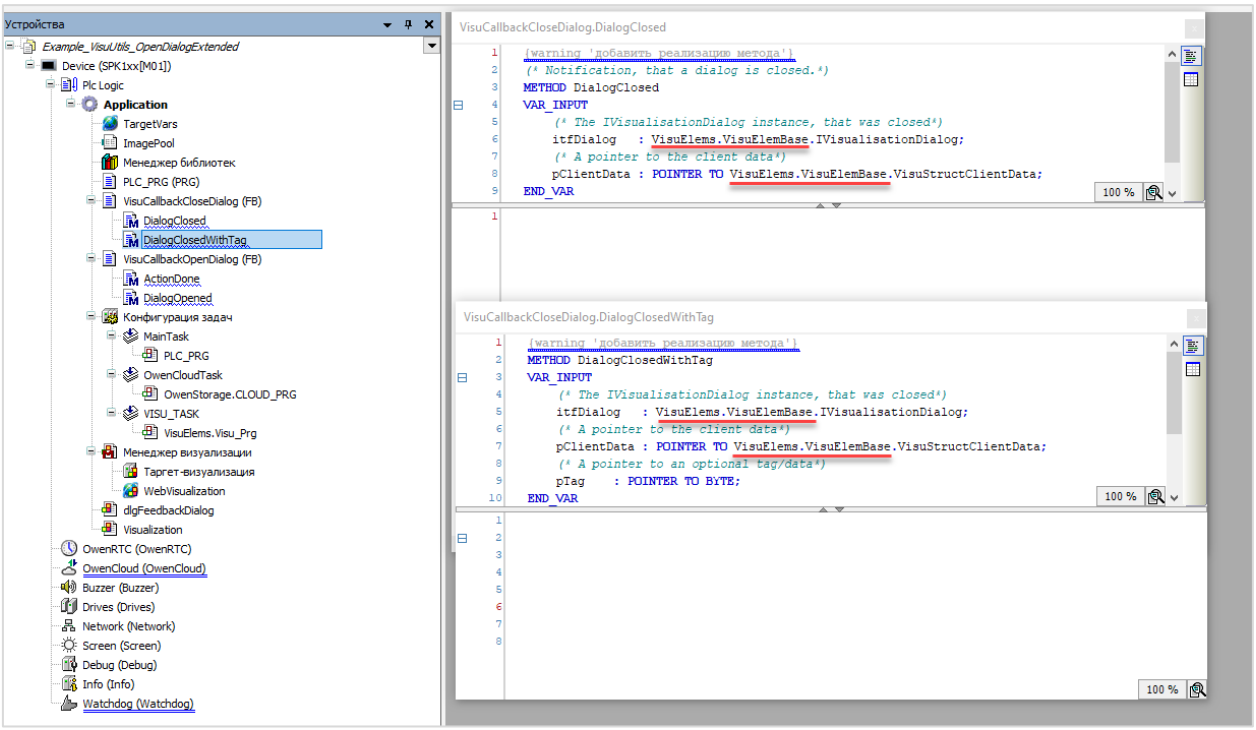


Рисунок 1.4.4.1 – Методы, полученные от интерфейса **IDialogCloseListenerWithTag**

Блок автоматически получит от интерфейса **IDialogCloseListenerWithTag** два метода – **DialogClosed** и **DialogClosedWithTag**. Если у вас возникнут ошибки компиляции – то допишите к типам переменных **itfDialog** и **pClientData** пространство имен **VisuElems.VisuElemsBase** (так называется библиотека, в состав которой входят эти типы).

Оба этих метода автоматически (по технологии callback) вызываются при закрытии любого из диалогов проекта. Общими аргументами методов являются экземпляр интерфейса закрытого диалога (**itfDialog**) и указатель на контекст клиента, который закрыл диалог (pClientData; мы уже обсуждали его в п. 1.2).

Контекст позволяет определить, какой именно клиент визуализации закрыл диалог (например, клиент таргет-визуализации или один из клиентов web-визуализации). Эта информация доступна только в том случае, если закрытие произошло путем нажатия на кнопку в визуализации (а, например, не через вызов экземпляра ФБ FbCloseDialog).

У метода **DialogClosedWithTag** есть также специальный вход **pTag** типа **POINTER TO BYTE**. На этот вход автоматически подставляется значение входа **pbyTagForDialogClosed** экземпляра блока **FbOpenDialogExtended** (я надеюсь, вы еще помните, что мы до сих пор рассматриваем именно этот блок?). Это позволяет при обработке закрытия диалога передать «наружу» (в нашем случае – в программу **PLC_PRG**) какую-то информацию – например, введенные в нем пользователем значения.

В рамках рассматриваемого примера реализуем следующую логику – если диалог закрыт нажатием кнопки **OK**, то будем считывать и возвращать в программу значения переменных диалога (т. е. переменных с рис. 1.4.2.1) – в частности, это позволит нам узнать выставленную пользователем оценку (**usiQualityOfService**).

```
METHOD DialogClosedWithTag
VAR_INPUT
   (* The IVisualisationDialog instance, that was closed*)
   itfDialog     : VisuElems.VisuElemBase.IVisualisationDialog;
   (* A pointer to the client data*)
   pClientData   : POINTER TO VisuElems.VisuElemBase.VisuStructClientData;
   (* A pointer to an optional tag/data*)
   pTag          : POINTER TO BYTE;
END_VAR
VAR
   pstFeedbackDialogData: POINTER TO dlgFeedbackDialog_VISU_STRUCT;
   stCloseFeedbackDialog: dlgFeedbackDialog_VISU_STRUCT;
END_VAR


IF itfDialog.GetName(FALSE) = 'dlgFeedbackDialog' AND
   itfDialog.GetResult() = VisuElems.VisuElemBase.Visu_DialogResult.OK THEN

   pstFeedbackDialogData := pTag;
   itfDialog.GetDialogInterface(ADR(stCloseFeedbackDialog));
   pstFeedbackDialogData^ := stCloseFeedbackDialog;

END_IF
```

В коде метода мы делаем именно то, что описали выше:

- проверяем, что был закрыт диалог с интересующим нас именем и что закрыт он был нажатием на кнопку с результатом **OK**;
- если проверка успешна – то инициализируем указатель на структуру переменных нашего диалога значением адреса, полученного на входе метода **pTag** (напомним, что это адрес, который мы передадим на вход **pbyTagForDialogClosed** экземпляра блока **FbOpenDialogExtended** – это случится уже на следующей странице);
- заполняем нашу структуру (объявленную в локальных переменных метода) текущими значениями переменных диалога;
- записываем эту структуру по инициализированному чуть выше указателю.

У вас, вероятно, уже возникли вопросы – что это за методы **itfDialog** мы используем (и какие они вообще есть), почему аргументом метода **GetName** является **FALSE** и т. д. Про это я расскажу в п. 1.4.6. Пока что сразу вернемся в программу, чтобы не потерять нить повествования.

Отмечу только, что при необходимости мы могли бы в методе **DialogClosedWithTag** использовать указатель на контекст клиента (**pClientData**) – например, если бы нам нужно было бы обрабатывать закрытие диалога только в том случае, если его закрыл клиент визуализации, принадлежащий определенной группе пользователей (например, “Admin”) или с определенным IP-адресом и т. д.

```
PROGRAM PLC_PRG
VAR
   fbOpenFeedbackDialog:             VU.FbOpenDialogExtended;
   xExecute:                         BOOL;

   stOpenFeedbackDialog:             dlgFeedbackDialog_VISU_STRUCT;
   stCloseFeedbackDialog:            dlgFeedbackDialog_VISU_STRUCT;

   fbVisuCallbackOpenDialog:         VisuCallbackOpenDialog;
   fbVisuCallbackCloseDialog:        VisuCallbackCloseDialog;

   uiFeedbackDialogOpenedCounter:    UINT;

   xInit:                            BOOL;
END_VAR


IF NOT(xInit) THEN

   stOpenFeedbackDialog.wsMessage := "Какое-то сообщение";
   stOpenFeedbackDialog.usiQualityOfService := 3;

   fbVisuCallbackOpenDialog.puiFeedbackDialogOpenedCounter :=
      ADR(uiFeedbackDialogOpenedCounter);

   xInit := TRUE;

END_IF


fbOpenFeedbackDialog
(
   xExecute                    := xExecute,
   itfClientFilter             := VU.Globals.AllClients,
   sDialogName                 := 'dlgFeedbackDialog',
   xModal                      := FALSE,
   pTopLeftPosition            := ,
   itfDialogOpenedListener      := fbVisuCallbackOpenDialog,
   pbyDialogInterfaceData       := ADR(stOpenFeedbackDialog),
   udiDialogInterfaceDataSize   := SIZEOF(stOpenFeedbackDialog),
   itfDialogCloseListener       := fbVisuCallbackCloseDialog,
   pbyTagForDialogClosed        := ADR(stCloseFeedbackDialog),
   dwDialogFlags                :=
);
```

Наконец-то мы добавили в программу вызов экземпляра блока, который рассматривали в этом пункте! Сразу объясню про «пустые» входы – **pTopLeftPosition** позволяет определить координаты открытия диалога (см. табл. 1.4.1). В рамках примера мне не кажется это интересным, так что я оставлю вход пустым – и диалог будет открыт по центру экрана. Что касается **dwDialogFlags** – то о них я немного расскажу в п. 1.4.7.

Давайте теперь более подробно обсудим, как работает наш код:

- по переднему фронту **xExecute** произойдет открытие диалога с именем **dlgFeedbackDialog**. Диалог будет открыт по центру экрана (см. комментарий чуть выше) и в немодальном режиме (так как **xModal** имеет значение **FALSE**) – то есть элементы экрана, поверх которого открыт диалог, останутся доступны для нажатия;
- при открытии диалога его переменным (см. рис. 1.4.2.1) будут присвоены значения из экземпляра структуры **stOpenFeedbackDialog** (см. входы **pbyDialogInterfaceData** и **udiDialogInterfaceDataSize**);
- при открытии диалога будет вызван экземпляр ФБ **VisuCallbackOpenDialog**, в котором вы сможете обработать открытие диалога (см. вход **itfDialogOpenedListener**). В нашем случае обработка сводиться к инкременту значения переменной **uiFeedbackDialogOpenedCounter**, адрес которой передается на вход экземпляра блока при его инициализации (в условии **IF NOT(xInit) THEN** …);
- при закрытии диалога будет вызван экземпляр ФБ **VisuCallbackCloseDialog**, в котором вы сможете обработать закрытие диалога (см. вход **itfDialogCloseListener**). В нашем случае обработка сводится к записи значений переменных диалога в экземпляр структуры **stCloseFeedbackDialog** (см. вход **pbyTagForDialogClosed**).

Теперь загрузим проект в контроллер и проверим его работу.

1.4.5. Проверка работы примера

Для начала присвоим переменной **xExecute** значение **TRUE** – это приведет к открытию диалога.

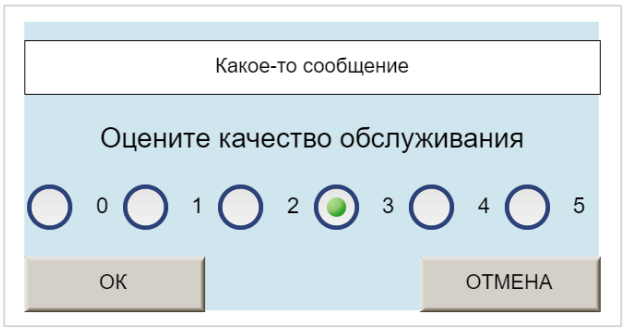
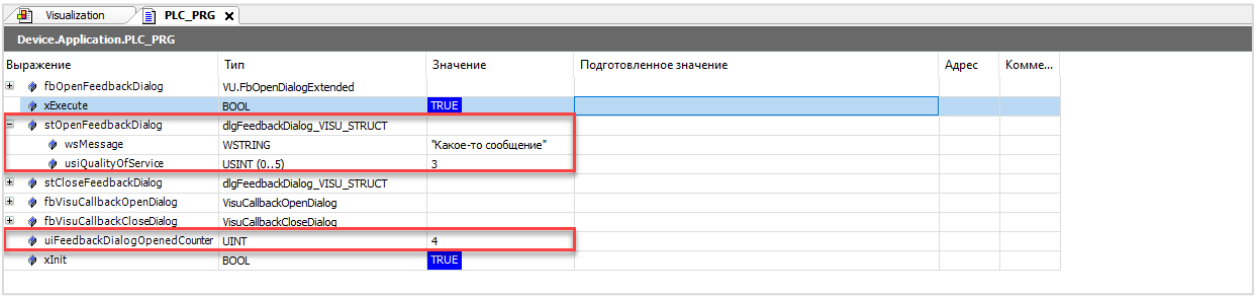


Рисунок 1.4.5.1 – Внешний вид диалога **dlgFeedbackDialog**, открытого в клиенте визуализации

Обратите внимание, что в прямоугольник и радио-кнопку при вызове были подставлены значения из экземпляра нашей структуры **stOpenFeedbackDialog**, а значение переменной **uiFeedbackDialogOpenedCounter** увеличилось – в моем случае на 4, потому что к моему контроллеру подключено 4 клиента визуализации. Так как мы используем фильтр **VU.Globals.AllClients**, то метод **DialogOpened** был вызван для каждого из 4-х клиентов и в каждом вызове значение переменной было увеличено на 1.

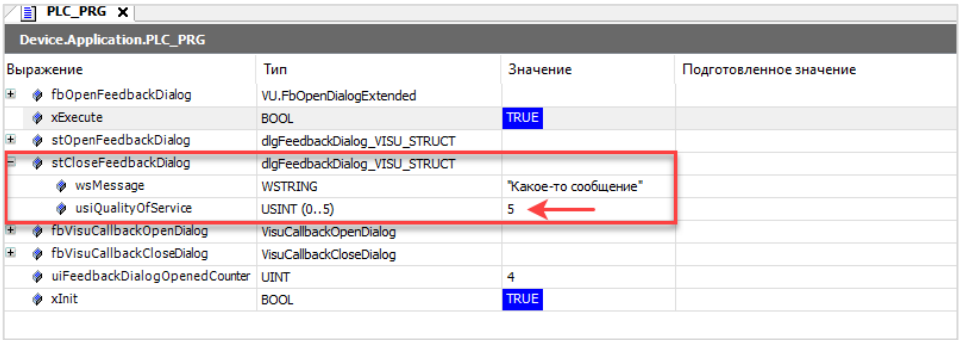

Visualization | PLC_PRG

Device.Application.PLC_PRG

Выражение	Тип	Значение	Подготовленное значение	Адрес	Комме...
fbOpenFeedbackDialog	VU.FbOpenDialogExtended				
xExecute	BOOL	TRUE			
stOpenFeedbackDialog	dlgFeedbackDialog_VISU_STRUCT				
wsMessage	WSTRING	"Какое-то сообщение"			
usiQualityOfService	USINT (0..5)	3			
stCloseFeedbackDialog	dlgFeedbackDialog_VISU_STRUCT				
fbVisuCallbackOpenDialog	VisuCallbackOpenDialog				
fbVisuCallbackCloseDialog	VisuCallbackCloseDialog				
uiFeedbackDialogOpenedCounter	UINT	4			
xInit	BOOL	TRUE			

Рисунок 1.4.5.2 – Экземпляр структуры со значениями, передаваемыми диалогу при его открытии, и переменная-счетчик открытия диалога

Выберите в диалоге качество обслуживания «5» и нажмите кнопку **OK**. Теперь мы увидим это значение в экземпляре структуры **stCloseFeedbackDialog**.


PLC_PRG

Device.Application.PLC_PRG

Выражение	Тип	Значение	Подготовленное значение
fbOpenFeedbackDialog	VU.FbOpenDialogExtended		
xExecute	BOOL	TRUE	
stOpenFeedbackDialog	dlgFeedbackDialog_VISU_STRUCT		
stCloseFeedbackDialog	dlgFeedbackDialog_VISU_STRUCT		
wsMessage	WSTRING	"Какое-то сообщение"	
usiQualityOfService	USINT (0..5)	5	
fbVisuCallbackOpenDialog	VisuCallbackOpenDialog		
fbVisuCallbackCloseDialog	VisuCallbackCloseDialog		
uiFeedbackDialogOpenedCounter	UINT	4	
xInit	BOOL	TRUE	

Рисунок 1.4.5.3 – Экземпляр структуры со значениями переменных диалога, переданных в программу при его закрытии

1.4.6. Интерфейс диалога (IVisualisationDialog)

В п. 1.4.4 в методе **DialogClosedWithTag** мы использовали экземпляр интерфейса диалога **IVisualisationDialog**. Это позволяло нам при закрытии диалога получить в коде метода доступ к информации о диалоге – например, его имени, значениям его переменных и т. д. Этот интерфейс является скрытым; его методы не описаны в пользовательской документации. Тем не менее, вы можете увидеть их список:

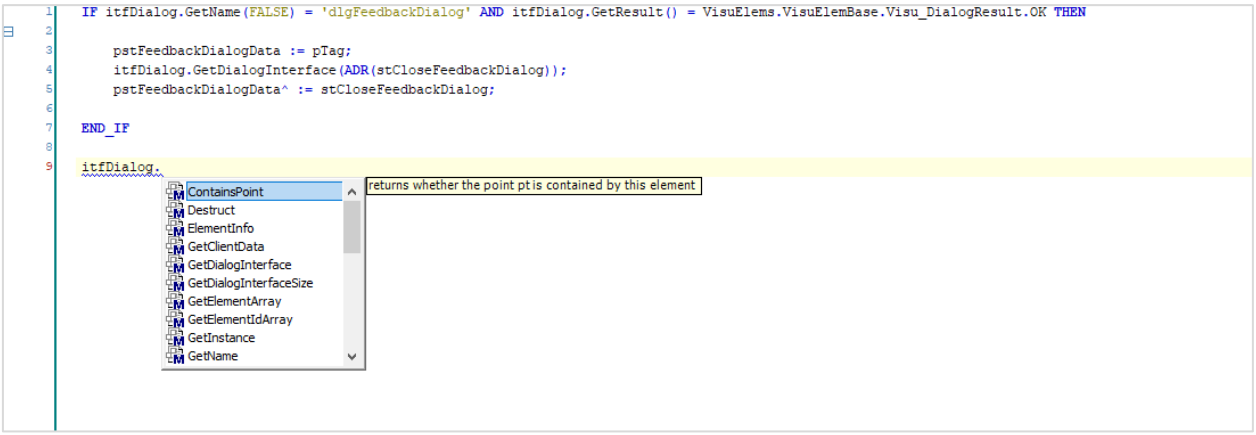


Рисунок 1.4.6.1 – Доступ к методам интерфейса диалога

Ниже я опишу некоторые из этих методов – те, которые понимаю и при этом считаю полезными для большинства задач.

Таблица 1.4.6 – Описание некоторых методов интерфейса **IVisualisationDialog**

Сигнатура	Описание
BOOL **GetDialogInterface**(POINTER TO DWORD pInterface)	Возвращает по указателю значения переменных диалога (т. е. экземпляр структуры типа <имя_диалога>_VISU_STRUCT. См. подробнее в п. 1.4.2 и 1.4.4
DINT **GetDialogInterfaceSize**()	Возвращает размер (в байтах) структуры переменных диалога
STRING **GetName**(BOOL bFullName)	Возвращает имя диалога. Если **bFullName** имеет значение **TRUE** – то имя включает в себя пространство имен (например, название библиотеки, если диалог входит в состав библиотеки)
POINTER TO STRING **GetNamespace**()	Возвращает указатель на пространство имен диалога (например, название библиотеки, если диалог входит в состав библиотеки)
VisuStructPoint **GetSize**()	Возвращает размер диалога в виде координат его правой нижней точки (т. е. точка отсчета – левая верхняя точка диалога)
Visu_DialogResult **GetResult**()	Возвращает результат закрытия диалога (см. информацию под таблицей)

Закрытие диалога выполняется либо нажатием на элемент визуализации с настроенным действием **Закрыть диалог**, либо с помощью экземпляра ФБ FbCloseDialog. В обоих случаях указывается результат закрытия диалога (для **FbCloseDialog** он задается на входе **dialogResult**):

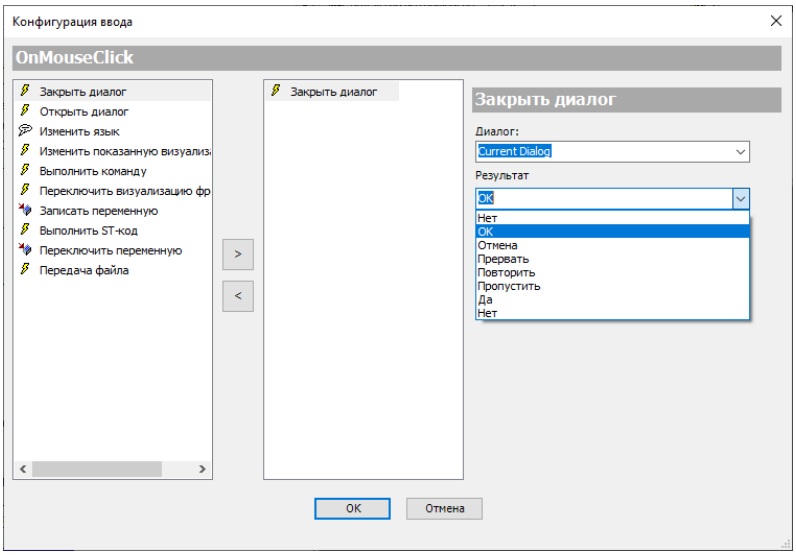


Рисунок 1.4.6.2 – Настройки действия **Закрыть диалог**

В настройках действия открытия диалога указываются результаты, при которых значения выходных переменных (**VAR_OUTPUT**) и входов-выходов (**VAR_IN_OUT**) диалога будут скопированы в привязанные к ним переменные программы.

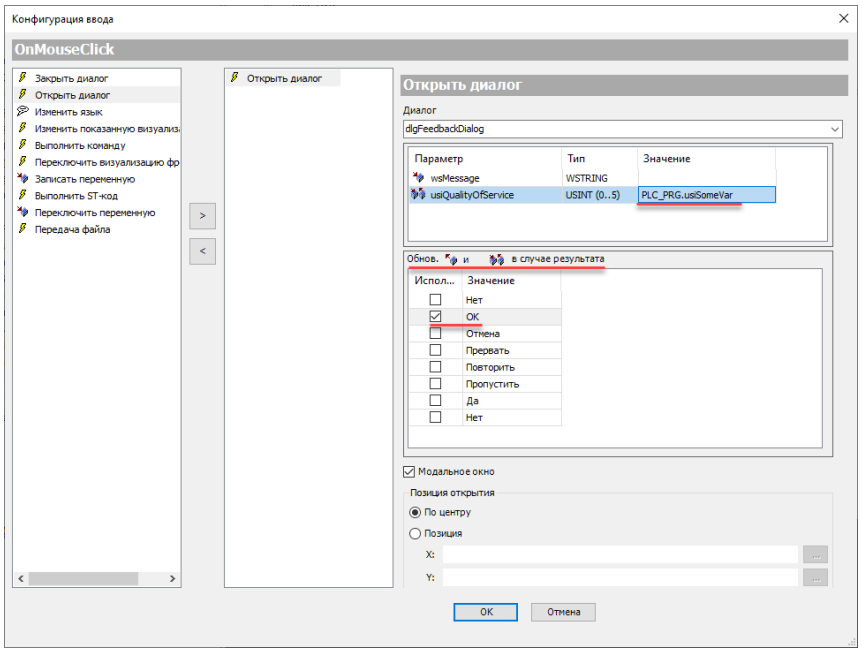


Рисунок 1.4.6.3 – Настройки действия **Открыть диалог**

В рамках примера мы открываем диалог из кода – так что никаких «привязанных переменных» и «ожидаемого результата» у нас нет. Вместо этого в методе **DialogClosedWithTag** мы используем метод интерфейса диалога **GetResult**, чтобы определить, нажатием на какую именно

кнопку (то есть на кнопку с каким настроенным результатом) был закрыт диалог и нужно ли нам что-то делать со значениями его переменных.

Метод возвращает значение перечисления **Visu_DialogResult**. Это перечисление входит в состав библиотеки **VisuElemBase** и, естественно, является скрытым. На это указывает **Marcel Prestel** (сотрудник **CODESYS Group**). Там же он приводит содержимое перечисление – оно совпадает с вариантами, доступными в действии **Закрыть диалог**:

```
TYPE Visu_DialogResult :
    (
        None ,
        OK ,
        Cancel ,
        Abort ,
        Retry ,
        Ignore ,
        Yes ,
        No
    );
END_TYPE
```

Рисунок 1.4.6.4 – Состав перечисления **Visu_DialogResult**

1.4.7. Флаги открытия диалога (dwDialogFlags)

Последнее, о чем я еще не упомянул, рассказывая о ФБ **FbOpenDialogExtended** – это о входе **dwDialogFlags**. Документация, как обычно, нас троллит – там указано, что это «дополнительные информационные флаги», описываемые неким элементом **Visu_InputFlags**. В баг-трекере есть предложение сделать его видимым для разработчиков с итоговым решением «не надо» (Won't Fix).

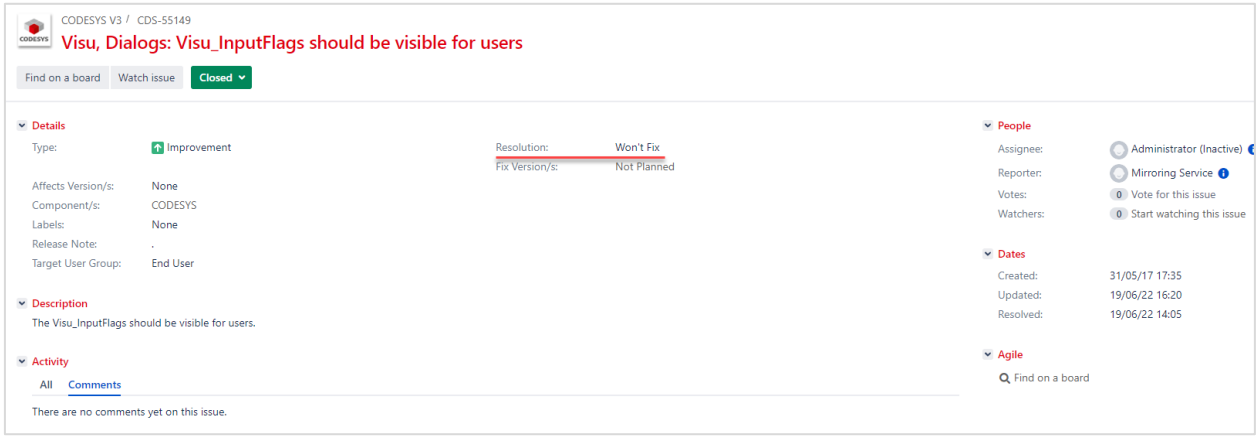


Рисунок 1.4.7.1 – Предложение по открытию для разработчиков объекта **Visu_InputFlags**

Поэтому перестаем надеяться на подсказки и просто пробуем объявить в программе объект такого типа (опытным путем можно выяснить, что он принадлежит библиотеке **VisuElemBase**):

```
VAR
    eVisuInputFlags:        VisuElems.VisuElemBase.Visu_InputFlags;
END_VAR
```

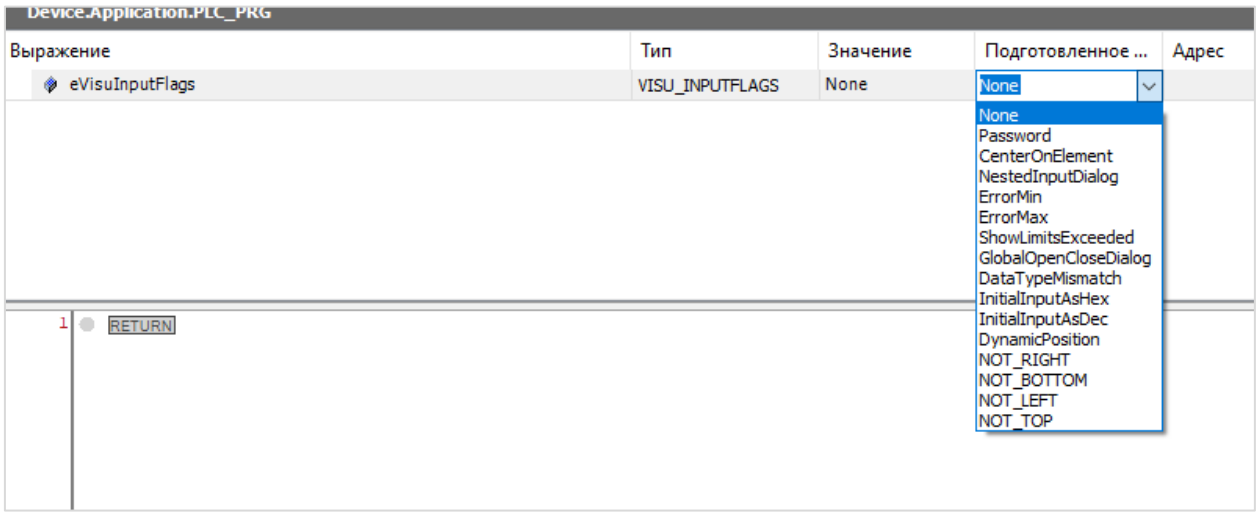


Рисунок 1.4.7.2 – Состав перечисления **Visu_InputFlags**

Итак, это перечисление. При этом каждый элемент перечисления соответствует биту маски (т. е. их значения – 0x01, 0x02, 0x04, 0x08, 0x10, 0x20 и т. д.) и их можно комбинировать:

```
dwDialogFlags := VisuElems.VisuElemBase.Visu_InputFlags.Password AND
   VisuElems.VisuElemBase.Visu_InputFlags.ShowLimitsExceeded;
```

Я не знаю, за что в точности отвечает каждый бит (и пока что не планировал пытаться выяснить это экспериментальным путем), но могу вам указать направление для исследований – это библиотека VisuDialogs. В ее диалогах и функциях используются переменные с названием **flags**. Если вы проанализируете фрагменты, в которых они используются – то сможете провести параллели с элементами перечисления (например, **16#0010** на скриншоте ниже – это не кто иной, как **ErrMax**).

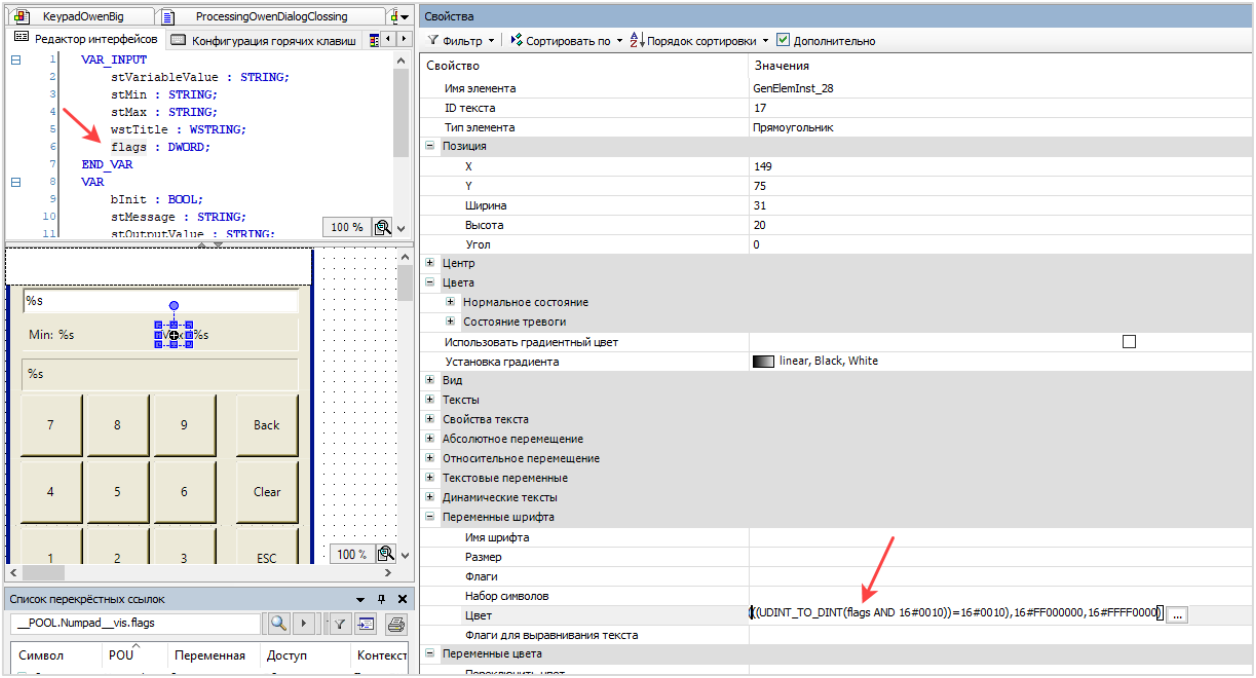


Рисунок 1.4.7.3 – Использование флагов диалогов в библиотеке **VisuDialogs**

Напоследок отмечу, что в документации CODESYS указано, что флаг **CenterOnElement** может использоваться только для текущего клиента визуализации (т. е. для **VU.Globals.CurrentClient**).

Если вы внимательно прочитали этот пункт до конца (и вспомнили, что уже где-то читали подобное начало фразы) – то, видимо, вы действительно интересуетесь нюансами визуализации CODESYS.

1.4.8. Диалоги ввода (Numpad, Keypad)

Стоит отметить, что ФБ **FbOpenDialog** и **FbOpenDialogExtended** не рассчитаны на открытие диалогов ввода (**Numpad** и **Keypad**) – при попытке открытия этих диалогов ничего не произойдет. На это указывает сотрудник **CODESYS Group** по имени **Marcel Prestel** на форуме CODESYS. В баг-трекере CODESYS есть несколько пожеланий (одно из них добавлено по результатам приведенной выше темы на форуме), связанных с поддержкой этого функционала в рамках библиотеки **Visu Utils**.

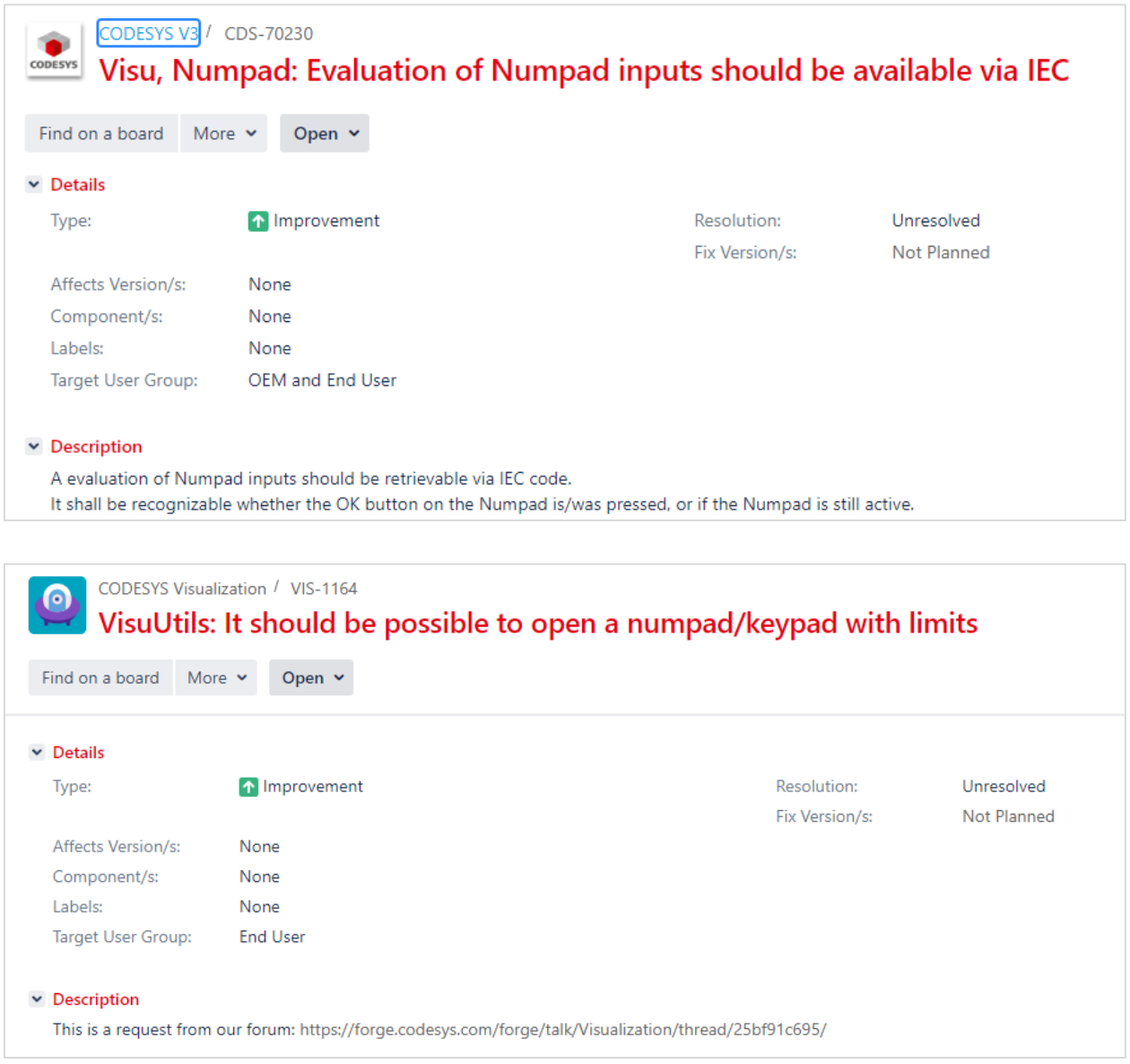


Рисунок 1.4.8.1 – Пожелания по возможности открытия диалогов Numpad/Keypad из кода программы в баг-трекере CODESYS

1.5. Закрытие диалога для клиентов визуализации (FbCloseDialog)

Функциональный блок **FbCloseDialog** предназначен для закрытия диалога для заданной группы клиентов. Как и остальные блоки библиотеки **Visu Utils** – данный блок соответствует модели поведения PLCopen Behaviour Model.

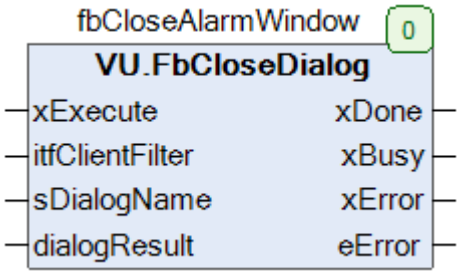


Рисунок 1.5.1 – Внешний вид ФБ **FbCloseDialog** на языке CFC

Таблица 1.5.1 – Описание входов и выходов ФБ **FbCloseDialog**

Название	Тип	Описание
Входы		
xExecute	BOOL	По переднему фронту происходит закрытие диалога для всех клиентов визуализации, соответствующих фильтру **itfClientFilter**
itfClientFilter	IVisualizationClientFilter	Экземпляр ФБ, реализующего интерфейс **IVisualizationClientFilter**. Используется для определения категории клиентов, для которых будет закрыт диалог. Пользователь может реализовать этот ФБ самостоятельно или воспользоваться одним из готовых, объявленных в списке глобальных переменных библиотеки (Globals)
sDialogName	STRING	Имя закрываемого диалога
dialogResult	VisuElems.VisuElemBase. Visu_DialogResult	Результат закрытия диалога. См. подробнее в п. 1.4.6
Выходы		
xDone	BOOL	Принимает значение **TRUE** после закрытия диалога
xBusy	BOOL	Имеет значение **TRUE**, пока блок находится в процессе работы
xError	BOOL	Принимает значение **TRUE** в случае возникновения ошибки в процессе работы блока
eError	VU.ERROR	Код ошибки

Принцип фильтрации клиентов мы уже рассмотрели в п. 1.3.

Что из себя представляет результат закрытия диалога – обсуждалось в п. 1.4.6.

Блок является довольно простым (особенно после того, как мы рассмотрели предыдущие блоки), так что приводить пример его использования я не буду – к этому моменту его создание должно уже быть для вас интуитивно понятным.

1.6. Передача файлов для клиентов визуализации (FBFileTransfer)

Функциональный блок FBFileTransfer предназначен для передачи данных между файловой системой контроллера и клиентом визуализации. Как и остальные блоки библиотеки **Visu Utils** – данный блок соответствует модели поведения PLCopen Behaviour Model.

Фактически блок представляет собой «программную» версию действия **Передача файла** из вкладки **Конфигурация ввода** (см. рис. 1.6.2). Для возможности использования действия и блока в конфигурационном файле контроллера должна быть разрешена передача файлов (для контроллеров ОВЕН она разрешена по умолчанию).

Как обычно, я сразу привожу ссылку на готовый пример: скачать

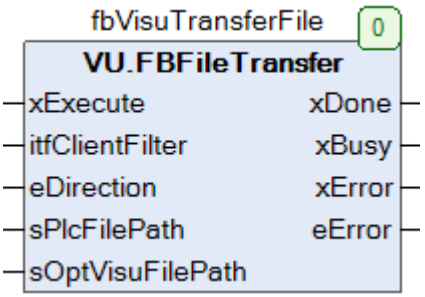


Рисунок 1.6.1 – Внешний вид ФБ **FBFileTransfer** на языке CFC

Таблица 1.6.1 – Описание входов и выходов ФБ **FBFileTransfer**

Название	Тип	Описание
Входы		
xExecute	BOOL	По переднему фронту происходит передача файла для всех клиентов визуализации, соответствующих фильтру **itfClientFilter**
itfClientFilter	IVisualizationClientFilter	Экземпляр ФБ, реализующего интерфейс **IVisualizationClientFilter**. Используется для определения категории клиентов, для которых будет передан файл. Пользователь может реализовать этот ФБ самостоятельно или воспользоваться одним из готовых, объявленных в списке глобальных переменных библиотеки (Globals)
eDirection	VisuElems. VisuElemBase. VisuEnumFile TransferDirection	Направление передачи файла. Возможные значения: **PLC_TO_VISU**, **VISU_TO_PLC**
sPlcFilePath	STRING(255)	Путь к файлу в файловой системе ПЛК. Начиная с определенной версии CODESYS (я не могу назвать ее точно) – абсолютные пути не подходят (в случае их

		использования возвращается ошибка **TRANSFER_FAILED**). Следует использовать файловые плейсхолдеры – см. информацию после таблицы
sOptVisuFilePath	STRING(255)	Согласно документации – это путь к файлу в клиенте визуализации, по которому должен быть сохранен файл, передаваемый из ПЛК (**eDirection = PLC_TO_VISU**). В случае пустой строки – в клиенте визуализации должно появится всплывающее окно для выбора пути к файлу. На практике у меня не получилось увидеть какого-либо результата (я задавал путь в файловой системе ПК, на котором был запущен клиент веб-визуализации)
Выходы		
xDone	BOOL	Принимает значение **TRUE** после завершения передачи файла
xBusy	BOOL	Имеет значение **TRUE**, пока блок находится в процессе работы
xError	BOOL	Принимает значение **TRUE** в случае возникновения ошибки в процессе работы блока
eError	VU.ERROR	Код ошибки

Вход **sPlcFilePath** определяет путь к файлу. В случае передачи файла из ПЛК в клиент визуализации (**PLC_TO_VISU**) – будет передан именно файл, размещенный по этому пути. В случае передачи файла из клиента в ПЛК (**VISU_TO_PLC**) в клиенте визуализации появится окно выбора файла в файловой системе клиента визуализации. После выбора файл будет передан в контроллер и сохранен по пути **sPlcFilePath**.

В современных версиях CODESYS в **sPlcFilePath** нельзя указывать абсолютный путь (например, **/root/CODESYS_WRK/PlcLogic/my_file.txt**). Вместо этого нужно указывать специальные заместители – файловые плейсхолдеры. Вот список стандартных файловых плейсхолдеров:

- **$$PlcLogic$$** – соответствует директории **PlcLogic**, которая содержит директории проекта пользователя (**Applicaton**, **visu** и т. д.);
- **$$visu$$** – директория сервера web-визуализации;
- **$$trend$$** – директория файлов трендов;
- **$$alarms$$** – директория файлов тревог;
- **$$TBF$$** – директория бэкапа (см. функцию Backup&Restore).

Т. е. в переменную **sPlcFilePath** нужно записывать, например, **‘$$PlcLogic$$/my_file.txt’**.

Производители контроллеров могут добавлять свои плейсхолдеры. Например, у контроллеров ОВЕН есть плейсхолдеры **$$USB$$** (корневая директория USB-накопителя, подключенного к контроллеру), **$$SD$$** (корневая директория SD-накопителя) и **$$FTP$$** (директория FTP-сервера контроллера).

Если вы используете контроллер, который подразумевает «ручное» редактирование конфиг-файла – то сможете добавить собственные плейсхолдеры. См. эту и эту статьи в онлайн-FAQ CODESYS.

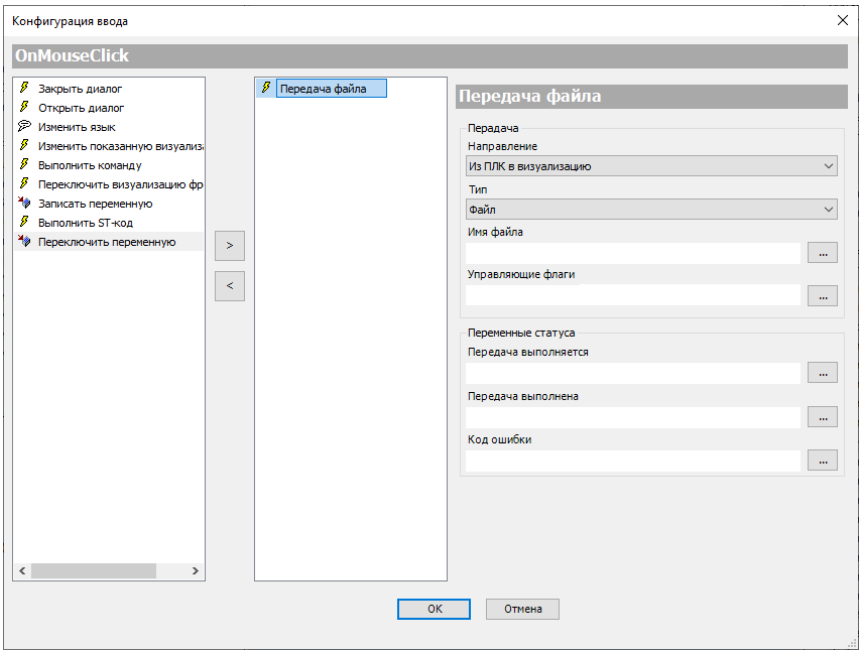


Рисунок 1.6.2 – Настройки действия **Передача файла**. ФБ **FBFileTransfer** является «программным» вариантом этого действия

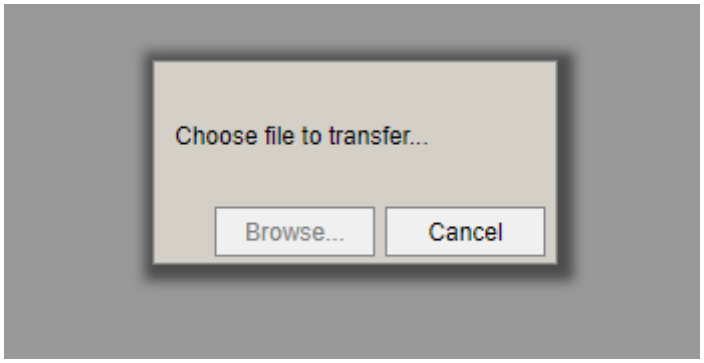


Рисунок 1.6.3 – Диалог выбора передаваемого в ПЛК файла в клиенте визуализации

Отмечу, что в менеджере библиотек у блока виден метод **OnCyclicActionDone**. Метод имеет спецификатор **Protected** – то есть доступен для вызова только внутри самой библиотеки. Вероятно, его просто забыли сделать скрытым.

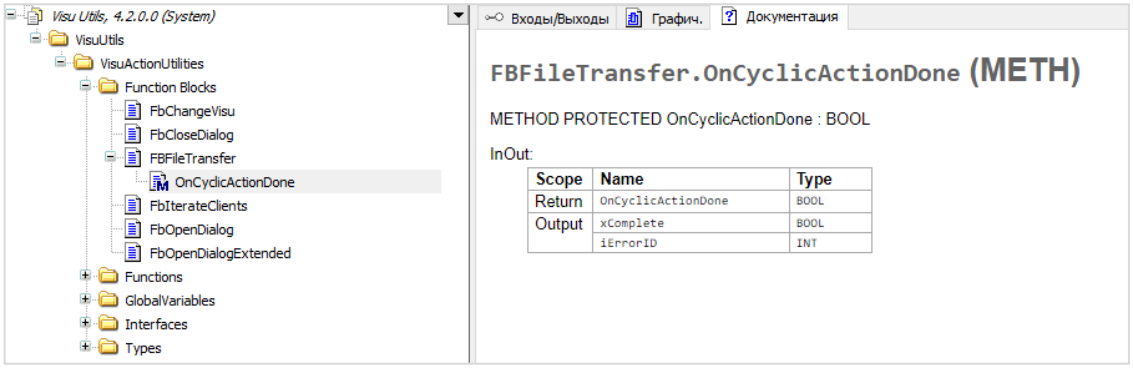


Рисунок 1.6.4 – Метод **OnCyclicActionDone** ФБ **FBFileTransfer**

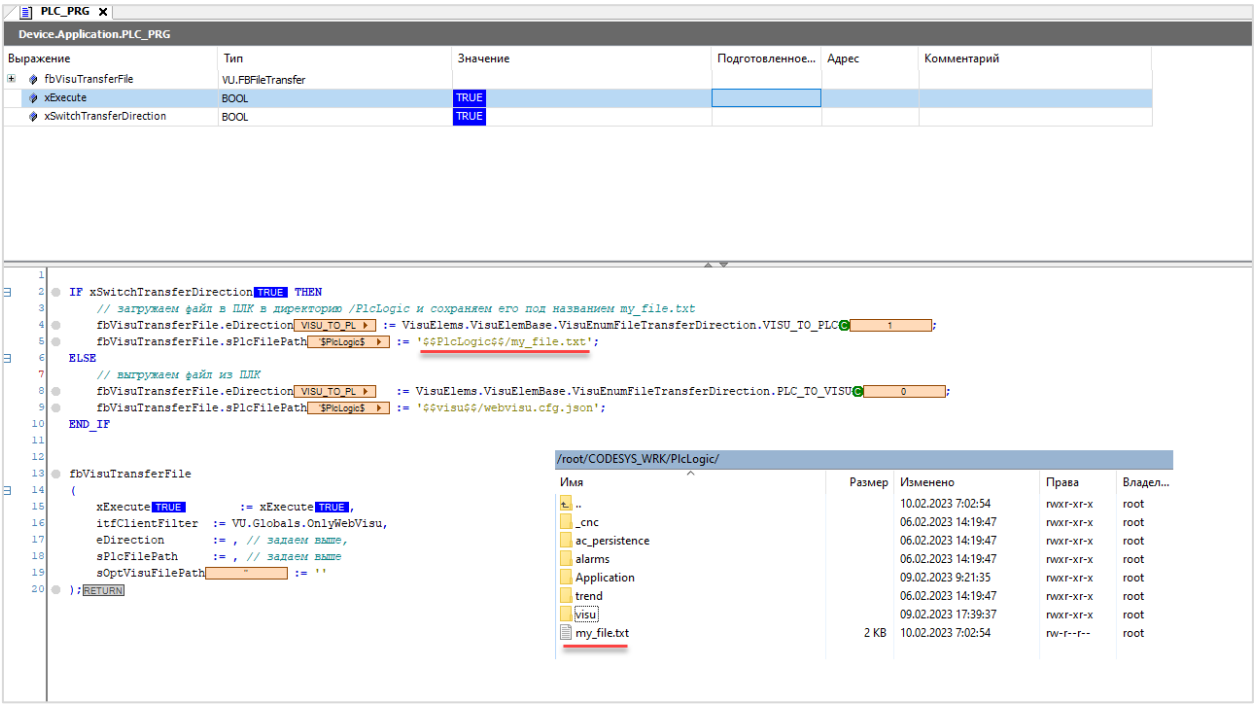


Рисунок 1.6.5 – Работа примера (для подключения к файловой системе контроллера использовалась утилита WinSCP)

1.7. Перспективы развития библиотеки

Разработчики CODESYS планируют расширять функционал библиотеки **Visu Utils** в следующих версиях среды. К уже зафиксированным[2] в баг-трекере задачам относятся:

- добавление функционала для управления пользователями визуализации из кода программы (см. рис. 1.3.6);
- добавление возможности открытия диалога ввода (Numpad/Keypad) из кода программы. См. рис. 1.4.8.1;
- добавления возможности определения в коде программы факта наличия в проекте визуализации и ее типа (таргет-визуализация, удаленная таргет-визуализация или web-визуализация);
- добавление возможности переключения между web-страницами (.htm) web-визуализации из кода программы;
- добавление набора синхронных функций в качестве альтернативы асинхронным ФБ библиотеки;
- добавление возможности отключения и включения web-сервера визуализации из кода программы;
- добавление возможности переключения типа ввода клиента визуализации из кода программы (виртуальная клавиатура или аппаратная клавиатура). Эта возможность будет добавлена в версии плагина визуализации **4.4.0.0**, выход которой запланирован на май 2023.

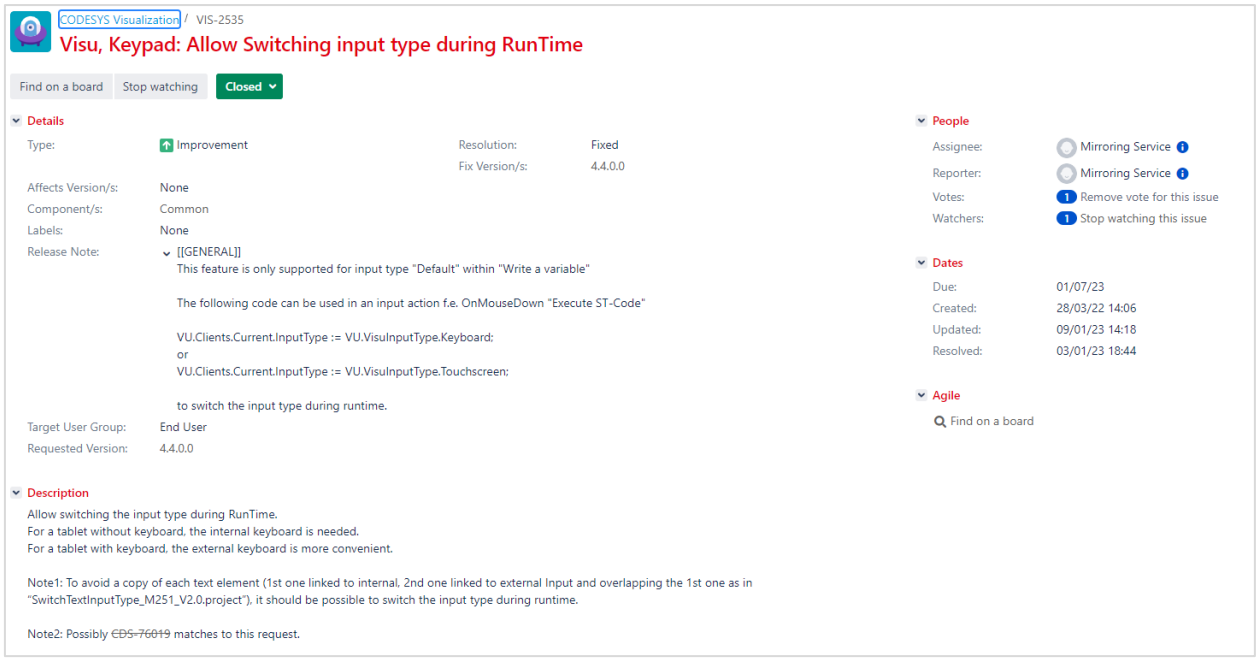


Рисунок 1.7.1 – Пожелание в баг-трекере CODESYS о добавлении в библиотеку **Visu Utils** возможности переключения типа ввода клиента визуализации

[2] Но надо учитывать, что не все зафиксированные пожелания в итоге оказываются реализованными.

2. Библиотека VisuUserManagement

2.1. Основная информация

Библиотека VisuUserManagement (также называемая **VisuUserMgmt**) реализует систему управления пользователями визуализации. Начиная с версии среды **CODESYS V3.5 SP18** и плагина **CODESYS Visualization 4.2.0.0** эта система объединена с системой пользователей контроллера (см. подробнее об этом здесь) – то есть для подключения к контроллеру и для авторизации в визуализации может использоваться один и тот же набор логинов/паролей. Новая объединенная система получила название «runtime-based user management». Прежний вариант отдельной системы пользователей визуализации все ещё доступен и называется «legacy user management».

Типовой подход к реализации проекта подразумевает, что действия пользователей (авторизация, выход из системы, изменение паролей и т. д.) производятся в визуализации путем нажатия кнопок с настроенным в конфигурации ввода действием **Управление пользователями**, в котором выбран соответствующий диалог:

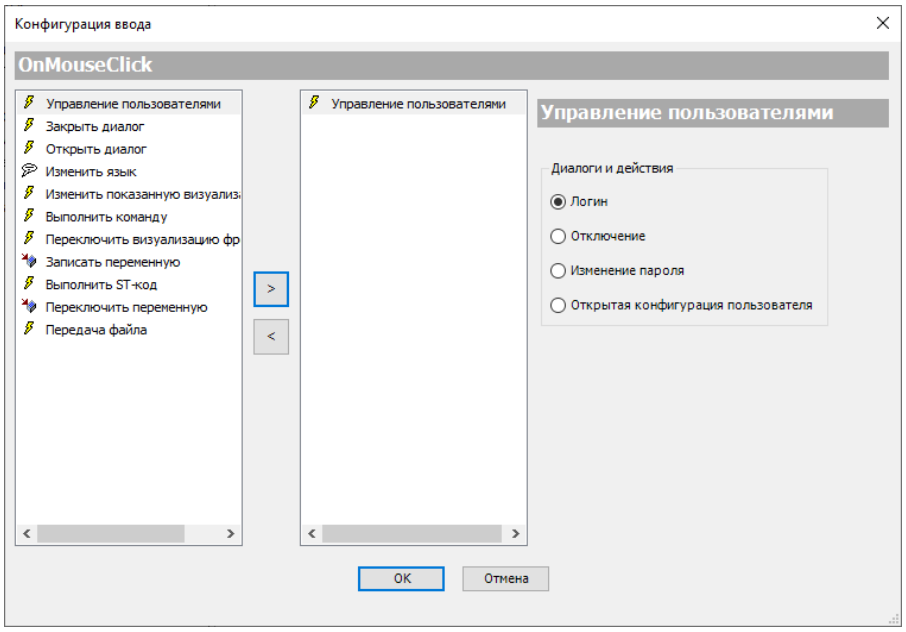


Рисунок 2.1.1 – Настройки действия управления пользователями в элементе визуализации

Но в ряде ситуаций такой подход оказывается неприемлем и требуется выполнение этих же операций в коде программы. Вот несколько примеров:

- контроллер получает информацию о входе в систему от считывателя смарт-карт и должен на ее основании авторизовать в системе соответствующего пользователя или организовать его выход из системы (логаут);
- требуется с заданной периодичностью (например, раз в месяц) автоматически менять пароли пользователей;
- требуется отображать в визуализации информацию о пользователях (их логины, группы и т. д.);
- требуется отображать в визуализации историю действия пользователей (авторизацию, выход из системы, изменение пароля и т. д.).

В подобных случаях разработчик может использовать библиотеку VisuUserManagement и ее вложенную библиотеку VisuUserMgmt3 Interfaces (строго говоря, для последней задачи упомянутой в списке задаче потребуется другая библиотека – об этом мы поговорим в п. 2.9).

Основным источником информации об использовании библиотеки является проект от разработчиков CODESYS. Он включает в себя три отдельных примера:

- **ApplicationRuntimeBased** – управление из кода «новой» (объединяющей пользователей контроллера и визуализации) системой пользователей;
- **ApplicationLegacyBased** – управление из кода «прежней» системой пользователей визуализации с использованием «нового» API (того же, что в **ApplicationRuntimeBased**);
- **Application_Old** – управление из кода «прежней» системой пользователей визуализации с использованием «старого» API.

Далее я рассмотрю несколько примеров использования «нового» API для «прежней» (legacy) системы пользователей визуализации.

В рамках примеров я буду использовать следующую конфигурацию пользователей:

Таблица 2.1.1 – Конфигурация пользователей, используемая в примерах

Группа пользователей	ID группы	Логин пользователя	Полное имя пользователя	Пароль	Комментарий
Admin	1	AdminUser	AdministratorName	1	С возможностью редактирования других пользователей и автоматическим выходом спустя 5 минут неактивности
Service	2	ServiceUser	ServiceName	2	
Operator	3	OperatorUser	OperatorName	3	

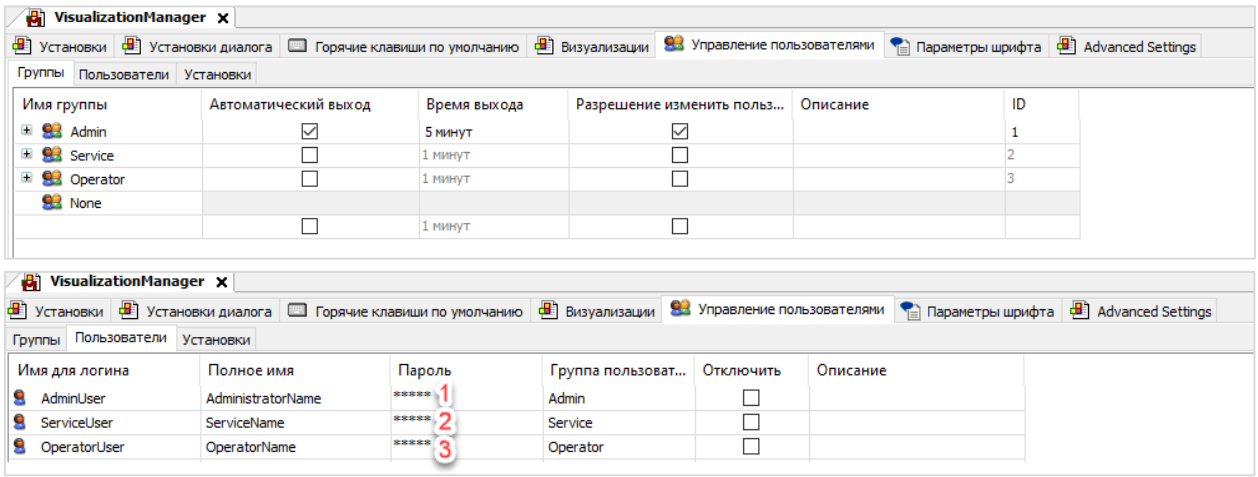


Рисунок 2.1.2 – Конфигурация пользователей, используемая в примерах

2.2. Структура и основные элементы библиотеки VisuUserManagement

Давайте перейдем в менеджер библиотек и посмотрим, что включает в себя библиотека **VisuUserManagement**:

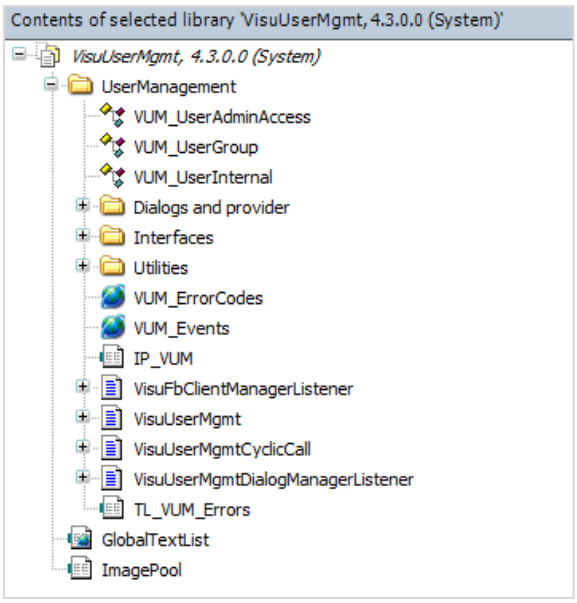


Рисунок 2.2.1 – Структура библиотеки **VisuUserManagement**

- перечисление VUM_UserAdminAccess определяет, есть ли у пользователя права администратора (**None** – нету, **FullAdmin** – полные права администратора, **RestrictAdmin** – некие «ограниченные» права администратора). Видимо, перечисление появилось вместе с «runtime-based user management» и описывает права пользователя контроллера. Используется всего в одном фрагменте библиотеки, и, на мой взгляд, не представляет особого интереса;
- структура VUM_UserGroup определяет параметры группы пользователей. Описание полей структуры приведено в табл. 2.2.1;
- структура VUM_UserIntermal определяет полный набор параметров пользователя, включая служебные, которые используются только внутри библиотеки (например, MD5-хеш пароля). Изначально вместо нее использовалась структура **VUM_User** из библиотеки VisuUserMgmt3_Interfaces (она является вложенной для **VisuUserManagement**), но потом разработчики поняли, что структура интерфейса не может быть расширена (т. е. в нее нельзя добавить новые поля в новой версии библиотеки без потери обратной совместимости) и создали отдельный вариант структуры для своих собственных нужд. Описание полей структуры приведено в табл. 2.2.2;
- папка Dialogs_and_provider включает в себя функции, ФБ, диалоги и фреймы, используемые для создания графического интерфейса управления пользователями (диалога логина, диалога изменения прав пользователей и т. д.). Документация по этому поводу практически отсутствует, так что рассказать о них особо нечего;

- папка Interfaces содержит интерфейсы (наборы прототипов методов и свойств), которые могут использоваться для работы с системой пользователей визуализации из кода программы;
- папка Utilities содержит функции и ФБ, используемые внутри библиотеки. Не представляет особого интереса для разработчика;
- список глобальных констант VUM_ErrorCodes содержит коды ошибок, которые могут возникнуть при работе с пользователями. По названию ошибки примерно понятен ее смысл – например, **ERR_VUM_FILE_NOT_FOUND** означает, что файл базы данных пользователей не найден, а **ERR_VUM_WRONG_PASSWORD** – что при авторизации был указан неверный пароль;
- список глобальных констант VUM_Events содержит идентификатор компонента сервера визуализации, который нужен библиотеке, чтобы подписаться на события этого компонента. Подробнее о работе с событиями в CODESYS я расскажу в отдельной статье, которую планирую опубликовать летом 2023 года;
- ФБ VisuFbClientManagerListener позволяет получить информацию о подключении и отключении клиентов визуализации. Блок реализует интерфейс **VisuElems.VisuElemBase.IClientManagerListener**. Выглядит интересно, но напрямую управления пользователями не касается. Давайте пока что просто запомним этот момент – мы рассмотрим его в п. 4;
- ФБ VisuUserMgmt является «сердцем» библиотеки – он реализует методы интерфейсов папки **Interfaces** (и не только их) и используется для работы с пользователями визуализации. Далее мы будем использовать объявленный внутри библиотеки глобальный экземпляр этого блока;
- ФБ VisuUserMgmtCyclicCall, судя по названию, что-то делает циклически. Нигде в документации и примерах он не упоминается – так что не представляет интереса;
- ФБ VisuUserMgmtDialogManagerListener позволяет получить информацию об открытии и закрытии диалога управления пользователями визуализации. В принципе, в п. 1.4 мы уже рассматривали, как получать информацию об открытии и закрытии диалогов, так что этот блок не представляет для нас особого интереса;
- списки текстов (**IP_VUM**, **TL_VUM_Errors**, **GlobalTextList**) и пул изображений (ImagePool) используются в диалогах управления пользователями. Не представляют интереса.

Таблица 2.2.1 – Описание полей структуры **VUM_UserGroup**

Поле	Тип	Описание
wstGroupName	WSTRING(79)	Имя группы пользователей
dwID	DWORD	ID группы пользователей
xAutoLogout	BOOL	**TRUE** – для группы задан автоматический выход из системы после истечения периода неактивности
tAutoLogout	TIME	Период неактивности для автоматического выхода
dwFlags	DWORD	Битовая маска флагов «специальных прав» пользователей. Как обычно – в документации эти флаги не описаны, приводится лишь один пример: USERGROUPFLAG_RIGHT_TO_CHANGE_USER

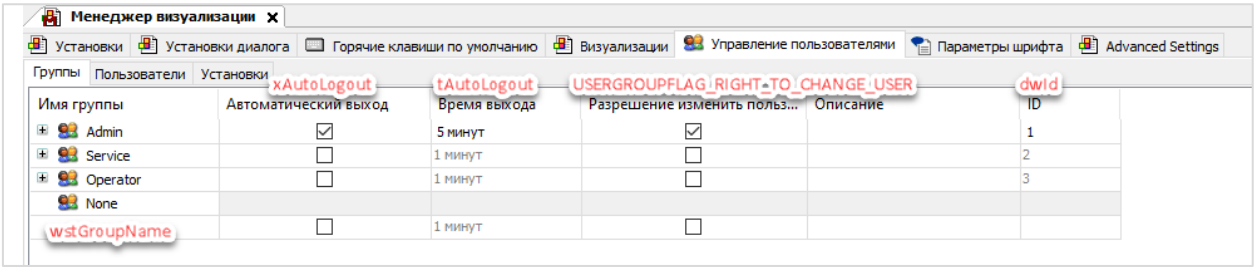


Рисунок 2.2.2 – Пояснения к полям структуры **VUM_UserGroup**

Таблица 2.2.2 – Описание полей структуры **VUM_UserInternal**

Поле	Тип	Описание
wstUserName	WSTRING(79)	Логин пользователя
wstFullName	WSTRING(79)	Полное имя пользователя
wstPassword	WSTRING(79)	Пароль
md5Hash	ARRAY [0..15] OF BYTE	Хеш пароля, рассчитанный по алгоритму MD5
dwUserGroupID	DWORD	ID «основной» группы пользователя
abyUserGroupIDs	см. информацию под таблицей	ID «дополнительных» групп, которым принадлежит пользователей
byUserGroupCount	BYTE	Число групп, которым принадлежит пользователь
bDeactivated	BOOL	**TRUE** – пользователь отключен
bChgPassFirstLogin	BOOL	**TRUE** – пароль пользователя должен быть изменен при первом входе в систему

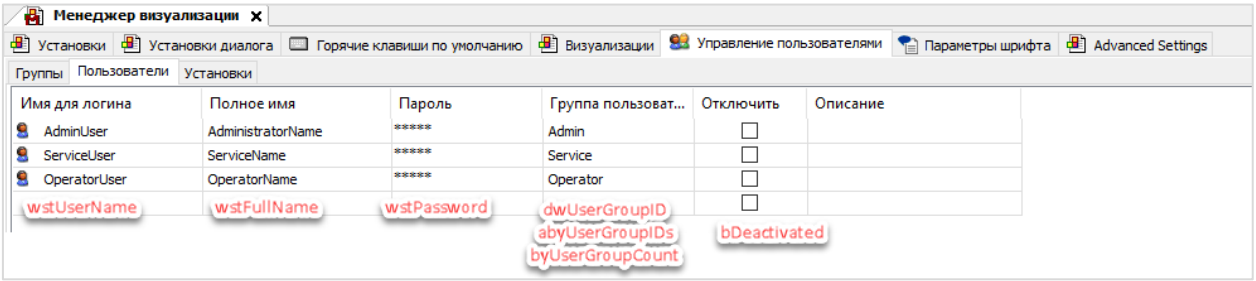


Рисунок 2.2.3 – Пояснения к полям структуры **VUM_UserInternal**

abyUserGroupIDs представляет собой массив:

ARRAY [0..VUM_Constants.VISU_VUM_MAX_GROUPS_PER_USER] OF BYTE

В этом массиве хранятся ID «дополнительных» групп, которым принадлежит пользователь – дело в том, что один пользователь может принадлежать сразу нескольким группам (например, **Admin** и **Operator**). Максимальное число таких групп зависит от константы **VISU_VUM_MAX_GROUPS_PER_USER** из списка глобальных констант **VUM_Constants** библиотеки **VisuUserMgmt3 Interfaces**, о которой мы еще поговорим далее. В текущих версиях CODESYS значение этой константы **10**. Кстати, заметим, что тип ID «основной» и «дополнительных» групп отличаются (**DWORD** и **BYTE**). Видимо, по ходу разработки было решено, что 255 групп пользователей должно быть достаточно для любого проекта.

Текущее число групп, которым принадлежит пользователь, записывается в поле **byUserGroupCount**.

Поле **bChgPassFirstLogin** определяет, должен ли пользователь изменить свой пароль при первом входе в систему. Это поле поддерживается только для runtime-based user management и соответствует галочке **Необходимо изменить пароль при первом логине**, которая присутствует в окне создания пользователя контроллера в узле **Device** на вкладке **Пользователи и группы**.

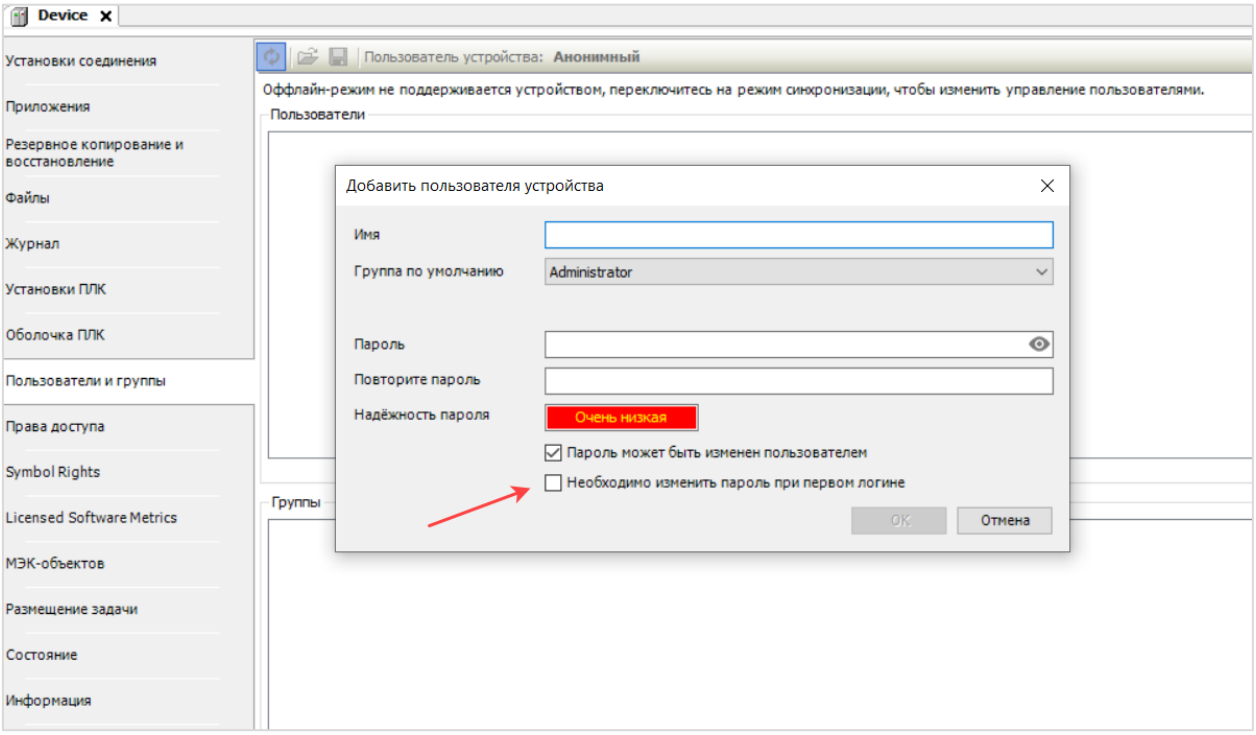


Рисунок 2.2.4 – Пояснения к полю **bChgPassFirstLogin**

2.3. Структура и основные элементы библиотеки VisuUserMgmt3 Interfaces

Библиотека **VisuUserManagement** имеет зависимость от интерфейсной библиотеки VisuUserMgmt3_Interfaces (номер в названии библиотеки периодически увеличивается). Эта библиотека содержит еще ряд структур, перечислений и интерфейсов, используемых библиотекой **VisuUserManagement**. Подразумевается, что в идеальном случае разработчик приложения должен использовать только эту библиотеку для работы с пользователями визуализации (т. е. библиотека **VisuUserManagement** является «системной», а **VisuUserMgmt3 Interfaces** – «пользовательской»).

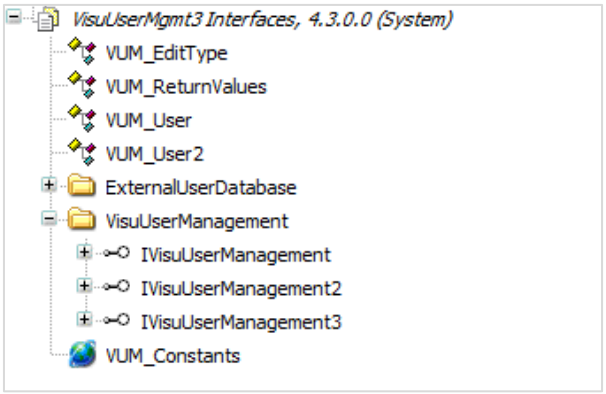


Рисунок 2.3.1 – Структура библиотеки **VisuUserMgmt3 Interfaces**

- перечисление VUM_EditType определяет операции, которые можно совершить с пользователем визуализации (например, **VUM_ADD** – добавление пользователя, **VUM_CHANGEPASSWORD** – изменение его пароля и т. д.);
- перечисление VUM_ReturnValues содержит коды ошибок, которые могут возникнуть при работе с пользователями. В значительной степени оно повторяет коды из списка глобальных констант VUM_ErrorCodes из библиотеки VisuUserManagement, но содержит больше кодов (например, **ERR_CHANGE_PASSWORD**, **ERR_VUM_EMPTY_USER_NAME_NOT_ALLOWED** и т. д.);
- структура VUM_User совпадает со структурой VUM_UserInternal из библиотеки **VisuUserManagement** с единственным отличием – в **VUM_User** отсутствует поле **bChgPassFirstLogin**;
- структура VUM_User2 представляет собой вариант структуры **VUM_User** с отсутствующими полями **wstPassword** и **md5Hash**;
- папка ExternalUserDatabase включает интерфейсы с прототипами методов, которые требуются, если разработчик контроллеров хочет реализовать собственную базу данных для хранения информации пользователей (если стандартная реализация его не устраивает);
- папка VisuUserManagement содержит интерфейсы (прототипы методов и свойств), которые могут использоваться для работы с системой пользователей визуализации из кода программы. Соответственно, это наиболее интересная для нас папка, и в основном мы будем рассматривать именно ее… У вас дежа-вю? Да, такая папка была и в библиотеке **VisuUserManagement**. Но там были интерфейсы с названиями **IVisuUserMgmt№,** а здесь – **IVisuUserManagement№;**

- список глобальных констант VUM_Constants включает константу **VISU_VUM_MAX_GROUPS_PER_USER**, определяющую максимальное число групп, которым может принадлежать пользователь. См. табл. 2.2.2 и информацию под таблицей.

2.4. Промежуточные итоги перед примерами

Выше мы рассмотрели состав библиотеки VisuUserManagement и состав библиотеки VisuUserMgmt3_Interfaces, от которой она является зависимой. Наибольший интерес вызывают методы интерфейсов **IVisuUserMgmt** и **IVisuUserManagement**, а также используемые ими структуры и перечисления. В следующих пунктах мы познакомимся с некоторыми из этих методов, решая конкретные задачи. Вот ссылка на полный проект, включающий все рассмотренные далее примеры: скачать

В проекте создано legacy user management (согласно рис. 2.1.2), а на экране визуализации добавлены кнопки логина/выхода из системы, а также индикатор и переключатель, видимые и активные только для пользователей группы **Admin**.

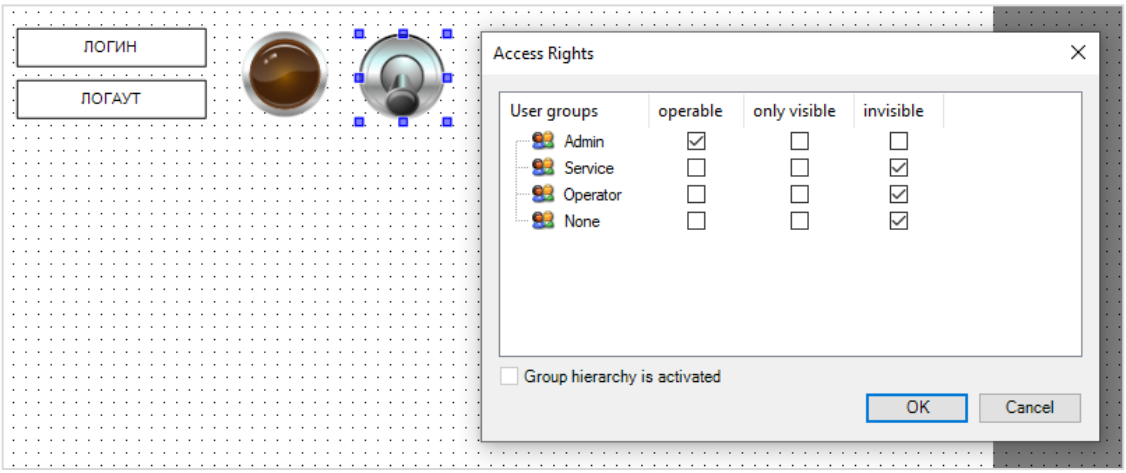


Рисунок 2.4.1 – Основная часть визуализации примера

Напоследок упомяну еще две библиотеки, связанные с управлением пользователями, которые не будут рассматриваться в данной статье:

- CmpUserMgr – эта библиотека отвечает за управление пользователями контроллера. Вместе с появлением runtime-based user management библиотека **VisuUserManagement** получила зависимость от библиотеки **CmpUserMgr**;
- **Visu User Mgmt Dialogs** – эта библиотека распространяется в исходниках и позволяет кастомизировать внешний вид диалогов управления пользователями (диалога логина, диалога изменения пароля и т. д.).

2.5. «Залогинивание», «разлогинивание» и изменение пароля пользователя

Начнем с рассмотрения трех основных операций, которые можно произвести с пользователями – авторизация в системе («залогинивание»), выход из системы («разлогинивание») и изменение пароля. Наша цель – осуществить всё это из кода. Давайте вернемся к библиотеке VisuUserMgmt3 Interfaces и изучим интерфейс IVisuUserManagement3 – мы найдем три метода с подходящими названиями (Login, Logout и ChangePassword).

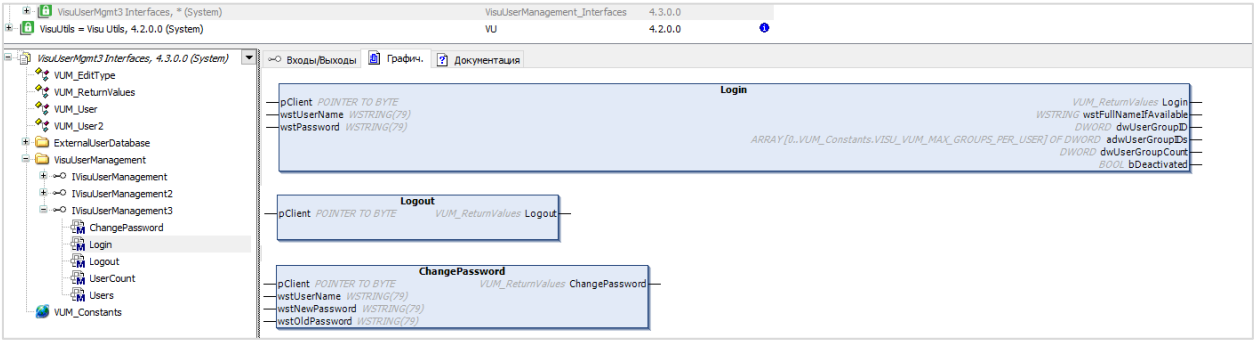


Рисунок 2.5.1 – Сигнатура методов интерфейса **IVisuUserManagement3**

У всех этих методов можно выделить две общие черты:

- одним из их аргументов является **pClient** – указатель на контекст клиента (если вы забыли, что это такое – то см. здесь);
- один из их возвращаемых значений является экземпляр перечисления VUM_ReturnValues, содержащий код ошибки или значение **ERR_OK**, если выполнение метода прошло без ошибок.

Метод **Login** помимо указателя на контекст клиента принимает на вход только его логин и пароль, а возвращает полное имя, информацию о группах, которым принадлежит пользователь, и флаг разрешения авторизации в системе.

Итак, нам потребуется контекст клиента. Поэтому берем пример из п. 1.2 и дополняем его следующим образом:

```
PROGRAM PLC_PRG
VAR
   fbGetVisuClientsInfo:          VU.FbIterateClients;
   xExecute:                      BOOL;
   fbClientIterationCallback:     VisuClientIteration;
   astVisuClientInfo:             ARRAY
       [1..Visu_Superglobal_Constants.VISU_MAX_NUMBER_OF_CLIENTS]
         OF VISU_CLIENT_DATA;
   i:                             INT;

   // Переменная переключателя и индикатора на экране визуализации
   xButton:                       BOOL;

   // Команда авторизации
   xLogin:                        BOOL;
   // Команда выхода из системы
   xLogout:                       BOOL;
   // Команда изменения пароля
   xChangePassword:               BOOL;

   // IP-адрес пользователя, для которого будет выполнена команда
   sUserIpAddr:                   STRING  := '10.2.8.133';

   // Логин пользователя
   wsUserName:                    WSTRING := "AdminUser";
   // Пароль пользователя
   wsUserPassword:                WSTRING := "1";
   // Устанавливаемый пароль (для команды xChangePassword)
   wsNewUserPassword:             WSTRING := "321";

   // Результат выполнения команды
   eVisuUserResult:                VisuUserManagement.VisuUserManagement_Interfaces.
                                      VUM_ReturnValues;
END_VAR
```

Область кода будет показана на следующей странице. Пока что обсудим новые переменные:

- **xButton** – эта переменная привязана к переключателю и индикатору на экране визуализации. Напомним, что переключатель и индикатор видимы и активны только для пользователя группы **Admin**. Переменная к ним привязана исключительно для интерактивности;
- **xLogin**, **xLogout**, **xChangePassword** – булевские команды для выполнения из кода соответствующих операций с клиентом визуализации;
- **sUserIpAddr** – IP-адрес компьютера, с которого я подключаюсь к web-визуализации. Таким образом, выполнение упомянутых выше операций будет происходить только для клиента, который работает с визуализацией именно на моем компьютере. В ваших проектах, соответственно, вы можете выбрать другой критерий для определения нужного вам клиента. См. также в п. 1.3 информацию о фильтрации клиентов;
- **wsUserName**, **wsUserPassword** – логин и пароль пользователя (см. рис. 2.2.3);
- **wsNewUserPassword** – новый пароль пользователя, устанавливаемый с помощью метода **ChangePassword**;
- **eVisuUserResult** – результат вызова методов работы с пользователями.

```
fbClientIterationCallback.pstVisuClientData := ADR(astVisuClientInfo);

fbGetVisuClientsInfo
(
   xExecute             := xExecute,
   // будем собирать информацию о всех клиентах визуализации
   itfClientFilter      := Vu.Globals.AllClients,
   // экземпляр ФБ, автоматически вызываемого для обработки клиентов
   itfIterationCallback := fbClientIterationCallback
);

// информация о всех интересующих клиентах получена
IF fbGetVisuClientsInfo.xDone THEN
   // и здесь можно что-то с ней сделать
END_IF

// Залогинивание
IF xLogin THEN

   FOR i := 1 TO Visu_Superglobal_Constants.VISU_MAX_NUMBER_OF_CLIENTS DO

      IF astVisuClientInfo[i].sIpAddr = sUserIpAddr THEN

         eVisuUseResult := VisuUserManagement.g_VisuUserMgmt.Login
           (astVisuClientInfo[i].pstClientData, wsUserName, wsUserPassword);

      END_IF
    END_FOR

    xLogin := FALSE;

END_IF

// Разлогинивание
IF xLogout THEN

   FOR i := 1 TO Visu_Superglobal_Constants.VISU_MAX_NUMBER_OF_CLIENTS DO

      IF astVisuClientInfo[i].sIpAddr = sUserIpAddr THEN

         eVisuUserResult := VisuUserManagement.g_VisuUserMgmt.Logout
           (astVisuClientInfo[i].pstClientData);

      END_IF
   END_FOR

   xLogout := FALSE;

END_IF

// Изменение пароля
IF xChangePassword THEN

   FOR i := 1 TO Visu_Superglobal_Constants.VISU_MAX_NUMBER_OF_CLIENTS DO

      IF astVisuClientInfo[i].sIpAddr = sUserIpAddr THEN

         eVisuUserResult := VisuUserManagement.g_VisuUserMgmt.ChangePassword
            (astVisuClientInfo[i].pstClientData, wsUserName, wsNewUserPassword,
            wsUserPassword);

      END_IF
   END_FOR

        xChangePassword := FALSE;

END_IF
```

Для всех трех команд используется один и тот же принцип:

- сначала мы проходимся по массиву информации о клиентах визуализации (**astVisuClientInfo**), определяя интересующего нас клиента (в моем случае это клиент с IP-адресом **sUserIpAddr**);
- найдя клиента – мы вызываем нужный нам метод, используя указатель на контекст клиента из массива, а также соответствующие переменные программы в качестве аргументов метода.

Для вызова метода мы обращаемся к **g_VisuUserMgmt** – это глобальный экземпляр ФБ **VisuUserMgmt**. В примерах от разработчиков CODESYS объявляется экземпляр какого-то из интерфейсов (например, в нашем случае в качестве него использовался бы **IVisuUserManagement3**), который инициализируется этим глобальным экземпляром, и далее вызов методов происходит через экземпляр интерфейса – но, на мой взгляд, можно обращаться к глобальному экземпляру напрямую. Особенно это удобно в том случае, если требуется вызывать методы разных интерфейсов – позволяет обойтись без дополнительных преобразований интерфейсов (__QUERYINTERFACE) в коде.

Справедливый вопрос, который вы можете задать – почему после вызова метода для найденного клиента не происходит выход из цикла с помощью оператора **EXIT**? Конечно, так можно сделать, но стоит учитывать, что, например, в рамках web-визуализации на одном ПК может быть открыто несколько вкладок, каждая из которых представляет собой отдельного web-клиента – и если сразу выйти из цикла, то метод будет вызван только для одного их этих клиентов. В рамках примера происходит проверка всего массива информации о клиентах – так что операция будет выполнена для всех web-клиентов моего ПК, подключенного к web-визуализации контроллера.

Давайте проверим пример в деле.

Для начала задайте переменной **sUserIpAddr** значение IP-адреса вашего ПК, загрузите проект примера в контроллер и запустите его. Перейдите в web-визуализацию – в данный момент она будет выглядеть следующим образом:

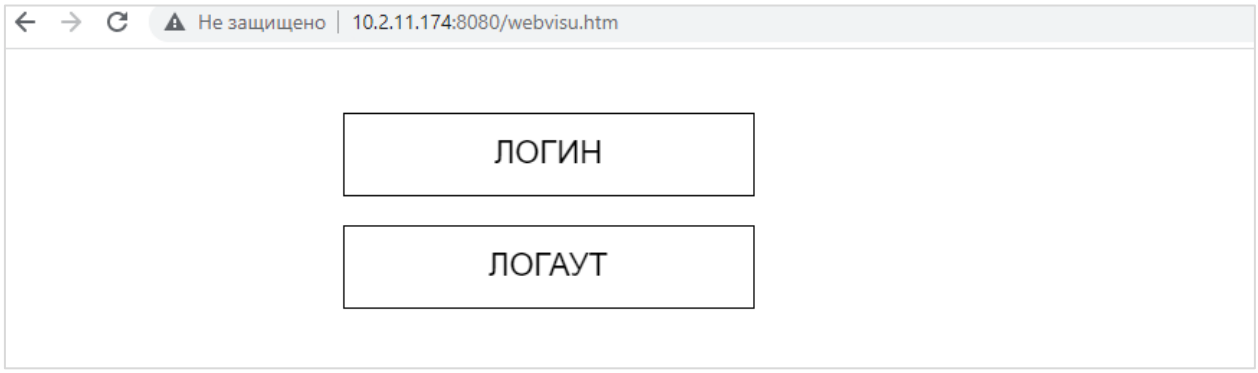


Рисунок 2.5.2 – Web-визуализация примера до авторизации пользователя **AdminUser**

Присвойте переменной **xExecute** значение **TRUE**, чтобы собрать информацию о клиентах. После этого присвойте переменной **xLogin** значение **TRUE** – в результате ваш клиент web-визуализации будет залогинен в системе как **AdminUser**.

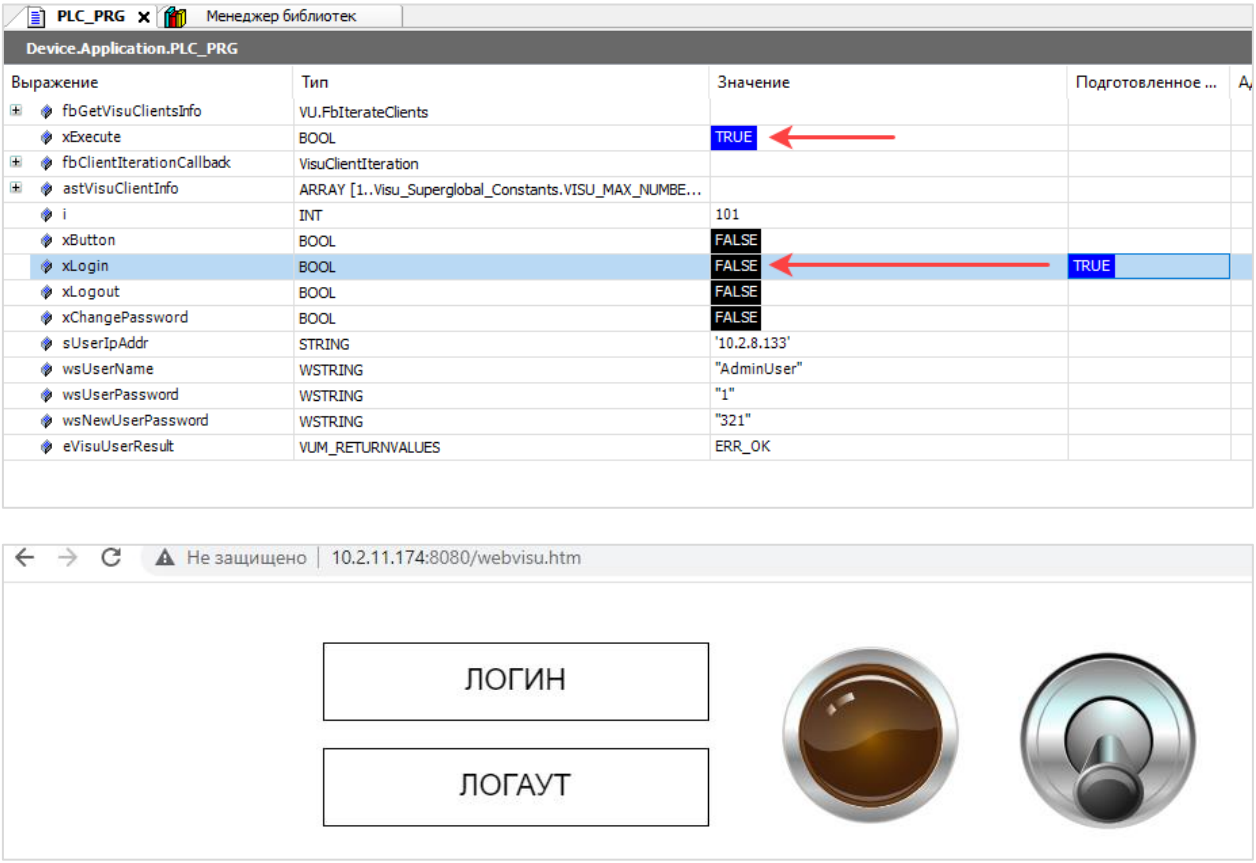


Рисунок 2.5.3 – Web-визуализация примера после авторизации пользователя **AdminUser**

Теперь присвойте переменной **xLogout** значение **TRUE**. Произойдет «разлогинивание» пользователя, и визуализация снова будет выглядеть как на рис. 2.5.2.

После этого присвойте переменной **xChangePassword** значение **TRUE**. Пароль пользователя **AdminUser** изменится на **321** (см. значение переменной **wsNewUserPassword**). Вы можете проверить это, использовав кнопку **ЛОГИН** для авторизации с новым паролем.

Справедливый вопрос – зачем в вызове метода **ChangePassword** передавать указатель на контекст клиента? Понятно, зачем это делать в методах **Login** и **Logout** – там мы работает с конкретным клиентом визуализации (в нашем случае – это клиент web-визуализации с известным нам IP-адресом). Выше мы видели, что **ChangePassword** успешно выполнился, даже когда мы были не залогинены. «Стоп» – скажете вы, – «Но ведь после логаута мы не обновили массив информации о клиентах. Может, поэтому метод и сработал?». И тут я должен напомнить, что информацию о клиентах мы собрали еще **до** вызова метода **Login**. В общем – в метод **ChangePassword** можно передать указатель на контекст любого клиента, и он успешно сработает. С другой стороны, требование передать контекст позволяет реализовать детектирование клиентов – например, вы можете вызывать этот метод только в том случае, если к визуализации подключился определенный клиент.

2.6. Сбор информации о клиентах

Теперь давайте получим в коде информацию о пользователях, созданных в управлении пользователями визуализации. Для этой операции в интерфейсе **IVisuUserManagement3** библиотеки **VisuUserMgmt3 Interfaces** доступны методы UserCount (определение числа пользователей) и Users (получение информации о пользователях).

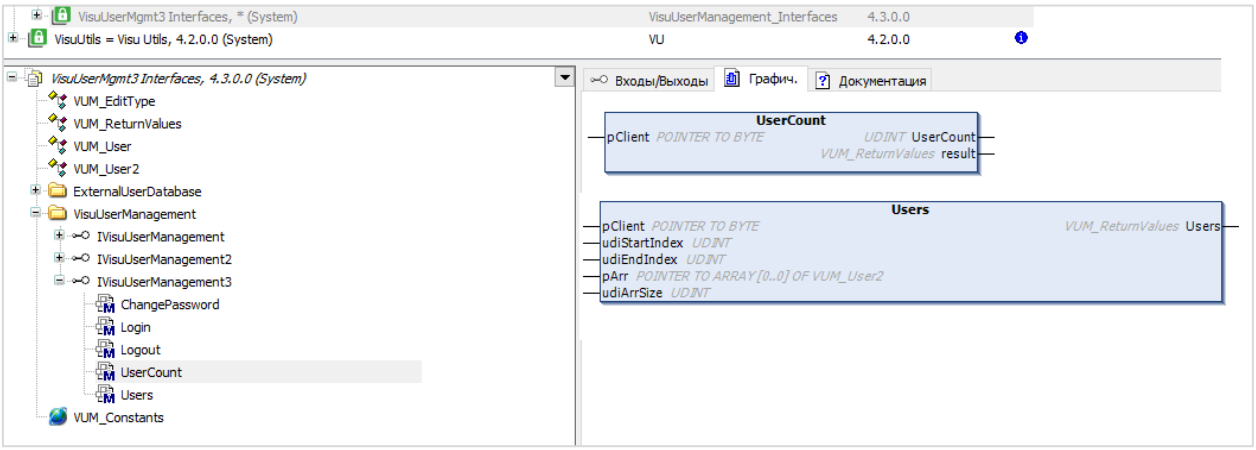


Рисунок 2.6.1 – Сигнатура методов **UserCount** и **Users**

Одним из аргументов обоих методов является указатель на контекст клиента. Как и в случае метода **ChangePassword** – в вызове методов можно передать контекст любого клиента. Метод **UserCount** возвращает число пользователей визуализации (обратите внимание, что фактически пользователей может быть больше, чем создано в менеджере визуализации на вкладке **Управление пользователями** – потому что они могут создаваться и удаляться в процессе работы приложения с помощью диалога управления пользователями (**Конфигурация ввода** – <событие> – **Управление пользователями** – **Открытая конфигурация пользователя**) или из кода программы и код ошибки (его тип – перечисление **VUM_ReturnValues**). Метод **Users** на выход возвращает только код ошибки.

Давайте подробнее обсудим входы метода **Users**:

- **pClient** – указатель на контекст клиента визуализации;
- **udiStartIndex** – номер начального пользователя, о котором будет получена **информация** (нумерация с нуля);
- **udiEndIndex** – номер конечного пользователя, о котором будет получена информация;
- **pArr** – указатель на массив структур VUM_User2, который после вызова метода будет заполнен информацией о пользователях;
- **udiArrSize** – число элементов массива, размещенного по указателю **pArr** (а не размер массива в байтах, как можно подумать по названию и описанию).

Меняя значения входов **udiStartIndex** и **udiEndIndex** перед новым вызовом метода – можно «порционно» считывать информацию о клиентах (например, по нажатию кнопок «вперед»/«назад» на экране, созданном для просмотра информации о пользователях).

Давайте доработаем код нашего примера из п. 2.5. Мы знаем, что в нашем проекте создано три пользователя (см. рис. 2.2.3) – но, как уже упоминалось, число пользователей может меняться прямо в процессе работы приложения. Поэтому давайте определимся, что в нашем проекте их будет не более **10** и объявим соответствующую константу.

```
PROGRAM PLC_PRG
VAR
   // объявление переменных из п. 2.5
   // ...

   // Команда сбора информации о пользователях
   xGetVisuUserInfo:              BOOL;
   // Массив информации о пользователях визуализации
   astVisuUserData:               ARRAY [1..c_udiMaxVisuUserCount] OF
         VisuUserManagement.VisuUserManagement_Interfaces.VUM_User2;
   // Количество пользователей визуализации
   udiVisuUserCurrentCount:       UDINT;
END_VAR
VAR CONSTANT
   // Максимальное ожидаемое количество пользователей визуализации
   c_udiMaxVisuUserCount:         UDINT := 10;
END_VAR



// код из п. 2.5
// ...

// Сбор информации о пользователях визуализации
IF xGetVisuUserInfo THEN

   FOR i := 1 TO Visu_Superglobal_Constants.VISU_MAX_NUMBER_OF_CLIENTS DO

      IF astVisuClientInfo[i].sIpAddr = sUserIpAddr THEN

         udiVisuUserCurrentCount := VisuUserManagement.g_VisuUserMgmt.UserCount
            (astVisuClientInfo[i].pstClientData, result => eVisuUserResult);
         udiVisuUserCurrentCount := MAX(1, udiVisuUserCurrentCount);

         eVisuUserResult := VisuUserManagement.g_VisuUserMgmt.Users
            (astVisuClientInfo[i].pstClientData, 0,  udiVisuUserCurrentCount - 1,
               ADR(astVisuUserData), c_udiMaxVisuUserCount );

      END_IF

   END_FOR

   xGetVisuUserInfo := FALSE;

END_IF
```

Мы действуем по уже известному нам из п. 2.5 принципу – проходимся по всем клиентам визуализации, находим среди них клиента с нужным нам IP-адресом и вызываем наши методы с использованием его контекста (напомню, что это, например, позволяет запретить сбор информации о пользователях, если к визуализации не подключен соответствующий клиент). В методе **Users** мы должны передать индекс первого и последнего из интересующих нас клиентов, при этом индексация ведется с нуля – поэтому если мы хотим получить информацию о всех клиентах, то должны в качестве конечного индекса передать число пользователей, полученное в вызове метода **UserCount**, и вычесть из него единицу. Это может привести к печальным последствиям, если по каким-то причинам метод **UserCount** вернет число пользователей **0** – тогда

мы вычтем из него 1, получим 4294967296 из-за эффекта переполнения, попытаемся собрать информацию о ≈4.3 млн. несуществующих пользователей, в процессе чего контроллер с подавляющей степенью вероятностью выпадет в исключение AccessViolation. Поэтому перед вызовом метода мы ограничиваем значение переменной **udiVisuUserCurrentCount** так, чтобы оно не могло быть меньше **1**.

В визуализации примера добавим на экран таблицу с информацией о пользователях. Я вывел только три поля из структуры VUM_User2 – **wstUserName** (имя для логина), **wstFullName** (полное имя) и **dwUserGroupID** (ID группы пользователей).

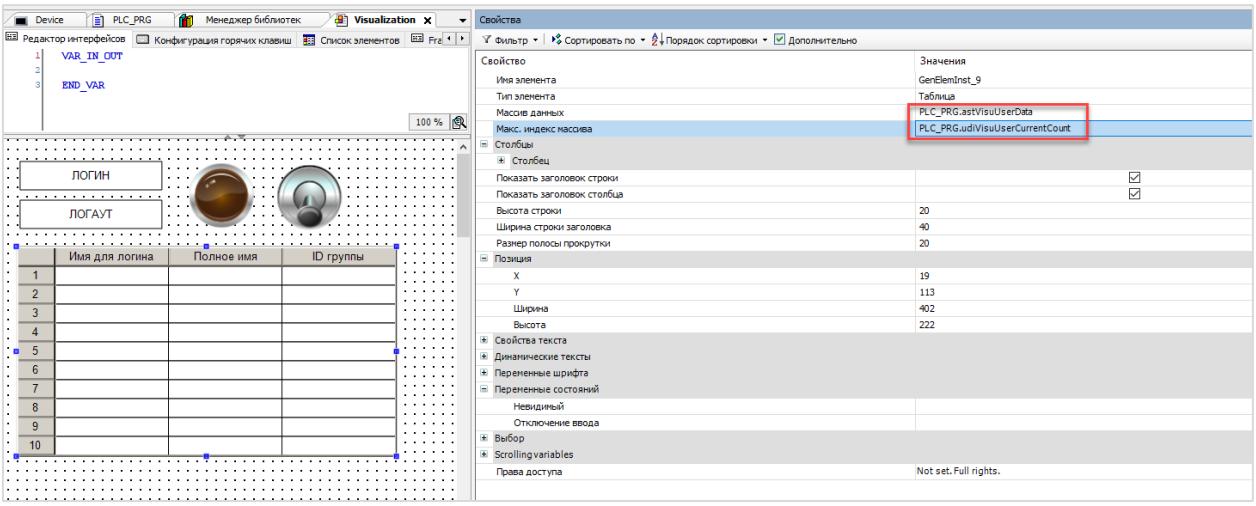


Рисунок 2.6.2 – Таблица с информацией о пользователях в визуализации примера

Загрузите проект примера в контроллер и запустите его.

Присвойте переменной **xExecute** значение **TRUE**, чтобы собрать информацию о клиентах. После этого присвойте переменной **xLogin** значение **TRUE** – в результате таблица будет заполнена информацией о пользователях. Сравните полученный результат с рис. 2.2.3 – мы получили именно ту информацию, которую задали на вкладке **Управление пользователями** в менеджере визуализации.

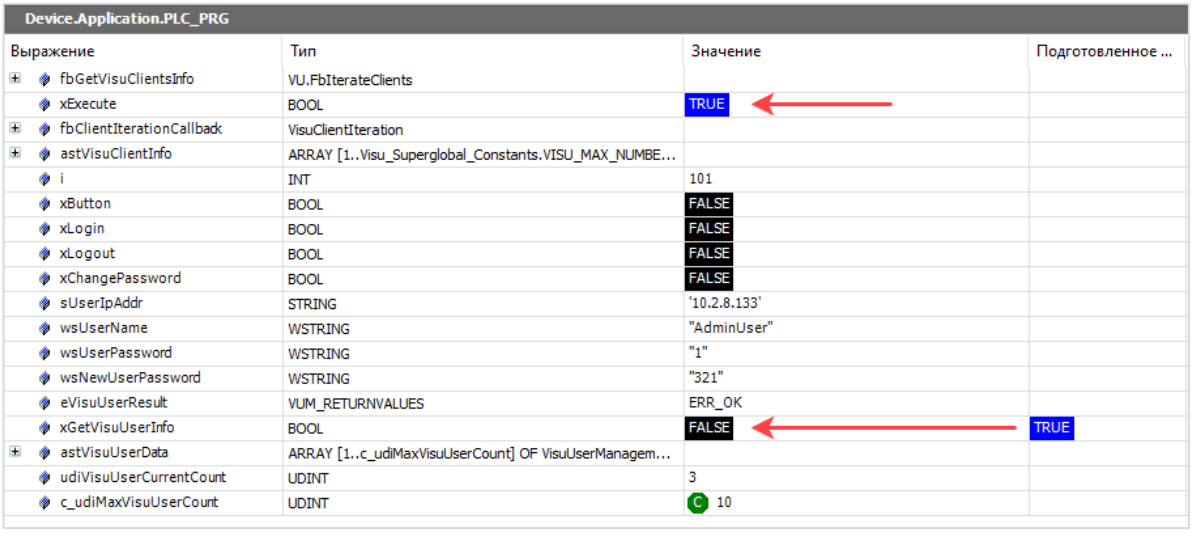


Рисунок 2.6.3 – Сбор информации о пользователях

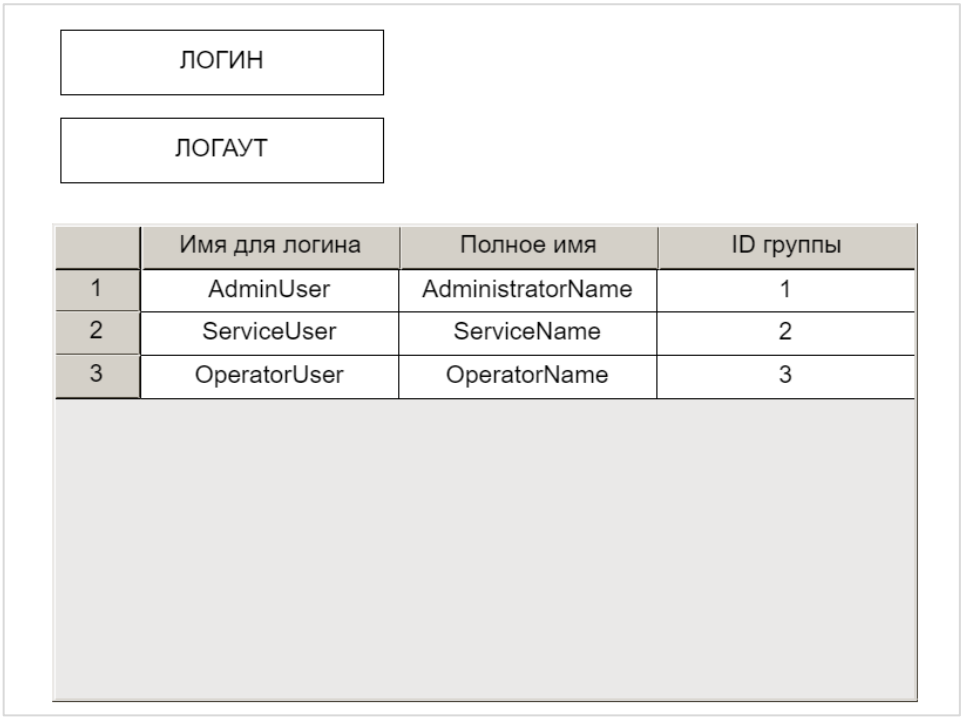


Рисунок 2.6.4 – Отображение информации о пользователях в визуализации примера

2.7. Добавление/удаление/изменение пользователей

В ряде случаев требуется добавлять и удалять пользователей визуализации из кода программы, а также изменять их данные (логины, принадлежность к группам и т. д). Например, это может быть необходимым, если вы синхронизируете профили пользователя контроллера с профилями какой-то другой информационной системы. Тогда эта система может передать в контроллер команду на выполнение соответствующей операции (например, создания пользователя), сопровождая ее нужной информацией (имя пользователя, его группа и т. д.).

В интерфейсах библиотеки VisuUserMgmt3 Interfaces нет подходящего метода[3]. Зато он есть в библиотеке VisuUserManagement – входит в состав интерфейса **IVisuUserMgmt4** под названием ChangeUser4:

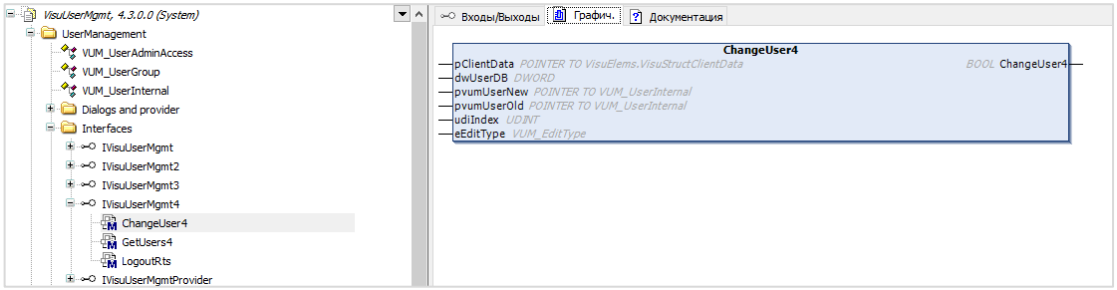


Рисунок 2.7.1 – Сигнатура метода **ChangeUser4**

Таблица 2.7.1 – Описание входов метода **ChangeUser4**

Название	Тип	Описание
pClientData	POINTER TO VisuElems.VisuElemBase.VisuStructClientData	Указатель на контекст клиента визуализации. Нужен только в случае использования runtime-based user management. Для выполнения команды **VUM_MODIFY** клиент должен быть авторизован как пользователь группы с правом «Разрешения изменения пользовательских данных» (см. **Менеджер визуализации – Управление пользователями – Группы**)
dwUserDB	DWORD	Идентификатор базы данных пользователей визуализации. Нужен только в случае использования legacy user management. Может быть получен с помощью вызова метода GetUserDBAsCopy
pvumUserNew	POINTER TO VUM_UserInternal	Структура данных пользователя визуализации
pvumUserOld		Указатель на структуру «текущих» данных пользователя визуализации. Нужна только для команды **VUM_MODIFY**
udiIndex	UDINT	Индекс пользователя в списке. Нужен только для команды **VUM_INSERT**
eEdtiType	VUM_EditType	Выполняемая команда (**VUM_ADD**, **VUM_INSERT**, **VUM_MODIFY** и **VUM_REMOVE**)

[3] Он появится в версии плагина визуализации **4.4.0.0**, выход которого запланирован на май 2023, будет размещен в интерфейсе **IVisuUserManagement4** и называться **EditUser**. Я решил не дожидаться выхода плагина (он может задержаться) и, кроме того, мне кажется уместным показать хотя бы один пример использования интерфейса библиотеки **VisuUserManagement**.

Метод возвращает **TRUE** после выполнения (что довольно странно; на мой взгляд, было бы логично, если бы он возвращал **UDINT** с кодом ошибки из списка глобальных констант VUM_ErrorCodes, как большинство остальных методов библиотеки **VisuUserMgmt**).

В рамках примера давайте реализуем добавление из кода программы нового пользователя визуализации с помощью команды **VUM_ADD**. Для этого доработаем код нашего примера:

```
PROGRAM PLC_PRG
VAR
   // объявление переменных из прошлых пунктов
   // ...
   // Команда добавления нового пользователя
   xAddVisuUser:                  BOOL;
   // Структура данных нового пользователя
   stAddUserInfo:                 VisuUserManagement.VUM_UserInternal;
   // Идентификатор базы данных пользователей
   dwUserDB:                      DWORD;
   // Экземпляр интерфейса для вызова методов работы с пользователями
   itfVisuUserMgmtIntern4:        VisuUserManagement.IVisuUserMgmt4;
END_VAR
VAR CONSTANT
   // Максимальное ожидаемое количество пользователей визуализации
   c_udiMaxVisuUserCount:         UDINT := 10;
END_VAR

// код из прошлых пунктов
// ...
// Добавление нового пользователя
IF xAddVisuUser THEN

   FOR i := 1 TO Visu_Superglobal_Constants.VISU_MAX_NUMBER_OF_CLIENTS DO

      IF astVisuClientInfo[i].sIpAddr = sUserIpAddr THEN

         dwUserDB := VisuUserManagement.g_VisuUserMgmtIntern.GetUserDBAsCopy();

        // Задаем данные нового пользователя
        stAddUserInfo.wstUserName   := "NewAdminUser";
        stAddUserInfo.wstFullName   := "NewAdministratorName";
        stAddUserInfo.dwUserGroupID := 1;
        stAddUserInfo.wstPassword   := "new";

        IF __QUERYINTERFACE(VisuUserManagement.g_VisuUserMgmtIntern,
              itfVisuUserMgmtIntern4) THEN

            itfVisuUserMgmtIntern4.ChangeUser4
            (
               // контекст клиента нужен только для runtime-based user management
               pClientData := 0,
               dwUserDB    := dwUserDB,
               pvumUserNew := ADR(stAddUserInfo),
               // данные о текущем пользователе; нужны только для команды VUM_MODIFY
               pvumUserOld := 0,
               // индекс пользователя в списке; нужен только для команды VUM_INSERT
               udiIndex    := 0,
               eEditType   := VisuUserManagement.VUM_EditType.VUM_ADD
            );
         END_IF

         itfVisuUserMgmtIntern4.SetNewUserDB(dwUserDB, TRUE);
         eVisuUserResult := itfVisuUserMgmtIntern4.GetLastError();
      END_IF
   END_FOR
   xAddVisuUser := FALSE;
END_IF
```

Мы действуем по уже привычному нам принципу – проходимся по всем клиентам визуализации и находим среди них клиента с нужным нам IP-адресом (напомню, что это, например, позволяет запретить добавление пользователей визуализации, если к визуализации не подключен соответствующий клиент). В рамках пример контекст этого клиента нам не нужен, так как мы используем legacy user management.

После нахождения ожидаемого нами клиента мы обращаемся к глобальному экземпляру **g_VisuUserMgmtIntern** и вызываем метод GetUserDBAsCopy для получения идентификатора базы данных пользователей – этот идентификатор потребуется нам для вызова метода ChangeUser4. Обратите внимание, что теперь мы обращаемся не к **g_VisuUserMgmt**, а к **g_VisuUserMgmtIntern**. Это связано с тем, что **g_VisuUserMgmt** реализует только интерфейсы библиотеки **VisuUserMgmt3 Interfaces**, а нам нужен метод интерфейса библиотеки **VisuUserManagement** – а их реализует как раз **g_VisuUserMgmtIntern**.

Далее заполняем структуру данных создаваемого нами пользователя – задаем ему имя (логин и полное имя), группу пользователей и пароль. В случае необходимости – вы можете заполнить и другие поля структуры VUM_UserInternal.

После этого можно вызвать метод **ChangeUser4**, но тут мы сталкиваемся с нюансом – **g_VisuUserMgmtIntern** не реализует интерфейс **IVisuUserMgmt4** (мне сложно сказать, с чем это связано; возможно, просто при обновлении библиотеки разработчики упустили этот момент). В принципе, мы могли бы использовать метод ChangeUser3 (из интерфейса IVisuUserMgmt3 – его **g_VisuUserMgmtIntern** реализует), но давайте все же, как и планировали, доберемся до метода ChangeUser4. Для этого в переменных программы мы заранее объявили экземпляр интерфейса **IVisuUserMgmt4** с названием **itfVisuUserMgmtIntern4** и теперь с помощью оператора __QUERYINTERFACE преобразуем **g_VisuUserMgmtIntern** в экземпляр нашего интерфейса. Мы уже использовали этот оператор в п. 1.2.3, помните?

Теперь мы можем обратиться к экземпляру нашего интерфейса и вызвать метод **ChangeUser4**. В нашем случае (для legacy user management и команды **VUM_ADD**) достаточно указать значения всего трех аргументов метода – **dwUserDB**, **pvumUserOld** и **eEditType**. Остальные входы можно оставить в значениях по умолчанию (для наглядности мы выше присвоили им **0**).

После вызова метода **ChangeUser4** нужно сохранить внесенные нами изменения в базу данных пользователя. Для этого следует вызвать метод SetNewUserDB. Первым аргументом метода является идентификатор базы данных пользователей, а второй (**bResultOK**) не документирован, но в примерах от разработчиков CODESYS для него всегда передается **TRUE**.

В конце мы вызываем метод GetLastError – это позволит нам определить, не случилось ли каких-нибудь ошибок во время вызовов предыдущих методов.

Давайте проверим наш пример на практике. Загрузите проект примера в контроллер и запустите его.

Присвойте переменной **xExecute** значение **TRUE**, чтобы собрать информацию о клиентах. После этого присвойте переменной **xAddVisuUser** значение **TRUE**. Теперь попробуйте войти в систему, нажав кнопку **ЛОГИН** и введя имя пользователя **NewAdminUser** и пароль **new**.

Device.Application.PLC_PRG

Выражение	Тип	Значение	Подготовленное ...
fbGetVisuClientsInfo	VU.FbIterateClients		
xExecute	BOOL	TRUE	
fbClientIterationCallback	VisuClientIteration		
astVisuClientInfo	ARRAY [1..Visu_Superglobal_Constants.VISU_MAX_NUMBE...		
i	INT	0	
xButton	BOOL	FALSE	
xLogin	BOOL	FALSE	
xLogout	BOOL	FALSE	
xChangePassword	BOOL	FALSE	
sUserIpAddr	STRING	'10.2.8.133'	
wsUserName	WSTRING	"AdminUser"	
wsUserPassword	WSTRING	"1"	
wsNewUserPassword	WSTRING	"321"	
eVisuUserResult	VUM_RETURNVALUES	ERR_OK	
xGetVisuUserInfo	BOOL	FALSE	
astVisuUserData	ARRAY [1..c_udiMaxVisuUserCount] OF VisuUserManagem...		
udiVisuUserCurrentCount	UDINT	0	
xAddVisuUser	BOOL	FALSE	TRUE
stAddUserInfo	VisuUserManagement.VUM_UserInternal		
dwUserDB	DWORD	0	
itfVisuUserMgmtIntem4	VisuUserManagement.IVisuUserMgmt4	16#00000000	
c_udiMaxVisuUserCount	UDINT	10	

Рисунок 2.7.2 – Выполнение команды создания нового пользователя

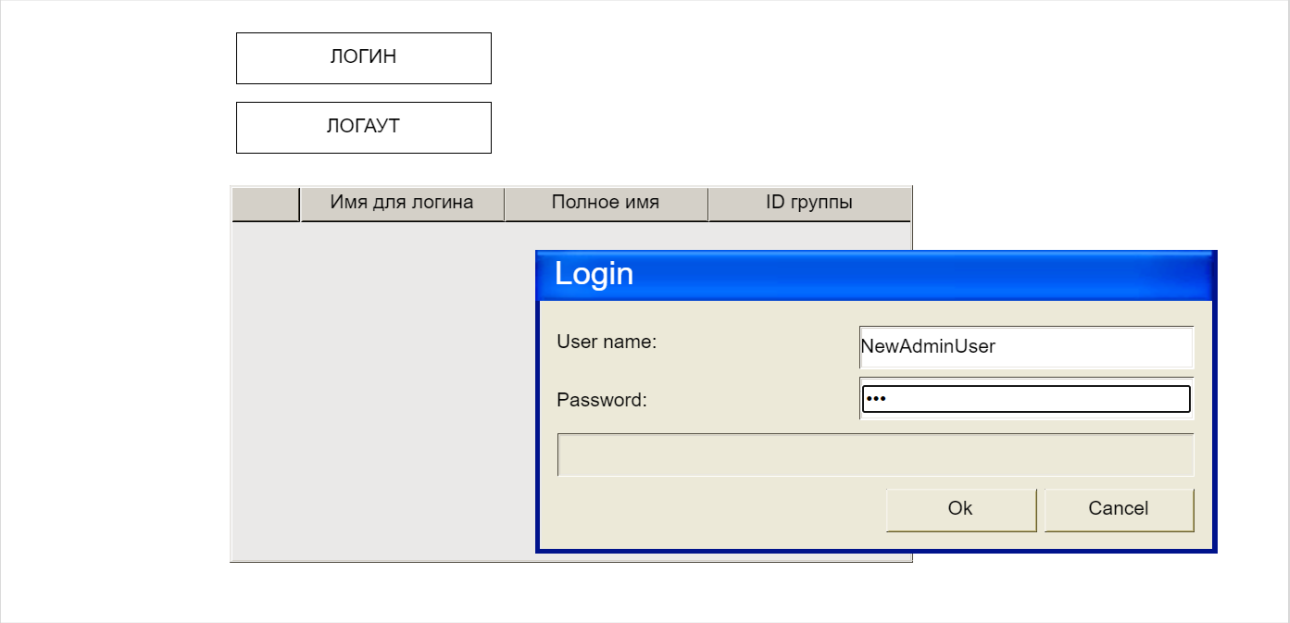


Рисунок 2.7.3 – Авторизация с логином и паролем созданного пользователя

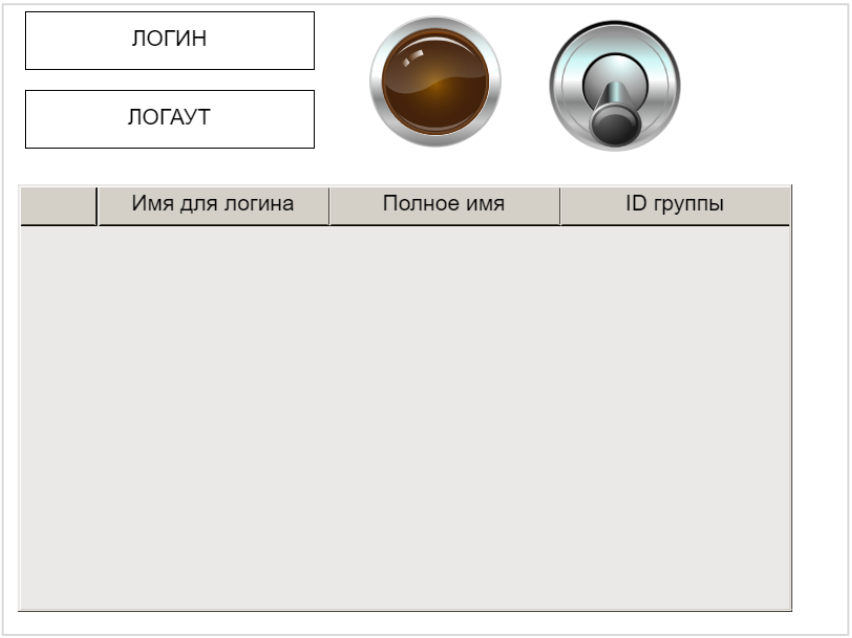


Рисунок 2.7.4 – Авторизация прошла успешно – отобразились переключатель и индикатор (они отображаются только для пользователей группы **Admin**, а созданный нами из кода пользователь принадлежит именно этой группе)

По аналогии вы можете использовать команды **VUM_INSERT**, **VUM_MODIFY** и **VUM_REMOVE**. При этом нужно учитывать следующие моменты:

- **VUM_MODIFY** – если используется runtime-based user management, то клиент должен быть авторизован как пользователь группы с правом «Разрешения изменения пользовательских данных» (см. **Менеджер визуализации - Управление пользователями – Группы**). На вход **pvumUserOld** следует передать указатель на текущие данные редактируемого пользователя, а на вход pvumUserNew – на значения, которые вы хотите получить в результате редактирования;
- **VUM_INSERT** – отличается от **VUM_ADD** тем, что пользователь добавляется не в конец списка пользователей, а на указанную позицию, определяемую входом **udiIndex**. Это имеет значение, когда вы отображаете список пользователей в визуализации;
- **VUM_REMOVE** – нельзя удалить пользователя, если один из клиентов сейчас авторизован под его логином и паролем.

2.8. Методы, которые мы не рассмотрели

Мы рассмотрели существенную часть методов интерфейсов библиотеки **VisuUserMgmt3 Interfaces**. Давайте обзорно рассмотрим те из них, что остались за кадром. Сначала еще раз вспомним структуру библиотеки:

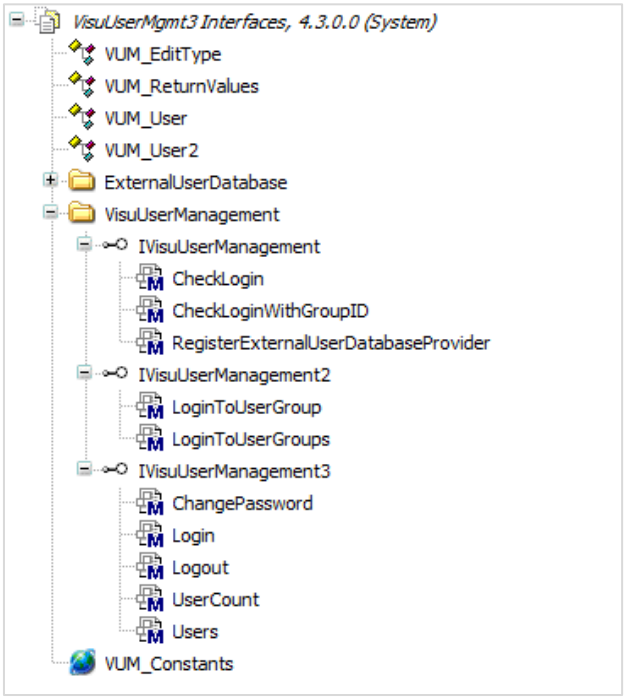


Рисунок 2.8.1 – Структура библиотеки **VisuUserMgmt3 Interfaces**

Все методы интерфейса **IVisuUserManagement3** мы рассмотрели в предыдущих пунктах.

В интерфейсе IVisuUserManagement есть два метода для проверки данных пользователя – CheckLogin и CheckLoginWithGroupID.

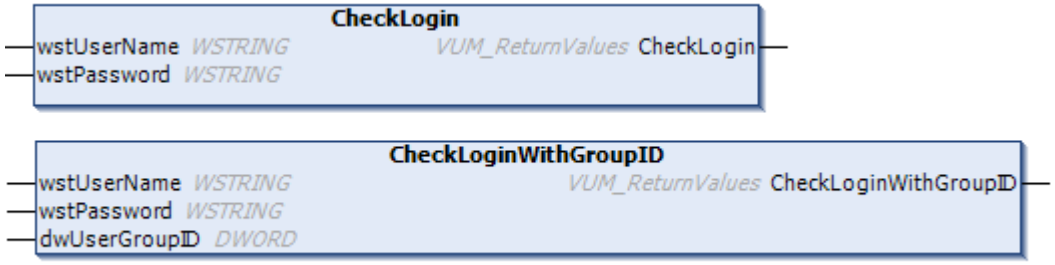


Рисунок 2.8.2 – Сигнатура методов **CheckLogin** и **CheckLoginWithGroupID**

Они крайне просты в использовании – вы передаете методу логин и пароль (в случае **CheckLoginWithGroupID** – еще и идентификатор группы) пользователя, а метод выполняет проверку наличия в базе данных пользователя с такими данными. Тип возвращаемого значения – уже известное нам перечисление VUM_ReturnValues. Если пользователь найден – то возвращается **ERR_OK**.

Метод RegisterExternalUserDatabaseProvider регистрирует экземпляр интерфейса **IExternalUserDatabaseProvider** или **IExternalUserDatabaseProvider2** (они присутствуют в папке **ExternalUserDatabase**, которую видно на рис. 2.8.1) для работы с внешней базой данных пользователей. Все эти интерфейсы нужны в том случае, если вас не устраивает встроенная база данных информации о пользователях и вы хотите реализовать собственную (например, вы планируете использовать в качестве такой базы данных LDAP-сервер предприятия).

В интерфейсе IVisuUserManagement2 есть два метода для авторизации пользователя – LoginToUserGroup и LoginToUserGroups. Основное их отличие от метода Login, который мы рассматривали в п. 2.5, заключается в том, что эти методы не проверяют наличие в базе данных пользователя с переданными им логином и паролем – то есть можно залогиниться под «несуществующим» пользователем. Отличие метода **LoginToUserGroups** от **LoginToUserGroup** в том, что он позволяет залогиниться, указав принадлежность сразу нескольким группам пользователей.

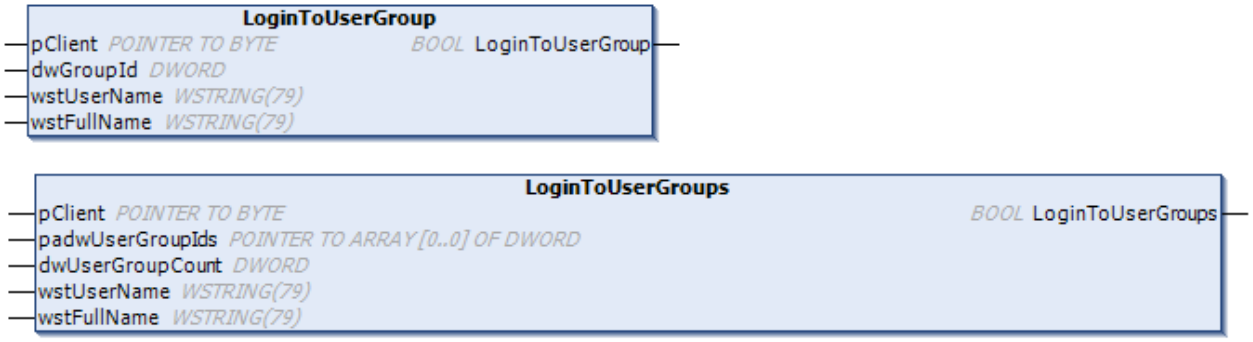


Рисунок 2.8.3 – Сигнатуры методов **LoginToUserGroup** и **LoginToUserGroups**

Про интерфейсы **VisuUserManagement** мы практически не говорили. Напомню, что эта библиотека является «системной» и подразумевается, что разработчик в большинстве случаев не должен ее использовать (эта библиотека используется разработчиками CODESYS для реализации самой системы управления пользователями) – вместо этого он должен работать с **VisuUserMgmt3 Interfaces**.

Структура библиотеки **VisuUserManagement** выглядит следующим образом:

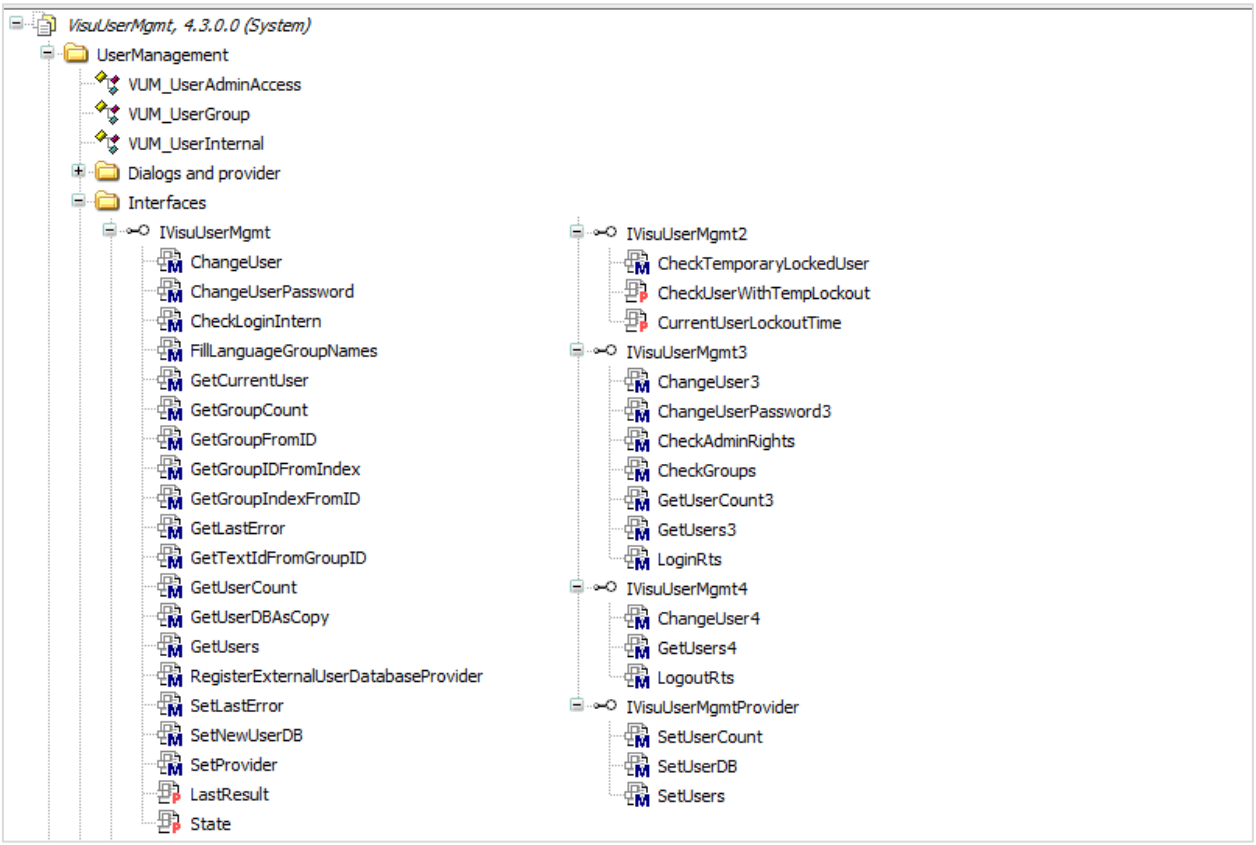


Рисунок 2.8.4 – Структура библиотеки VisuUserManagement

Многие методы по назначению аналогичны соответствующим методам библиотеки VisuUserMgmt3 Interfaces – например, **ChangeUser**, **ChangeUserPassword**, **CheckLoginIntern**, **GetUsers** и т. д.

Часть методов нацелена на получение информации из базы данных пользователей – например, метод GetGroupCount возвращает число групп пользователей. Некоторые методы связаны со спецификой реализации отдельных моментов библиотеки – например, перед вызовом метода ChangeUser нужно вызвать GetUserDBAsCopy, чтобы получить идентификатор базы данных пользователей, а после вызова метода ChangeUser – вызвать метод SetNewUserDB, чтобы сохранить внесенные изменения.

Метод и свойства интерфейса IVisuUserMgmt2 посвящены получению информации о «временно заблокированных» пользователях (тех, которые превысили допустимое число попыток авторизации, заданное в менеджере визуализации на вкладке **Управление пользователями – Установки**).

Некоторые методы в процессе выпуска новых версий CODESYS дополнялись новыми аргументами и, соответственно, появлялись их новые версии с порядковыми номерами – например, **GetUsers3**, **GetUsers4** и т. д.

В целом, большинство из этих методов не документированы и рассказать о них что-то интересное довольно сложно. Примеры использования некоторых методов доступны в демонстрационном проекте от разработчиков CODESYS (в приложении **Application_Old**).

2.9. Получении информации о действиях пользователя визуализации

Рассматриваемый в этом пункте вопрос выходит за пределы библиотеки **VisuUserManagement**. Но поскольку он непосредственно касается работы с пользователями визуализации – то я решил оставить его именно здесь. Кроме того, это кажется мне удачным мостиком к следующим разделам документа.

Вопрос заключается в следующем – как в коде программы получить информацию о действиях, которые совершают пользователи визуализации?

Давайте сразу обсудим, о каких именно действиях идет речь. В идеальном случае хотелось бы получать информацию вообще обо всех действиях пользователей (в стиле: «Пользователь Admin нажал на прямоугольник Rectangle1 и ввел значение 123») – т. е. иметь полноценный аудиторский след. В текущих версиях CODESYS такой возможности нет; в баг-трекере есть пожелание по ее реализации (и даже видно, что какие-то работы ведутся), но из текста неясно, будет ли на первом этапе этот функционал поддержан именно для визуализации.

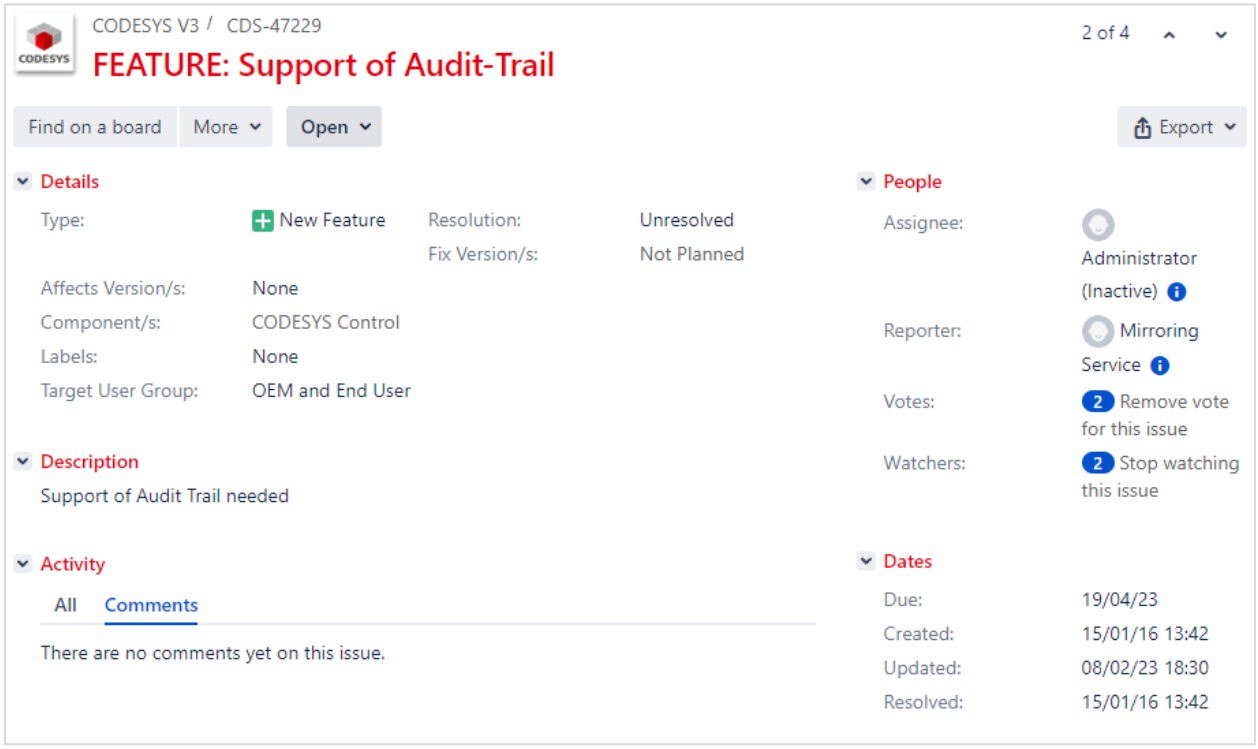


Рисунок 2.9.1 – Пожелание в баг-трекере CODESYS о поддержке аудиторского следа

Но уже в текущих версиях CODESYS можно получить информацию о 4 событиях:

- успешной попытке авторизации в системе;
- неуспешной попытке авторизации в системе (когда были введены некорректные логин и/или пароль);
- изменение пароля пользователя;
- выход из системы.

Давайте добавим этот функционал в наш проект. В рамках примера я добавлю в визуализацию таблицу тревог, в которой буду отображать все упомянутые выше события.

Для начала создадим функциональный блок **VisuUserMgmtEventHandler**, который будет реализовывать (IMPLEMENTS) интерфейс VisuElems.VisuElemBase.IUserMgmtEventHandler.

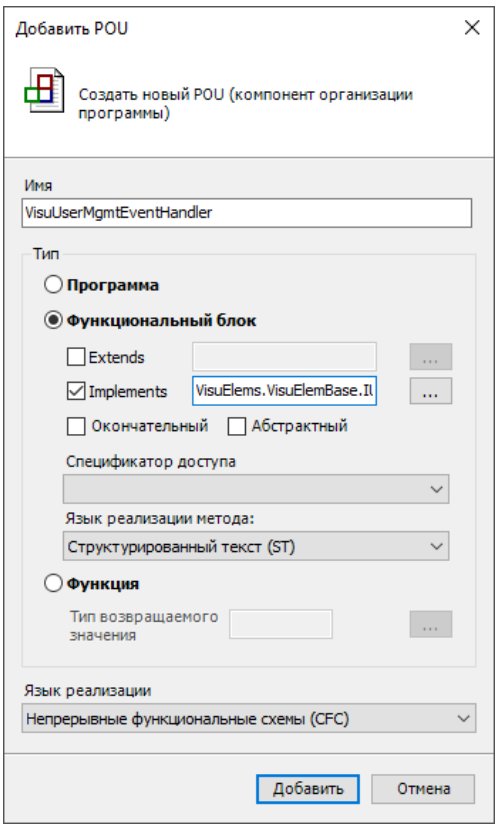


Рисунок 2.9.2 – Создание ФБ с реализацией интерфейса **VisuElems.VisuElemBase.IUserMgmtEventHandler**

Интерфейс **IUserMgmtEventHandler** из библиотеки **VisuElemBase**, входящей в состав библиотеки **VisuElems**, содержит прототипы 4 методов, соответствующих 4 событиям, упомянутым на прошлой странице. После создания ФБ эти методы будут созданы автоматически – правда, возможно, вы столкнетесь с ошибками компиляции. Чтобы устранить их – в области входных переменных каждого метода допишите для типа переменной **pClient** пространство имен **VisuElems.VisuElemBase**.

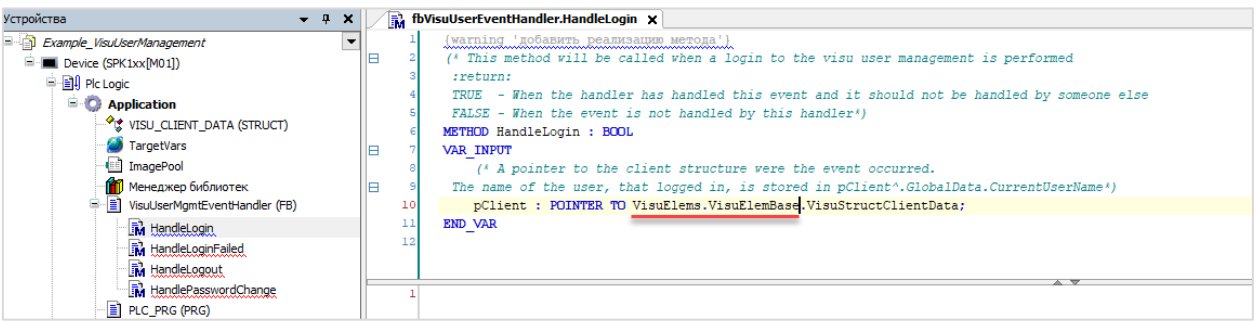


Рисунок 2.9.3 – Указываем пространство имен для структуры **VisuStructClientData**

Если вам показалось, что мы уже когда-то делали что-то подобное – то вам не показалось. Созданием ФБ, реализующей интерфейс, мы первый раз занимались в п. 1.2.1, а дописывание пространства имен для устранения ошибок компиляции – в п. 1.4.3.

Итак, наш ФБ содержит 4 метода, которые будут автоматически (без нашего участия) вызываться при возникновении определенных событий в визуализации:

- **HandleLogin** – при успешном входе пользователя в систему;
- **HandleLoginFailed** – при неуспешной попытке авторизации в системе (когда были введены некорректные логин и/или пароль);
- **HandleLogout** – при выходе пользователя из системы;
- **HandlePasswordChange** – при изменении пароля пользователя.

Аргументом каждого из методов является указатель на контекст клиента **pClient** – указатель на структуру данных клиента визуализации, которую мы уже множество раз использовали. У метода **HandleLoginFailed** в дополнение к нему есть вход **udiError**, который содержит код ошибки из списка глобальных констант VUM_ErrorCodes библиотеки **VisuUserManagement** – там мы сможем определить, какая же именно ошибка возникла при неудачной попытке логина (например, это может быть **ERR_VUM_WRONG_PASSWORD**).

Каждый метод имеет тип **BOOL**. Если присвоить ему значение **TRUE** – то данное событие больше не будет обрабатываться ни одним обработчиком. Это имеет смысл в тех случаях, когда разные «подсистемы» вашего проекта должны получать информацию об одних и тех же событиях – и если вы хотите в некоторых случаях «блокировать» получение информации другими «подсистемами» (в контексте рассматриваемого функционала сложно привести реалистичный пример; но в п. 7.3 мы будет «перехватывать» нажатия на экран визуализации – и если вы хотите осуществить «блокировку» экрана (запретив обработку нажатий подсистемой визуализации CODESYS), то следует в вызове соответствующего метода присваивать его выходу значение **TRUE**). Если выход метода будет иметь значение **FALSE**, то после завершения работы метода это событие может быть передано экземплярам других блоков, реализующих данный интерфейс.

В коде методов мы можем выполнить нужные нам операции. В нашем примере при возникновении события я буду формировать строку с сообщением о данном событии и увеличивать значение счетчика событий. Эту информацию нужно как-то вернуть в программу пользователя, в которой будет вызываться экземпляр нашего ФБ. Ранее мы обсуждали возможные способы в п. 1.2.3; я воспользуюсь уже известным нам способом – верну значения по указателям, переданным на входы ФБ. Для этого объявим в ФБ два входа:

VisuUserMgmtEventHandler

```
1  FUNCTION_BLOCK VisuUserMgmtEventHandler IMPLEMENTS VisuElems.VisuElemBase.IUserMgmtEventHandler
2  VAR_INPUT
3      // Счетчик событий
4      pudiEventCounter:          POINTER TO UDINT;
5      // Сообщение о последнем событии
6      pwsLastMessage:            POINTER TO WSTRING;
7  END_VAR
8  VAR_OUTPUT
9  END_VAR
10 VAR
11 END_VAR
```

Рисунок 2.9.4 – Объявление входов ФБ **VisuUserMgmtEventHandler**

Так как действия, выполняемые для каждого из событий, будут одними и теми же (инкремент счетчика и формирование строки сообщения), то, чтобы избежать дублирования кода, я создам дополнительный метод **prvEventHandler**:

```
METHOD prvEventHandler : BOOL
VAR_INPUT
   wsUserName:                    WSTRING;
   wsMessage:                     WSTRING;
END_VAR
VAR
   wsFullMessage:                 WSTRING;
END_VAR

// Код метода

pudiEventCounter^ := pudiEventCounter^ + 1;

wsFullMessage := WCONCAT(wsUserName, ": ");
wsFullMessage := WCONCAT(wsFullMessage, wsMessage);

pwsLastMessage^ := wsFullMessage;
```

Код **HandeLogin**, **HandleLoginFailed** и остальных «интерфейсных» методов будет состоять всего из одной строки:

```
// код метода HandleLogin
prvEventHandler(pClient^.GlobalData.CurrentUserName, "вход пользователя");

// код метода HandleLoginFailed
prvEventHandler("Неизвестный", "неудачная попытка входа");

// код метода HandleLogout
prvEventHandler(pClient^.GlobalData.CurrentUserName, "выход пользователя");

// код метода HandlePasswordChange
prvEventHandler(pClient^.GlobalData.CurrentUserName, "изменения пароля");
```

В качестве значений аргументов метода **prvEventHandler** мы передаем имя пользователя визуализации, извлеченное из контекста клиента (путь к нему отображается в автоматически сгенерированном комментарии к методу, что довольно удобно) и текст сообщения для события. Единственное исключение – метод **HandleLoginFailed**; поскольку в случае его вызова имя пользователя визуализации неизвестно (ведь он не смог авторизоваться), то вместо него мы подставляем «Неизвестный».

Теперь объявим экземпляр созданного нами ФБ в программе – вместе с несколькими дополнительными переменными.

```
PROGRAM PLC_PRG
VAR
   // объявление переменных из прошлых пунктов
   // ...
   // Флаг запуска программы
   xInit:                                  BOOL;
   // ФБ для обработки событий пользователей визуализации
   fbVisuUserEventHandler:                 VisuUserMgmtEventHandler;
   // Счетчик событий
   udiEventCounter:                        UDINT;
   // Сообщение о последнем событии
   wsLastMessage:                          WSTRING;
END_VAR
VAR CONSTANT
   // Максимальное ожидаемое количество пользователей визуализации
   c_udiMaxVisuUserCount:         UDINT := 10;
END_VAR
```

В коде программы на первом цикле добавим инициализацию входных указателей экземпляра блока (опять же, мы уже делали это в п. 1.2.3) и регистрацию нашего экземпляра в качестве обработчика событий пользователей визуализации – для этого мы вызывает соответствующий метод глобального обработчика разных событий (**g_VisuEventManager**), объявленный в списке глобальных переменных **Visu_Globals** библиотеки **VisuElemsBase**; подробнее об этом списке мы поговорим в п. 3.2.

```
IF NOT(xInit) THEN

   VisuElems.VisuElemBase.Visu_Globals.g_VisuEventManager.SetUserMgmtEventHandler
      (fbVisuUserEventHandler);

   fbVisuUserEventHandler.pudiEventCounter := ADR(udiEventCounter);
   fbVisuUserEventHandler.pwsLastMessage   := ADR(wsLastMessage);

   xInit := TRUE;

END_IF

// код программы из прошлых пунктов
// ...
```

Осталось добавить в проект конфигурацию тревог, а на экран визуализации – таблицу тревог. Я не буду подробно останавливаться на этом моменте – вы можете найти информацию о таблице тревог в этом видео, а также в документе CODESYS V3.5. Визуализация.

Покажу лишь основные детали. События пользователей визуализации я буду интерпретировать как тревогу класса **Info** со способом подтверждения **REP** (то есть без возможности квитирования).

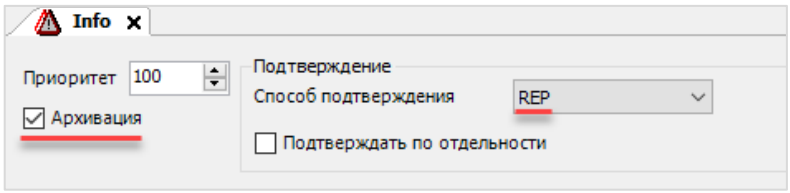


Рисунок 2.9.5 – Настройки класса тревог **Info**

В группе тревог создадим тревогу со следующими настройками:

- **Способ наблюдения** – Изменение (т. е. тревога будет однократно генерироваться при изменении значения целочисленной переменной, привязанной в столбце **Детали**);
- **Детали** – PLC_PRG.udiEventCounter (наша переменная, представляющая собой счетчик событий, значение которой мы инкрементируем в методе **prvEventHandler**);
- **Класс** – Info;
- **Сообщение** – <LATCH1> (т. е. текстом сообщения будет является значение переменной, привязанной в столбце **Триггерная переменная 1**);
- **Триггерная переменная 1** – PLC_PRG.wsLastMessage (наша переменная, представляющая собой текст сообщения, значение которой мы инкрементируем в методе **prvEventHandler**.

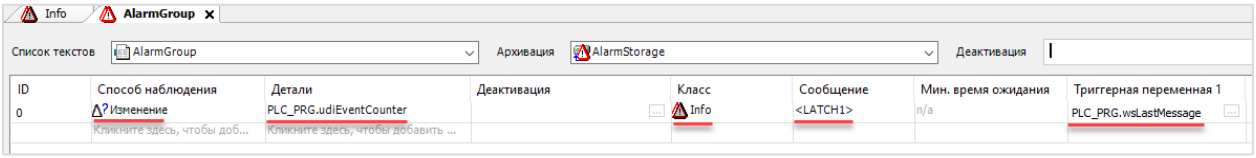


Рисунок 2.9.6 – Настройки тревоги

На экране визуализации добавим таблицу тревог. К ее управляющей переменной **История** привяжем локальную переменную, которую объявим в интерфейсе экрана визуализации. Зададим ей значение **TRUE** – чтобы в таблице сразу при старте приложения отображалась история тревог.

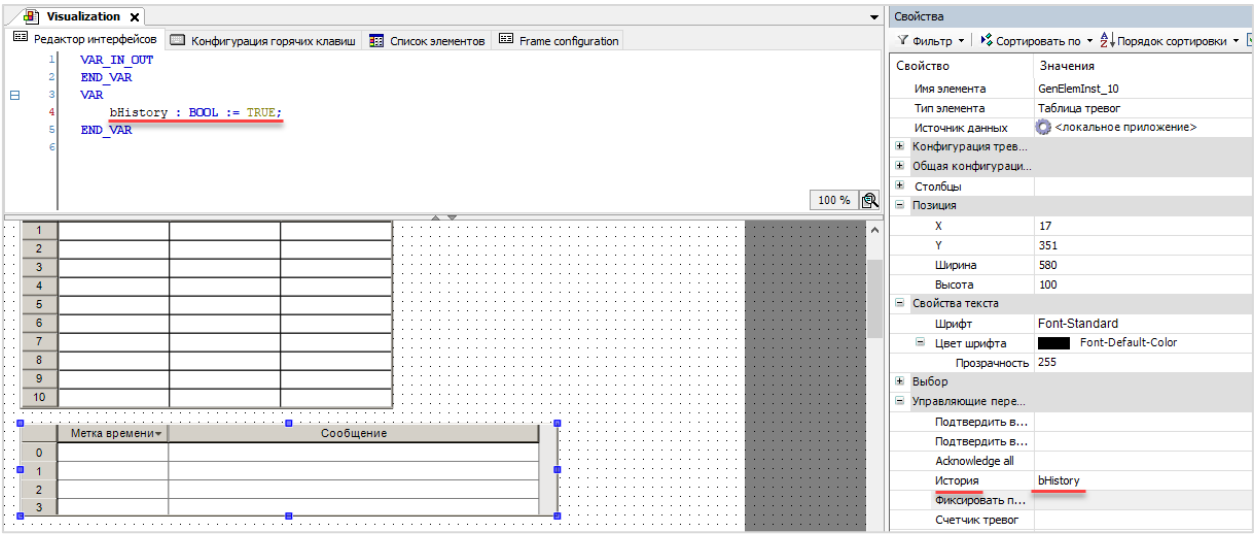


Рисунок 2.9.7 – Настройки таблицы тревог

Давайте проверим наш пример на практике. Загрузите проект примера в контроллер и запустите его. С помощью кнопки **ЛОГИН** попробуйте авторизоваться с неверным паролем, а затем – с верным (логины и пароли проекта примера указаны в табл. 2.1.1). Далее выйдите из системы с помощью кнопки **ЛОГАУТ**. В таблице тревог отобразятся все 3 произошедших события:

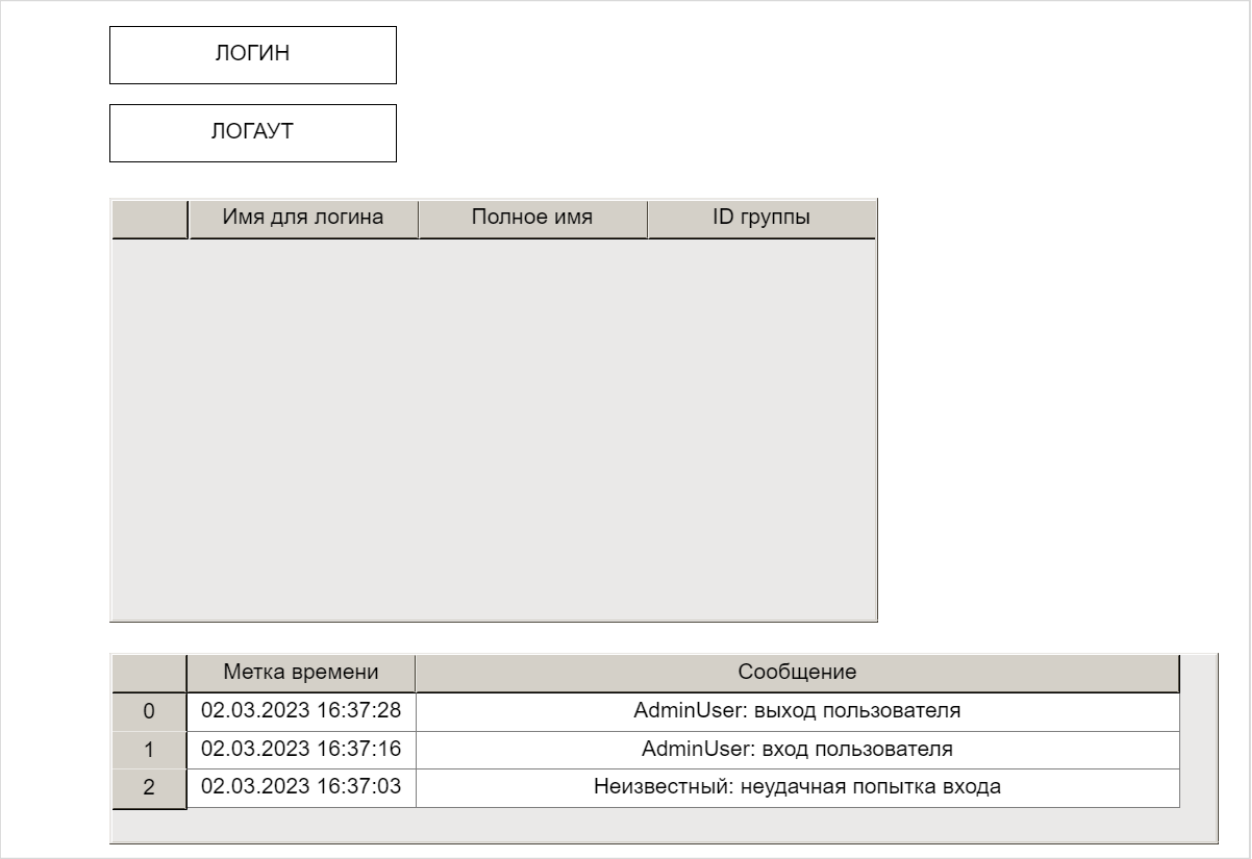


Рисунок 2.9.8 – Отображение событий, связанных с пользователями визуализации, в таблице тревог

Соответственно, события будут формироваться и в тех случаях, если вы будете осуществлять эти действия (авторизация, логаут и т. д.) из кода – как мы делали в п. 2.5.

2.10. Заключение

В данном разделе мы рассмотрели, как работать с пользователями визуализации из кода программы, применяя для этого библиотеки **VisuUserMgmt3 Interfaces** и **VisuUserManagement**. Вполне возможно, что часть рассмотренного функционала в будущем будет перенесена в библиотеку Visu Utils – в баг-трекере CODESYS есть соответствующее пожелание.

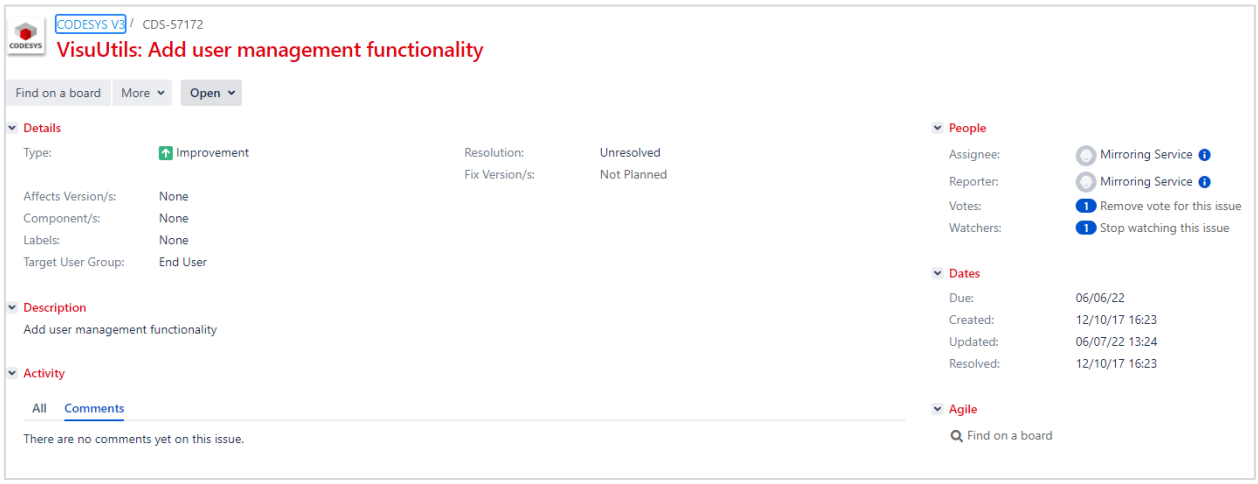


Рисунок 2.10.1 – Пожелание в баг-трекере CODESYS о добавлении в библиотеку **Visu Utils** функционала по работе с пользователями визуализации (**VisuUserManagement**)

В прошлом пункте (п. 2.9) мы плавно перешли к возможностям библиотеки **VisuElemsBase**, показав, как с ее помощью получать информацию о событиях, связанных с пользователями визуализации. В следующих разделах мы рассмотрим и другие возможности этой библиотеки. Начнем с ее общего обзора.

3. Библиотека VisuElemBase

3.1. Основная информация

Библиотека VisuElemBase реализует существенную часть функционала, используемого подсистемой визуализации CODESYS. Обычно библиотеку нет смысла добавлять в проект отдельно – она входит в состав библиотеки VisuElems, которая автоматически добавляется в менеджер библиотек при наличии в проекте визуализации. Сама **VisuElems** особенного интереса для разработчика не представляет – она содержит всего пару недокументированных функций и ФБ.

Библиотека **VisuElemBase** включает в себя две корневые папки – Private Implementation и Public Parts. В папке **Private Implementation** содержатся объекты, используемые самими разработчиками CODESYS – для реализации подсистемы визуализации, ее отладки, профилирования и т. д. Эти объекты практически не документированы и подразумевается, что они не должны использоваться другими разработчиками. В папке **Public Parts** содержатся объекты, которые как раз должны представлять для нас интерес; некоторые из них даже сопровождаются документацией.

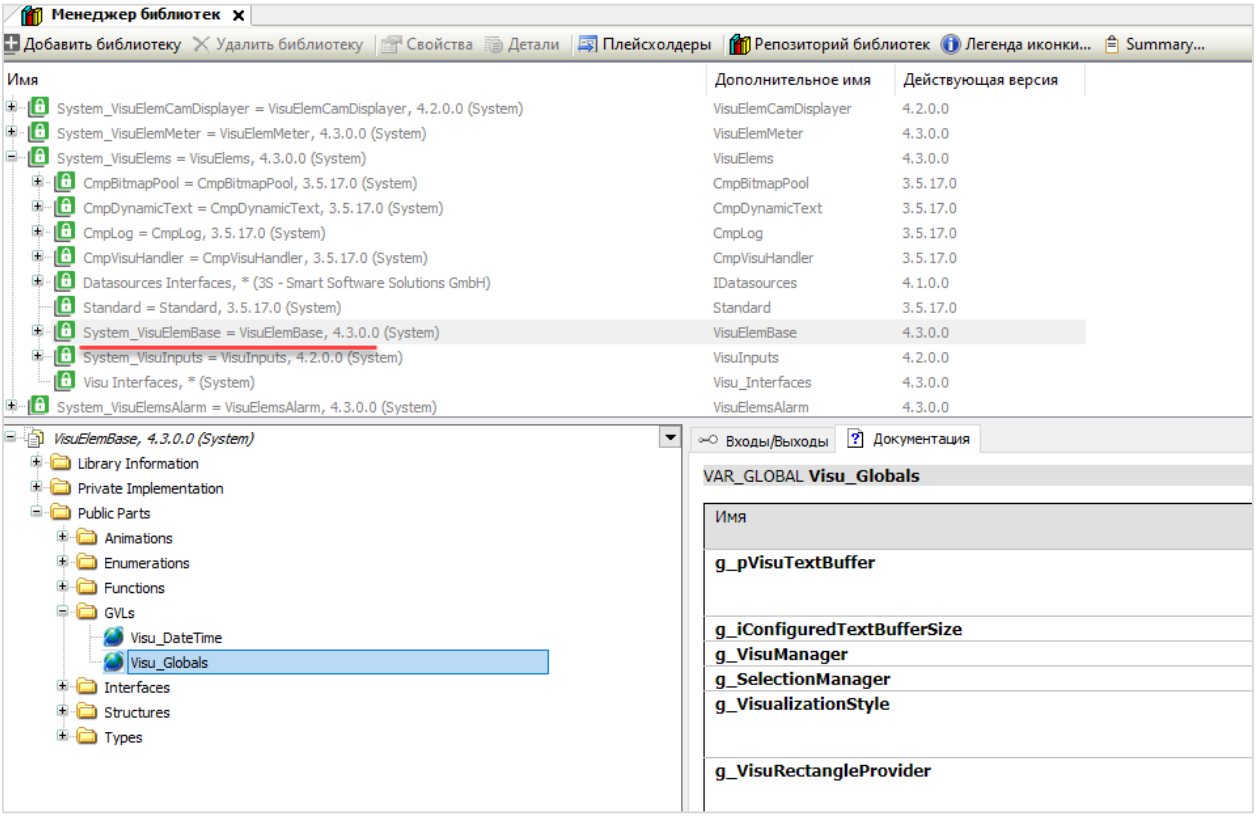


Рисунок 3.1.1 – Структура библиотеки **VisuElemsBase**

В следующих пунктах я обзорно расскажу о содержимом подпапок GVLs и Interfaces – именно они представляют наибольший интерес для разработчиков, которые планируют работать с визуализацией CODESYS из кода программы.

3.2. Списки глобальных переменных и констант

Начнем со списка глобальных переменных Visu_Globals.

Таблица 3.2.1 – Элементы списка глобальных переменных **Visu_Globals**

Название	Тип	Описание
g_pVisuTextBuffer	POINTER TO ARRAY [0..Visu_Constants. VISU_TEXTBUFFER_SIZE] OF WORD	Указатель на глобальный текстовый буфер и его размер в байтах. Насколько я понимаю – этот буфер используется для форматирования строк, отображаемых в элементах визуализации. По умолчанию размер буфера равен **-1** – это значение, по всей видимости, обрабатывается особым образом. Возможно, в этом случае память под буфер выделяется динамически по мере необходимости или же определяется значением глобальной константы **VisuElems.Visu_Constants. VISU_TEXTBUFFER_SIZE**
g_iConfiguredTextBufferSize	INT	
g_VisuManager	IVisuManager2	Глобальный экземпляр интерфейса менеджера визуализации. См. п. 3.4
g_SelectionManager	ISelectionManager	Глобальный экземпляр интерфейса менеджера выбора элементов и имитации нажатия на элемент. См п. 6
g_VisualizationStyle	IVisualizationStyle	Глобальный экземпляр обработчика стиля визуализации. Позволяет получить информацию о параметрах стиля
g_VisuRectangleProvider	IApplication RectangleProvider	Глобальный экземпляр обработчика жестов
Current-переменные могут использоваться только в визуализации – то есть вы можете привязать их к элементам визуализации, использовать в коде действий **Выполнить ST-код** этих элементов и т. д. Но использовать их в коде проекта бессмысленно, потому что они будут иметь некорректные значения. Эти переменные содержат информацию о «текущем» клиенте визуализации и, соответственно, нуждаются в контексте клиента – а он существует только в визуализации. Единственное исключение – CURRENTVISU (см. пояснение ниже)		
CURRENTVISU	STRING	Для использования этой переменной нужно установить галочку **Использовать переменную CurrentVisu** в **Менеджере визуализации**. После установки галочки визуализация становится «однопользовательской» – то есть все клиенты визуализации синхронизируются (например, переключение экрана для одного клиента приводит к переключению экрана для всех клиентов). Чтение значения переменной позволяет

		определить название текущего открытого экрана. Запись значения переменной позволяет переключить экран визуализации
CURRENTLANGUAGE	STRING	Название языка визуализации для текущего клиента визуализации (в текущих версиях CODESYS язык переключается только для всех клиентов сразу)
CurrentUserGroupId	DWORD	ID группы пользователей для текущего клиента визуализации
CurrentUserGroupName	WSTRING(511)	Имя группы пользователей для текущего клиента визуализации
CurrentUserName	WSTRING	Имя пользователя для текущего клиента визуализации («имя для логина»)
CurrentFullUserName	WSTRING	Полное имя пользователя для текущего клиента визуализации
CurrentUseAutoLogoutTime	BOOL	**TRUE** – для группы пользователей, которой принадлежит текущий клиент визуализации, в **Управлении пользователями** установлена галочка **Автоматический выход**
CurrentRemainingAutoLogoutTime	TIME	Время, оставшееся до автоматического разлогинивания текущего клиента визуализации, вследствие его неактивности
CurrentClientType	Visu_ClientType	Тип текущего клиента визуализации
VISU_CYCLE_TIME_ON_ OPEN_DIALOGS	UDINT	Справка туманно указывает, что эти настройки связаны с оптимизацией производительности открытия диалогов
OPEN_DIALOGS_DISABLE_ CHANGING_INVALIDATION_RECT	BOOL	
g_VisuKineticScrollingSizeFactor	REAL	Судя по названиям – эти настройки связаны с полосой прокрутки (например, она может присутствовать у таблиц). Никакой точной информации о них нет
g_VisuKineticScrollingDamping	REAL	
g_VisuTouchScrollingOutsideElem	BOOL	
g_VisuLineJoinMiterLimit	REAL	Никакой информации нет. Возможно, связана с одноименным параметром HTML5 Canvas?
g_MaxTwoDigitYear	UINT	Эта переменная задает максимальное значение года, который отображается в визуализации при форматировании %t[YY]. Поскольку в этом случае отображаются только две последние цифры года, то неясно, к какому веку этот год относится. Значение по умолчанию – 2099 (любое двухзначное значение года относится к 21 веку). Если, например, задать 2050 – то значения 0…50 будут относиться к 21 веку, а 51…99 – к 20 веку (1951..1999)
g_DateTimeValuesInIECSyntax	BOOL	Эта настройка влияет на формат ввода переменных даты и времени в элементах визуализации (DT, DATE,

		TIME и т. д.). **TRUE** – используется МЭК-синтаксис (T#11h22m), **FALSE** – используется синтаксис форматирования (11:22). Настройка появилась в **V3.5 SP12** (CDS-46739), до этого поддерживался только МЭК-синтаксис
g_VisuRedundSyncFrameIndex	BOOL	**TRUE** – при использовании плагина резервирования (**CODESYS Redundancy**) индексы экранов фреймов визуализаций активного и пассивного ПЛК синхронизируются См. подробнее по ссылке
g_VisuRedundValueChanged	VisuEnumRedundancyValueChanged	Режим синхронизации данных, вводимых в визуализации активного ПЛК, с пассивным ПЛК (в случае использования **CODESYS Redundancy**)
SECURE_TAPPING_TIMEOUT	DWORD	Представьте, что клиент web-визуализации зажал кнопку и в этот момент связь между клиентом и ПЛК прервалась. В результате с точки зрения программы ПЛК кнопка останется зажатой. Данная настройка позволяет задать период отсутствия связи с клиентом web-визуализации, после которого зажатые им кнопки будут автоматически «отжаты» (VIS-219)
g_xForceDlgPosAsDesigned	BOOL	Режим обработки определения координат открытия диалога в тех случаях, когда предыдущая позиция не найдена (начиная с **CODESYS V3.5 SP17** появилась возможность «перетаскивать» диалоги по экрану и после закрытия следующее открытие диалога этим же клиентом происходило именно по последним установленным координатам). Справка не поясняет, каким режимам соответствуют значения **TRUE** и **FALSE**
g_xUseGlobalDlgPos	BOOL	**TRUE** – для нового подключившегося клиента визуализации диалог будет открываться по координатам, определенным в **Менеджере визуализации** на вкладке **Установки диалога**. **FALSE** – диалог будет открываться по последним установленным координатам
g_VisuTrendWithTimeSelectorUpdateBehaviour	UINT	Режим применения настроек тренда. Подробнее см. по ссылке (VIS-627)

Мы рассмотрели все «видимые» элементы списка, но следует упомянуть о ряде скрытых:

- **g_ClientManager** – глобальный экземпляр интерфейса менеджера клиентов визуализации. Мы рассмотрим его в п. 3.5;
- **g_VisuEventManager** – глобальный экземпляр обработчика событий визуализации. Мы рассмотрим его в п. 7.1.

Также в библиотеке есть список глобальных переменных Visu_DateTime, который содержит единственный элемент **DisplayUTC** типа **BOOL**. Если он имеет значение **TRUE** – при отображении времени в элементах визуализации отображается всемирное координированное время (UTC), если **FALSE** – то отображается локальное время. Это влияет на элементы:

- отображающие системное время (т.е. у которых в параметре **Тексты** указаны спецификаторы формата времени, но не привязано текстовой переменной);
- тренд;
- таблицу тревог и баннер тревог;
- аналоговые часы и элемент выбора даты и времени (если к нему не привязана переменная).

Информация из документации на **Visu_Globals** позволяет нам узнать о существовании списка глобальных констант **Visu_Constants** (см. тип переменной **pVisuTextBuffer**) и списка глобальных переменных **Private_Visu_Globals** (см. начальное значение переменной **g_VisualizationStyle**; я не приводил его в табл. 3.2.1, но вы можете увидеть его в документации).

Документации по этим спискам нет, но, тем не менее, вы можете увидеть их «открытое» содержимое, используя команду **Вид – Просмотр**:

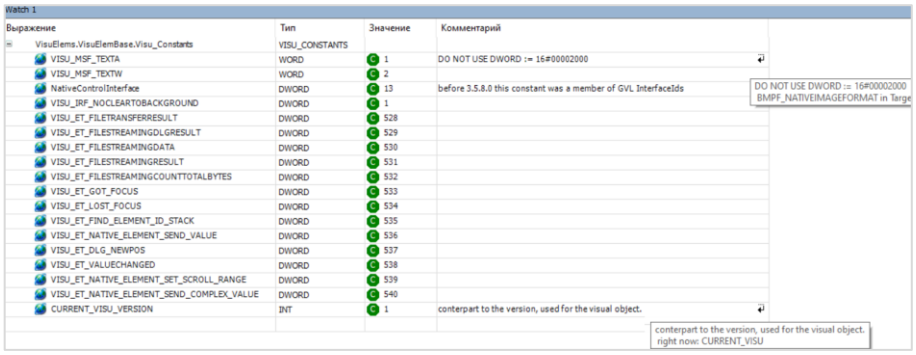


Рисунок 3.2.1 – Элементы списка глобальных констант **Visu_Constants**

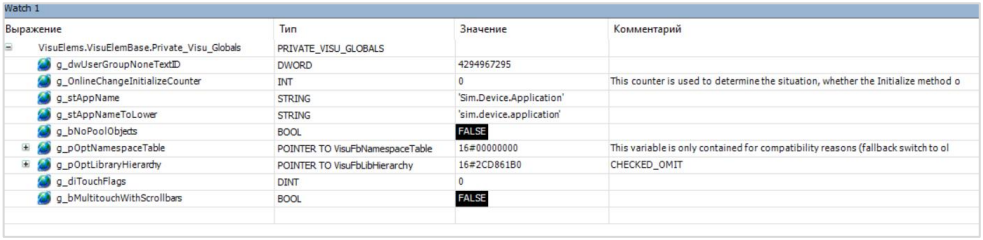


Рисунок 3.2.2 – Элементы списка глобальных переменных **Private_Visu_Globals**

Если вы крайне интересуетесь нюансами визуализации CODESYS – то можете увидеть «legacy» версии списков **Visu_Globals** и **Visu_Constants** в библиотеке **VisuElemFunctionality.**

3.3. Интерфейсы

В этом пункте я сделаю краткий обзор некоторых интерфейсов, присутствующих в библиотеке **VisuElemBase**. Я буду рассматривать только те интерфейсы, которые считаю интересными для разработчика и назначение которых мне понятно. Именно интерфейсы являются точкой входа для использования того или иного функционала – пользователь создает ФБ, реализующий нужный ему интерфейс (или использует глобальный экземпляр ФБ, уже объявленный в библиотеке), добавляет код в его методы, регистрирует экземпляр этого ФБ в обработчике (если это требуется для конкретного интерфейса) и т. д. Подробности будут рассмотрены в пунктах, описывающих соответствующие интерфейсы.

Таблица 3.3.1 – Некоторые интерфейсы библиотеки **VisuElemBase**

Название	Назначение методов интерфейса
ICustomEventHandler	Обработка пользовательских событий, сформированных с помощью библиотеки CmpEventMgr. В будущем я планирую написать отдельную статью об этой библиотеке
IDialogCloseListener	Обработка закрытия диалога. Мы использовали этот интерфейс в п. 1.4.4
IDialogCloseListenerWithTag	Обработка закрытия диалога с получением информации о переменных диалога. Мы использовали этот интерфейс в п. 1.4.4
IDialogManager9	Открытие, закрытие и получение информации о диалогах. В современных версиях CODESYS рекомендуется вместо этого интерфейса использовать ФБ FbOpenDialog и FbOpenDialogExtended из библиотеки **Visu Utils**
IFrameManager2	Работа с фреймами (переключение экранов в фреймах и т. д.). Мы рассмотрим эти интерфейсы в п. 5
IFrameManagerBase	
IGestureEventHandler3	Обработка жестов клиентов визуализации. См. пример
IInputOnElementEventHandler	Обработка нажатий на элемент визуализации. Мы рассмотрим этот интерфейс в п. 7.4
IKeyEventHandler	Обработка событий клавиатуры. Мы рассмотрим этот интерфейс в п. 7.2
IMouseEventHandler	Обработка событий курсора. Мы рассмотрим этот интерфейс в п. 7.3
ISelectionManager	Выбор элементов визуализации и имитация нажатия на элемент. Мы рассмотрим этот интерфейс в п. 6
ISpecialEventHandler	Обработка специальных событий клиентов визуализации. Используется только разработчиками CODESYS
IUserMgmtEventHandler	Обработка событий пользователей визуализации (авторизация, логаут и т. д.). Мы рассматривали этот интерфейс в п. 2.9
IValueChangedListener	Обработка изменения значения переменных, привязанных к элементу визуализации. Мы рассмотрим этот интерфейс в п. 7.6
IVisuManager3	Переключение визуализации, получение информации о текущей открытой визуализации и т. п. В современных версиях CODESYS рекомендуется вместо этого интерфейса использовать ФБ FbChangeVisu и FbIterateClients из библиотеки **Visu Utils**. Но еще этот интерфейс, например, позволяет получить экземпляр **FrameManager**'а – мы воспользуемся этим в п. 5
IVisuManagerBase	
IVisuUserEventManager	Регистрация экземпляров обработчиков, реализующих интерфейсы <…>EventHandler. Мы рассмотрим его в п. 7

3.4. Менеджер визуализации (g_VisuManager)

g_VisuManager – это экземпляр функционального блока, реализующего интерфейс IVisuManager3 (который по цепочке наследует интерфейсы **IVisuManager2**, **IVisuManager** и IVisuManagerBase), объявленный в списке глобальных переменных Visu_Globals. На рисунке ниже приведено перечисление методов этого объекта:

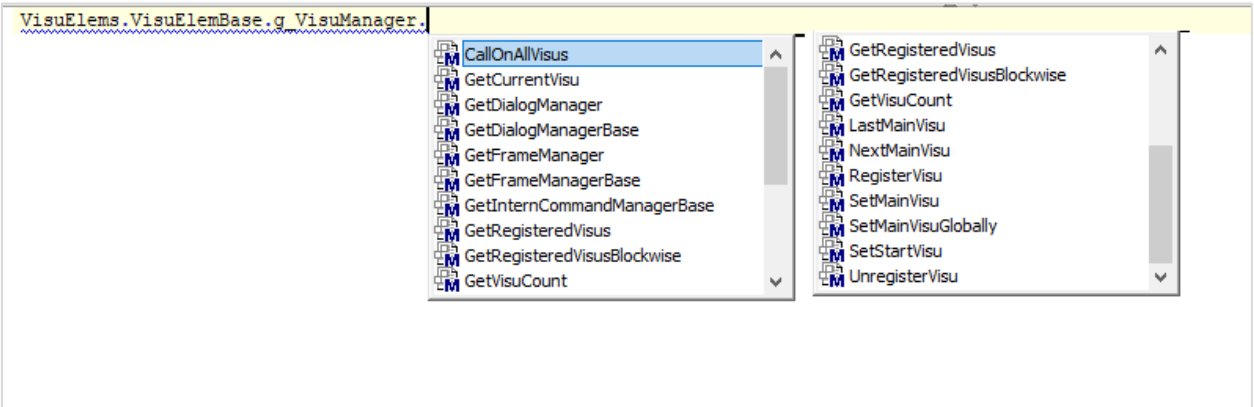


Рисунок 3.4.1 – Методы **g_VisuManager**

Я не буду подробно рассматривать эти методы, потому что существенная часть реализуемого ими функционала в современных версиях CODESYS доступна в библиотеке Visu Utils (и одной из ключевых целей разработки **Visu Utils** было как раз предоставить «чистое API» для работы с визуализацией). Например:

- вместо методов **SetMainVisu** и **SetMainVisuGlobally** (переключение экрана визуализации для конкретного клиента и для конкретного типа клиентов) можно использовать ФБ FbChangeVisu;
- вместо методов **LastMainVisu** и **NextMainVisu** (получение информации о предыдущем экране визуализации, на котором находится клиент, и том, на который он попадет после нажатия кнопку с действием **Следующая визуализация**) можно использовать ФБ FbIterateClients, который, в том числе, позволяет получить и эту информацию).

Есть и ряд методов, у которых нет аналогов в **Visu Utils** – например, метод **GetVisuCount** позволяет определить число экранов визуализации проекта.

В п. 5 мы используем метод **GetFrameManager**, чтобы с помощью него инициализировать экземпляр интерфейса **IFrameManager** и вызывать его методы для работы с фреймами из кода программы.

Напоследок уточню, что не стоит ассоциировать **g_VisuManager** с компонентом **Менеджер визуализации** в дереве проекта – оба этих объекта нужны для работы подсистемы визуализации, но синонимичными они не являются.

3.5. Менеджер клиентов (g_ClientManager)

g_ClientManager – это экземпляр функционального блока, реализующего неизвестные мне интерфейсы (упомяну только **IClientManagerListener3**, о котором расскажу в п. 4). Этот экземпляр объявлен в списке глобальных переменных Visu_Globals (при этом он является скрытым и не отображается в описании в менеджере библиотек). На рисунке ниже приведено перечисление методов этого объекта:

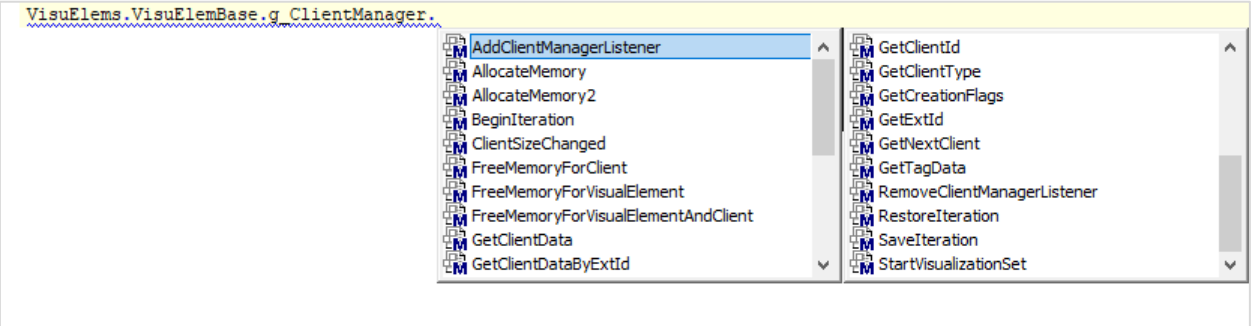


Рисунок 3.5.1 – Методы **g_ClientManager**

Я не буду подробно рассматривать эти методы, потому что существенная часть реализуемого ими функционала в современных версиях CODESYS доступна в библиотеке Visu Utils (и одной из ключевых целей разработки **Visu Utils** было как раз предоставить «чистое API» для работы с визуализацией). В частности, вместо методов **BeginIteration/GetClientData/GetNextClient** можно использовать ФБ FbIterateClients. Методы, названия которых начинаются со слов **Allocate** и **Free**, являются системными и используются для выделения/освобождения памяти, используемой для хранения данных визуализации.

Если вы хотите посмотреть, как использовались методы **BeginIteration/GetClientData/GetNextClient** – то можете ознакомиться с этим примером (создан в **V3.5 SP11 Patch 5**).

Методы **AddClientManagerListener**, **ClientSizeChanged** и **StartVisualizationSet** будут рассмотрены в п. 4.

3.6. Библиотека VisuElemBase и задача VISU_TASK

Еще одна причина разработки библиотеки Visu Utils – это предоставление разработчику возможности вызывать экземпляры ФБ для работы с визуализацией в любой задаче проекта CODESYS. Вызов блоков, реализующих некоторые (и какие именно – не документировано) интерфейсы библиотеки VisuElemBase в контексте задач, отличных от задачи VISU_TASK, может привести к неконкретизированным разработчиками CODESYS проблемам (насколько я помню – одной из проблем является возникновение исключения в приложении контроллера).

Это упоминается в баг-трекере CODESYS:

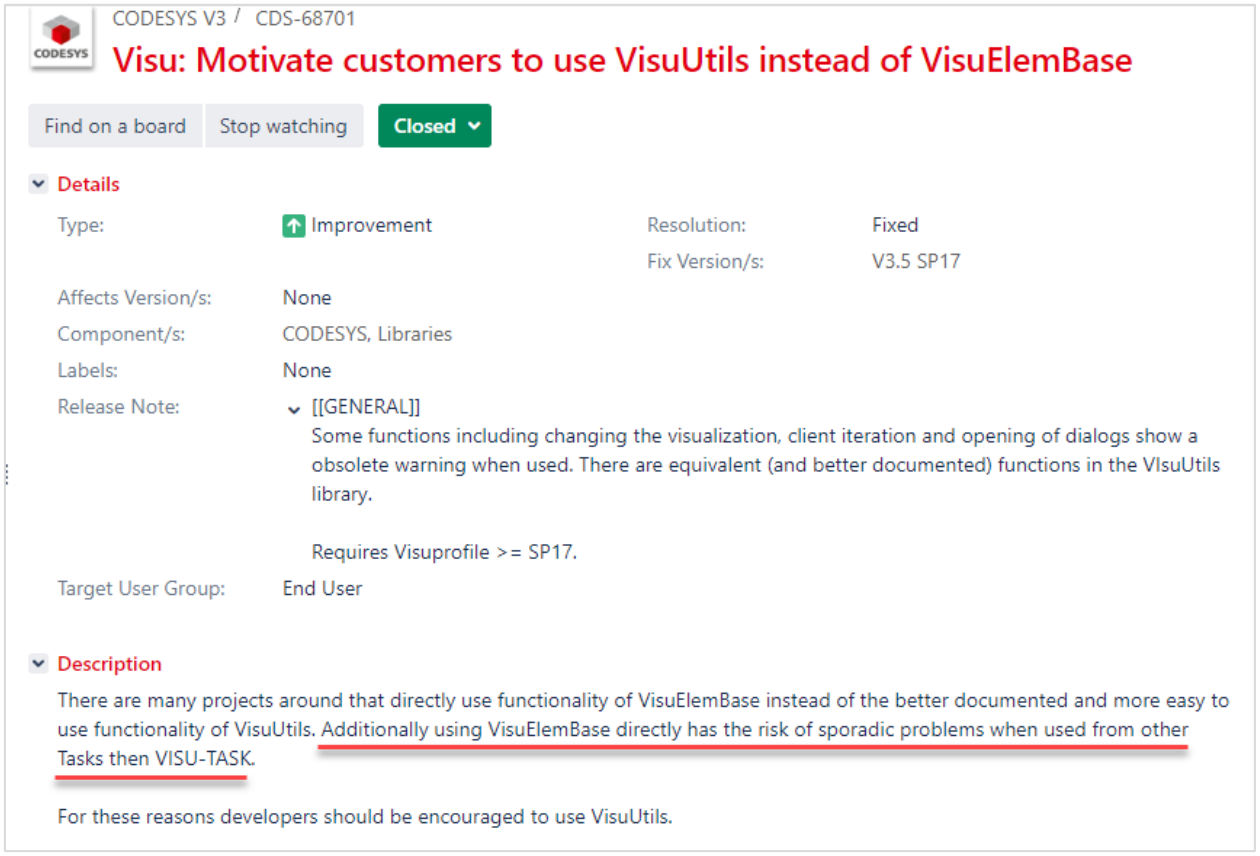


Рисунок 3.6.1 – Задача из баг-трекера CODESYS по мотивации разработчиков переходить на библиотеку **Visu Utils**. В рамках задачи были добавлены предупреждения компилятора при использовании тех объектов библиотеки **VisuElemBase**, которые считаются «устаревшими»

Вы можете справедливо заметить, что в п. 2.9 мы разместили код получения информации о действиях пользователей визуализации в программе, привязанной к задаче **MainTask**. На это я замечу, что в данном случае наш пример всё равно работал корректно. Но в целом вы правы - поэтому в следующих пунктах использование функционала библиотеки **VisuElemBase** будет производиться только в контексте задачи **VISU_TASK**.

3.7. Заключение

Мы обзорно рассмотрели основные элементы библиотеки VisuElemBase. В следующих разделах я более подробно расскажу об ее конкретных объектах, приведя примеры:

- обработки подключения/отключения клиентов визуализации;
- работы с фреймами;
- имитации нажатий на элементы визуализации;
- обработки действий пользователя (нажатий клавиш, событий курсора и т. д.).

4. Обработка событий клиентов визуализации (IClientManagerListener)

4.1. Основная информация

В п. 1.2 мы рассмотрели, как получить информацию о клиентах визуализации с помощью ФБ **FbIterateClients** из библиотеки **Visu Utils**. В примере из пункта информация о клиентах собиралась по команде из программы пользователя; совсем несложно реализовать сбор информации на периодической основе, заведя на вход **xExecute** выход генератора импульсов.

Но иногда было бы полезно получать информацию о клиентах событийно. Типовой пример – нужно определить в программе, что подключился новый клиент и в зависимости от его параметров (например, типа или разрешения дисплея) открыть для него нужный экран визуализации.

Давайте ретроспективно вспомним некоторые моменты из предыдущих пунктов:

- в п. 1.2.2 мы первый раз познакомились с контекстом клиента – структурой VisuElems.VisuElemBase.VisuStructClientData, содержащей огромное количество информации о клиенте визуализации;
- в п. 1.2.3 мы первый раз использовали callback – технологию, при которой вызов экземпляра ФБ осуществляем не мы сами в своем коде, а какой-либо другой программный компонент (например, подсистема визуализации);
- в п. 2.2, рассматривая содержимое библиотеки **VisuUserManagement**, я коротко упомянул ФБ **VisuFbClientManagerListener**, реализующий интерфейс **VisuElems.VisuElemBase.IClientManagerListener**. Пришло время рассмотреть его подробнее.

Интерфейс **IClientManagerListener** является скрытым – поэтому он не упомянут в таблице 3.2. Но нельзя сказать, что информации о нем вообще нет – в 2016 году **Максим Сироткин** (сотрудник компании ПК Пролог – официального дистрибьютора CODESYS в России; вероятно, вы слышали о нем, если читали мою статью об истории библиотек диалогов визуализации в CODESYS или использовали контроллеры Berghof) на московской **CODESYS User Conference** в своем выступлении рассказал об ответственной разработке продвинутой визуализации и продемонстрировал пример подобного подхода (основанный на примере от разработчиков CODESYS).

В нашем примере мы будем рассматривать третью версию интерфейса – **IClientManagerListener3**. Она включает в себя 4 метода:

- **ClientCreated** – событие о подключении нового клиента визуализации;
- **ClientDestroyed** – событие об отключении клиента визуализации;
- **ClientSizeChanged** – событие об изменении размера экрана клиента визуализации (например, при изменении размеров вкладки в web-браузере);
- **StartVisualizationSet** – установка стартовой визуализации для подключившегося клиента визуализации.

Ссылка на готовый пример: скачать

4.2. Обзор примера

Пример содержит три экрана визуализации:

- **CommonStartVisualization** – «дефолтный» экран визуализации, указанный в качестве стартового и в компоненте **Таргет-визуализация**, и в компоненте **Web-визуализация**;
- **TargetStartVisualization** – стартовый экран визуализации для таргет-визуализации (конечно, можно было бы сразу выбрать его в **компоненте Таргет-визуализация** – но не придирайтесь, это просто пример для демонстрации работы конкретного функционала);
- **WebStartVisualization** – стартовый экран визуализации для клиент web-визуализации с IP-адресом **10.2.8.133** (это IP-адрес моего ПК; замените его на свой).

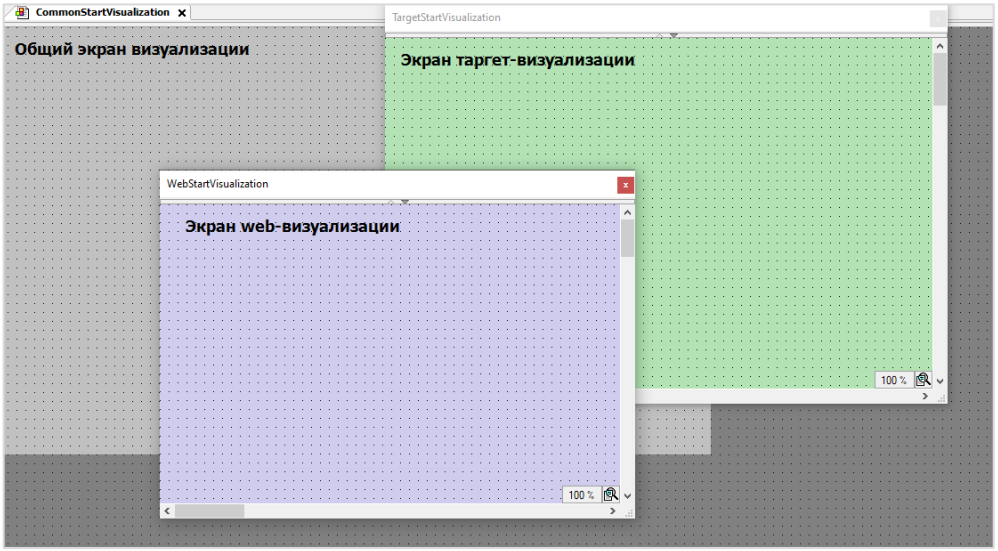


Рисунок 4.2.1 – Экраны визуализации примера

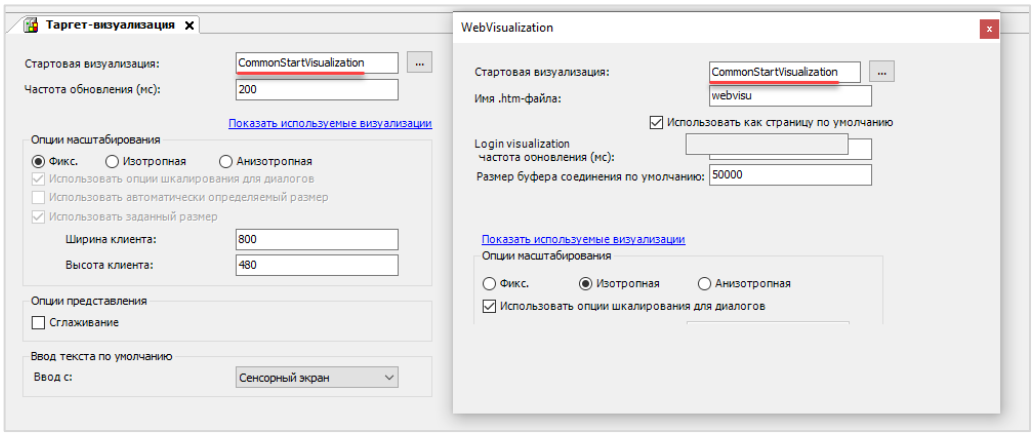


Рисунок 4.2.2 – Настройки стартового экрана визуализации

В рамках примера мы:

- организуем счетчик подключенных клиентов визуализации;
- организуем счетчик изменений размера дисплеев клиентов;
- организуем выбор стартового экрана:
  - для пользователя таргет-визуализации стартовым будет экран **TargetStartVisualization**;
  - для пользователя web-визуализации IP-адресом **10.2.8.133** (у вас он будет своим) стартовым будет экран **WebStartVisualization**;
  - для пользователей web-визуализации с другими IP стартовым будет экран **CommonStartVisualization**.

Функционал библиотеки **VisuElemBase** следует использовать в задаче VISU_TASK. В нашем примере весь код будет размещен в программе **VISU_LOGIC**, привязанной к этой задаче.

4.3. Описание примера

Начнем с создания ФБ **VisuClientManagerListener**, реализующего (**IMPLEMENTS**) интерфейс **VisuElems.VisuElemBase.IClientManagerListener3** (как обычно, название интерфейса не влезает полностью в окно создания POU):

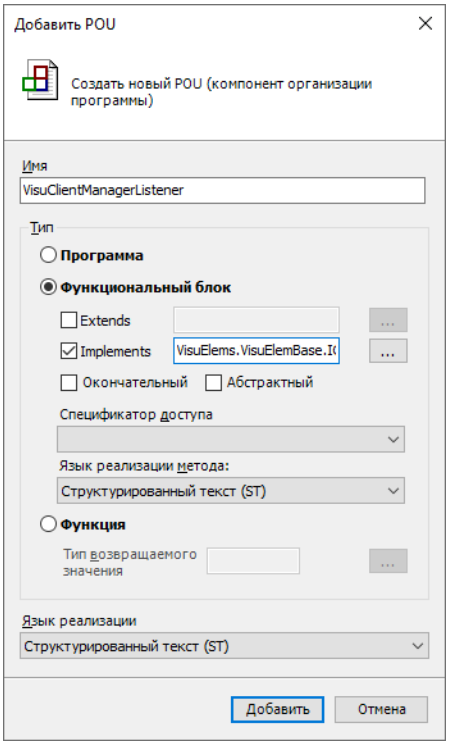


Рисунок 4.3.1 – Создание ФБ, реализующего интерфейс **VisuElems.VisuElemBase.IClientManagerListener3**

У созданного ФБ будет присутствовать только один метод – **ClientSizeChanged**. Дело в том, что интерфейс **IClientManagerListener3** действительно содержит прототип только этого метода – а остальные наследует (**EXTENDS**) от интерфейсов **IClientManagerListener2** и **IClientManagerListener**. Поэтому оставшиеся методы (**ClientCreated**, **ClientDestroyed** и **StartVisualizationSet**) добавим самостоятельно:

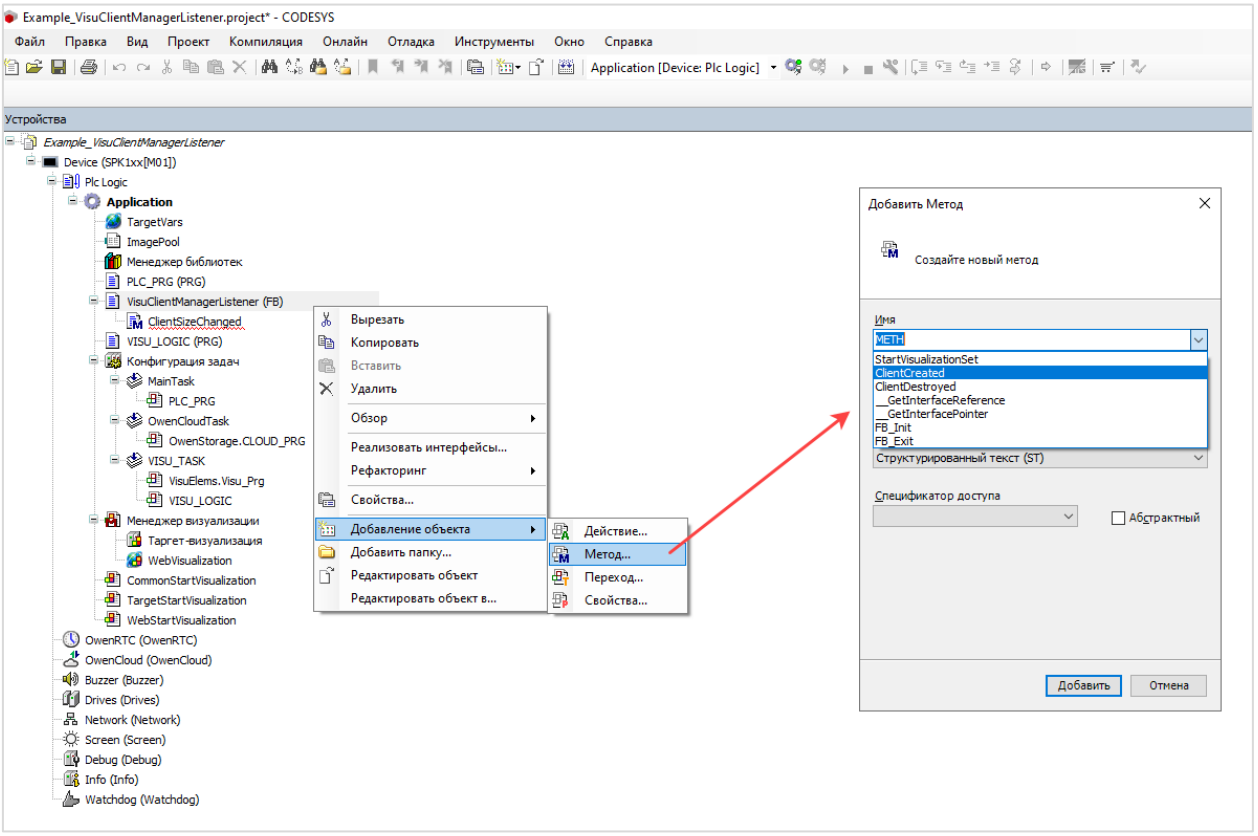


Рисунок 4.3.2 – Добавление методов родительских интерфейсов

Методы **__GetInterfaceReference** и **__GetInrerfacePointer** являются системными (это можно определить по префиксу «__») и поэтому нам не интересны – все равно использовать их не получится. Методы **FB_Init** и **FB_Exit** являются «стандартными» – их можно добавить для любого ФБ. Метод **FB_Init** автоматически вызывается в процессе инициализации (например, при загрузке приложения), а метод **FB_Exit** – при очистке ресурсов (например, при удалении приложения или загрузке нового приложения). В данном примере мы не будем их использовать.

Теперь нужно устранить ошибки компиляции – мы уже делали это в прошлых пунктах, поэтому помним, что для этого нужно в каждом методе к типу **VisuStructClientData** добавить пространство имен **VisuElems.VisuElemBase**:

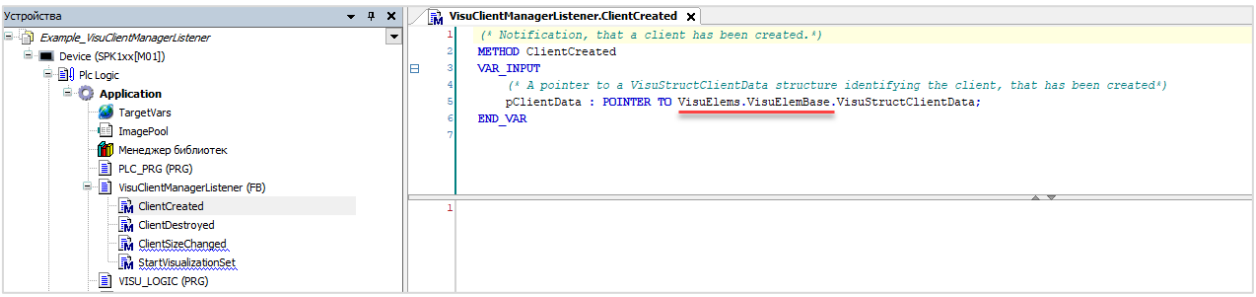


Рисунок 4.3.3 - Указываем пространство имен для **VisuStructClientData**

Объявим в ФБ указатели на две переменных-счетчика:

```
FUNCTION_BLOCK VisuClientManagerListener IMPLEMENTS
   VisuElems.VisuElemBase.IClientManagerListener3
VAR_INPUT
   // Указатель на счетчик клиентов визуализации
   pudiVisuClientCouter:       POINTER TO UDINT;
   // Указатель на счетчик количества изменений разрешения экранов клиентов
   pudiVisuClientSizeChanged:  POINTER TO UDINT;
END_VAR
VAR_OUTPUT
END_VAR
VAR
END_VAR
```

Теперь добавим реализацию методов. Методы **ClientCreated**, **ClientDestroyed** и **ClientSizeChanged** будут содержать всего по одной строке, в которой будет производиться нужная операция со счетчиком:

```
METHOD ClientCreated : BOOL
VAR_INPUT
   pClientData    : POINTER TO VisuElems.VisuElemBase.VisuStructClientData;
END_VAR

pudiVisuClientCouter^ := pudiVisuClientCouter^ + 1;
```

```
METHOD ClientDestroyed : BOOL
VAR_INPUT
   pClientData    : POINTER TO VisuElems.VisuElemBase.VisuStructClientData;
END_VAR

pudiVisuClientCouter^ := pudiVisuClientCouter^ - 1;
```

```
METHOD ClientSizeChanged : BOOL
VAR_INPUT
   pClientData    : POINTER TO VisuElems.VisuElemBase.VisuStructClientData;
END_VAR

pudiVisuClientSizeChanged^ := pudiVisuClientSizeChanged^ + 1;
```

В методе **StartVisualizationSet** мы будем определять стартовый экран клиента визуализации. Напомним, что в рамках примера мы выбрали следующую логику:

- для пользователя таргет-визуализации стартовым будет экран **TargetStartVisualization**;
- для пользователя web-визуализации IP-адресом **10.2.8.133** (у вас он будет своим) стартовым будет экран **WebStartVisualization**;
- для пользователей web-визуализации с другими IP стартовым будет экран **CommonStartVisualization**.

```
METHOD StartVisualizationSet : BOOL
VAR_INPUT
   pClientData   : POINTER TO VisuElems.VisuElemBase.VisuStructClientData;
   // название стартового экрана, заданное в дереве проекта
   // в узле Таргет-визуализация или WebVisualization – в зависимости от типа клиента
   stStartVisu   : STRING;
END_VAR
VAR
   sIpAddr:                  STRING;
   xIsIpAddr:                BOOL;
   fbClientTagDataHelper:    VisuElems.VisuElemBase.VisuFbClientTagDataHelper;
END_VAR


fbClientTagDataHelper(pClientData := pClientData,
   xIPv4Valid => xIsIpAddr, stIPv4 => sIpAddr);

IF pClientData^.GlobalData.ClientType =
   VisuElems.VisuElemBase.Visu_ClientType.TargetVisualization THEN

   VisuElems.VisuElemBase.g_VisuManager.SetMainVisu(pClientData,
     'TargetStartVisualization');

ELSIF pClientData^.GlobalData.ClientType =
   VisuElems.VisuElemBase.Visu_ClientType.WebVisualization AND xIsIpAddr AND
      sIpAddr = '10.2.8.133' THEN

   VisuElems.VisuElemBase.g_VisuManager.SetMainVisu(pClientData,
      'WebStartVisualization');

ELSE
   VisuElems.VisuElemBase.g_VisuManager.SetMainVisu(pClientData,
      'CommonStartVisualization');

END_IF
```

Сначала мы извлекаем из контекста клиента информацию об его IP-адресе с помощью экземпляра ФБ VisuElems.VisuElemBase.VisuFbClientTagDataHelper. Если у клиента нет IP-адреса (например, это касается клиента таргет-визуализации) – то выход **xIPv4Valid** будет иметь значение **FALSE**.

Далее мы проверяем тип клиента (pClientData^.GlobalData.ClientType) и в зависимости от него (а также от IP-адреса) организуем переключения клиента на нужный экран с помощью метода **VisuElems.VisuElemBase.g_VisuManager.SetMainVisu**.

Компилятор выдаст предупреждение, что этот метод признан устаревшим и порекомендует использовать функциональный блок FbChangeVisu из библиотеки **Visu Utils**:

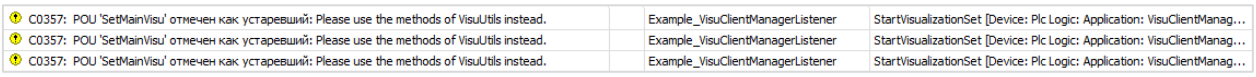


Рисунок 4.3.4 – Предупреждение компилятора об использовании устаревшего метода

Но тут начинаются проблемы – мы помним, что ФБ из **Visu Utils** являются асинхронными и их выполнение может занять несколько циклов контроллера; с другой стороны, методы **IClientManagerListener**'а вызываются только на один цикл задачи **VISU_TASK**. Кроме того, для переключения конкретного экрана для конкретного клиента с заданным IP-адресом нам пришлось бы настроить фильтр клиентов. В общем, на мой взгляд, в данной ситуации разумнее проигнорировать предупреждение компилятора и использовать «устаревшие» методы, рассчитывая на то, что в будущем разработчики CODESYS добавят в библиотеку **Visu Utils** аналогичные синхронные функции:

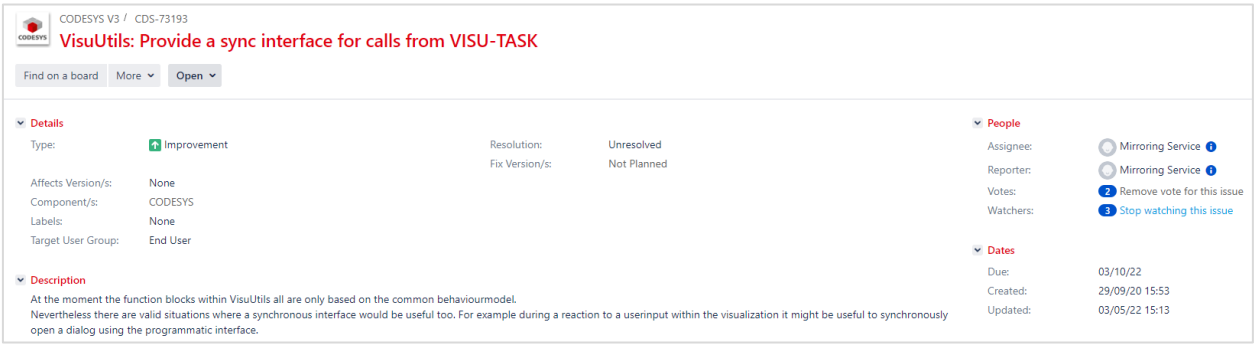


Рисунок 4.3.5 – Пожелание в баг-трекере CODEYS по добавлению в библиотеку **Visu Utils** синхронных функций

Мы реализовали в методе **StartVisualizationSet** довольно простую логику; в вашем проекте вы можете реализовать свою. Для этого полезно будет знать, что разрешение экрана клиента можно получить с помощью следующих полей структуры контекста:

- pClientData^.rClientRect.ptBottomRight.iX
- pClientData^. rClientRect.ptBottomRight.iY

Как мы помним, из их значений потребуется вычесть единицу.

В примере мы использовали перечисление VisuElems.VisuElemBase.Visu_ClientType, которое определяет тип клиента визуализации. Помимо таргет-визуализации и web-визуализации оно содержит клиента «сервисной» визуализации (отображаемой в среде разработки при подключении к контроллеру) – см. **VisuElems.VisuElemBase.Visu_ClientType.ProgrammingSystem**.

Теперь перейдем в программу **VISU_LOGIC** (напомним, она осознанно привязана к задаче **VISU_TASK**, потому что именно в этой задаче требуется вызывать функционал библиотеки **VisuElemBase**). Объявим в программе экземпляр нашего ФБ и проведем его инициализацию. В процессе инициализации мы однократно:

- инициализируем экземпляр интерфейса **IClientManagerListener3** экземпляром нашего ФБ;
- зарегистрируем этот экземпляр в подсистеме визуализации с помощью метода **AddClientManagerListener**. Этот метод принадлежит объекту **g_ClientManager**, который объявлен в списке глобальных переменных Visu_Globals библиотеки **VisuElemBase**;
- передаем на входы блока (напомним, они представляют собой указатели) адреса локальных переменных нашей программы.

```
PROGRAM VISU_LOGIC
VAR
   fbVisuClientManagerListener:      VisuClientManagerListener;
   itfClientManagerListener3:        VisuElems.VisuElemBase.IClientManagerListener3;

   udiVisuClientCounter:             UDINT;
   udiVisuClientSizeChangedCounter:  UDINT;

   xInit:                            BOOL;
END_VAR



IF NOT(xInit) THEN

   itfClientManagerListener3 := fbVisuClientManagerListener;

   VisuElems.VisuElemBase.Visu_Globals.g_ClientManager.
      AddClientManagerListener(ADR(itfClientManagerListener3) );

   fbVisuClientManagerListener.pudiVisuClientCouter := ADR(udiVisuClientCounter);
   fbVisuClientManagerListener.pudiVisuClientSizeChanged :=
      ADR(udiVisuClientSizeChangedCounter);

   xInit := TRUE;

END_IF
```

Вообще, подобный «инициализационный» код мы могли бы вынести в отдельную программу (назвав ее, например, **VISU_INIT**) и привязать ее к полю Call **after visu initialization** в **Менеджере визуализации** (см. рис. 4.3.6) – тогда она автоматически была бы однократно вызвана в контексте задачи **VISU_TASK** после завершения инициализации подсистемы визуализации CODESYS. Но в этом случае пришлось бы продумывать, как обращаться к переменным этой программы из других программ проекта. Очевидным решением являлось бы использование глобальных переменных – но в п. 1.2.3 я уже описывал минусы такого подхода. Поэтому здесь и далее я буду проводить однократную инициализацию прямо в программе **VISU_LOGIC** с помощью флага **xInit**, значение которого буду менять в своем коде.

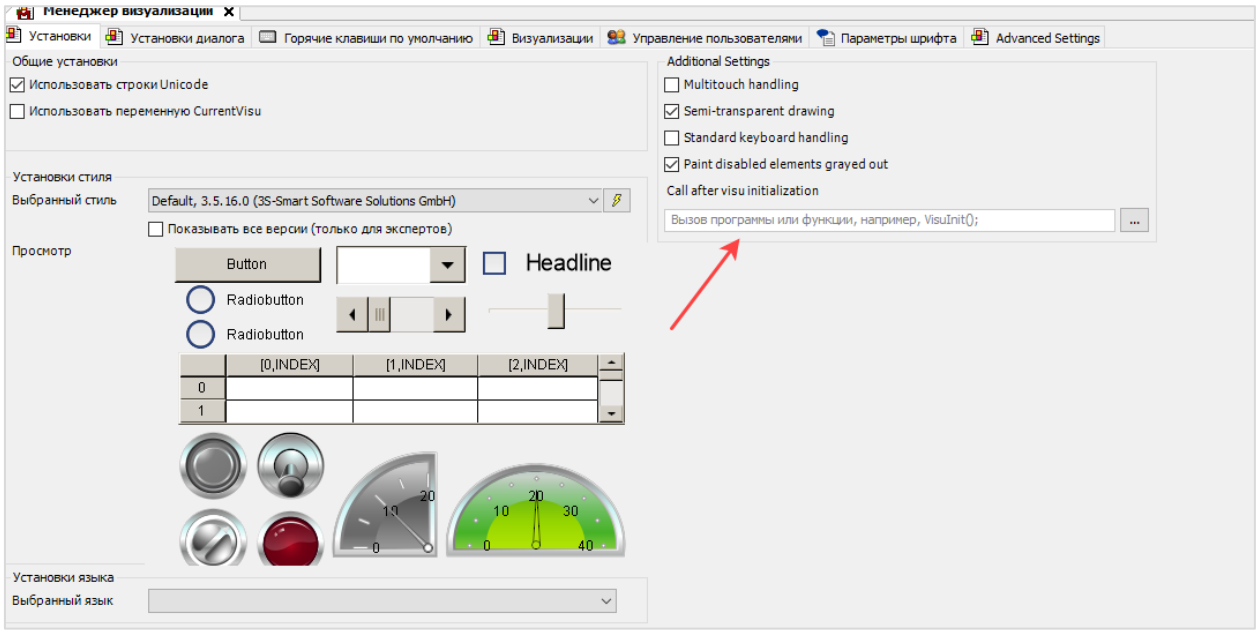


Рисунок 4.3.6 – Поле **Call after visu initialization** в менеджере визуализации

Проект полностью готов. Давайте загрузим его в контроллер и проверим, как он работает.

4.4. Проверка работы примера

После запуска проекта примера на экране панельного контроллера отобразится экран **TargetStartVisualization**. При этом счетчик клиентов (**udiVisuClientCounter**) останется равным **0**, а счетчик числа изменений разрешения клиентов (**udiVisuClientSizeChangedCounter**) будет равным **2** – это интересное наблюдение, которое проявляется на устройствах с таргет-визулизацией и которое я не могу прокомментировать.

Откройте web-визуализацию контроллера на ПК (или другом устройстве) с IP-адресом, который вы задали в коде метода **StartVisualizationSet** (в моем случае это **10.2.8.133**, у вас он будет своим). В web-браузере отобразится экран **WebStartVisualization**, а значения **udiVisuClientCounter** и **udiVisuClientSizeChangedCounter** увеличатся на **1**.

Откройте web-визуализацию контроллера на устройстве с другим IP-адресом или же откройте любой из экранов в редакторе CODESYS, сохраняя подключение к контроллеру. Отобразится экран **CommonStartVisualization** (даже если в редакторе CODESYS вы попробовали открыть другой экран), а значения **udiVisuClientCounter** и **udiVisuClientSizeChangedCounter** увеличатся на **1**.

Изменяйте размеры вкладки web-браузера или вкладки визуализации в редакторе CODESYS. При каждом изменении значение **udiVisuClientSizeChangedCounter** будет увеличиваться на 1.

При закрытии каждой вкладки web-визуализации в браузере или в редакторе CODESYS значение **udiVisuClientCounter** будет уменьшаться на **1**.

Таким образом можно в коде отследить подключение/отключение/изменение размера экрана клиентов визуализации и определить для них стартовый экран.

5. Работа с фреймами из кода программы (IFrameManager)

5.1. Основная информация

Фрейм – это элемент визуализации CODESYS, который позволяет открыть в плоскости одного экрана визуализации другой. В других средах разработки и ПО (например, SCADA) аналогичный функционал может называться «шаблонами экранов», дашбордами (dashboard), «панелями мониторинга» и т. д. Посмотрите на рисунок 0 – «Отопление», «Подпитка», «Обратка», а также верхний статус-бар и нижняя панель кнопок являются не просто наборами размещенных рядом друг с другом элементов, а фреймами. Вот как выглядит в редакторе CODESYS экран с этого рисунка и один из его фреймов:

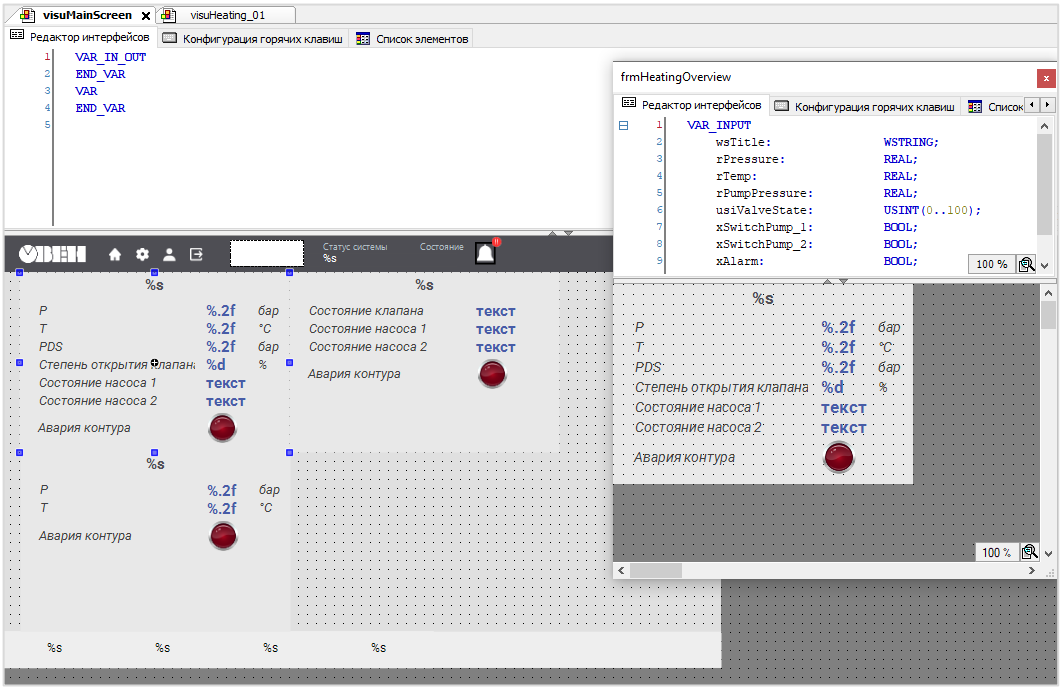


Рисунок 5.1.1 – Пример элемента **Фрейм** (выделен на экране визуализации **visuMainScreen**) и экрана этого фрейма

Есть несколько основных способов использования фреймов:

- отображение в фрейме разных экранов (например, экрана обзорной мнемосхемы технологического объекта, экрана его настройки, экрана тревог и т. д.);
- отображение в фрейме нескольких экземпляров одного экрана с разными значениями переменных (например, экрана объекта 1, экрана объекта 2 и т. д. – при этом все объекты представлены одним экраном, и при переключении экранов фрейма происходит просто «подстановка» в них значений параметров нужного объекта);
- отображение в фрейме одного экрана. Этот подход используется, чтобы избежать копирования одних и тех же элементов по разным экранам – см., например, статус-бар на рисунке выше. Если потребуется внести изменения в статус-бар – то достаточно будет сделать это на экране его фрейма, а не редактировать все экраны проекта.

В конфигурации элемента **Фрейм** добавляются экраны, которые можно будет отображать в этом фрейме. Далее я буду называть их «**экранами фрейма**». Каждому экрану автоматически присваивается индекс; нумерация индексов ведется с **0**. Я буду называть их «**индексами экранов фрейма**».

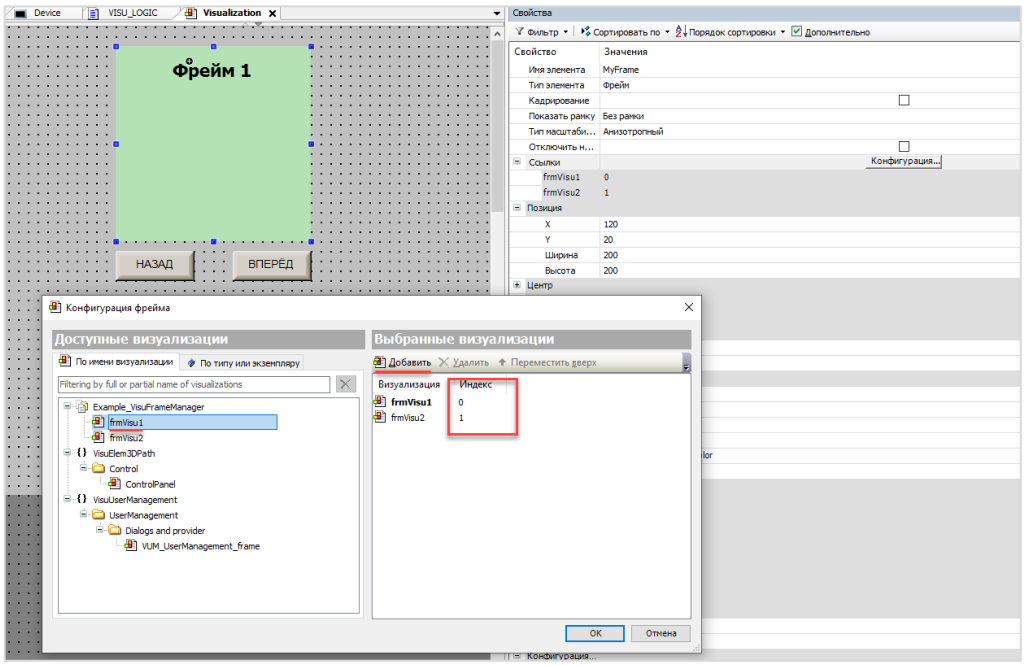


Рисунок 5.1.2 – Пример конфигурации фрейма

Подсистема визуализации CODESYS размещает все элементы **Фрейм** проекта CODESYS в массиве. Для нас неважно, как именно это организовано, но следует знать, что нумерация этого массива ведется с **0**. Индекс этого массива для определенного фрейма я буду называть «**индексом фрейма**». Не путайте их с индексами экранов фрейма (см. выше)!

При работе с фреймами из кода программы будет иметь значения **имя элемента**, заданное в одноименном свойстве элемента визуализации. Это один из тех редких случаев, когда это имя имеет значение для разработчика.

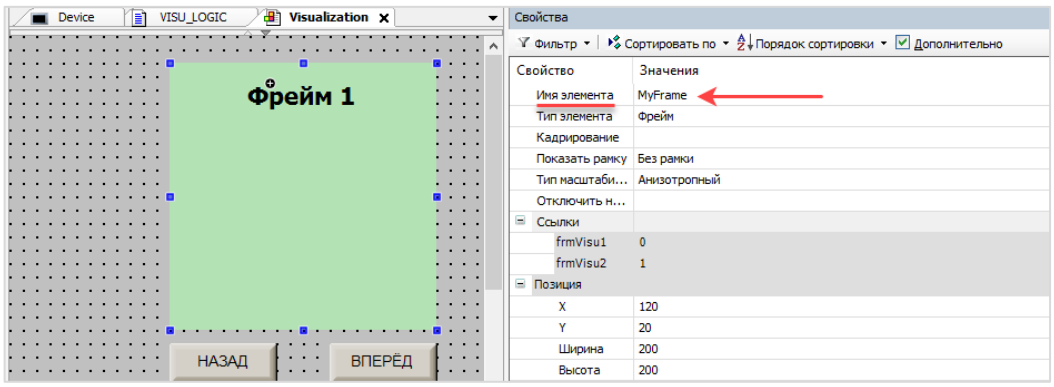


Рисунок 5.1.3 – Имя элемента **Фрейм**

В этом пункте я выделил **жирным** термины, которую буду использовать в дальнейшем.

5.2. Обзор примера

Пример содержит экран визуализации **Visualization**, на котором размещен элемент **Фрейм** с именем **MyFrame** (см. рис. 5.1.3). В конфигурации элемента настроено два экрана фрейма (см. рис. 5.1.2):

- **frmVisu1** с индексом **0**;
- **frmVisu2** с индексом **1**.

Под фреймом расположены кнопки «ВПЕРЁД»/«НАЗАД», которые предназначены для переключения экранов фрейма оператором. В конфигурации ввода этих кнопок настроено действие **Переключить визуализацию фрейма**, выбран элемент визуализации **MyFrame**, выбран тип содержания **Переключать визуализацию по шагам** и тип действия **На следующий** (для кнопки «ВПЕРЁД) и **На предыдущий** (для кнопки «НАЗАД»). Таким образом, кнопки позволяют листать экраны фрейма «по кругу» в обоих направлениях.

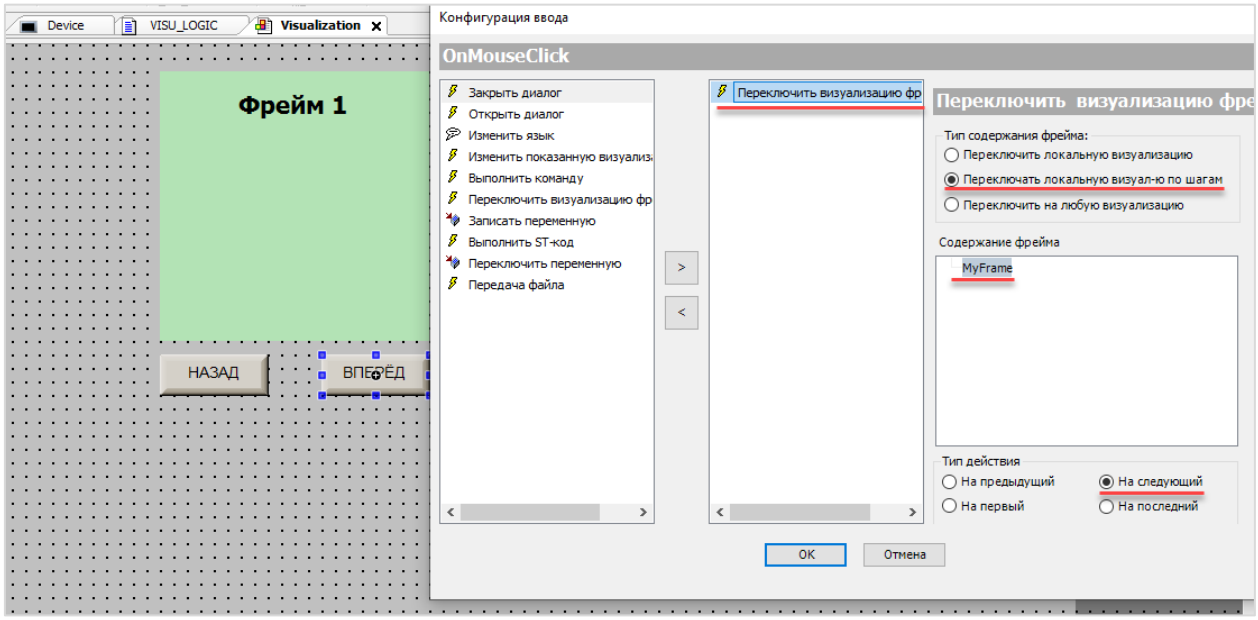


Рисунок 5.2.1 – Настройки действий кнопки «ВПЕРЁД»

Поговорим немного про другие «типы содержания»:

- **Переключить локальную визуализацию** – позволяет для элемента выбрать конкретный экран фрейма из списка всех экранов элемента. При нажатии на кнопку будет произведено открытие в элементе этого экрана (см. рис. 5.2.2);
- **Переключить на любую визуализацию** – позволяет для выбранного из списка элемента (**Присвоить**) или элемента, чье имя задано через строковую переменную (**Присвоить выражение**) указать переменную типа **INT** (**индекс для выбора**), с помощью изменения которой можно будет открыть экран с данным индексом в этом элементе (см. рис. 5.2.3).

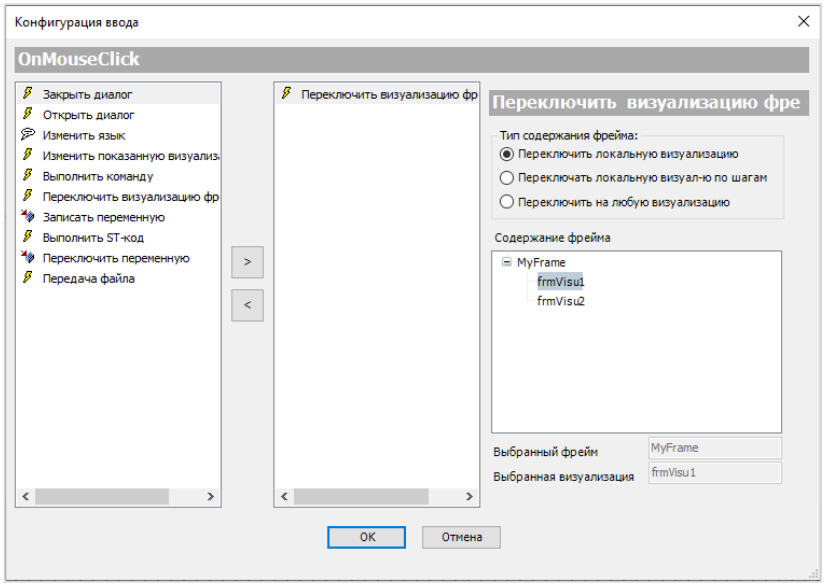


Рисунок 5.2.2 – Настройки для типа содержания «**Переключить локальную визуализацию**»

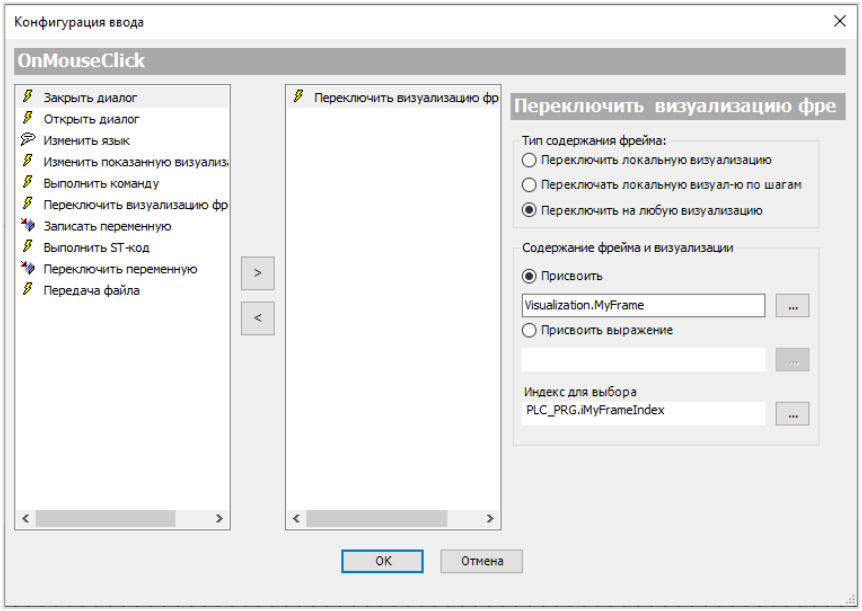


Рисунок 5.2.3 – Настройки для типа содержания «**Переключить на любую визуализацию**»

Последний вариант подходит для переключения экранов фреймов из кода программы, но позволяет сделать это только сразу для всех клиентов визуализации. Поэтому в рамках примера давайте решим следующие задачи:

- считаем информацию о фреймах проекта в переменные программы;
- определим, какой экран в конкретном фрейме открыт сейчас у конкретного клиента;
- организуем переключение экранов в конкретном фрейме для конкретных клиентов.

Готовый пример доступен по ссылке: скачать

5.3. Обзор интерфейса IFrameManager2

Интерфейс IFrameManager2 входит в состав библиотеки **VisuElemBase** и содержит методы, используемые для работы с фреймами из кода программы. Интерфейс наследует интерфейсы IFrameManager и IFrameManagerBase. Далее приведено описание методов интерфейса.

5.3.1. Метод GetFrameCount

Метод **GetFrameCount** возвращает число элементов **Фрейм**, размещенных на экранах визуализации данного проекта.

Рисунок 5.3.1.1 – Сигнатура метода **GetFrameCount**

5.3.2. Метод GetRegisterFrames

Метод **GetRegisterFrames** возвращает число элементов **Фрейм**, размещенных на экранах визуализации данного проекта, а также имена этих элементов, размещая их в массиве строк по указателю **pDataResult**. Вход **iSize** определяет число элементов массива, выделенное пользователем под хранение этой информации (а не размер массива в байтах, как может показаться из названия).

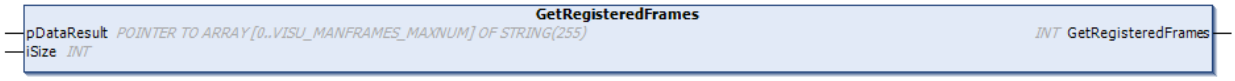


Рисунок 5.3.2.1 – Сигнатура метода **GetRegisterFrames**

Константа **VisuElems.VisuElemBase.VISU_MANFRAMES_MAXNUM** в прошлых версиях CODESYS определяла максимальное число элементов **Фрейм** в пользовательском проекте. В современных версиях это ограничение было снято.

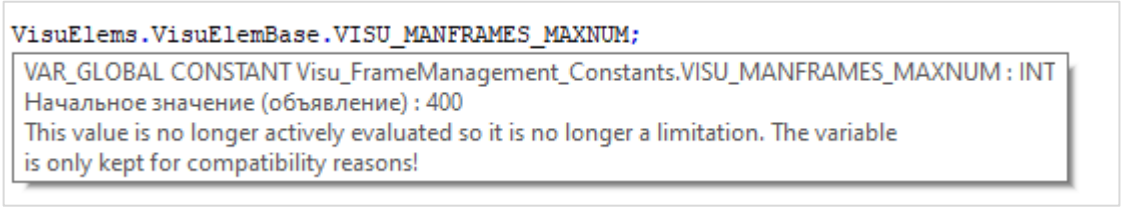


Рисунок 5.3.2.2 – Информация о константе **VisuElems.VisuElemBase.VISU_MANFRAMES_MAXNUM**

5.3.3. Метод GetVisuCount

Метод **GetVisuCount** возвращает число экранов, добавленных в конфигурации элемента **Фрейм** с путем **stFramePath**. Путь как минимум включает в себя имя экрана визуализации, на котором размещен фрейм, и имя элемента. Например, если фрейм с именем **MyFrame** размещен на экране визуализации **Visualization**, то путь к нему будет выглядеть как **Visualization.MyFrame**. Кроме того, путь может включать в себя пространство имен библиотеки (если фрейм создан в библиотеке) и имена других фреймов (если элемент **Фрейм** размещен внутри другого элемента **Фрейм**).

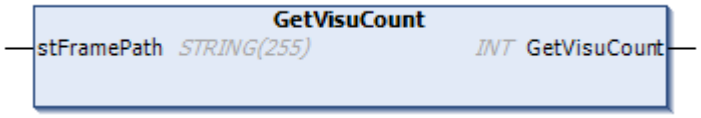


Рисунок 5.3.3.1 – Сигнатура метода **GetVisuCount**

5.3.4. Метод GetVisuName

Метод **GetVisuName** возвращает имя экрана с индексом **iIndex**, добавленного в конфигурации элемента **Фрейм** с путем **stFramePath** (подробнее про путь см. в п. 5.3.3). Если вход **bFullName** имеет значение **TRUE**, то метод вернет строку, которая помимо имени экрана будет включать пространства имен (например, если экран создан внутри библиотеки). Если вход **bFullName** имеет значение **FALSE**, то метод вернет только имя экрана без пространств имен.

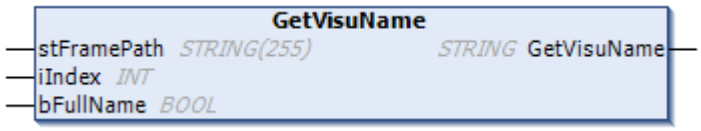


Рисунок 5.3.4.1 – Сигнатура метода **GetVisuName**

5.3.5. Метод GetSelectedVisu

Метод **GetSelectedVisu** возвращает индекс экрана, открытого в фрейме с путем **stFramePath** (подробнее про путь см. в п. 5.3.3) у клиента визуализации, определяемого указателем на контекст **pClientData**.

Рисунок 5.3.5.1 – Сигнатура метода **GetSelectedVisu**

5.3.6. Методы RegisterFrame и UnregisterFrame

Метод **RegisterFrame** используется для регистрации элемента визуализации, реализующего интерфейс IFrameElement и имеющего экземпляр этого интерфейса **frame**, по пути **stPath** (подробнее про путь см. в п. 5.3.3) в подсистеме визуализации CODESYS. Метод **UnregisterFrame** выполняет процедуру дерегистрации. В целом, не подразумевается, что эти методы будет использовать разработчик приложения (они нужны самой подсистеме визуализации CODESYS). Интересный диалог по этому поводу можно прочитать здесь (и посмотреть в приложенном проекте openVis.project попытку использования этого метода). Сотрудник **CODESYS Group** по **имени Marcel Prestel** (уже неоднократно упоминаемый на страницах документа) делает ценное замечание – в текущих версиях CODESYS нет возможности из кода программы изменить список экранов, привязанных к элементу **Фрейм**.

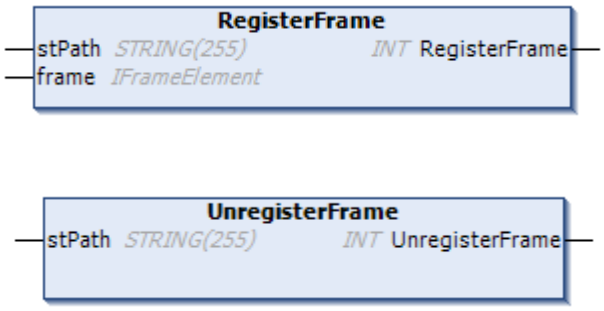


Рисунок 5.3.6.1 – Сигнатура методов **RegisterFrame** и **UnregisterFrame**

5.3.7. Метод SwitchToVisuGlobally

Метод **SwitchToVisuGlobally** открывает в элементе **Фрейм** с путем **stFramePath** (подробнее про путь см. в п. 5.3.3) экран с индексом **iIndex** для всех клиентов категории **clientsToChange** (типа Visu_ClientType). Категории могут комбинироваться с помощью оператора OR – мы рассмотрим это в нашем примере.

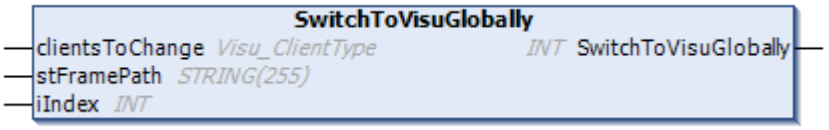


Рисунок 5.3.7.1 – Сигнатура метода **SwitchToVisuGlobally**

5.3.8. Метод SwitchToVisu

Метод **SwitchToVisu** открывает в элементе **Фрейм** с путем **stFramePath** (подробнее про путь см. в п. 5.3.3) экран с индексом **iIndex** для клиента визуализации, определяемого указателем на контекст **pClientData**.

Рисунок 5.3.8.1 – Сигнатура метода **SwitchToVisu**

5.3.9. Метод SwitchToVisu2

Метод **SwitchToVisu2** открывает в элементе **Фрейм** с путем **stFramePath** (подробнее про путь см. в п. 5.3.3) экран с индексом **iIndex** для клиента визуализации, определяемого указателем на контекст **pClientData**. Если вход **xUpdVar** имеет значение **TRUE**, то при открытии также будет произведено обновление переменных интерфейса фрейма (**VAR_INPUT** и **VAR_IN_OUT**). Если вход **xUpdLastIndex** имеет значение **TRUE**, то этот экран для данного фрейма будет помечен как «последний открытый» (это нужно при использовании технологии **CODESYS Redundancy** для синхронизации визуализаций резервируемых контроллеров).

Рисунок 5.3.9.1 – Сигнатура метода **SwitchToVisu2**

5.3.10. Метод GetFrameByIndex

Метод **GetFrameByIndex** возвращает имя элемента **Фрейм** на основании его индекса в массиве фреймов (**index**). Массив может быть получен с помощью метода GetRegisterFrames.

Рисунок 5.3.10.1 – Сигнатура метода **GetFrameByIndex**

5.3.11. Метод GetSelectedVisuByIndex

Метод **GetSelectedVisuByIndex** является аналогом метода GetSelectedVisu, но вместо пути фрейма используется его индекс в массиве фреймов (**index**), который может быть получен с помощью метода GetRegisterFrames.

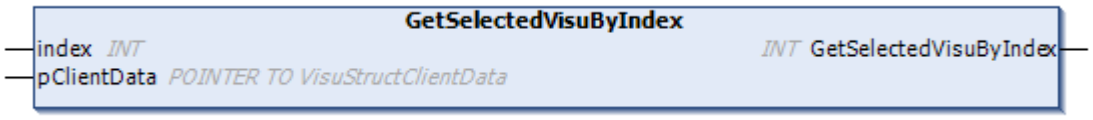


Рисунок 5.3.11.1 – Сигнатура метода **GetSelectedVisuByIndex**

5.3.12. Метод SwitchToVisuByIndex

Метод **SwitchToVisuByIndex** является аналогом метода SwitchToVisu2, но вместо пути фрейма используется его индекс в массиве фреймов (**index**), который может быть получен с помощью метода GetRegisterFrames.

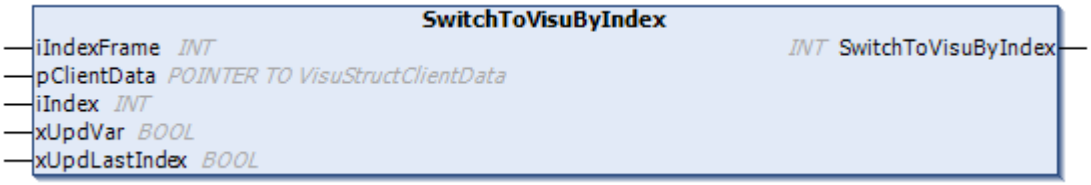


Рисунок 5.3.12.1 – Сигнатура метода **SwitchToVisuByIndex**

5.4. Описание примера

Некоторые из рассмотренных в прошлом пункте методов (например, GetSelectedVisu и SwitchToVisu2) требуют в качестве одного из аргументов указатель на контекст клиента визуализации. Поэтому данный пример основан на примере получения информации о клиентах визуализации из п. 1.2.

Для начала я покажу переменные примера:

```
// Программа привязана к задаче VISU_TASK
PROGRAM VISU_LOGIC
VAR
   // ФБ сбора информации о клиентах
   fbGetVisuClientsInfo:                  VU.FbIterateClients;
   // Команда сбора информации о клиентах
   xExecute:                              BOOL;
   // ФБ обработки клиентов
   fbClientIterationCallback:             VisuClientIteration;
   // Массив информации о клиентах
   astVisuClientInfo:                     ARRAY
      [1..Visu_Superglobal_Constants.VISU_MAX_NUMBER_OF_CLIENTS] OF VISU_CLIENT_DATA;

   // Экземпляры интерфейсов FrameManager'a
   // Мы получаем IFrameManager в вызове GetFrameManager() и конвертируем его в
   // IFrameManager2
   itfFrameManager:                       VisuElems.VisuElemBase.IFrameManager;
   itfFrameManager2:                      VisuElems.VisuElemBase.IFrameManager2;

   // Число элементов Фрейм в проекте пользователя
   iFrameInApplicationCount:              INT;
   // Число экранов для заданного фрейма
   iVisuInFrameCount:                     INT;
   // Массив названий элементов Фрейм для всего проекта
   asRegisterFrameElementNames:   ARRAY [1..c_iMaxFrameElementCount] OF STRING(255);
   // Индексы экранов, открытых в заданном фрейме, для всех клиентов визуализации
   aiSelectedVisuOfFrameVisuIndex:        ARRAY
     [1..Visu_Superglobal_Constants.VISU_MAX_NUMBER_OF_CLIENTS] OF INT :=
       [Visu_Superglobal_Constants.VISU_MAX_NUMBER_OF_CLIENTS(-1)];
   // Названия экранов, открытых в заданном фрейме, для всех клиентов визуализации
   aiSelectedVisuOfFrameVisuName:         ARRAY
      [1..Visu_Superglobal_Constants.VISU_MAX_NUMBER_OF_CLIENTS] OF STRING;

   // Команда открытия в фрейме экрана frmVisu2 для клиента таргет-визуализации
   xSwitchTargetClientToFrmVisu2:         BOOL;
   // Команда открытия в фрейме экрана frmVisu1 для всех клиентов web-визуализации
   xSwitchWebClientsToFrmVisu1:           BOOL;
   // Команда открытия в фрейме экрана frmVisu1 для всех клиентов всех типов
   // визуализации
   xSwitchAllClientsToFrmVisu1:           BOOL;

   // Команда инициализации
   xInit:                                 BOOL;
   // Счетчик циклов
   i:                                     INT;
END_VAR
VAR CONSTANT
   // Имя элемента Фрейм на экране Visualization (см. свойство "Имя элемента")
   c_sFrameElementName:                    STRING := 'Visualization.MyFrame';
   // Максимальное число элементов Фрейм, которое мы планируем использовать в проекте
   c_iMaxFrameElementCount:                INT    := 10;
END_VAR
```

Константа **c_sFrameElementName** содержит имя элемента **Фрейм**, используемого в нашем примере (см. рис. 5.1.3), а константа **c_iMaxFrameElementCount** определяет максимальное число элементов **Фрейм**, которое мы планируем использовать в нашем проекте (она используется для задания верхней границы массива имен элементов типа Фрейм **asRegisterFrameElementNames**, который мы будет заполнять в вызове метода GetRegisterFrames).

Теперь рассмотрим несколько первых фрагментов кода:

```
IF NOT(xInit) THEN

   // передаем адрес массива, который будет заполняться информацией о клиентах
   // визуализации
   fbClientIterationCallback.pstVisuClientData := ADR(astVisuClientInfo);

   // получаем экземпляр FrameManager'a
   itfFrameManager := VisuElems.VisuElemBase.g_VisuManager.GetFrameManager();

   // конвертируем его к itfFrameManager2, чтобы иметь возможность вызывать
   // SwitchToVisu2 и другие методы интерфейса IFrameManager2
   __QUERYINTERFACE(itfFrameManager, itfFrameManager2);

   // определяем число элементов Фрейм в проекте
   iFrameInApplicationCount :=
      itfFrameManager2.GetRegisteredFrames(ADR(asRegisterFrameElementNames),
         c_iMaxFrameElementCount);
   // определяем число экранов в нашем фреймее
   iVisuInFrameCount := itfFrameManager2.GetVisuCount(c_sFrameElementName);

   xInit := TRUE;

END_IF

fbGetVisuClientsInfo
(
   xExecute             := xExecute,
   itfClientFilter      := VU.Globals.AllClients,
   itfIterationCallback := fbClientIterationCallback
);

// информация о всех интересующих клиентах получена
IF fbGetVisuClientsInfo.xDone THEN

   // определяем для каждого клиента индекс и имя экрана, открытого в фрейме
   FOR i := 1 TO fbClientIterationCallback.iCurrentClientCount DO

      aiSelectedVisuOfFrameVisuIndex[i]  :=
         itfFrameManager2.GetSelectedVisu(c_sFrameElementName,
            astVisuClientInfo[i].pstClientData);

      aiSelectedVisuOfFrameVisuName[i]   :=
         itfFrameManager2.GetVisuName(c_sFrameElementName,
            aiSelectedVisuOfFrameVisuIndex[i], TRUE);

   END_FOR

END_IF
```

При первом вызове программы **VISU_LOGIC** мы однократно выполняем ряд действий, используя для этого переменную **xInit**:

- передаем в **fbClientIterationCallback** (экземпляр ФБ **VisuClientIteration**) адрес массива информации о клиентах (**astVisuClientInfo**);
- обращаемся к объекту **g_VisuManager** из списка глобальных переменных Visu_Globals библиотеки **VisuElemBase** и вызываем его метод **GetFrameManager()**, присваивая результат в локальную переменную **itfFrameManager**. Таким образом, мы инициализируем наш экземпляр интерфейса и далее сможем вызывать его методы. Но в вызове **GetFrameManager()** мы получаем экземпляр интерфейса IFrameManager; если нужно использовать методы интерфейса **IFrameManager2** – то следует выполнить приведение интерфейсов с помощью оператора __QUERYINTERFACE. После его вызова экземпляр интерфейса itfFrameManager2 будет инициализирован – и теперь с помощью него мы сможет вызывать и методы интерфейса **IFrameManager2**, и методы интерфейса **IFrameManager** (потому что интерфейс **IFrameManager2** наследует интерфейс **IFrameManager**);
- мы сразу вызываем методы GetRegisteredFrames (для получения имен элементов **Фрейм** нашего проекта) и GetVisuCount (для получения числа экранов в нашем фрейме с именем **MyFrame**), потому что эта информация не может измениться в процессе работы приложения.

Далее по команде пользователя (по переднему фронту локальной переменной **xExecute**) будет вызываться **fbGetVisuCleintsInfo** (экземпляр блока **FbIterateClients**), в результате чего массив **astVisuClientInfo** будет заполнен информацией о клиентах визуализации. Когда это произойдет (то есть когда выход **xDone** примет значение **TRUE**) – мы пройдемся по массиву информации о клиентах (число клиентов при этом записывается на выход **iCurrentClientCount** экземпляра блока **VisuClientIteration** с именем **fbClientIterationCallback**) и для каждого клиента вызовем методы GetSelectedVisu и GetVisuName. В первый метод мы передаем указатель на контекст клиента, получая индекс экрана, открытого в нашем фрейме для данного клиента, а с помощью второго метода конвертируем этот индекс в имя соответствующего экрана).

Кроме того, мы обрабатываем 3 команды переключения экранов фрейма:

```
// открываем в фрейме экран frmVisu2 для клиента таргет-визуализации
IF xSwitchTargetClientToFrmVisu2 THEN

   FOR i := 1 TO fbClientIterationCallback.iCurrentClientCount DO

      IF astVisuClientInfo[i].eClientType = VU.VisuClientType.TargetVisualization THEN

         // 1 - индекс экрана frmVisu2 (см. конфигурацию фрейма)
        itfFrameManager2.SwitchToVisu2(c_sFrameElementName,
          astVisuClientInfo[i].pstClientData, 1, FALSE, FALSE);

      END_IF

   END_FOR

   xSwitchTargetClientToFrmVisu1 := FALSE;

END_IF
```

```
// открываем в фрейме экран frmVisu1 для клиента web-визуализации
IF xSwitchWebClientsToFrmVisu1 THEN

   FOR i := 1 TO fbClientIterationCallback.iCurrentClientCount DO

      IF astVisuClientInfo[i].eClientType = VU.VisuClientType.WebVisualization THEN

      // 0 - индекс экрана frmVisu1 (см. конфигурацию фрейма)
      itfFrameManager2.SwitchToVisu2(c_sFrameElementName,
         astVisuClientInfo[i].pstClientData, 0, FALSE, FALSE);

      END_IF

   END_FOR

   xSwitchWebClientsToFrmVisu2 := FALSE;

END_IF


// открываем в фрейме экран frmVisu1 для всех клиентов всех типов визуализации
IF xSwitchAllClientsToFrmVisu1 THEN

   itfFrameManager2.SwitchToVisuGlobally
   (
      // формируем битовую маску типов клиентов визуалиазации
      clientsToChange := VisuElems.VisuElemBase.Visu_ClientType.ProgrammingSystem OR
         VisuElems.VisuElemBase.Visu_ClientType.TargetVisualization OR
            VisuElems.VisuElemBase.Visu_ClientType.WebVisualization,
      stFramePath := c_sFrameElementName,
      // 0 - индекс экрана frmVisu1 (см. конфигурацию фрейма)
      iIndex              := 0
   );

  xSwitchAllClientsToFrmVisu1 := FALSE;

END_IF
```

Первые две команды обрабатываются аналогичным образом – мы проходимся по массиву информации о клиентах и для клиентов нужного типа (для первой команды это таргет-визуализация, для второй – web-визуализация) вызываем метод SwitchToVisu2, передавая ему указатель на контекст клиента и индекс экрана, который нужно для этого клиента открыть. Описание остальных входов метода приведено в п. 5.3.9. В своем проекте вы можете реализовать нужную вам логику – например, переключить экран фрейма для клиента web-визуализации с нужным IP-адресом (для этого потребуется всего лишь добавить в условие внутри IF дополнительные проверки).

Третья команда переключает экран в фрейме для вообще всех клиентов всех типов визуализации – поэтому контекст клиентов (и, соответственно, обход массива **astVisuClientInfo**) здесь не нужен – мы просто вызываем метод SwitchToVisuGlobally. Из интересного тут вход **clientsToChange**, который определяет категорию клиентов, для которой мы переключаем экран фрейма. Этот вход имеет тип Visu_ClientType (перечисление). В перечислении описываются три категории:

- **ProgrammingSystem** (значение **16#01**) – сервисная визуализация, отображаемая в редакторе визуализации при подключении к контроллеру;
- **TargetVisualization** (значение **16#02**) – таргет-визуализация;
- **WebVisualization** (значение **16#04**) – web-визуализация.

Как можно заметить – значение каждого элемента перечисления соответствует одному биту битовой маски (биту 0, 1 и 2 соответственно). Поэтому если требуется переключить экран фрейма для нескольких категорий клиентов сразу – то нужно сформировать битовую маску категорий и передать ее на вход **clientsToChange**. Именно это мы и делаем в вызове метода, комбинируя элементы перечисления через оператор **OR**. Компилятор выдаст предупреждение о неявной конверсии из **INT** в **UINT** (видимо, перечисление основано на типе **INT**) – можете не обращать на него внимания.

Проект полностью готов. Давайте загрузим его в контроллер и проверим, как он работает.

5.5. Проверка работы примера

Сразу после запуска примера в переменную **iFrameInApplicationCount** будет записано значение **1**, а в переменную **iVisuInFrameCount** – значение **2**. Всё логично – в нашем проекте используется один элемент **Фрейм**, в конфигурации которого добавлено два экрана.

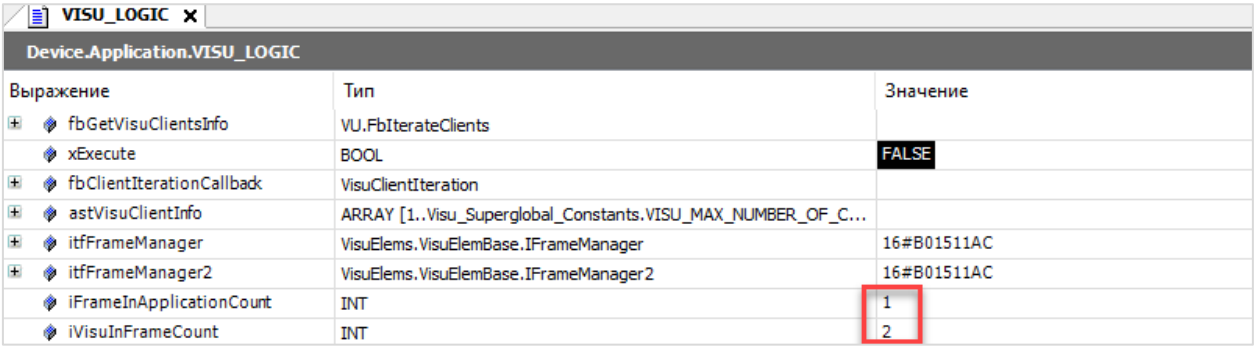


Рисунок 5.5.1 – Число фреймов проекта и числро экранов фрейма с именем **MyFrame**

На экране панельного контроллера в фрейме отображается экран **Фрейм 1** (**frmVisu1**). Подключитесь к web-визуализации контроллера и откройте там в фрейме экран **Фрейм 2** (**frmVisu2**) с помощью кнопки «ВПЕРЁД».

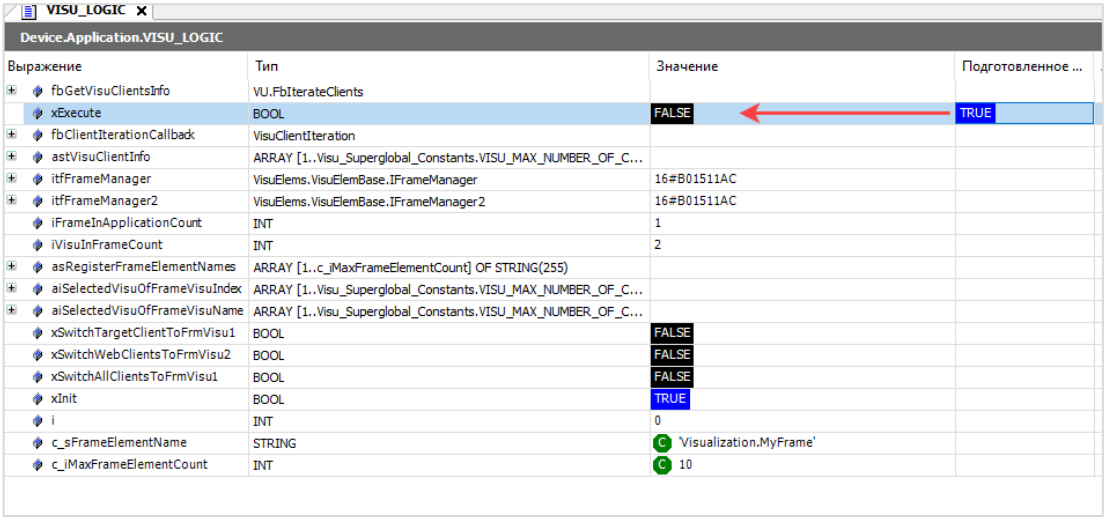


Рисунок 5.5.2 – Выполнение команды сбора информации о клиентах визуализации

Теперь посмотрим содержимое массивов **aiSelectedVisuOfFrameVisuIndex** и **asSelectedVisuOfFrameVisuName**:

aiSelectedVisuOfFrameVisuIndex	ARRAY [1..Visu_Superglobal_Constants.VISU_MAX_NUMBER_OF_C...	
aiSelectedVisuOfFrameVisuI...	INT	0
aiSelectedVisuOfFrameVisuI...	INT	1
aiSelectedVisuOfFrameVisuI...	INT	-1

aiSelectedVisuOfFrameVisuName	ARRAY [1..Visu_Superglobal_Constants.VISU_MAX_NUMBER_OF_C...	
aiSelectedVisuOfFrameVisu...	STRING	'frmVisu1'
aiSelectedVisuOfFrameVisu...	STRING	'frmVisu2'
aiSelectedVisuOfFrameVisu...	STRING	''

Рисунок 5.5.3 – Информация о том, какие экраны фрейма открыты у разных клиентов визуализации

Действительно, именно это и мы и ожидали увидеть. Отметим, что индексы в массивах **aiSelectedVisuOfFrameVisuIndex** и **asSelectedVisuOfFrameVisuName** совпадают с индексами массива информации о клиентах **astVisuClientInfo** – таким образом, вы можете установить между ними соответствие. Вообще, более изящным решением было бы вместо этих двух массивов (**aiSelectedVisuOfFrameVisuIndex** и **asSelectedVisuOfFrameVisuName**) добавить два поля (**iSelectedVisuOfFrameVisuIndex** и **sSelectedVisuOfFrameVisuName**) в структуру **VISU_CLIENT_INFO**. Но в рамках примера я решил не изменять исходную структуру.

Теперь присвоим переменным **xSwitchTargetClientToFrmVisu2** и **xSwitchWebClientsToFrmVisu1** значение **TRUE**. В результате в таргет-визуализации в фрейме будет открыт экран **Фрейм 2** (**frmVisu2**), а в web-визуализации – экран **Фрейм 1** (**frmVisu1**).

Откроем в web-визуализации в фрейме экран **Фрейм 2** с помощью кнопки «ВПЕРЁД». Присвоим переменной **xSwitchAllClientsToFrmVisu1** значение **TRUE** – в результате для всех клиентов визуализации независимо от их типа в фрейме будет открыт экран **Фрейм 1**.

Итак, мы рассмотрели:

- как считать информацию о фреймах проекта в переменные программы;
- как определить, какой экран в конкретном фрейме открыт сейчас у конкретного клиента;
- как переключать экраны в конкретном фрейме для конкретных клиентов и для категорий клиентов различного типа.

6. Выбор элемента и имитация нажатия из кода программы (ISelectionManager)

6.1. Основная информация

Существует два типовых варианта управления технологическим процессом с использованием визуализации CODESYS:

- при использовании панельных контроллеров с сенсорным экраном или отображения web-визуализации на смартфоне: оператор нажимает на нужные элементы визуализации с помощью пальца, стилуса и т. п.;
- при отображении визуализации на ПК: оператор использует для нажатий на элементы мышь.

Но есть отдельные специфические случаи, когда ни один из этих вариантов не подходит. Например, панельный контроллер может быть установлен в шкафу автоматики за бронестеклом, поэтому использовать для работы с визуализацией его сенсорный экран нет возможности. Также нет возможности для использования клавиатуры и/или мыши (например, контроллер не поддерживает работу с USB HID устройствами или их просто негде разместить – нет выдвижной полки, как в серверных стойках). На дверце шкафа размещены кнопки, которые подключены к дискретным входам контроллера (или модуля ввода, который опрашивает контроллер). Требуется организовать работу с визуализацией (переключение между элементами и нажатие на них) с помощью этих кнопок.

Это приводит нас к задаче переключения между элементами визуализации и имитации нажатий на них из кода программы.

Приведенный выше сценарий не является единственным, где это может потребоваться – например, я встречал такие требования к визуализации: «при открытии диалога авторизации в поле пароля должен мигать курсор приглашения к вводу». Соответственно, для этого также потребуется сымитировать нажатие на элемент.

Для начала я расскажу о галочке **Standard keyboard handling** («Стандартная обработка клавиатуры»), которая может быть установлена в **Менеджере визуализации** (по умолчанию она снята) – это упростит понимание одного из следующих пунктов (если точнее – метода SelectNextElement).

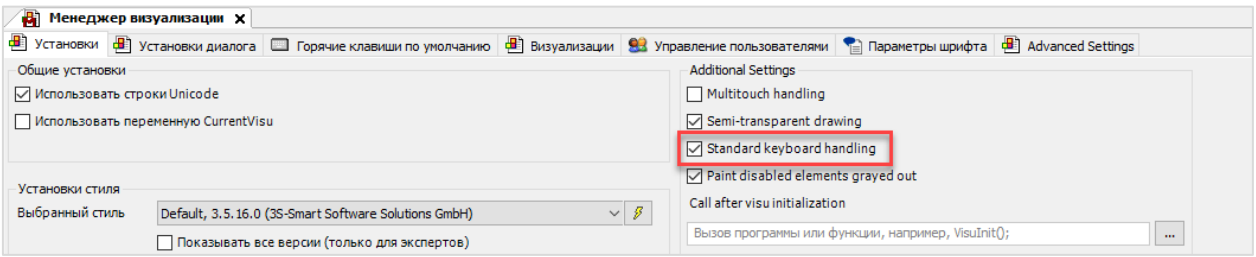
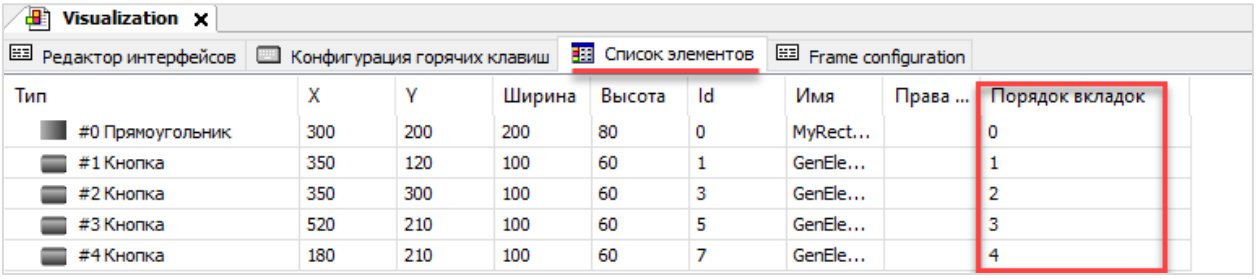


Рисунок 6.1.1 – Галочка **Standard keyboard handling** в установках **Менеджера визуализации**

Эта галочка позволяет управлять элементами визуализации, используя для этого только клавиатуру (например, если ваш промышленный компьютер не подразумевает подключение мыши). При этом не требуется вносить какие-то изменения в настройки элементов визуализации. При активации галочки некоторые клавиши начинают обрабатываться особым образом:

Таблица 6.1.1 – Описание клавиш, используемых при «стандартной обработке клавиатуры»

Клавиша	Описание
	Выделение (установка фокуса ввода) следующего элемента. Для выделения доступны только элементы ввода (переключатели, ползунки, элементы с настроенной вкладкой **Конфигурация ввода** и т. д.) – то есть элементы, способные реагировать на нажатие. Порядок выделения определяется номерами элементов в столбце **Порядок вкладок** на вкладке **Список элементов** экрана визуализации (см. рис. 6.1.2). Если выделенный элемент является таблицей, то сначала выделяется левая верхняя ячейка. Далее каждое нажатие на **TAB** приводит к выделению ячейки следующего столбца. После выделения ячейки последнего столбца происходит переход на следующую строку и т. д. Для выделения доступны только ячейки столбцов с настроенной вкладкой **Конфигурация ввода**. Если выделяемый элемент является фреймом или группой, то каждое нажатие приводит к выделение следующего элемента фрейма/группы, который поддерживает ввод, в порядке, определяемом номерами в столбце **Порядок вкладок**. Если в визуализации открывается модальный диалог, то будут выделяться элементы диалога, поддерживающие ввод. Для немодальных диалогов выделение элементов не поддерживается – выделяться будут элементы на экране визуализации, поверх которого открыт диалог. После выделения последнего элемента (с наибольшим номером в столбце **Порядок вкладок**) следующее нажатие клавиши не приведет к выделению первого элемента (с наименьшим номером)
	Выделение предыдущего элемента
	Выделение следующего ближайшего элемента, размещенного в направлении стрелки
	Нажатие на элемент (аналог события **OnMouseDown**). Отпускание клавиши аналогично событию **OnMouseUp**
	Отмена ввода (для элементов с настроенным действием **Записать переменную**). После нажатия курсор приглашения к вводу исчезает, но элемент остается выделенным



Тип	X	Y	Ширина	Высота	Id	Имя	Права ...	Порядок вкладок
#0 Прямоугольник	300	200	200	80	0	MyRect...		0
#1 Кнопка	350	120	100	60	1	GenEle...		1
#2 Кнопка	350	300	100	60	3	GenEle...		2
#3 Кнопка	520	210	100	60	5	GenEle...		3
#4 Кнопка	180	210	100	60	7	GenEle...		4

Рисунок 6.1.2 – Столбец **Порядок вкладок** в списке элементов экрана визуализации определяет порядок выделения элементов при нажатии клавиш **Tab** и **Shift+Tab**

Еще один момент – в данном пункте, как и в прошлом, для нас будут иметь значения имена элементов визуализации, задаваемые в их свойствах:

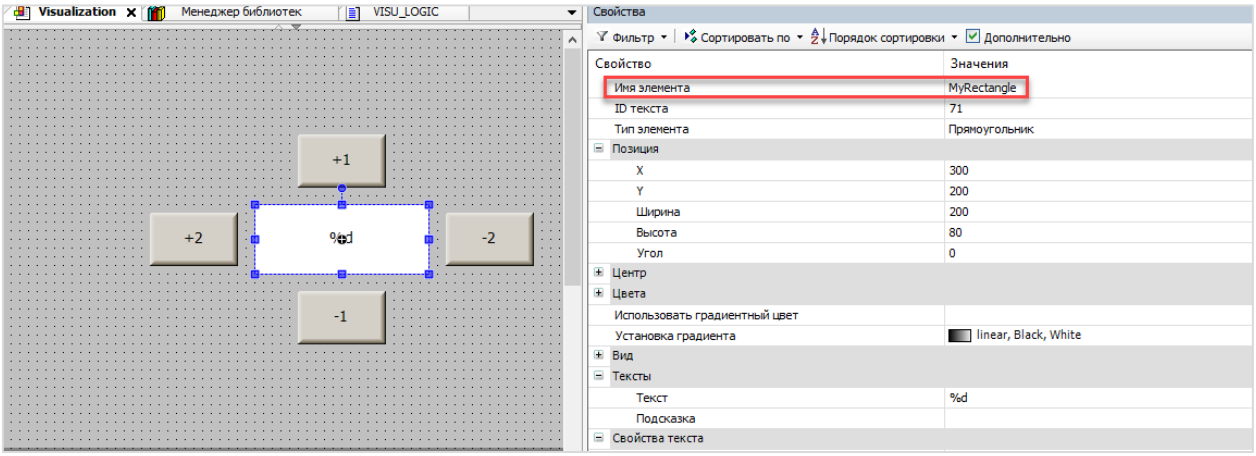


Рисунок 6.1.3 – Имя элемента визуализации

Это имя будет использоваться в методе SelectElementAt для выделения заданного элемента визуализации из кода программы.

И последнее – при установке галочки **Standard keyboard handling** в **Менеджере визуализации** – указанные в табл. 6.1.1 клавиши перестают обрабатываться как «горячие клавиши» (см., например, в свойство элемента визуализации **Конфигурация ввода – Горячая клавиша**). Причина в следующем: разработчики CODESYS считают, что если нажатие клавиши будет приводить сразу к двум событиям (переключению выделения и выполнению события горячей клавиши) – то это может быть неожиданно и неочевидно для пользователя.

6.2. Обзор примера

Пример содержит экран визуализации **Visualization**, на котором размещен прямоугольник с именем MyRectangle, к которому привязана целочисленная переменная. В конфигурации элемента для события **OnMouseDown** настроено действие **Записать переменную**. Вокруг прямоугольника расположены 4 кнопки, к каждой из которых для события **OnMouseDown** настроено действие **Выполнить ST-код**. В коде происходит изменение значение переменной прямоугольника на указанное число (+1, -1 и т. д.). Для всех элементов настроены номера в столбце **Порядок вкладок**.

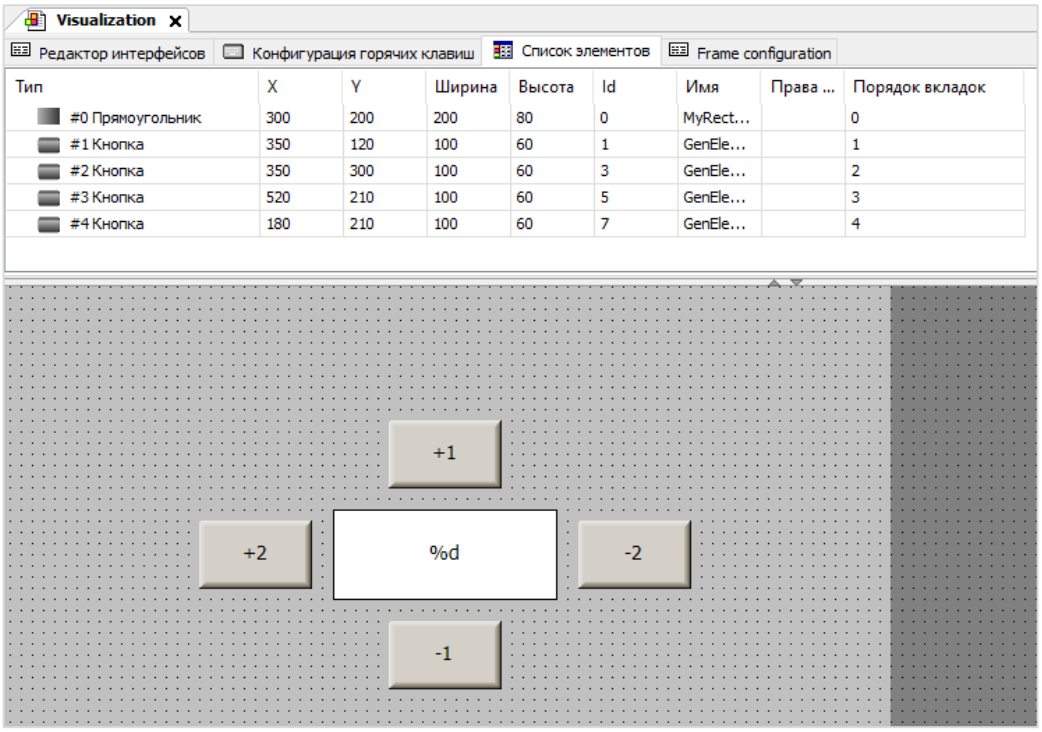


Рисунок 6.2.1 – Экран визуализации примера

Готовый пример доступен по ссылке: скачать

6.3. Обзор интерфейса ISelectionManager

Интерфейс ISelectionManager входит в состав библиотеки VisuElemBase и содержит методы, используемые для переключения фокуса элементов и имитации нажатия на них из кода программы. Далее приведено описание свойств и методов интерфейса. Все свойства доступны для чтения и записи.

6.3.1. Свойство EnabledSelectionType

Насколько я понимаю, свойство **EnabledSelectionType** (тип **DWORD**) позволяет из кода программы «снять» галочку Standard keyboard handling, отключив таким образом возможность переключения между элементами с помощью клавиатуры. Для этого нужно присвоить свойству значение **VisuElems.VisuElemBase.Visu_Selection_Constants.VISU_SELECTION_ENABLED_NONE**. Я не знаю, какое значение нужно присвоить, чтобы «установить» галочку из кода – надо сказать, что список глобальных констант **Visu_Selection_Constants** не описан в пользовательской документации и довольно скудно описан в OEM-документации.

6.3.2. Свойство FrameOffset

Свойство **FrameOffset** (тип **INT**) позволяет задать смещение в пикселях для области выделения «внутрь» элемента в пикселях. При значении **0** – область выделения совпадает с границами элемента.

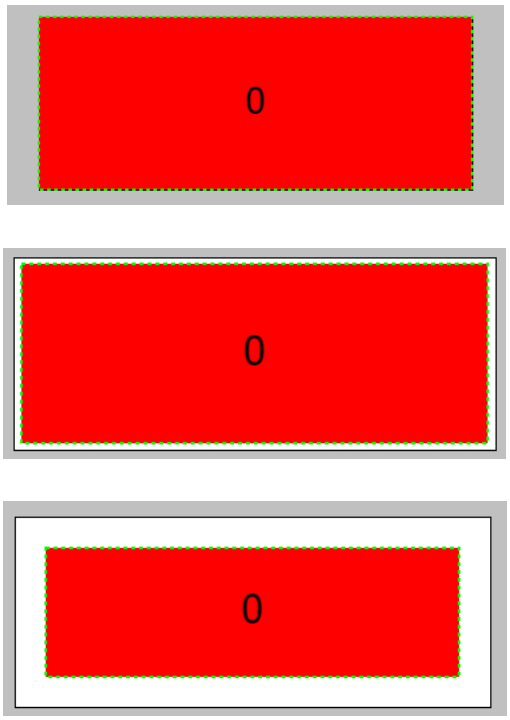


Рисунок 6.3.2.1 – Влияние значений свойства **FrameOffset** на внешний вид области выделения (сверху вниз: **FrameOffset** = **0**, **FrameOffset** = **3**, **FrameOffset** = **13**)

6.3.3. Свойство SelectionColors

Свойство **SelectionColors** (тип – структура VisuStructColors) позволяет настроить цвет области выделения. Структура содержит два поля:

- **dwFrameColor** – цвет заливки области выделения;
- **dwFillColor** – цвет контура области выделения.

Цвета представлены типом DWORD, в который должно быть записано значение в формате ARGB32.

6.3.4. Свойство SelectionLook

Свойство **SelectionLook** (тип – структура **VisuStructElementLook**) настроить внешний вид области выделения. Структура содержит три поля:

- **iLineWidth** (тип INT) – ширина контура области выделения;
- **dwFillFlags** (тип **DWORD**) – битовая маска флагов заливки области выделения. Флаги определены в перечислении VisuEnumBrushStyle. Доступны только два флага – **BS_HOLLOW** (заливка отсутствует) и **BS_SOLID** (сплошная заливка). В маске должен быть установлен только один из флагов;
- **dwFrameFlags** (тип **DWORD**) – битовая маска флагов типа контура области выделения. Флаги определены в перечислении VisuEnumPenStyle. Они могут комбинироваться через оператор **OR**.

6.3.5. Метод SelectElement

Метод **SelectElement** выделяет элемент визуализации, определяемый экземпляром интерфейса **IVisualElement**, для клиента визуализации с контекстом **pClientData**. У разработчика нет документированных способов получить экземпляр интерфейса элемента – поэтому про этот метод мне рассказать нечего.

Рисунок 6.3.5.1 – Сигнатура метода **SelectElement**

6.3.6. Метод SelectElementAt

Метод **SelectElementAt** выделяет элемент визуализации с путем **stPosition** для клиента визуализации, определяемого контекстом **pClientData**. Путь включает в себя все пространства имен (например, пространство имен библиотеки, если открыт диалог библиотеки), имя экрана визуализации (или диалога) и имя элемента, заданное в одноименном свойстве (см. рис. 6.1.3).

Например, для элемента с именем **MyRectangle**, размещенном на экране визуализации **Visualization**, путь будет выглядеть как **Visualization.MyRectangle**.

Метод возвращает код ошибки. Возможные коды ошибок:

Таблица 6.3.6.1 – Возможные коды ошибок методов интерфейса **ISelectionManager**

Название	Значение	Описание
VISU_SELECTION_OK	0	Операция успешно выполнена
VISU_SELECTION_NONE	1	Не удалось выделить элемент (например, при попытке выделить следующий элемент, когда достигнут элемент с последним порядковым номером в столбце **Порядок вкладок)**
VISU_SELECTION_ERR_ WRONG_ELEMENT_POSITION	2	Не удалось выделить элемент с заданным номером (например, номер превышает число выделяемых элементов на экране). Я предполагаю, что такая ошибка может вернуться только при вызове метода SelectElement
VISU_SELECTION_ERR_ ELEMENT_NOT_SELECTABLE	3	Не удалось выделить элемент, так как он не является выделяемым (например, элемент – индикатор). Я предполагаю, что такая ошибка может вернуться только при вызове метода SelectElement
VISU_SELECTION_KEY_HANDLED	4	Не удалось выделить элемент, так как на экране визуализации открыт модальный диалог
VISU_SELECTION_KEEP_IN_ELEMENT	5	Не удалось выделить элемент, так как он «активен» (например – в элементе **Combobox** открыт выпадающий список. В этом случае переключить выделение нельзя)
VISU_SELECTION_DISABLED	6	Судя по названию ошибки – не удалось выделить элемент, так как выделение отключено (не установлена галочка Standard keyboard handling или использовано свойство EnabledSelectionType). Но у меня не получилось ее воспроизвести – в описанной ситуации методы возвращают **0**, хотя выделения и не происходит

Рисунок 6.3.6.1 – Сигнатура метода **SelectElementAt**

6.3.7. Метод SelectNextElement

Метод **SelectNextElement** выделяет «следующий» элемент визуализации для клиента визуализации, определяемого контекстом **pClientData**. Вход **dwSelectionType** определяет способ выбора следующего элемента, а вход **dwGroupType** – режим выделения для сгруппированных элементов (сгруппированных с помощью команды контекстного меню элемента или размещенных в элементе **Группа**).

Таблица 6.3.7.1 – Возможные значения входа **dwSelectionType**

Название	Значение	Описание
VISU_SELECTION_TAB	16#01	Имитация нажатия на **Tab** (выделение элемента со следующим номером в столбце **Порядок вкладок**)
VISU_SELECTION_SHIFTTAB	16#02	Имитация нажатия на **Shift+Tab** (выделение элемента с предыдущим номером в столбце **Порядок вкладок**)
VISU_SELECTION_FIRST	16#03	Выделение элемента с наименьшим номером в столбце **Порядок вкладок**
VISU_SELECTION_LAST	16#04	Выделение элемента с наибольшим номером в столбце **Порядок вкладок**
VISU_SELECTION_LEFT	16#05	Имитация нажатия на соответствующую клавишу-стрелку (выделение ближайшего элемента, расположенного в соответствующем направлении)
VISU_SELECTION_UP	16#06	
VISU_SELECTION_RIGHT	16#07	
VISU_SELECTION_DOWN	16#08	

Таблица 6.3.7.2 – Возможные значения входа dwGroupType

Название	Значение	Описание
VISU_SELECTION_GROUP_SINGLE	16#01	Возможность выделения каждого элемента в группе
VISU_SELECTION_GROUP_BLOCK	16#02	Возможность выделения только группы элементов целиком

Как показывает практика – **VISU_SELECTION_GROUP_BLOCK** в текущих версиях не срабатывает (точнее, работает также, как **VISU_SELECTION_GROUP_SINGLE**). Еще одно наблюдение – в web-визуализации выделение элемента внутри группы сначала визуально не отображается, но если изменить размеры вкладки web-визуализации – то отобразится. При этом при выделении с помощью физической клавиши **Tab** – элементы выделяются и подсвечиваются корректно.

Метод возвращает код ошибки. Возможные коды ошибок описаны в табл. 6.3.6.1.

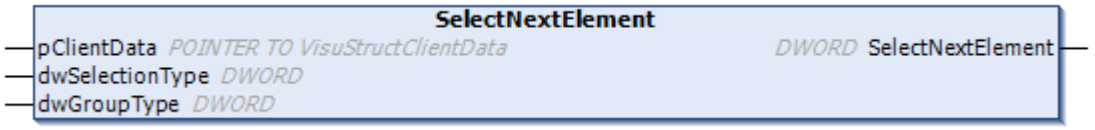


Рисунок 6.3.7.1 – Сигнатура метода **SelectNextElement**

6.3.8. Метод DoSelectedAction

Метод **DoSelectedAction** выполняет действие над выделенным элементом (имитирует нажатие клавиши) для клиента визуализации, определяемого контекстом **pClientData**. Вход **pEvent** представляет собой указатель на структуру имитируемого события типа **VisuElems.VisuElemBase.VisuStructEvent**. Структура включает в себя около десятка полей, но разработчик в большинстве случаев должен присвоить значение только одному из них – **EventTag** (тип **DWORD**). При работе с **SelectionManager** имеет смысл только одно событие – **VisuElems.VisuElemBase.VISU_ET_MOUSEDOWN** (соответствующее событию **OnMouseDown** вкладки **Конфигурация ввода** в настройках элемента). Я экспериментировал с другими событиями (**VISU_ET_MOUSEUP**, **VISU_ET_MOUSEMOVE**, **VISU_ET_MOUSEDBLCK** и т. д. – см. список глобальных констант **Visu_Constants** в библиотеке **VisuElemFunctionality**) – но у меня не получилось с их помощью добиться генерации событий **OnMouseUp/OnMouseMove/OnMouseClick**.

Метод возвращает код ошибки. Возможные коды ошибок описаны в табл. 6.3.6.1.

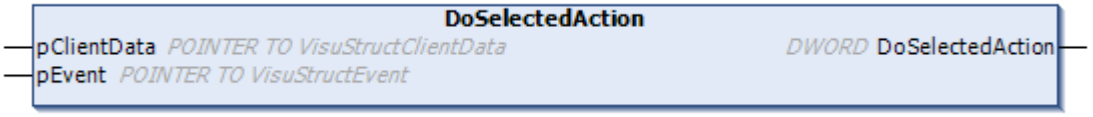


Рисунок 6.3.8.1 – Сигнатура метода **DoSelectedAction**

6.3.9. Метод ResetSelection

Метод **ResetSelection** отменяет выделение текущего элемента для клиента визуализации, определяемого контекстом **pClientData**.

Метод возвращает код ошибки. Возможные коды ошибок описаны в табл. 6.3.6.1.

Рисунок 6.3.9.1 – Сигнатура метода **ResetSelection**

6.4. Описание примера

Рассмотренные в прошлом пункте методы требуют в качестве одного аргументов указатель на контекст клиента визуализации. Поэтому данный пример основан на примере получения информации о клиентах визуализации из п. 1.2.

Для начала я покажу переменные примера:

```
// Программа привязана к задаче VISU_TASK
PROGRAM VISU_LOGIC
VAR
   // ФБ сбора информации о клиентах
   fbGetVisuClientsInfo:                  VU.FbIterateClients;
   // Команда сбора информации о клиентах
   xExecute:                              BOOL;
   // ФБ обработки клиентов
   fbClientIterationCallback:             VisuClientIteration;
   // Массив информации о клиентах
   astVisuClientInfo:                     ARRAY
      [1..Visu_Superglobal_Constants.VISU_MAX_NUMBER_OF_CLIENTS] OF VISU_CLIENT_DATA;

   // Внешний вид области выделения
   stSelectionLook:                    VisuElems.VisuElemBase.VisuStructElementLook;
   // Цвета заливки и фона области выделения
   stSelectionColors:                  VisuElems.VisuElemBase.VisuStructColors;

   // Команда инициализации
   xInit:                                 BOOL;
   // Счетчик циклов
   i:                                     INT;

   // Команда выделения элемента по имени (c_sVisuElementName)
   xSelectElementByPath:                  BOOL;
   // Команда выделения следующего элемента
   xSelectNextElement:                    BOOL;
   // Команда сброса выделения
   xResetSelection:                       BOOL;
   // Команда имитации нажатия мыши на выделенный элемент
   xDoSelectedAction:                     BOOL;

   // Результат вызова метода
   dwResult:                              DWORD;

   // Структура события визуализации
   stActionEvent:                         VisuElems.VisuElemBase.VisuStructEvent;

   // Переменная, привязанная к прямоугольнику на экране визуализации
   iVisuValue:                            INT;
END_VAR
VAR CONSTANT
   // Имя прямоугольника на экране Visualization (см. свойство Имя элемента)
   c_sVisuElementName:                    STRING := 'Visualization.MyRectangle';
   // IP-адрес клиента web-визуализации, для которого будут вызываться
   // методы SelectionManager. Измените его на свой!
   c_sVisuClientIpAddr:                   STRING := '10.2.8.133';
END_VAR
```

Константа **c_sVisuElementName** содержит имя элемента **Прямоугольник**, используемого в нашем примере (см. рис. 6.1.3). Константа **c_sVisuClientIpAddr** содержит IP-адрес клиента web-визуализации, для которого мы в рамках примера будет вызывать методы **SelectionManager**'а. Не забудьте изменить этот IP-адрес на ваш.

Теперь рассмотрим несколько первых фрагментов кода:

```
IF NOT(xInit) THEN

   // передаем адрес массива, который будет заполняться
   // информацией о клиентах визуализации
   fbClientIterationCallback.pstVisuClientData := ADR(astVisuClientInfo);

   // если сделать так - то действие галочки Standard keyboard handling
   // в Менеджере визуализации будет отключено
   // VisuElems.VisuElemBase.g_SelectionManager.EnabledSelectionType :=
         VisuElems.VisuElemBase.Visu_Selection_Constants.VISU_SELECTION_ENABLED_NONE;

   // ширина области выделения в пикселях
   stSelectionLook.iLineWidth := 2;
   // битовая маска флагов типа заливки области выделения
   // (см. перечисление VisuElems.VisuElemBase.VisuEnumBrushStyle)
   stSelectionLook.dwFillFlags :=  VisuElems.VisuElemBase.VisuEnumBrushStyle.BS_SOLID;
   // битовая маска флагов типа контура области выделения
   // (см. перечисление VisuElems.VisuElemBase.VisuEnumPenStyle)
   stSelectionLook.dwFrameFlags := VisuElems.VisuElemBase.VisuEnumPenStyle.PS_SOLID OR
      VisuElems.VisuElemBase.VisuEnumPenStyle.PS_DOT;

   VisuElems.VisuElemBase.g_SelectionManager.SelectionLook := stSelectionLook;

   // цвет заливки области выделения
   stSelectionColors.dwFillColor := 16#FFFF0000;
   // цвет контура области выделения
   stSelectionColors.dwFrameColor := 16#FF00FF00;

   VisuElems.VisuElemBase.g_SelectionManager.SelectionColors := stSelectionColors;

   // смещение области выделения "внутрь" элемента в пикселях
   VisuElems.VisuElemBase.g_SelectionManager.FrameOffset := 3;

   xInit := TRUE;

END_IF

fbGetVisuClientsInfo
(
   xExecute             := xExecute,
   itfClientFilter      := VU.Globals.AllClients,
   itfIterationCallback := fbClientIterationCallback
);

// информация о всех интересующих клиентах получена
IF fbGetVisuClientsInfo.xDone THEN

   // и теперь можно с ними что-то сделать
   FOR i := 1 TO fbClientIterationCallback.iCurrentClientCount DO

   //

   END_FOR

END_IF
```

При первом вызове программы **VISU_LOGIC** мы однократно выполняем ряд действий, используя для этого переменную **xInit**:

- передаем в **fbClientIterationCallback** (экземпляр ФБ **VisuClientIteration**) адрес массива информации о клиентах (**astVisuClientInfo**);
- обращаемся к объекту **g_SelectionManager** из списка глобальных переменных Visu_Globals библиотеки **VisuElemBase** и присваиваем значения его свойствам stSelectionLook, stSelectionColors и FrameOffset, настраивая таким образом внешний вид области выделения. Запись свойства EnabledSelectionType закомментирована, так как это бы отключило **SelectionManager** и весь функционал примера перестал бы работать. Этот закомментированный код оставлен на тот случай, если вам потребуется отключать обработку выделения из кода программы.

Далее по команде пользователя (по переднему фронту локальной переменной **xExecute**) будет вызываться **fbGetVisuCleintsInfo** (экземпляр блока FbIterateClients), в результате чего массив **astVisuClientInfo** будет заполнен информацией о клиентах визуализации.

После этого можно использовать методы **g_SelectionManager**:

```
// команда выделения элемента по имени (c_sVisuElementName)
IF xSelectElementByPath THEN

   FOR i := 1 TO Visu_Superglobal_Constants.VISU_MAX_NUMBER_OF_CLIENTS DO

      IF astVisuClientInfo[i].eClientType = VU.Visu_ClientType.WebVisualization AND
         astVisuClientInfo[i].sIpAddr = c_sVisuClientIpAddr THEN

         dwResult :=
            VisuElems.VisuElemBase.Visu_Globals.g_SelectionManager.SelectElementAt
               (astVisuClientInfo[i].pstClientData, c_sVisuElementName);
      END_IF

   END_FOR

   xSelectElementByPath := FALSE;

END_IF

// команда выделения следующего элемента (имитация нажатия на Tab)
IF xSelectNextElement THEN

   FOR i := 1 TO Visu_Superglobal_Constants.VISU_MAX_NUMBER_OF_CLIENTS DO

      IF astVisuClientInfo[i].eClientType = VU.Visu_ClientType.WebVisualization AND
         astVisuClientInfo[i].sIpAddr = c_sVisuClientIpAddr THEN

         dwResult :=
            VisuElems.VisuElemBase.Visu_Globals.g_SelectionManager.SelectNextElement
            (
               pClientData     := astVisuClientInfo[i].pstClientData,
               dwSelectionType :=
                  VisuElems.VisuElemBase.Visu_Selection_Constants.VISU_SELECTION_TAB,
               dwGroupType     := VisuElems.VisuElemBase.Visu_Selection_Constants.
                  VISU_SELECTION_GROUP_SINGLE
            );
      END_IF

   END_FOR

   xSelectNextElement := FALSE;

END_IF
```

```
// команда сброса выделения
IF xResetSelection THEN

   FOR i := 1 TO Visu_Superglobal_Constants.VISU_MAX_NUMBER_OF_CLIENTS DO

      IF astVisuClientInfo[i].eClientType = VU.Visu_ClientType.WebVisualization AND
         astVisuClientInfo[i].sIpAddr = c_sVisuClientIpAddr THEN

         dwResult := VisuElems.VisuElemBase.Visu_Globals.g_SelectionManager.
           ResetSelection(astVisuClientInfo[i].pstClientData);

       END_IF

   END_FOR

   xResetSelection := FALSE;

END_IF


// команда имитации нажатия мыши на выделенный элемент
IF xDoSelectedAction THEN

   FOR i := 1 TO Visu_Superglobal_Constants.VISU_MAX_NUMBER_OF_CLIENTS DO

      IF astVisuClientInfo[i].eClientType = VU.Visu_ClientType.WebVisualization AND
         astVisuClientInfo[i].sIpAddr = c_sVisuClientIpAddr THEN

         // для SelectionManager имеет смысл только это событие
         // (активирует событие OnMouseDown элемента визуализации)
         stActionEvent.EventTag := VisuElems.VisuElemBase.VISU_ET_MOUSEDOWN;

        dwResult := VisuElems.VisuElemBase.Visu_Globals.g_SelectionManager.
           DoSelectedAction(astVisuClientInfo[i].pstClientData, ADR(stActionEvent) );

      END_IF

   END_FOR

   xDoSelectedAction := FALSE;

END_IF
```

Все команды обрабатываются схожим образом – мы проходимся по массиву информации о клиентах и находим там нужных клиентов (в нашем случае – клиента web-визуализации с IP-адресом, определяемом константой **c_sVisuClientIpAddr**). Для этих клиентов мы вызываем соответствующий метод, передавая в качестве одного из его аргументов указатель на контекст клиента. В своем проекте вы можете реализовать нужную вам логику обработки клиентов – например, после подключения нового клиента переключить его на нужный экран авторизации (как это сделать – см. в п. 1.3), подставить в поле ввода логина предлагаемое по умолчанию имя пользователя (например, на основании IP-адреса), и с помощью методов SelectElementAt и DoSelectedAction сымитировать нажатие на поле ввода пароля для появления курсора приглашения к вводу.

Проект полностью готов. Давайте загрузим его в контроллер и проверим, как он работает.

6.5. Проверка работы примера

Откройте web-визуализацию контроллера. После этого перейдите в программу **VISU_LOGIC** и присвойте переменной **xExecute** значение **TRUE** для сбора информации о клиентах визуализации.

Device.Application.VISU_LOGIC

Выражение	Тип	Значение	Подготовленное значение
fbGetVisuClientsInfo	VU.FbIterateClients		
xExecute	BOOL	FALSE	TRUE
fbClientIterationCallback	VisuClientIteration		
astVisuClientInfo	ARRAY [1..Visu_Superglobal_Constants.VISU_MAX_NUMBER_OF_C...		
stSelectionLook	VisuElems.VisuElemBase.VisuStructElementLook		
stSelectionColors	VisuElems.VisuElemBase.VisuStructColors		
xInit	BOOL	TRUE	
i	INT	0	
xSelectElementByPath	BOOL	FALSE	
xSelectNextElement	BOOL	FALSE	
xResetSelection	BOOL	FALSE	
xDoSelectedAction	BOOL	FALSE	
dwResult	DWORD	0	
stActionEvent	VisuElems.VisuElemBase.VisuStructEvent		
iVisuValue	INT	0	
c_sVisuElementName	STRING	'Visualization.MyRectangle'	
c_sVisuClientIpAddr	STRING	'10.2.8.133'	

Рисунок 6.5.1 – Выполнение команды сбора информации о клиентах визуализации

Присвойте переменной **xSelectNextElement** значение **TRUE**. В результате будет выделен элемент с наименьшим порядковым номером в столбце **Порядок вкладок** (см. рис. 6.1.2) – то есть прямоугольник. Настройки внешнего вида области выделения были заданы при инициализации (см. первый фрагмент программы).

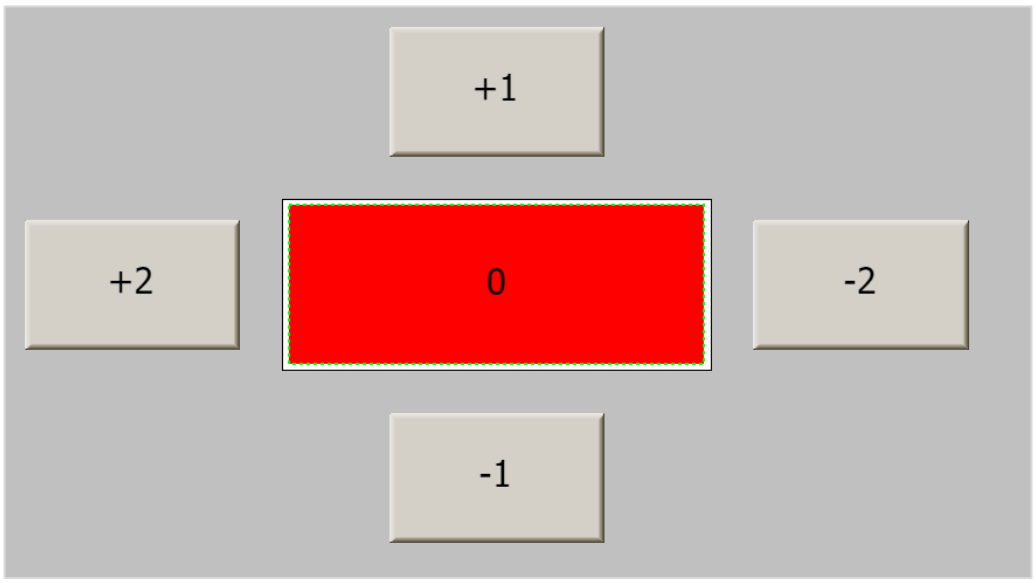


Рисунок 6.5.2 – Выделение элемента из кода программы

Присвойте переменной **xSelectNextElement** еще несколько раз значение **TRUE**. Каждый раз будет выделяться следующий (по возрастанию порядкового номера в столбце **Порядок вкладок**) элемент.

Присвойте переменной **xSelectElementByPath** значение **TRUE**. Снова будет выделен элемент **Прямоугольник**. Напомню, что в этом случае будет произведен вызов метода SelectElementAt, который позволяет выделить элемент с заданным путем (в примере путь к элементу определяется константой **c_sVisuElementName**).

Присвойте переменной **xDoSelectedAction** значение **TRUE**. Будет сымитировано нажатие на элемент, в результате чего появится поле ввода (потому что в конфигурации ввода элемента для события **OnMouseDown**, которое мы имитируем, настроено действие **Записать переменную**).

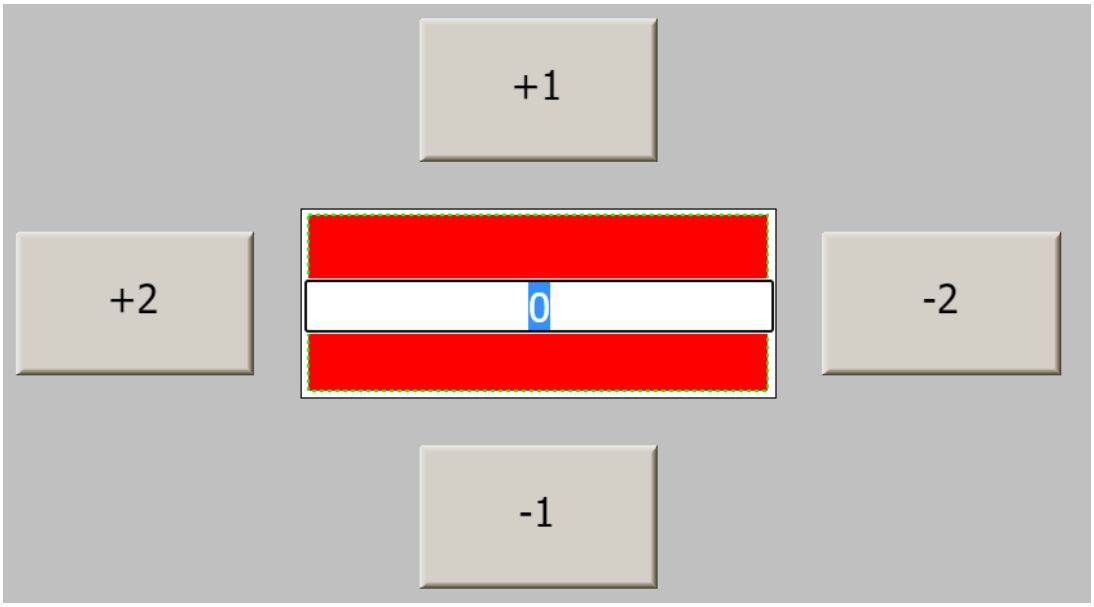


Рисунок 6.5.3 – Имитация нажатия на элемент из кода программы

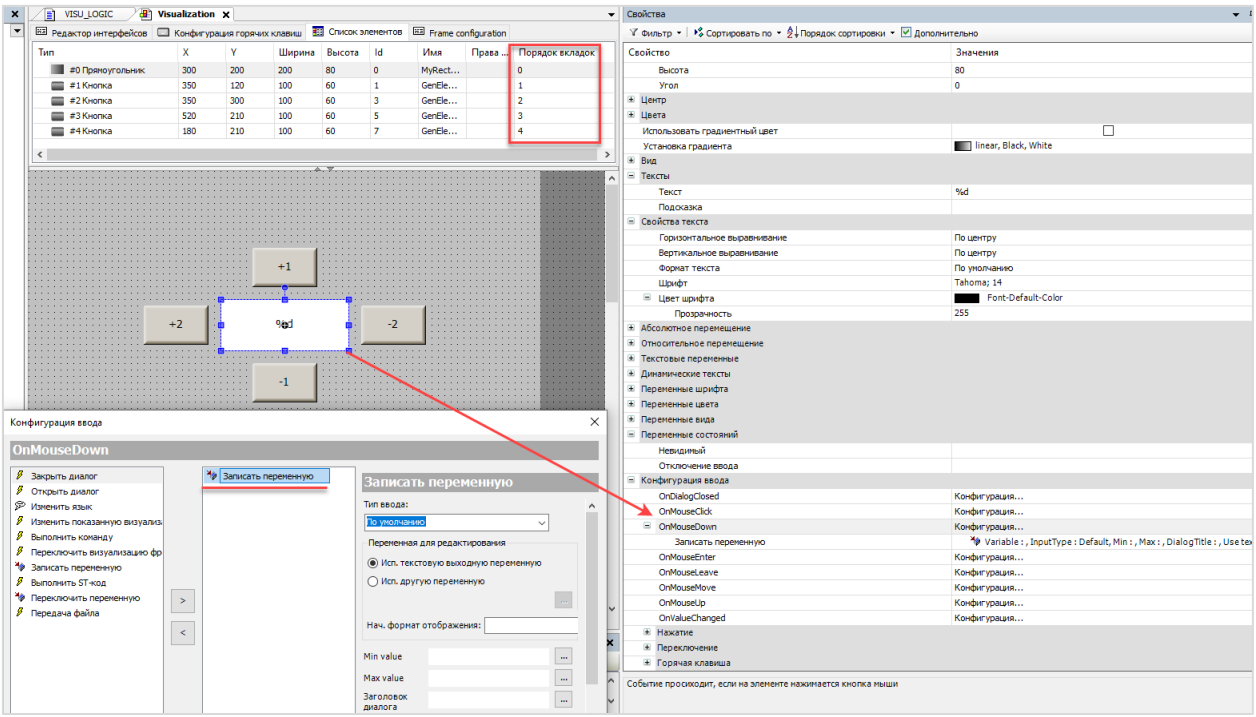


Рисунок 6.5.4 – Настройки события **OnMouseDown** для элемента **Прямоугольник**

Введите с помощью клавиатуры какое-нибудь число и нажмите **Enter**. Сымитировать эти действия с помощью **SelectionManager** не получится – придется произвести их вручную.

После этого присвойте переменной **xResetSelection** значение **TRUE**. Это приведет к отключению выделения прямоугольника.

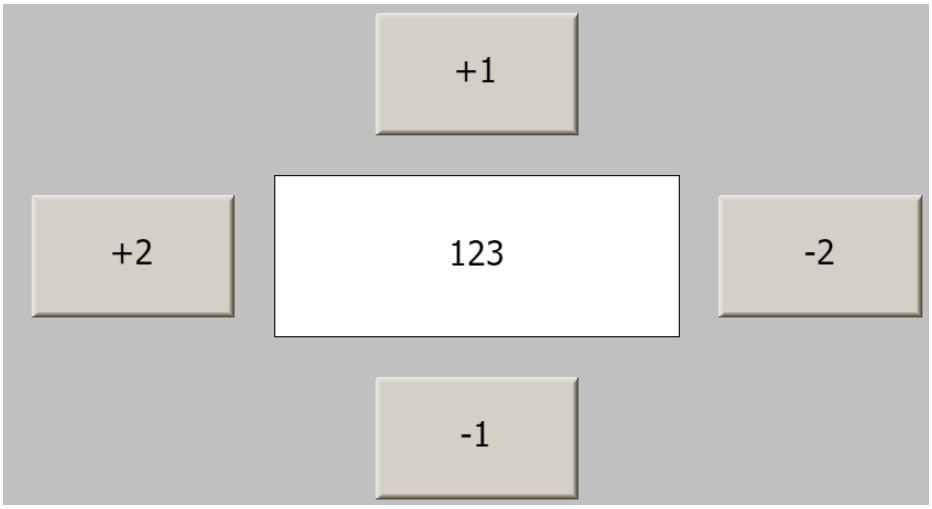


Рисунок 6.5.5 – Отключение выделения элемента из кода программы

Итак, мы рассмотрели, как осуществить выделение элементов визуализации из кода программы и имитировать нажатия на них.

7. Обработка событий пользователя (IVisuUserEventManager)

7.1. Основная информация

В процессе работы с визуализацией CODESYS оператор применяет какие-то средства ввода – сенсорный экран панельного контроллера, мышь, клавиатуру и т. п. Он использует их для перемещения курсора, нажатий на элементы визуализации, ввода новых значений и т. д. В некоторых случаях требуется считывать информацию об этих событиях в коде программы – например, определить, какую клавишу нажал конкретный клиент визуализации или где сейчас находится курсор конктертного клиента. Кроме того, может потребоваться запретить обработку событий – например, если экран панельного контроллера переходит в «спящий» режим (обычно при этом яркость подсветки снижается до минимума), то первое нажатие на экран, приводящее к пробуждению, не должно обрабатываться (потому что клиент не видит, куда нажимает – и, например, случайно может нажать на кнопку запуска технологического процесса).

В библиотеке VisuElemBase есть ряд интерфейсов для обработки подобных событий:

- ICustomEventHandler – обработка произвольных пользовательских событий, зарегистрированных с помощью библиотеки CmpEventMgr. Не будет рассматриваться в рамках данного документа;
- **IEditBoxInputHandler** – обработка ввода нового значения с помощью элемента визуализации. Будет рассмотрен в п. 7.5;
- IGestureEventHandler – обработка жестов (при включенной в **Менеджере визуализации** галочке **Multitouch handling**). Не будет рассматриваться в рамках данного документа. См. пример от разработчиков CODESYS;
- IInputOnElementEventHandler – обработка действия с элементом визуализации (нажатие на элемент, отпускание и т. д.). Будет рассмотрен в п. 7.4;
- IKeyEventHandler – обработка нажатия клавиши клавиатуры. Будет рассмотрен в п. 7.2;
- IMouseEventHandler – обработка нажатия или перемещения курсора мыши (или курсора экрана панельного контроллера). Будет рассмотрен в п. 7.3;
- ISpecialEventHandler – обработка неких «специальных» событий. Интерфейс используется только в системных библиотеках CODESYS и не предназначен для использования сторонними разработчиками. Не будет рассматриваться в рамках данного документа;
- IUserMgmtEventHandler – обработка событий пользователей визуализации (вход в систему, изменения пароля и т. д.). Был рассмотрен в п. 2.9;
- IValueChangedListener – обработка изменений значений переменных, привязанных элементам визуализации. Основное отличие от **IEditBoxInputHandler** – вызывается не только при вводе клиентом нового значения с помощью аппаратной или экранной клавиатуры, но и при изменении значения из-за других обстоятельств. Будет рассмотрен в п. 7.6.

Каждый интерфейс обычно включает в себя всего один метод, автоматически вызываемый (по технологии callback) при соответствующем событии. Исключением является **IMouseEventHandler**, у которого 2 события (нажатие/отпускание курсора и перемещение курсора) и **IUserMgmtEventHandler**, у которого их 4 (авторизация/неудачная попытка авторизации/изменение пароля/выход из системы).

Работа с любым из интерфейсов происходит по одному и тому же сценарию:

- сначала нужно создать функциональный блок, который реализует (**IMPLEMENTS**) данный интерфейс;
- далее добавить код в метод(-ы), полученный от этого интерфейса. Одним из входов метода будет указатель на контекст клиента (**pClientData**) – с помощью него можно определить клиента визуализации, для которого был вызван метод. Другие входы содержат информацию о конкретном событии (код нажатой клавиши, координаты курсора и т. д.). Каждый метод имеет выход типа **BOOL**. Если присвоить ему значение **TRUE** – то обрабатываемое событие не будет передано другим системным обработчикам – в том числе, подсистеме визуализации. Это, например, позволяет отключить стандартную обработку курсора – то есть сделать так, чтобы нажатие на элемент визуализации не приводило к каким-либо эффектам;
- затем нужно объявить в программе, привязанной к задаче VISU_TASK, экземпляр вашего функционального блока;
- в коде программы нужно зарегистрировать этот экземпляр в обработчике соответствующего события с помощью соответствующего метода глобального объекта **VisuElems.VisuElemBase.Visu_Globals.g_VisuEventManager**[4], реализующего интерфейс IVisuUserEventManager. Если метод возвращает **FALSE** – это значит, что пользовательский обработчик данного события уже зарегистрирован в другом фрагменте проекта, и поэтому зарегистрировать второй обработчик не получилось;
- метод экземпляра вашего блока будет вызван автоматически при наступлении соответствующего события;
- если требуется – в коде программы вы можете передать значения на входы экземпляра вашего блока, считать значения с его выходов и организовать его вызов (если нужно циклически выполнять код, размещенный в теле блока).

В следующих пунктах мы рассмотрим большинство из упомянутых выше интерфейсов и проверим их работу на конкретных примерах.

[4] Исключением является интерфейс **IValueChangedListener**, экземпляр которого регистрируется через другой глобальный объект. См. подробнее в п. 7.6.

7.2. Обработка нажатия клавиш (IKeyEventHandler)

7.2.1. Основная информация

При работе с визуализацией может использоваться аппаратная клавиатура. В этой ситуации может потребоваться выполнить определенное действие (или набор действий) при нажатии на заданную клавишу, а также определить в коде программы, что на эту клавишу нажал конкретный клиент визуализации.

Визуализация CODESYS поддерживает механизм «горячих клавиш». Они могут быть настроены в трех местах проекта:

- в **Менеджере визуализации** на вкладке **Горячие клавиши по умолчанию** (см. рис. 7.2.1.1). Здесь настраиваются горячие клавиши, которые будут обрабатываться на любом экране визуализации. Список доступных действий в целом соответствует действиям вкладки **Конфигурации ввода** (отсутствуют действия **Записать переменную** и **Переключить визуализацию в фрейме**, так как на этом уровне неизвестен контекст, требуемый для выполнения этих действий);
- в интерфейсе конкретного экрана визуализации на вкладке **Конфигурация горячих клавиш** (см. рис. 7.2.1.2). Здесь настраиваются горячие клавиши, которые будут обрабатываться только на данном экране. Список доступных действий совпадает с действиями вкладки **Конфигурация ввода** (добавлено дополнительное действие – **Выполнить событие мыши**, которое представляет собой имитацию нажатия на заданный элемент визуализации);
- в свойстве элемента визуализации (**Конфигурация ввода – Горячая клавиши**; см. рис. 7.2.1.3). Нажатие на выбранную здесь клавишу приведет к имитации действия с элементом (нажатие на элемент, отпускание элемента или клик, объединяющий последовательное нажатие и отпускание). Это приведет к формированию соответствующего события в элементе (**OnMouseDown**, **OnMouseUp** или **OnMouseClick** соответственно). Настроенная здесь клавиша автоматически добавляется на вкладку **Конфигурация горячих клавиш** экрана визуализации.

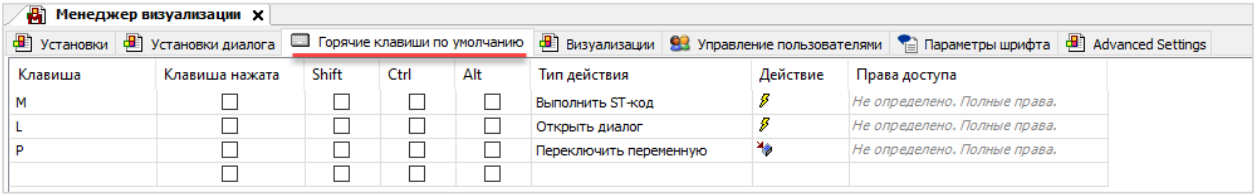


Рисунок 7.2.1.1 – Настройка горячих клавиш в **Менеджере визуализации**

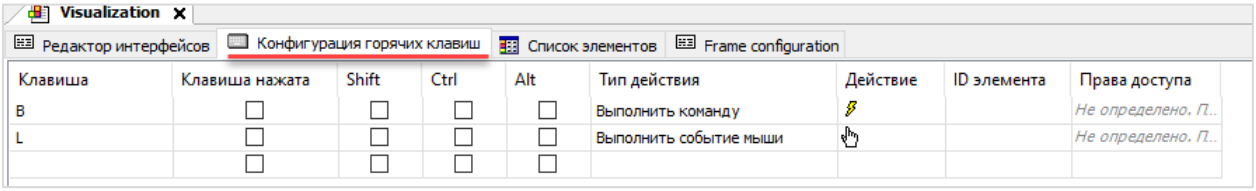


Рисунок 7.2.1.2 – Настройка горячих клавиш экрана визуализации

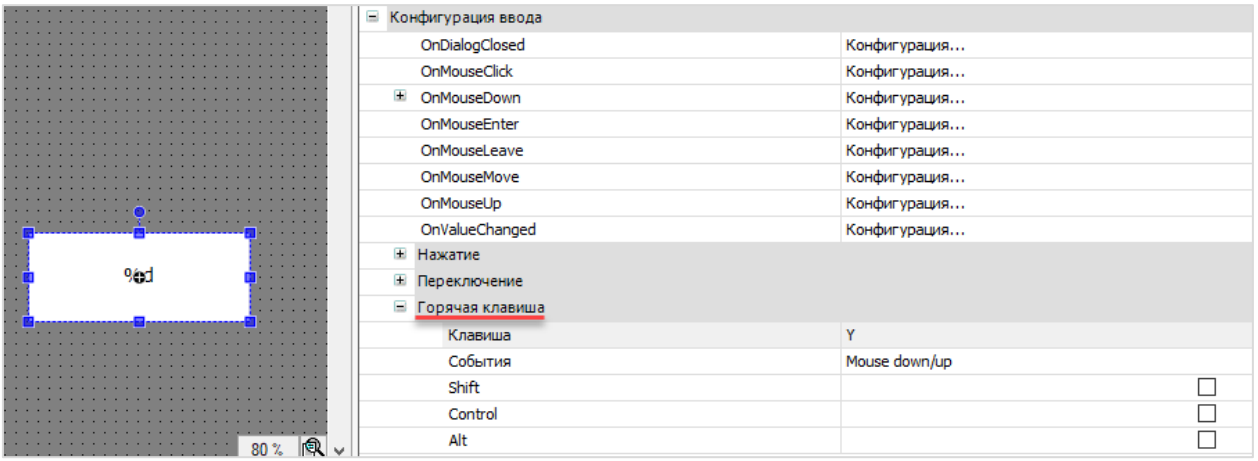


Рисунок 7.2.1.3 – Настройка горячей клавиши элемента визуализации

Если на разных вкладках (см. рисунки выше) для одной и той же клавиши настроены разные действия – то при нажатии на эту клавишу все они выполняются (при условии, что нажатие происходит на экране визуализации, для которого настроена эта клавиша на рис. 7.2.1.2 и 7.2.1.3).

Для горячей клавиши можно настроить клавиши-модификаторы (**Shift**, **Ctrl** и **Alt**). В этом случае действие будет выполнено только при нажатии соответствующей комбинации клавиш.

Надо сказать, что по умолчанию в качестве горячей клавиши можно выбрать не любую клавишу – например, не будут доступны ~, [,] и клавиши других спецсимволов. Чтобы появилась возможность выбора этих горячих клавиш – производитель контроллера должен добавить их в таргет-файл (если вы внезапно разрабатываете ПЛК с CODESYS – то см. пункт **visualization\keyboardusage** в **RuntimeSystemDocumentaion**). При этом такие клавиши будут обрабатываться только в web-визуализации. Чтобы они обрабатывались в таргет-визуализации – нужно добавить их в конфиг-файл (**CODESYSControl.cfg**). См. пункт **CmpTargetVisuItf** в **RuntimeSystemDocumentaion**, из него перейдите в **Settings** и там см. параметры **AdditionalKeyCodeRangeMin**, **AdditionalKeyCodeRangeMax** и **AdditionalKeyCodeOffset** (последний вам вряд ли потребуется).

См. также описание механизма «стандартной обработки клавиш» (**Standard keyboard handling**) в п. 6.1.

В сервисной визуализации CODESYS горячие клавиши не обрабатываются (по крайней мере, в текущей версии CODESYS).

Далее мы рассмотрим интерфейс IKeyEventHandler, который позволит нам обработать нажатие на клавишу в коде программы.

7.2.2. Обзор интерфейса IKeyEventHandler

Интерфейс IKeyEventHandler входит в состав библиотеки VisuElemBase и содержит единственный метод HandleKeyEvent. Этот метод вызывается при нажатии и отпускании любой клавиши клавиатуры. Если клавиша зажата – то метод вызывается с некоторой периодичностью (я не могу обозначить, с какой именно). Вообще, я не рекомендую использовать метод для обработки зажатия клавиш из-за задержек в обработке, которые будут описаны в конце п. 7.2.4.

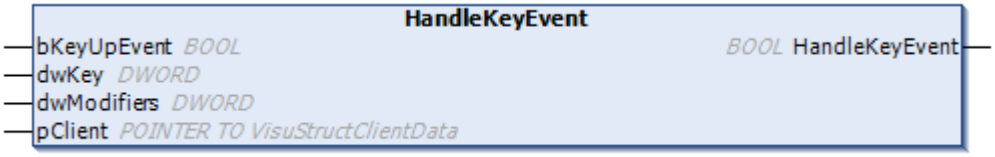


Рисунок 7.2.2.1 – Сигнатура метода **HandleKeyEvent**

- **bKeyUpEvent** имеет значение **FALSE**, если клавиша нажата и **TRUE** – если отпущена;
- **dwKey** содержит код нажатой клавиши (см. сайт для определения клавиатурных кодов);
- **dwModifiers** содержит битовую маску клавиш-модификаторов, нажатых вместе с клавишей **dwKey** (бит 0 – **Shift**, бит 1 – **Alt**, бит 2 – **Ctrl**);
- **pClient** представляет собой указатель на контекст клиента (структуру данных клиента визуализации), который нажал или отпустил кнопку.

Если на выход метода присвоить значение **TRUE** – то данное событие не будет обработано другими обработчиками (в т. ч. подсистемой визуализации CODESYS – то есть в этом случае действия, привязанные к горячей клавише в менеджере визуализации или редакторе визуализации, не будут выполнены).

Экземпляр ФБ, реализующего данный интерфейс и являющегося обработчиком соответствующего события, должен быть зарегистрирован с помощью вызова метода VisuElems.VisuElemBase.Visu_Globals.g_VisuEventManager.SetKeyEventHandler. Единственным входом метода является экземпляр данного интерфейса (соответственно, подойдет и экземпляр ФБ, реализующего данный интерфейс). Метод возвращает **TRUE** в случае успешной регистрации обработчика и **FALSE** – если обработчик данного интерфейса уже был зарегистрирован (для рассматриваемых в п. 7 интерфейсов нельзя зарегистрировать несколько обработчиков).

7.2.3. Обзор и описание примера

Приведенный ниже пример достаточно синтетический, но, тем не менее, позволяет ознакомиться с рассматриваемым в данном пункте функционалом.

На экране визуализации **Visualization** настроена горячая клавиша **Q** – по ее нажатию происходит инверсия значения переменной **xVisuLamp** программы **VISU_LOGIC**, привязанной к индикатору на экране визуализации. В переменные программы **VISU_LOGIC** считывается следующая информация (она же будет отображаться на экране визуализации):

- код последней нажатой клавиши;
- тип клиента визуализации, который произвел последнее нажатие (для клиента web-визуализации – еще и его IP-адрес);
- счетчик нажатий на клавишу **Q** клиентом web-визуализации с IP-адресом **10.2.8.133** (это IP-адрес моего ПК; вы замените его на свой). Нажатия на клавишу **Q** другими клиентами визуализации не приводят к увеличению значения счетчика.

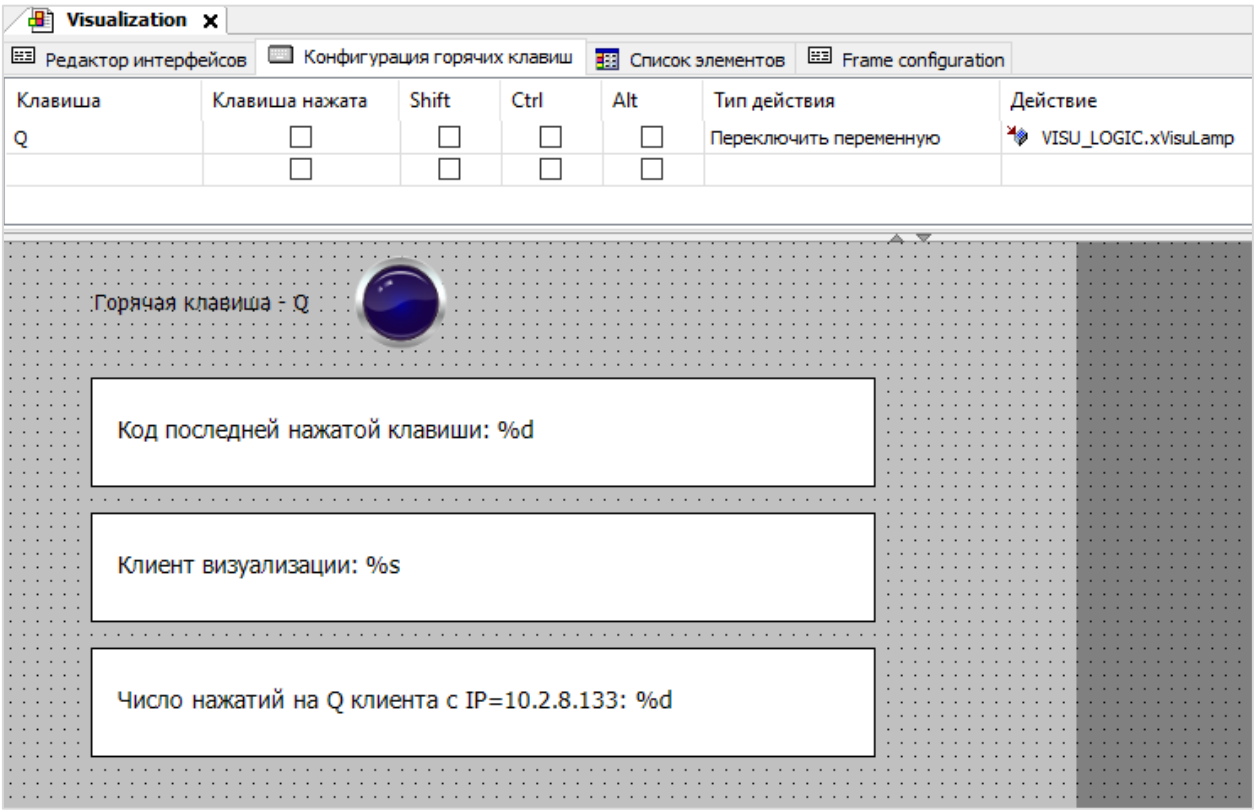


Рисунок 7.2.3.1 – Внешний вид экрана примера

Готовый пример доступен по ссылке: скачать

Для начала создадим ФБ, который будет реализовывать (**IMPLEMENTS**) интерфейс VisuElems.VisuElemBase.IKeyEventHandler. Назовем его **KeyEventHandler**.

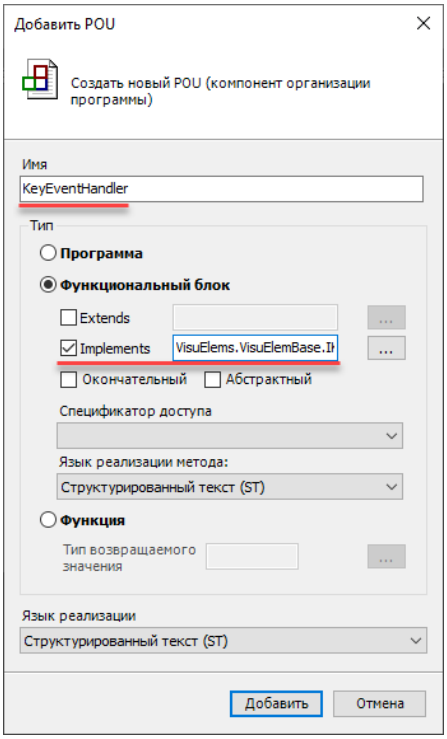


Рисунок 7.2.3.2 – Создание ФБ, реализующего интерфейс **VisuElems.VisuElemBase.IKeyEventHandler**

В его автоматически полученном от интерфейса методе HandleKeyEvent по привычке допишем пространство имен для переменной **pClient**, содержащей указатель на контекст клиента:

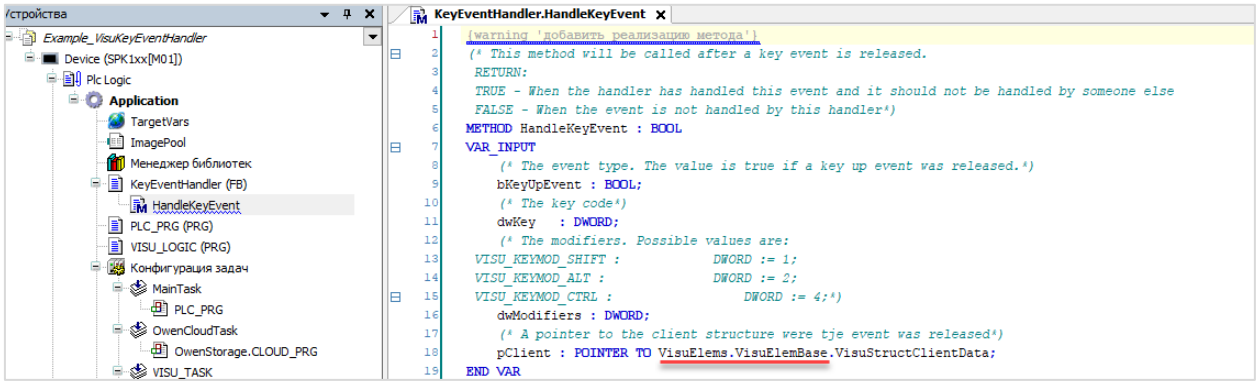


Рисунок 7.2.3.3 – Добавление пространства имен для переменной **pClient**

Объявим в области входов ФБ указатели на значения, которые мы хотим получить в нашей программе:

```
FUNCTION_BLOCK KeyEventHandler IMPLEMENTS VisuElems.VisuElemBase.IKeyEventHandler
VAR_INPUT
   pdwKeyCode:                    POINTER TO DWORD;
   psClientDefinition:            POINTER TO STRING;
   puiSpecificClientEventCounter: POINTER TO UINT;
END_VAR
```

Теперь добавим реализацию метода **HandleKeyEvent**. Я не перевожу автоматически созданные комментарии – метод уже был описан в п. 7.2.2.

```
(* This method will be called after a key event is released.
RETURN:
   TRUE - When the handler has handled this event and it should not be
      handled by someone else
   FALSE - When the event is not handled by this handler *)
METHOD HandleKeyEvent : BOOL
VAR_INPUT
   (* The event type. The value is true if a key up event was released.*)
   bKeyUpEvent  : BOOL;
   (* The key code*)
   dwKey        : DWORD;
   (* The modifiers. Possible values are:
      VISU_KEYMOD_SHIFT : DWORD := 1;
      VISU_KEYMOD_ALT :   DWORD := 2;
      VISU_KEYMOD_CTRL :  DWORD := 4;*)
   dwModifiers  : DWORD;
   (* A pointer to the client structure were tje event was released*)
   pClient      : POINTER TO VisuElems.VisuElemBase.VisuStructClientData;
END_VAR
VAR
   sIpAddr:                 STRING;
   xIsIpAddr:               BOOL;
   fbClientTagDataHelper:   VisuElems.VisuElemBase.VisuFbClientTagDataHelper;
END_VAR
```

Локальные переменные объявлены нами. Функциональный блок **VisuFbClientTagDataHelper** позволяет извлечь из контекста клиента интересную информацию – в частности, IP-адрес клиента web-визуализации. Он будет сохранен в нашу переменную **sIpAddr**. В переменную **xIsIpAddr** будет установлен флаг наличия у клиента IP-адреса.

Код метода будет выглядеть следующим образом:

```
pdwKeyCode^ := dwKey;

fbClientTagDataHelper(pClientData := pClient, xIPv4Valid => xIsIpAddr,
   stIPv4 => sIpAddr);

IF pClient^.GlobalData.ClientType =
   VisuElems.VisuElemBase.Visu_ClientType.WebVisualization THEN

   psClientDefinition^ := CONCAT('Web-visu, IP = ', sIpAddr);

   // если клиент с IP-адресом 10.2.8.133 отпустил (bKeyUpEvent)
   // клавишу с кодом 81 (Q), то увеличиваем значение счетчика
   IF bKeyUpEvent AND dwKey = 81 AND sIpAddr = '10.2.8.133' THEN
      puiSpecificClientEventCounter^ := puiSpecificClientEventCounter^ + 1;
   END_IF

ELSIF pClient^.GlobalData.ClientType =
   VisuElems.VisuElemBase.Visu_ClientType.TargetVisualization THEN

   psClientDefinition^ := 'Target-visu';

END_IF
```

Код нажатой клавиши будет записан по указателю **pdwKeyCode**. Напомню, что заранее изучить коды клавиш можно, например, на этом сайте.

Далее мы вызываем экземпляр ФБ **VisuFbClientTagDataHelper**, чтобы определить IP-адрес клиента web-визуализации и записать его в переменную **sIpAddr**. В переменную **xIsIpAddr** будет установлен флаг наличия у клиента IP-адреса.

В операторе **IF** мы проверяем тип клиента визуализации, который нажал или отпустил клавишу. Если это клиент web-визуализации, то мы формируем строку с обозначением его типа и IP-адресом, и записываем ее по указателю **psClientDefinition**. После этого во вложенном **IF** мы проверяем три условия:

- что текущий клиент – это клиент web-визуализации с IP-адресом **10.2.8.133** (это IP-адрес моего ПК; измените его на свой);
- что этот клиент отпустил клавишу (в этом случае вход метода **bKeyUpEvent** получает значение **TRUE**);
- что это была клавиша с кодом **81** (код клавиши **Q**).

Если все три условия выполняются – то мы инкрементируем значение счетчика по указателю **puiSpecificClientEventCounter**.

Обратите внимание, что мы ориентируемся на отпускание клавиши. Дело в том, что если мы будем проверять нажатие клавиши (в этом случае вход метода **bKeyUpEvent** имеет значение **FALSE**), то значение счетчика будет циклически увеличиваться всё время, пока клавиша будет удерживаться в нажатом состоянии (по моим наблюдениям – период цикла не особо коррелирует с интервалом вызова задачи **VISU_TASK**; я еще упомяну этот момент, когда мы запустим пример).

Если же тип нашего клиента – таргет-визуализация, то мы просто формируем строку с обозначением его типа и записываем ее по указателю **psClientDefinition**.

В программе **VISU_LOGIC**, привязанной к задаче VISU_TASK, объявим экземпляр нашего ФБ и вспомогательные переменные:

```
// Программа привязана к задаче VISU_TASK
PROGRAM VISU_LOGIC
VAR
   // Обработчик нажатий клавиш
   fbKeyEventHandler:                KeyEventHandler;

   // Команда инициализации
   xInit:                            BOOL;

   // Переменная, привязанная к лампе
   xVisuLamp:                        BOOL;

   // Код нажатой клавиши
   dwKeyCode:                        DWORD;
   // Тип клиента визуализации
   sClientDefinition:                STRING;
   // Счетчик нажатий на клавишу Q для клиента с IP = 10.2.8.133
   // (см. KeyEventHandler.HandleKeyEvent)
   uiSpecificClientEventCounter:     UINT;
END_VAR
```

Код программы довольно прост – при ее старте мы однократно регистрируем экземпляр нашего ФБ в качестве обработчика нажатий клавиш и передаем на его входные указатели адреса переменных программы.

```
IF NOT(xInit) THEN

   fbKeyEventHandler.pdwKeyCode           := ADR(dwKeyCode);
   fbKeyEventHandler.psClientDefinition   := ADR(sClientDefinition);
   fbKeyEventHandler.puiSpecificClientEventCounter :=
      ADR(uiSpecificClientEventCounter);

   VisuElems.VisuElemBase.Visu_Globals.g_VisuEventManager.SetKeyEventHandler
      (fbKeyEventHandler);

   xInit := TRUE;

END_IF
```

Проект полностью готов. Давайте загрузим его в контроллер и проверим, как он работает.

7.2.4. Проверка работы примера

Подключитесь к web-визуализации контроллера и нажимайте на различные клавиши. Наблюдайте за отображением вашего IP-адреса и кодов нажимаемых клавиш (см. сайт для определения кодов). Каждое нажатие на клавишу **Q** будет инвертировать значение переменной, привязанной к индикатору – за счет соответствующей настройки горячей клавиши экрана визуализации (см. рис. 7.2.3.1). Если на **Q** нажмет клиент web-визуализации с IP-адресом, указанным в коде метода **HandleKeyEvent** – то значение счетчика нажатий, отображаемое внизу экрана, увеличится на единицу. Нажатие на клавишу **Q** других клиентов визуализации не приведет к изменению значения счетчика.

Если вы будете зажимать и удерживать клавиши (особенно это наглядно с клавишей **Q**), то после отпускания клавиши визуализация будет «тормозить» – переключение индикатора и увеличение счетчика нажатий может произойти с задержкой до нескольких десятков секунд (в зависимости от времени удержания клавиши). Поэтому не рекомендуется использовать концепцию удерживания клавиш при работе с визуализацией CODESYS.

На базе примера вы можете разработать свою логику обработки нажатий на различные кнопки различными клиентами визуализации, а также другую информацию, которую можно получить от метода интерфейса.

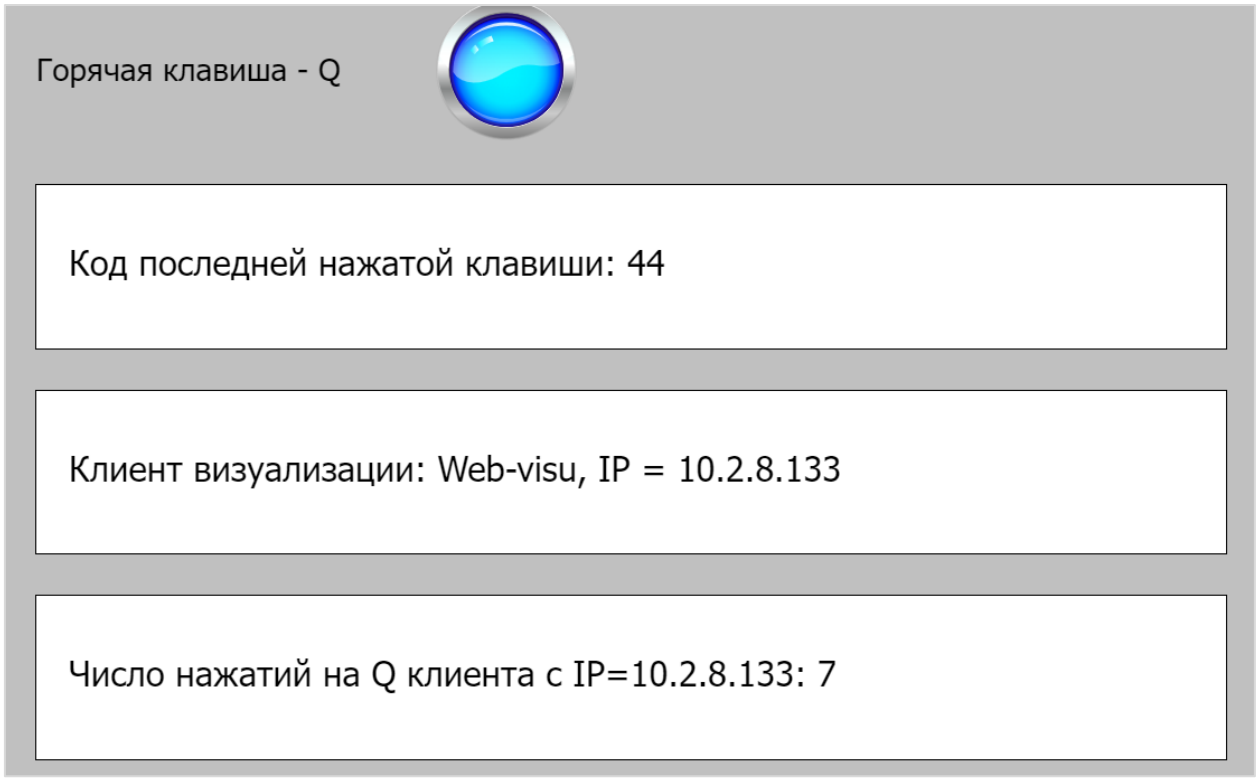


Рисунок 7.2.4.1 – Работа с web-визуализацией примера

7.3. Обработка курсора (IMouseEventHandler)

7.3.1. Основная информация

В прошлом пункте мы разобрались с обработкой нажатий клавиш. Теперь поговорим об обработке курсора. Оператор управляет курсором с помощью пальца/стилуса (такой подход используется при работе с сенсорным экраном панельного контроллера или смартфона, на котором открыта web-визуализация) или мыши (такой подход используется при работе с web-визуализацией, открытой на ПК; реже мышь подключается к панельному контроллеру и используется для работы с его таргет-визуализацией).

Две основные операции, производимые с курсором – это перемещение его по экрану и нажатие с его помощью в нужную точку экрана (обычно – на нужный элемент визуализации).

В некоторых ситуациях может потребоваться считывать информацию о положении курсора конкретного клиента визуализации в переменные программы. Например, для панельных контроллеров ОВЕН СПК был разработан компонент **Screen**, позволяющий уменьшить яркость подсветки экрана до заданного значения после неактивности оператора в течение заданного времени. Под «неактивностью» подразумевается отсутствие перемещения и нажатий курсора – поэтому как раз и требовалось получать информацию о его состоянии в коде программы. Кроме того, если экран панельного контроллера переходит в «спящий» режим (обычно при этом яркость подсветки снижается до минимума), то первое нажатие на экран, приводящее к пробуждению, не должно обрабатываться (потому что клиент не видит, куда нажимает – и, например, случайно может нажать на кнопку запуска технологического процесса). Соответственно, для этого требовалось отключить обработку курсора из кода программы.

Далее мы рассмотрим интерфейс IMouseEventHandler, который позволит нам получить информацию о положении курсора и нажатиях на него в коде программы, а также при необходимости отключить его обработку подсистемой визуализации.

7.3.2. Обзор интерфейса IMouseEventHandler

Интерфейс IMouseEventHandler входит в состав библиотеки VisuElemBase и содержит два метода: HandleMouseBtnEvent и HandleMouseMoveEvent. Первый из них вызывается подсистемой визуализации при нажатии на курсор, а второй – при перемещении курсора.

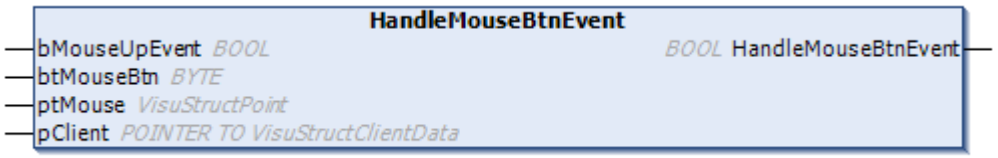


Рисунок 7.3.2.1 – Сигнатура метода **HandleMouseBtnEvent**

- **bMouseUpEvent** имеет значение **FALSE**, если курсор нажат и **TRUE** – если отпущен;
- **btMouseBtn** содержит код кнопки мыши (в текущих версиях CODESYS не определяется, как именно кнопка мыши была нажата, вход всегда имеет значение **0**);
- **ptMouse** содержит структуру с координатами курсора (тип структуры – CmpVisuHandler.VisuElemVisuStructPoint; она включает два поля типа **INT** – **iX** и **iY** – координаты по осям X и Y соответственно);
- **pClient** представляет собой указатель на контекст клиента (структуру данных клиента визуализации), который нажал или отпустил курсор.

У метода **HandleMouseMoveEvent** присутствуют только два входа: **ptMouse** (структуру с координатами текущего положения курсора) и **pClient** (указатель на контекст клиента).

Рисунок 7.3.2.2 – Сигнатура метода **HandleMouseMoveEvent**

Координаты всегда рассчитываются относительно самого «верхнеуровневого» экрана из открытых в данный момент – то есть если в визуализации открыт диалог, из которого открыт еще один диалог и курсор находится в нем – то координаты будут рассчитываться относительно верхней левой точки этого «верхнего» диалога.

Если на выход метода присвоить значение **TRUE** – то данное событие не будет обработано другими обработчиками (в т. ч. подсистемой визуализации CODESYS – то есть в этом случае нажатие, отпускание и перемещение курсора не будет обрабатываться).

Экземпляр ФБ, реализующего данный интерфейс и являющегося обработчиком соответствующего события, должен быть зарегистрирован с помощью вызова метода VisuElems.VisuElemBase.Visu_Globals.g_VisuEventManager.SetMouseEventHandler. Единственным входом метода является экземпляр данного интерфейса (соответственно, подойдет и экземпляр ФБ, реализующего данный интерфейс). Метод возвращает **TRUE** в случае успешной регистрации обработчика и **FALSE** – если обработчик данного интерфейса уже был зарегистрирован (для рассматриваемых в п. 7 интерфейсов нельзя зарегистрировать несколько обработчиков).

7.3.3. Обзор и описание примера

Приведенный ниже пример достаточно синтетический, но, тем не менее, позволяет ознакомиться с рассматриваемым в данном пункте функционалом.

В отличие от предыдущих примеров документа – пример сделан для виртуального контроллера **CODESYS Control Win V3**. Дело в том, что для панельного контроллера СПК зарегистрировать обработчик курсора не получится – это уже сделано в библиотеке, обслуживающей компонент **Screen** (см. подробнее в начале п. 7.3.1). В п. 7.3.5 я опишу подготовленные нами системные переменные, которые частично могут компенсировать это решение. Вы также можете запустить пример на контроллере ОВЕН ПЛК210 или ПЛК200, поменяв в проекте таргет-файл.

На экране визуализации **Visualization** размещен индикатор, отображающий значение переменной **xVisuLamp** программы **VISU_LOGIC**. Эта же переменная привязана к переключателю, нажатие на который инвертирует ее значение. С помощью чекбокса можно отключить обработку нажатий курсора для клиентов web-визуализации – после этого нажатие на переключатель в клиенте web-визуализации не будет приводить к какому-либо эффекту. Повторно включить обработку можно будет из таргет-визуализации, сервисной визуализации среды CODESYS или же изменив значение переменной **xDisableCursorHandlingForWebVisu** программы **VISU_LOGIC**.

Также на экране отображаются:

- координаты последнего нажатия/отпускания курсора для последнего клиента визуализации, который совершил данное действие;
- координаты последнего перемещения курсора для последнего клиента визуализации, который совершил данное действие;
- тип клиента визуализации, который произвел последнее нажатие/отпускание курсора (для клиента web-визуализации – еще и его IP-адрес);
- счетчик нажатий курсора (если точнее – его отпусканий) клиентом web-визуализации с IP-адресом **127.0.0.1** (это localhost-адрес ПК, на котором запущен виртуальный контроллер). Нажатие курсора другими клиентами визуализации не приводит к увеличению значения счетчика.

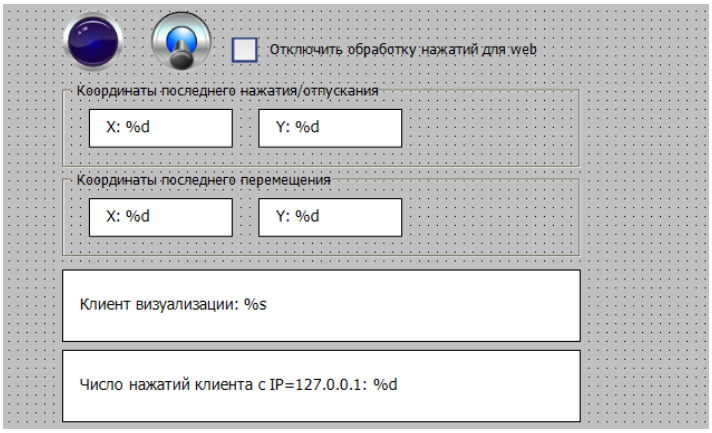


Рисунок 7.3.3.1 – Внешний вид экрана примера

Готовый пример доступен по ссылке: скачать

Для начала создадим ФБ, который будет реализовывать (**IMPLEMENTS**) интерфейс VisuElems.VisuElemBase.IMouseEventHandler. Назовем его **MouseEventHandler**.

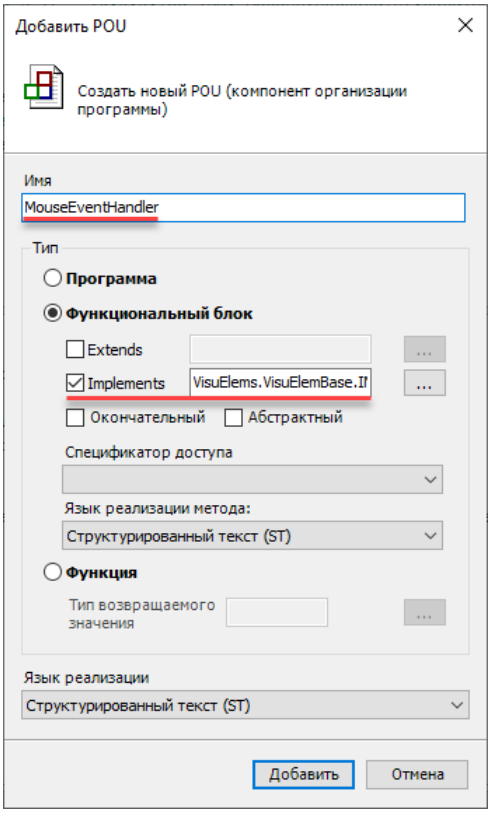


Рисунок 7.3.3.2 – Создание ФБ, реализующего интерфейс **VisuElems.VisuElemBase.IMouseEventHandler**

В его автоматически полученных от интерфейса методах допишем пространства имен для входов **ptMouse** и **pClient**, чтобы избежать ошибок компиляции:

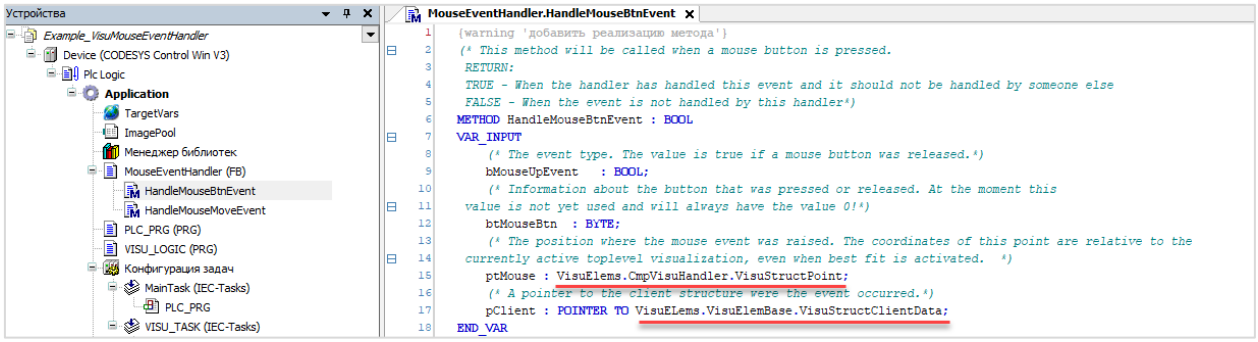


Рисунок 7.3.3.3 – Добавление пространства имен для переменных. Такую же процедуру надо провести в методе **HandleMouseMoveEvent**

Объявим в области входов ФБ указатели на значения, которые мы хотим получить в нашей программе:

```
FUNCTION_BLOCK MouseEventHandler IMPLEMENTS VisuElems.VisuElemBase.IMouseEventHandler
VAR_INPUT
   pxDisableCursorHandlingForWebVisu:  POINTER TO BOOL;
   psClientDefinition:                 POINTER TO STRING;
   puiSpecificClientEventCounter:      POINTER TO UINT;
   pstCurrentCursorPosition:      POINTER TO VisuElems.CmpVisuHandler.VisuStructPoint;
   pstLastCursorPressPosition:    POINTER TO VisuElems.CmpVisuHandler.VisuStructPoint;
END_VAR
VAR_OUTPUT
END_VAR
VAR
END_VAR
```

Теперь добавим реализацию методов **HandleMouseBtnEvent** и **HandleMouseMoveEvent**. Я не перевожу автоматически созданные комментарии – методы уже были описаны в п. 7.3.2.

```
{warning 'добавить реализацию метода'}
(* This method will be called when a mouse button is pressed.
    RETURN:
    TRUE - When the handler has handled this event and it should not be
    handled by someone else
    FALSE - When the event is not handled by this handler
*)
METHOD HandleMouseBtnEvent : BOOL
VAR_INPUT
   (* The event type. The value is true if a mouse button was released.*)
   bMouseUpEvent : BOOL;
   (* Information about the button that was pressed or released. At the
   moment this value is not yet used and will always have the value 0!*)
   btMouseBtn    : BYTE;
   (* The position where the mouse event was raised. The coordinates of this
   point are relative to the currently active toplevel visualization, even
   when best fit is activated.  *)
   ptMouse       : VisuElems.CmpVisuHandler.VisuStructPoint;
   (* A pointer to the client structure were the event occurred.*)
   pClient       : POINTER TO VisuELems.VisuElemBase.VisuStructClientData;
END_VAR
VAR
   sIpAddr:                   STRING;
   xIsIpAddr:                 BOOL;
   fbClientTagDataHelper:   VisuElems.VisuElemBase.VisuFbClientTagDataHelper;
END_VAR
```

Локальные переменные объявлены нами. Функциональный блок **VisuFbClientTagDataHelper** позволяет извлечь из контекста клиента интересную информацию – в частности, IP-адрес клиента web-визуализации. Он будет сохранен в нашу переменную **sIpAddr**. В переменную **xIsIpAddr** будет установлен флаг наличия у клиента IP-адреса.

Код метода будет выглядеть следующим образом:

```
pstLastCursorPressPosition^ := ptMouse;

fbClientTagDataHelper(pClientData := pClient, xIPv4Valid => xIsIpAddr,
   stIPv4 => sIpAddr);

IF pClient^.GlobalData.ClientType =
   VisuElems.VisuElemBase.Visu_ClientType.WebVisualization THEN

   HandleMouseBtnEvent := pxDisableCursorHandlingForWebVisu^;

   psClientDefinition^ := CONCAT('Web-visu, IP = ', sIpAddr);

   // если клиент с IP-адресом 127.0.1 отпустил (bMouseUpEvent) курсор,
   // то увеличиваем значение счетчика
   IF bMouseUpEvent AND sIpAddr = '127.0.0.1' THEN
      puiSpecificClientEventCounter^ := puiSpecificClientEventCounter^ + 1;
   END_IF

ELSIF pClient^.GlobalData.ClientType =
   VisuElems.VisuElemBase.Visu_ClientType.TargetVisualization THEN

   psClientDefinition^ := 'Target-visu';

END_IF
```

Координаты последнего нажатия/отпускания курсора записываются по указателю **pstLastCursorPressPosition**.

Далее мы вызываем экземпляр ФБ **VisuFbClientTagDataHelper**, чтобы определить IP-адрес клиента web-визуализации и записать его в переменную **sIpAddr**. В переменную **xIsIpAddr** будет установлен флаг наличия у клиента IP-адреса.

В операторе **IF** мы проверяем тип клиента визуализации, который нажал или отпустил курсор. Если это клиент web-визуализации, то мы передаем на выход метода значение по указателю **pxDisableCursorHandlingForWebVisu** (это позволяет отключить обработку курсора подсистемой визуализации CODESYS – для этого нужно будет передать на выход метода значение **TRUE**) и формируем строку с обозначением его типа и IP-адресом, и записываем ее по указателю **psClientDefinition**.

После этого во вложенном **IF** мы проверяем два условия:

- что текущий клиент – это клиент web-визуализации с IP-адресом **127.0.0.1** (то есть это клиент web-визуализации, открытой на том же ПК, на котором запущен виртуальный контроллер);
- что этот клиент отпустил курсор (в этом случае вход метода **bMouseUpEvent** получает значение **TRUE**).

Если оба условия выполняются – то мы инкрементируем значение счетчика по указателю **puiSpecificClientEventCounter**.

Обратите внимание, что метод вызывается однократно в момент нажатия или отпускания. Если курсор зажат – то метод не будет вызываться циклически.

Реализация метода **HandleMouseMoveEvent** состоит из одной строки кода – в ней мы записываем по указателю **pstCurrentCursorPosition** координаты текущего положения курсора.

```
{warning 'добавить реализацию метода'}
(* This method will be called when the mouse was moved.
   RETURN:
   TRUE - When the handler has handled this event and it should not be
   handled by someone else
   FALSE - When the event is not handled by this handler
*)
METHOD HandleMouseMoveEvent : BOOL
VAR_INPUT
   (* The position where the mouse was moved to. The coordinates of this
   point are relative to the currently active toplevel visualization, even
   when best fit is activated. *)
   ptMouse       : VisuElems.CmpVisuHandler.VisuStructPoint;
   (* A pointer to the client structure were the event occurred.*)
   pClient       : POINTER TO VisuELems.VisuElemBase.VisuStructClientData;
END_VAR
```

```
pstCurrentCursorPosition^ := ptMouse;
```

В программе **VISU_LOGIC**, привязанной к задаче VISU_TASK, объявим экземпляр нашего ФБ и вспомогательные переменные:

```
// Программа привязана к задаче VISU_TASK
PROGRAM VISU_LOGIC
VAR
   // Обработчик курсора
   fbMouseEventHandler:                  MouseEventHandler;
   // Команда инициализации
   xInit:                                BOOL;

   // Переменная, привязанная к лампе и кнопке
   xVisuLamp:                            BOOL;

   // Тип клиента визуализации
   sClientDefinition:                    STRING;
   // Счетчик нажатий на кнопку мыши для клиента с IP = 127.0.0.1
   //(см. MouseEventHandler.HandleMouseBtnEvent)
   uiSpecificClientEventCounter:  UINT;

   // Текущие координаты курсора для последнего клиента, перемещавшего курсор
   stCurrentCursorPosition: VisuElems.CmpVisuHandler.VisuStructPoint;
   // Координаты последнего нажатия курсора любым из клиентов
   stLastCursorPressPosition: VisuElems.CmpVisuHandler.VisuStructPoint;

   // Отключение обработки курсора для клиентов web-визуализации
   xDisableCursorHandlingForWebVisu:     BOOL;
END_VAR
```

Код программы довольно прост – при ее старте мы однократно регистрируем экземпляр нашего ФБ в качестве обработчика курсора и передаем на его входные указатели адреса переменных программы.

```
IF NOT(xInit) THEN

   fbMouseEventHandler.pxDisableCursorHandlingForWebVisu :=
      ADR(xDisableCursorHandlingForWebVisu);
   fbMouseEventHandler.psClientDefinition := ADR(sClientDefinition);
   fbMouseEventHandler.puiSpecificClientEventCounter :=
      ADR(uiSpecificClientEventCounter);
   fbMouseEventHandler.pstLastCursorPressPosition :=
      ADR(stLastCursorPressPosition);
   fbMouseEventHandler.pstCurrentCursorPosition :=
      ADR(stCurrentCursorPosition);

  VisuElems.VisuElemBase.Visu_Globals.g_VisuEventManager.SetMouseEventHandler
    (fbMouseEventHandler);

   xInit := TRUE;

END_IF
```

Проект полностью готов. Давайте загрузим его в виртуальный контроллер и проверим, как он работает.

7.3.4. Проверка работы примера

Подключитесь к web- или таргет-визуализации виртуального контроллера. Перемещайте курсор и нажимайте на экран левой кнопкой мыши. Наблюдайте отображение координат текущего положения курсора и координат нажатия/отпускания. Если вы работаете с web-визуализацией, открытой на том же ПК, что и виртуальный контроллер (т. е. клиент web-визуализации определяется как клиент с IP-адресом **127.0.0.1** – localhost) – то после каждого нажатия и отпускания курсора счетчик нажатий будет увеличиваться на **1**. Если установить галочку **Отключить обработку нажатий для web**, то нажатия в web-визуализации перестанут обрабатываться подсистемой визуализации CODESYS – то есть вы не сможете снять галочку и изменить состояние переключателя. Снять ее можно будет из таргет-визуализации, сервисной визуализации среды CODESYS или же изменив значение переменной **xDisableCursorHandlingForWebVisu** программы **VISU_LOGIC**.

На базе примера вы можете разработать свою логику обработки курсора для различных клиентов визуализации, а также другую информацию, которую можно получить от методов интерфейса.

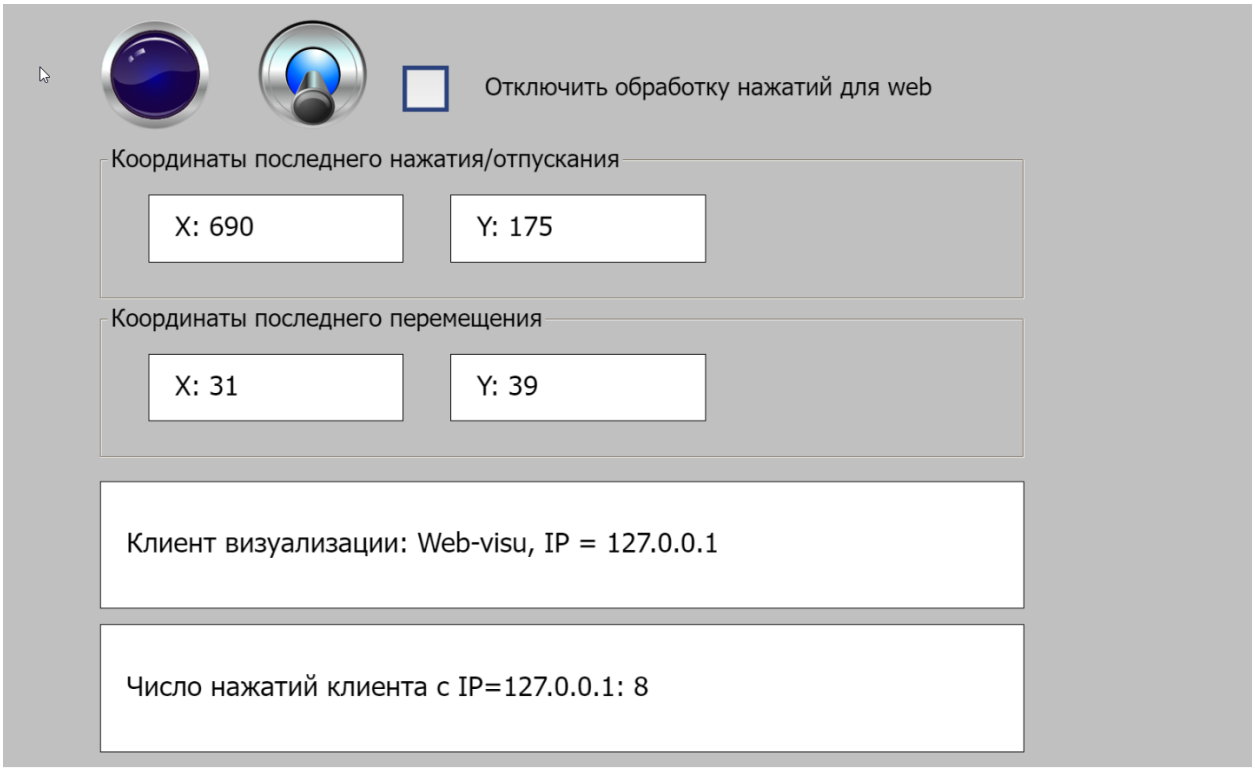


Рисунок 7.3.5.1 – Работа с web-визуализацией примера

7.3.5. Дополнительная информация про обработку курсора для ОВЕН СПК

Как я уже упоминал ранее – повторить пример пункта на сенсорном панельном контроллере ОВЕН СПК не получится, потому что обработчик курсора будет зарегистрирован в библиотеке компонента **Screen**, и зарегистрировать еще один обработчик в приложении пользователя не выйдет (метод SetMouseEventHandler вернет **FALSE**).

Чтобы в некоторой степени компенсировать этот момент – в компонент были добавлены следующие переменные, доступные в приложении пользователя и предоставляющие информацию о курсоре таргет-визуализации:

- Screen.cursorX (**INT**) – координата последнего нажатия курсора по оси X;
- Screen.cursorY (**INT**) – координата последнего нажатия курсора по оси Y;
- Screen.countTouch (**UDINT**) – число нажатий на экран с момента включения;
- Screen.uiRotateAngle (**UINT**) – угол поворота экрана (0/90/180/270).

Обработать курсор для клиентов web-визуализации СПК с помощью интерфейса IMouseEventHandler не получится. Но так как в большинстве задач основным способом работы с панельным контроллером является таргет-визуализация, то это не кажется мне существенным недостатком.

Напоследок я упомяну еще об одном моменте, который не касается непосредственно СПК, но тоже связан с обработкой курсора. В контексте визуализации доступна системная переменная **ptMouse** (типа **VisuElemVisuStructPoint**), которая содержит координаты последнего нажатия курсора. Например, эту переменную можно использовать в настройках кнопки открытия диалога – чтобы он открывался непосредственно рядом с этой кнопкой (по умолчанию диалог открывается по центру экрана).

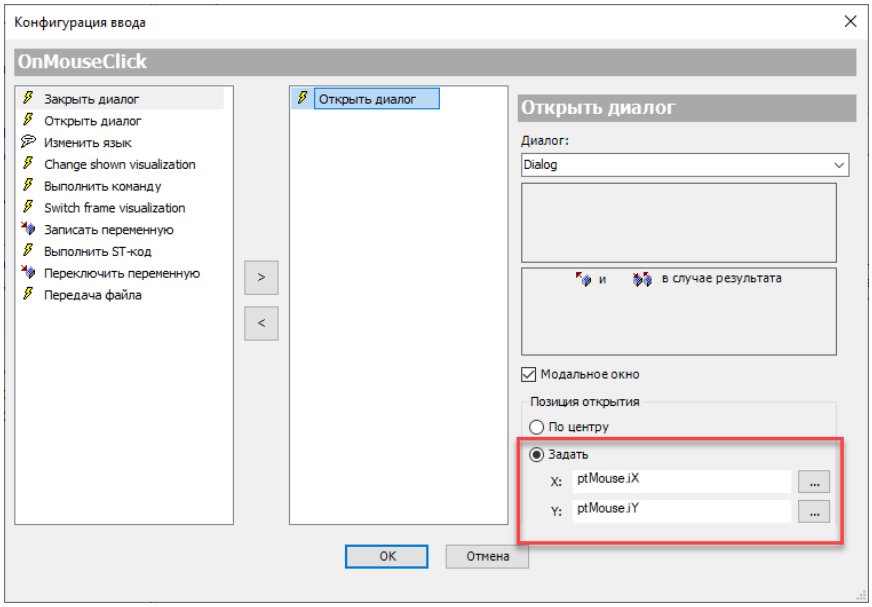


Рисунок 7.3.5.1 – Использование системной переменной **ptMouse** в настройках действия **Открыть диалог**

7.4. Обработка событий в элементе визуализации (IInputOnElementEventHandler)

7.4.1. Основная информация

Однажды разработчикам CODESYS пришло письмо от инженеров Mitsubishi India. В нем они описали задачу, в ходе решения которой столкнулись с трудностями – им было нужно, чтобы панельный контроллер издавал звуковой сигнал при нажатии на любой из «активных» элементов визуализации (кнопку, переключатель и т. д.). Такая звуковая индикация нажатий характерна для панелей оператора – обычно она либо присутствует по умолчанию, либо включается одной галочкой в конфигураторе панели.

Рисунок 7.4.1.1 – Пожелание в баг-трекере CODESYS, приведшее к появлению интерфейса **IInputOnElementEventHandler**

Разработчики CODESYS обдумали ситуацию и версии **V3.5 SP17** добавили еще один интерфейс визуализации с названием **IInputOnElementEventHandler**, чтобы разработчик приложения мог получить в коде программе информацию о том, что с элементом было выполнено какое-то действие (например, нажатие).

7.4.2. Обзор интерфейса IInputOnElementEventHandler

Интерфейс IInputOnElementEventHandler входит в состав библиотеки VisuElemBase и содержит единственный метод HandleInputOnElementEvent. Этот метод вызывается при возникновении в элементе события, настроенного во вкладке **Конфигурация ввода**. Метод поддерживает только 4 события из доступных – **OnMouseDown**, **OnMouseUp**, **OnMouseClick** и **OnMouseMove**. Для остальных событий метод не вызывается. В событии должно быть настроено хотя бы одно действие. Если в элементе последовательно происходит несколько событий (например, сначала **OnMouseDown**, а потом **OnMouseUp**) – то метод будет вызван для каждого из них.

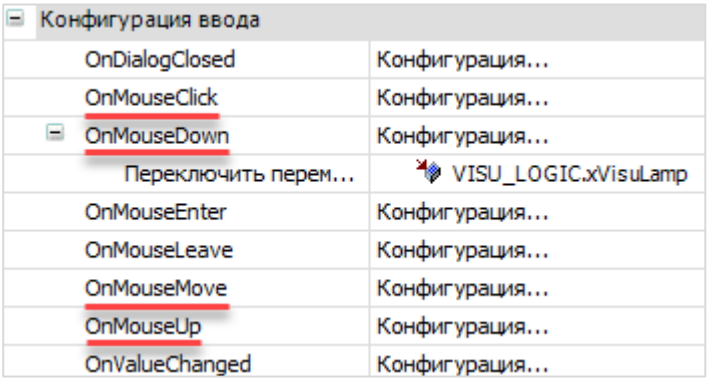


Рисунок 7.4.2.1 – События элемента визуализации, приводящие к вызову метода

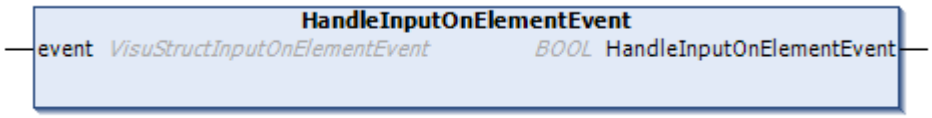


Рисунок 7.4.2.1 – Сигнатура метода **HandleInputOnElementEvent**

Метод имеет единственный вход – экземпляр структуры типа VisuStructInputOnElementEvent. Структура включает 3 поля:

- **pClient** – указатель на контекст клиента визуализации, выполнившего действия с элементом;
- **itfVisualElement** – экземпляр интерфейса элемента, с которым произошло событие. Методы элемента поверхностно (3.5 страниц на 16 методов) описаны в OEM-документации. Например, метод **GetText** возвращает текст элемента визуализации;
- **eType** – экземпляр перечисления VisuEnumInputOnElementType, описывающий событие, произошедшее в элементе (**MouseDown**, **MouseUp**, **MouseClick** или **MouseMove**).

Выход метода не используется.

Экземпляр ФБ, реализующего данный интерфейс и являющегося обработчиком соответствующего события, должен быть зарегистрирован с помощью вызова метода VisuElems.VisuElemBase.Visu_Globals.g_VisuEventManager.SetInputOnElementEventHandler.
Единственным входом метода является экземпляр данного интерфейса (соответственно, подойдет и экземпляр ФБ, реализующего данный интерфейс). Метод возвращает **TRUE** в случае успешной

регистрации обработчика и **FALSE** – если обработчик данного интерфейса уже был зарегистрирован (для рассматриваемых в п. 7 интерфейсов нельзя зарегистрировать несколько обработчиков).

7.4.3. Обзор и описание примера

Приведенный ниже пример достаточно синтетический, но, тем не менее, позволяет ознакомиться с рассматриваемым в данном пункте функционалом.

На экране визуализации **Visualization** расположены два индикатора (красный и синий), к которым привязаны переменные программы VISU_LOGIC **xLampRed** и **xLampBlue** соответственно. К расположенным под ними кнопками привязано действие **Переключить переменную** – причем для кнопки красного индикатора оно настроено в событии **OnMouseDown**, а для кнопки синего индикатора – в событии **OnMouseUp**. В переменные программы **VISU_LOGIC** считывается следующая информация (она же будет отображаться на экране визуализации):

- тип клиента визуализации, который произвел последнее нажатие на любую из кнопок (для клиента web-визуализации – еще и его IP-адрес);
- имя последнего произошедшего события ввода (**MouseDown** или **MouseUp**);
- ID элемента, в котором произошло событие (см. столбец **Id** во вкладке **Список элементов** экрана визуализации). Более удобного способа идентифицировать элемент в современных версиях CODESYS нет – в частности, определить имя элемента с помощью **itfVisualElement** не получится (это сообщила техподдержка CODESYS).

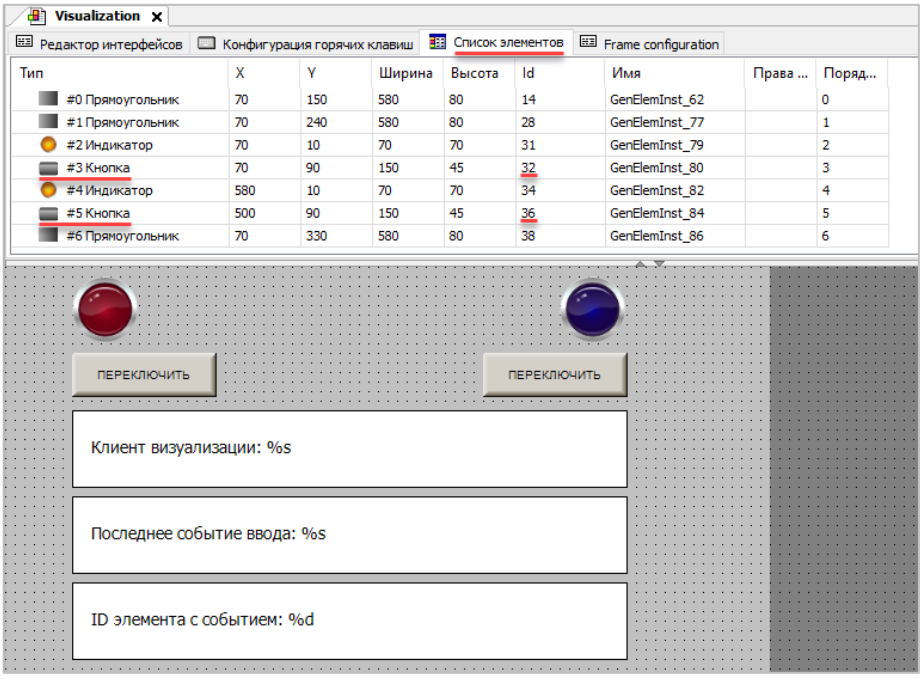


Рисунок 7.4.3.1 – Внешний вид экрана примера

Готовый пример доступен по ссылке: скачать

Для начала создадим ФБ, который будет реализовывать (**IMPLEMENTS**) интерфейс VisuElems.VisuElemBase.IInputOnElementEventHandler. Назовем его **InputOnElementEventHandler**.

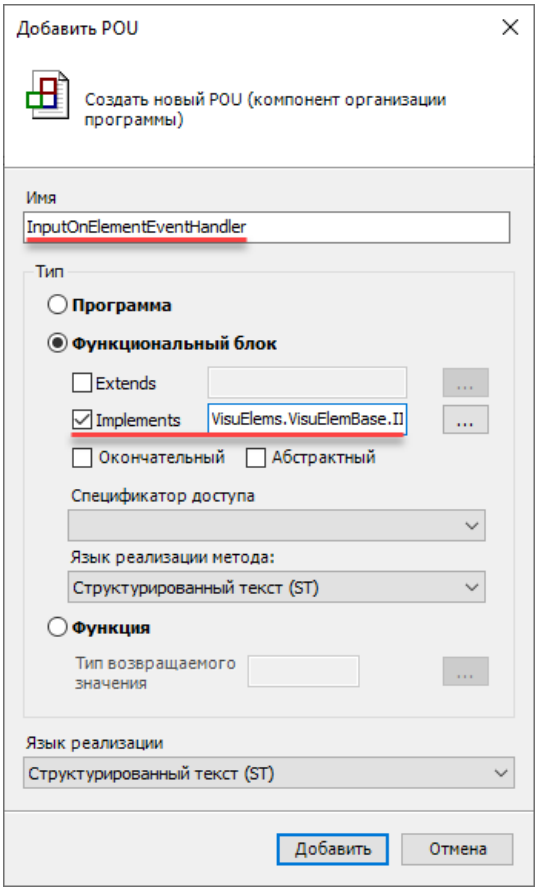


Рисунок 7.4.3.2 – Создание ФБ, реализующего интерфейс **VisuElems.VisuElemBase.IInputOnElementEventHandler**

В его автоматически полученном от интерфейса методе **HandleInputOnElementEvent** допишем пространство имен для входа event, чтобы избежать ошибок компиляции:

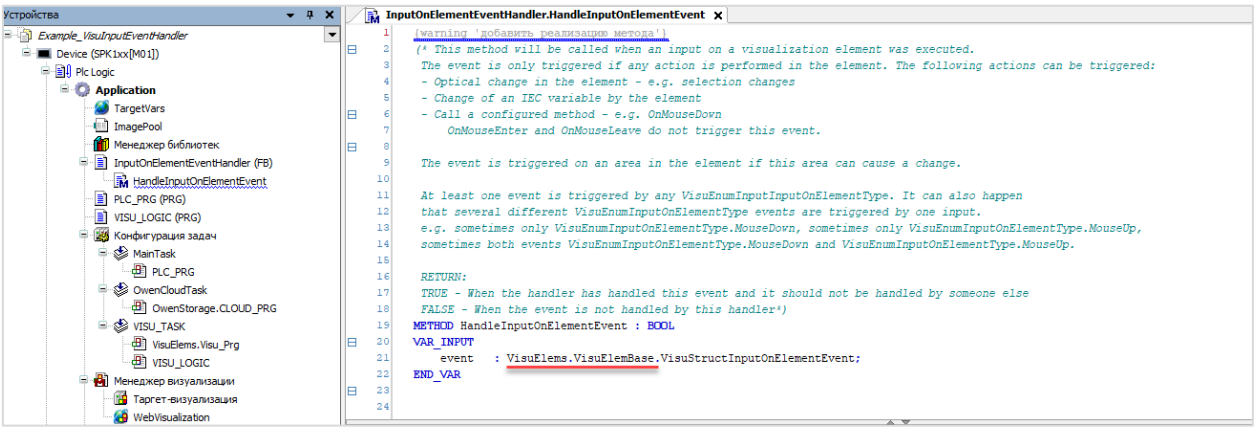


Рисунок 7.4.3.3 – Добавление пространства имен для переменной

Объявим в области входов ФБ указатели на значения, которые мы хотим получить в нашей программе:

```
FUNCTION_BLOCK InputOnElementEventHandler IMPLEMENTS
   VisuElems.VisuElemBase.IInputOnElementEventHandler
VAR_INPUT
   psClientDefinition:            POINTER TO STRING;
   psLastInputEvent:              POINTER TO STRING;
   paiLastInputElementId:         POINTER TO ARRAY
      [0..VisuElems.VisuElemBase.Visu_Selection_Constants.MAX_POSITION_DEPTH] OF INT;
END_VAR
```

Заметим, что мы будем передавать в программу не просто ID элемента визуализации, в котором произошло событие, а целый массив. Это связано с тем, что элемент может быть размещен внутри фрейма или «цепочки» вложенных фреймов. Например, представим, что на экране визуализации расположен фрейм с ID = **42**, внутри которого расположен еще один фрейм с ID = **2**, на экране которого размещен элемент с ID = **4**, в котором приозошло событие. Тогда массив будет заполнен следующими значениями: [**42**, **2**, **4**, -1, -1, … , -1] – то есть сначала в нем будут последовательно записаны ID фреймов, затем – ID элемента, а «неиспользованные» элементы массива сохранят значение **-1** (выбор значения **-1** связан с тем, что ID может иметь значение **0**). Константа **MAX_POSITION_DEPTH** определяет допустимую «глубину вложенности» фреймов. В текущих версиях CODESYS ее значение равно **19**.

Теперь добавим реализацию метода **HandleInputOnElementEvent**. Я не перевожу автоматически созданные комментарии – метод уже был описан в п. 7.4.2.

```
{warning 'добавить реализацию метода'}
(* This method will be called when an input on a visualization element was executed.
 The event is only triggered if any action is performed in the element. The following
actions can be triggered:
 - Optical change in the element - e.g. selection changes
 - Change of an IEC variable by the element
 - Call a configured method - e.g. OnMouseDown
     OnMouseEnter and OnMouseLeave do not trigger this event.

 The event is triggered on an area in the element if this area can cause a change.

 At least one event is triggered by any VisuEnumInputInputOnElementType. It can also
happen
 that several different VisuEnumInputOnElementType events are triggered by one input.
 e.g. sometimes only VisuEnumInputOnElementType.MouseDown, sometimes only
VisuEnumInputOnElementType.MouseUp,
 sometimes both events VisuEnumInputOnElementType.MouseDown and
VisuEnumInputOnElementType.MouseUp.

 RETURN:
 TRUE - When the handler has handled this event and it should not be handled by
someone else
 FALSE - When the event is not handled by this handler*)
```

```
METHOD HandleInputOnElementEvent : BOOL
VAR_INPUT
   event : VisuElems.VisuElemBase.VisuStructInputOnElementEvent;
END_VAR
VAR
   sIpAddr:                    STRING;
   xIsIpAddr:                  BOOL;
   fbClientTagDataHelper:   VisuElems.VisuElemBase.VisuFbClientTagDataHelper;
END_VAR
VAR CONSTANT
   // Названия обрабатываемых событий
   c_asInputEventNames:  ARRAY [0..3] OF STRING :=
     ['MouseDown', 'MouseUp', 'MouseClick', 'MouseMove'];
END_VAR
```

Локальные переменные и константа объявлены нами. Функциональный блок **VisuFbClientTagDataHelper** позволяет извлечь из контекста клиента интересную информацию – в частности, IP-адрес клиента web-визуализации. Он будет сохранен в нашу переменную **sIpAddr**. В переменную **xIsIpAddr** будет установлен флаг наличия у клиента IP-адреса.

Константа c**_asInputEventNames** содержит имена обрабатываемых событий. Мы будем использовать ее для конвертации кода события (**event.eType**) в его название для отображения в визуализации.

Код метода будет выглядеть следующим образом:

```
psLastInputEvent^ := c_asInputEventNames[event.eType];

paiLastInputElementId^ := event.pClient^.Inputdata.InputInfo.aiInputPosition;

fbClientTagDataHelper(pClientData := event.pClient, xIPv4Valid => xIsIpAddr,
   stIPv4 => sIpAddr);

IF event. pClient^.GlobalData.ClientType =
   VisuElems.VisuElemBase.Visu_ClientType.WebVisualization THEN

   psClientDefinition^ := CONCAT('Web-visu, IP = ', sIpAddr);

ELSIF event.pClient^.GlobalData.ClientType =
   VisuElems.VisuElemBase.Visu_ClientType.TargetVisualization THEN

   psClientDefinition^ := 'Target-visu';

END_IF
```

По указателю **psLastInputEvent** записывается название последнего события (оно формируется путем выбора из массива **c_asInputEventNames** элемента с индексом, значение которого совпадает с кодом события; на уровне создания массива специально установлено соответствие между кодами и соответствующими строками).

Массив идентификаторов, определяющих элемент визуализации, в котором произошло событие, извлекается из контекста клиента и записывается по указателю **paiLastInputElementId**.

Далее мы вызываем экземпляр ФБ **VisuFbClientTagDataHelper**, чтобы определить IP-адрес клиента web-визуализации и записать его в переменную **sIpAddr**. В переменную **xIsIpAddr** будет установлен флаг наличия у клиента IP-адреса.

В операторе **IF** мы проверяем тип клиента визуализации, который произвел событие в элементе визуализации. Если это клиент web-визуализации, то мы формируем строку с обозначением его типа и IP-адресом, и записываем ее по указателю **psClientDefinition**. Для клиента таргет-визуализации мы просто записываем название его типа.

В программе **VISU_LOGIC**, привязанной к задаче VISU_TASK, объявим экземпляр нашего ФБ и вспомогательные переменные:

```
// Программа привязана к задаче VISU_TASK
PROGRAM VISU_LOGIC
VAR
   // Обработчик событий элементов визуализации
   fbInputEventHandler:                                    InputOnElementEventHandler;

   // Команда инициализации
   xInit:                                                  BOOL;

   // Переменные, привязанные к лампам
   // На кнопках настроено их переключение
   xLampRed:                                               BOOL;
   xLampBlue:                                              BOOL;

   // Тип клиента визуализации
   sClientDefinition:                                      STRING;
   // Тип события
   sLastInputEvent:                                        STRING;
   // Массив с ID элемента (см. экран визуализации - Список элементов - ID)
   // Массив используется по той причине, что элемент может быть размещен в фрейме
   // (или в фрейме, вложенном в другие фреймы)
   // В этом случае в массив сначала записываются ID элементов Фрейм.
   // Последнее значимое (не равное -1) число в массиве определяет ID самого элемента
   aiLastInputElementId:          ARRAY
      [0..VisuElems.VisuElemBase.Visu_Selection_Constants.MAX_POSITION_DEPTH] OF INT;
END_VAR
```

Код программы довольно прост – при ее старте мы однократно регистрируем экземпляр нашего ФБ в качестве обработчика событий элементов визуализации и передаем на его входные указатели адреса переменных программы.

```
IF NOT(xInit) THEN

   fbInputEventHandler.psClientDefinition    := ADR(sClientDefinition);
   fbInputEventHandler.psLastInputEvent      := ADR(sLastInputEvent);
   fbInputEventHandler.paiLastInputElementId := ADR(aiLastInputElementId);

  VisuElems.VisuElemBase.Visu_Globals.g_VisuEventManager.SetInputOnElementEventHandler
     (fbInputEventHandler);

  xInit := TRUE;

END_IF
```

Проект полностью готов. Давайте загрузим его в контроллер и проверим, как он работает.

7.4.4. Проверка работы примера

Подключитесь к web- или таргет-визуализации контроллера. Нажимайте на кнопки **Переключить** и наблюдайте отображение информации о клиенте, который нажал кнопку, типе события, произошедшего в кнопке, и ID элемента (см. рис. 7.4.3.1), в котором произошло событие.

На базе примера вы можете разработать свою логику обработки событий, учитывая в ней всю перечисленную выше информацию, а также другую информацию, которую можно получить от метода интерфейса.

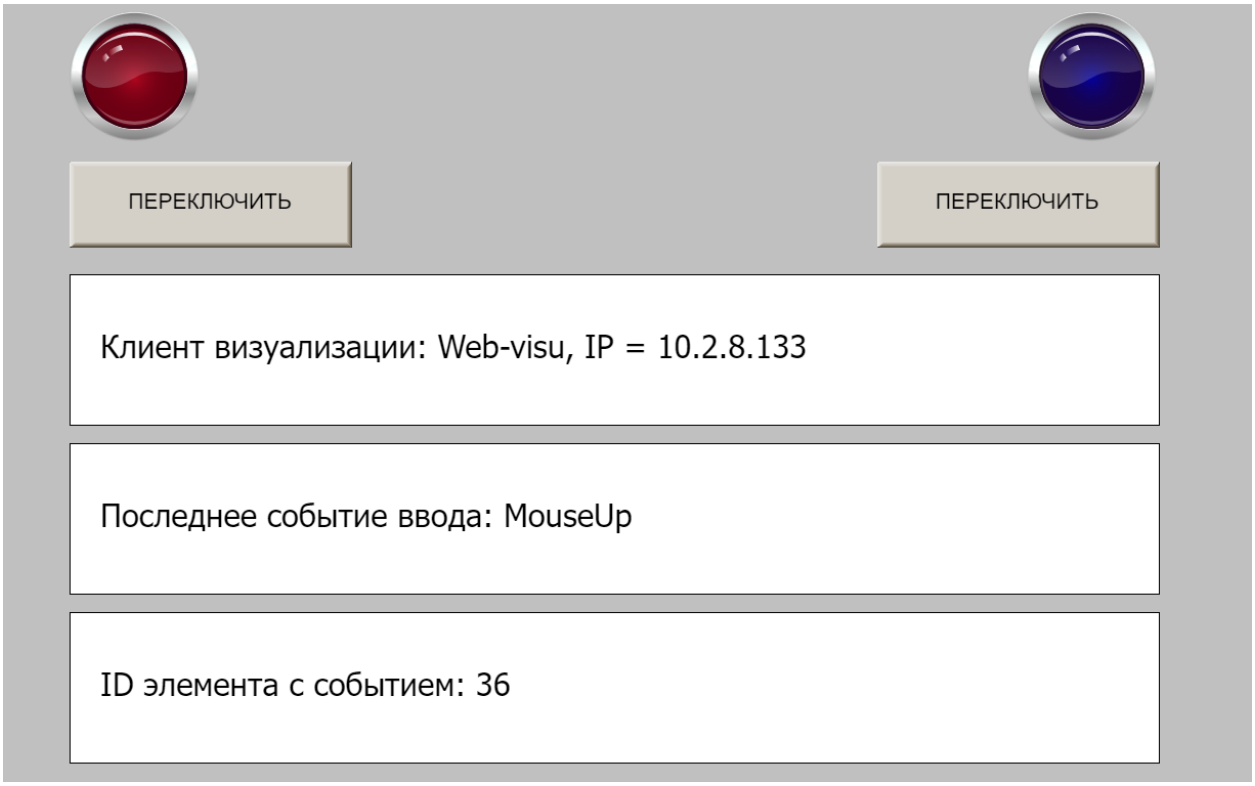


Рисунок 7.4.4.1 – Работа с web-визуализацией примера

7.5. Обработка ввода значения в элементе визуализации (IEditBoxInputHandler)

7.5.1. Основная информация

Оператор, работающий с визуализацией, может вводить новые значения уставок, настроек и других переменных с помощью аппаратной клавиатуры или диалогов ввода (Numpad, Keypad и т. д.). В некоторых ситуациях может потребоваться определить в коде проекта факт изменения значений переменных визуализации и произвести какую-то обработку (например, валидацию). Для этого используется интерфейс **IEditBoxInputHandler**.

7.5.2. Обзор интерфейса IEditBoxInputHandler

Интерфейс **IEditBoxInputHandler** входит в состав библиотеки VisuElemBase и содержит единственный метод **VariableWritten**. Этот метод вызывается при завершении ввода нового значения с помощью элемента визуализации, для которого настроено действие **Записать переменную**. Завершением ввода считается нажатие на кнопку **ОК** в диалоге ввода или нажатие на **Enter** при вводе с помощью аппаратной клавиатуры. Если у элемента не настроено действие **Записать переменную** – то метод вызываться не будет (например, он не будет вызываться при изменении значения с помощью ползунка).

В текущей версии библиотеки VisuElemBase (**4.3.0.0**) интерфейс является скрытым и не отображается в менеджере библиотек.

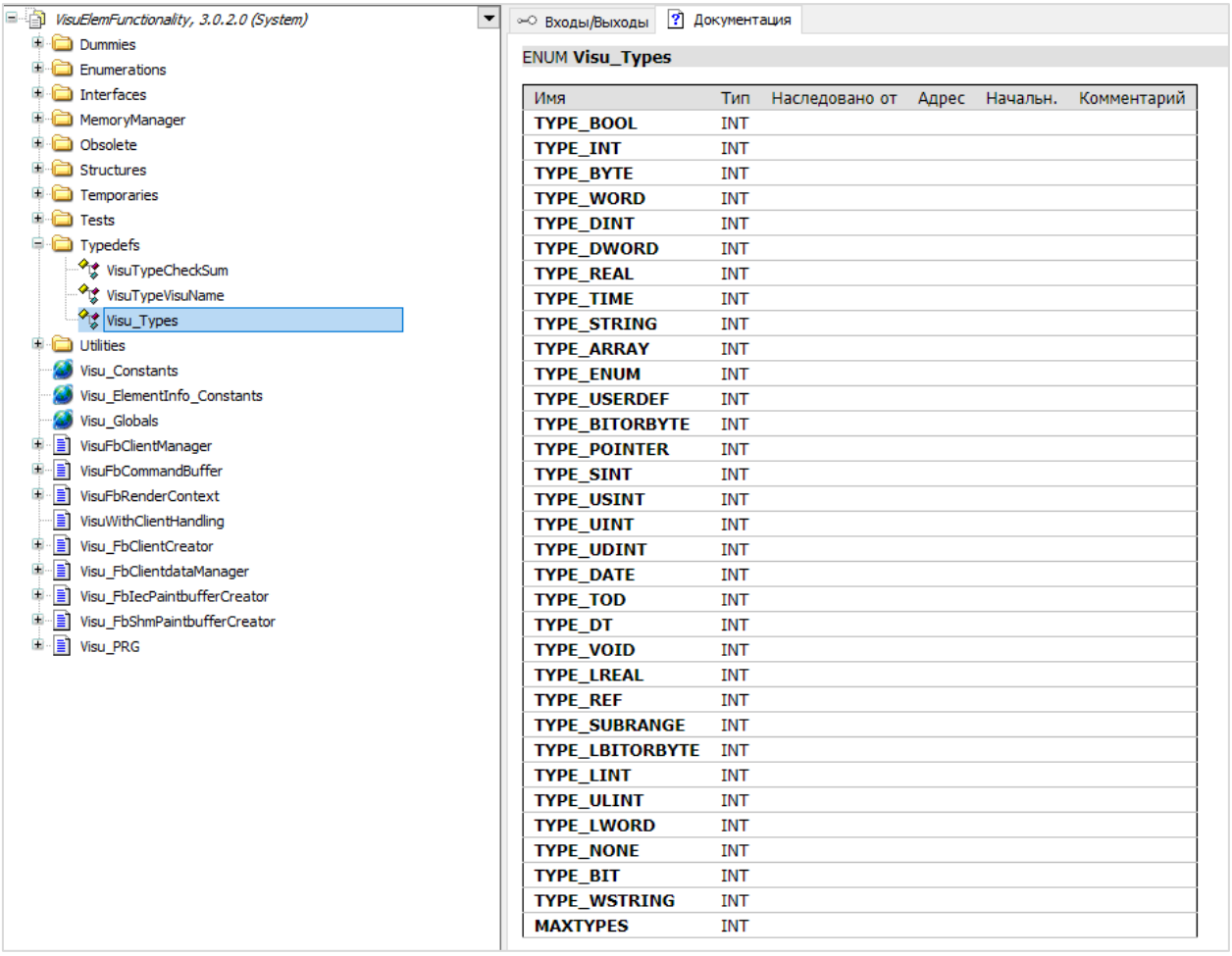
Метод **VariableWritten** имеет следующие входы:

- **pVar** (POINTER TO BYTE) – указатель на введенное значение. Указатель является нетипизированным – разработчик сам должен привести его к указателю на тип переменной, привязанной к элементу;
- **varType** (VisuElems.CmpVisuHandler.Visu_Types**)** – тип введенной переменной (соответствует типу переменной, привязанной к элементу визуализации, с помощью которого был произведен ввод). Перечисление **Visu_Types** в библиотеке CmpVisuHandler является скрытым и не отображается в менеджере библиотек. Посмотреть его содержимое можно в устаревшей библиотеке **VisuElemFunctionality** в папке **Typedefs** (см. рис. 7.5.2.1);
- **iMaxSize** (INT) – размер переменной в байтах. Является валидным только для строк (потому что при объявлении строки можно указать ее максимальную длину, и в отличие от всех остальных типов – размер памяти, занимаемой строкой, нельзя определить, не зная это значение), для остальных типов возвращается отрицательное число;
- **pClient** – указатель на контекст клиента визуализации, который совершил ввод значения с помощью элемента визуализации.

Выход метода не используется.

Экземпляр ФБ, реализующего данный интерфейс и являющегося обработчиком соответствующего события, должен быть зарегистрирован с помощью вызова метода **VisuElems.VisuElemBase.Visu_Globals.g_VisuEventManager.SetEditBoxEventHandler**. Единственным входом метода является экземпляр данного интерфейса (соответственно, подойдет и экземпляр ФБ, реализующего данный интерфейс). Метод возвращает **TRUE** в случае успешной регистрации обработчика и **FALSE** – если обработчик данного интерфейса уже был зарегистрирован (для рассматриваемых в п. 7 интерфейсов нельзя зарегистрировать несколько обработчиков).

Как можно заметить – разработчик заранее должен определить, ввод значений в какие элементы визуализации ему необходимо обрабатывать в коде и учесть типы переменных, привязанных к этим элементам. В примере мы также рассмотрим, как определить ID элемента, на который было совершено нажатие, чтобы однозначно его идентифицировать.



Имя	Тип	Наследовано от	Адрес	Начальн.	Комментарий
TYPE_BOOL	INT				
TYPE_INT	INT				
TYPE_BYTE	INT				
TYPE_WORD	INT				
TYPE_DINT	INT				
TYPE_DWORD	INT				
TYPE_REAL	INT				
TYPE_TIME	INT				
TYPE_STRING	INT				
TYPE_ARRAY	INT				
TYPE_ENUM	INT				
TYPE_USERDEF	INT				
TYPE_BITORBYTE	INT				
TYPE_POINTER	INT				
TYPE_SINT	INT				
TYPE_USINT	INT				
TYPE_UINT	INT				
TYPE_UDINT	INT				
TYPE_DATE	INT				
TYPE_TOD	INT				
TYPE_DT	INT				
TYPE_VOID	INT				
TYPE_LREAL	INT				
TYPE_REF	INT				
TYPE_SUBRANGE	INT				
TYPE_LBITORBYTE	INT				
TYPE_LINT	INT				
TYPE_ULINT	INT				
TYPE_LWORD	INT				
TYPE_NONE	INT				
TYPE_BIT	INT				
TYPE_WSTRING	INT				
MAXTYPES	INT				

Рисунок 7.5.2.1 – Состав перечисления **Visu_Types** в библиотеке **VisuElemFunctionality**. В свежих версиях CODESYS в это перечисление (перенесенное в библиотеку **CmpVisuHandler** и сделанное там скрытым) добавлены новые элементы – например, TYPE_LDATE, TYPE_LTOD и т. д.

7.5.3. Обзор и описание примера

Приведенный ниже пример достаточно синтетический, но, тем не менее, позволяет ознакомиться с рассматриваемым в данном пункте функционалом.

На экране визуализации **Visualization** расположены два прямоугольника, к которым привязаны переменные программы VISU_LOGIC **iVisuValue** (типа **INT**) и **rVisuValue** (типа **REAL**), для которых настроено действие **Записать переменную**.

В переменные программы **VISU_LOGIC** считывается следующая информация (она же будет отображаться на экране визуализации):

- тип клиента визуализации, который произвел последнее нажатие на любую из кнопок (для клиента web-визуализации – еще и его IP-адрес);
- код типа последней записанной переменной (да, я поленился создать функцию, которое бы конвертировала его в строковое название типа);
- значение последней записанной переменной в строковом виде;
- ID элемента, в котором произошло событие (см. столбец **Id** во вкладке **Список элементов** экрана визуализации).

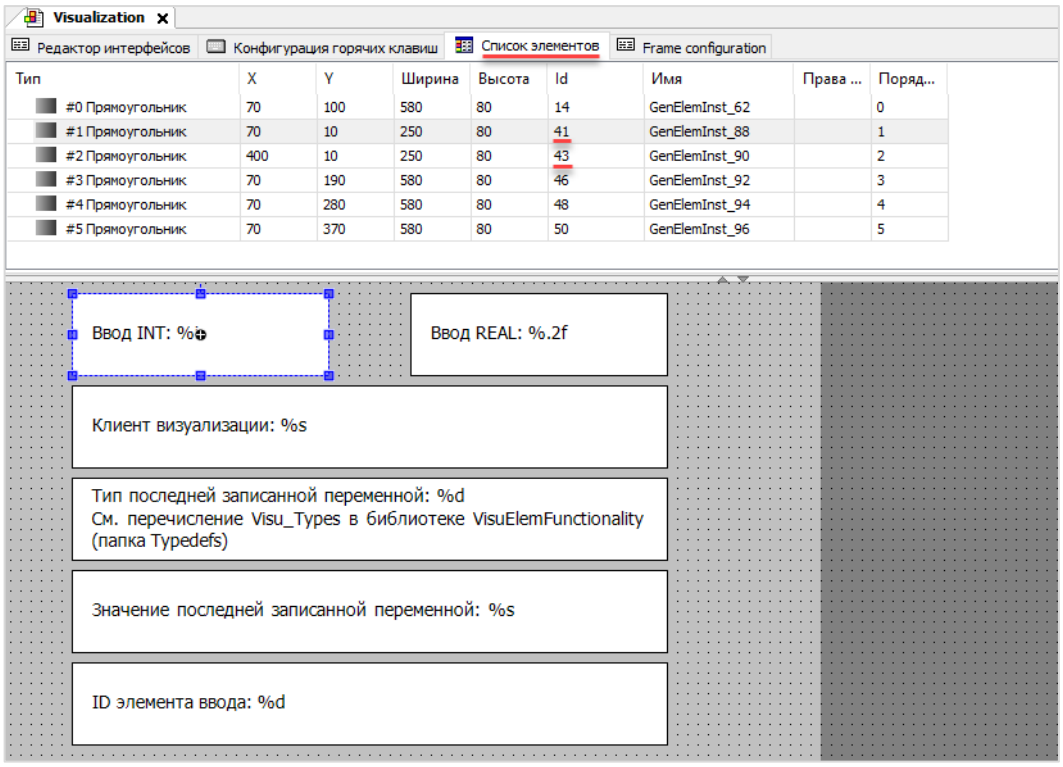


Рисунок 7.5.3.1 – Внешний вид экрана примера

Готовый пример доступен по ссылке: скачать

Для начала создадим ФБ, который будет реализовывать (**IMPLEMENTS**) интерфейс **VisuElems.VisuElemBase.IEditBoxInputHandler**. Назовем его **EditBoxInputHandler**.

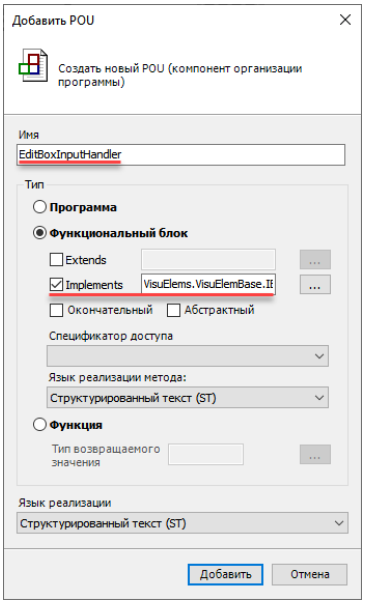


Рисунок 7.5.3.2 – Создание ФБ, реализующего интерфейс **VisuElems.VisuElemBase.IEditBoxInputHandler**

В его автоматически полученном от интерфейса методе **VariableWritten** допишем пространства имен для входов **varType** и **pClient**, чтобы избежать ошибок компиляции:

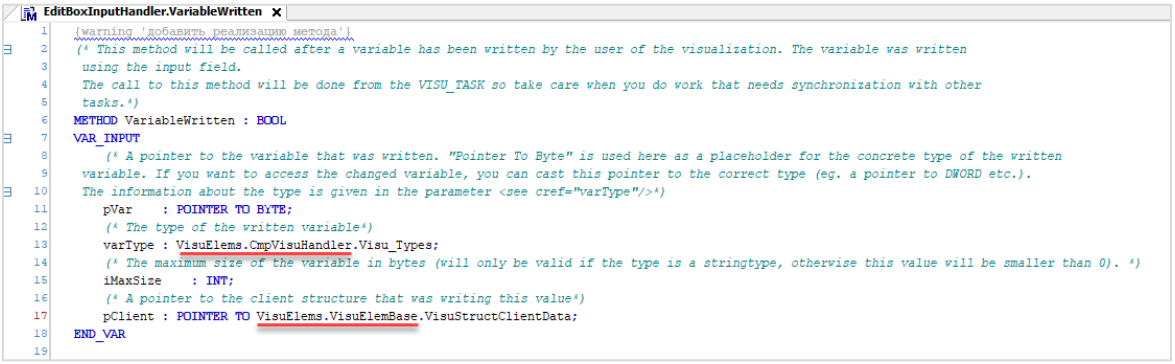


Рисунок 7.5.3.3 – Добавление пространства имен для переменных

Объявим в области входов ФБ указатели на значения, которые мы хотим получить в нашей программе:

```
FUNCTION_BLOCK EditBoxInputHandler IMPLEMENTS
   VisuElems.VisuElemBase.IEditBoxInputHandler
VAR_INPUT
   peLastEditVarType:              POINTER TO VisuElems.CmpVisuHandler.Visu_Types;
   piNewValue:                     POINTER TO INT;
   prNewValue:                     POINTER TO REAL;
   psClientDefinition:             POINTER TO STRING;
   psLastEditVarValue:             POINTER TO STRING;
   paiLastInputElementId:          POINTER TO ARRAY
     [0..VisuElems.VisuElemBase.Visu_Selection_Constants.MAX_POSITION_DEPTH] OF INT;
END_VAR
```

По указателю **piNewValue** будет размещаться последнее записанное значение типа **INT**, а по указателю **prNewValue** – последнее записанное значение типа **REAL**. По указателю **psLastEditVarValue** будет размещаться последнее из записанных значений любого из типов в строковом виде.

Заметим, что мы будем передавать в программу не просто ID элемента визуализации, в котором произошло событие, а целый массив. Это связано с тем, что элемент может быть размещен внутри фрейма или «цепочки» вложенных фреймов. Например, представим, что на экране визуализации расположен фрейм с ID = **42**, внутри которого расположен еще один фрейм с ID = **2**, на экране которого размещен элемент с ID = **4**, в котором приозошло событие. Тогда массив будет заполнен следующими значениями: [**42**, **2**, **4**, -1, -1, … , -1] – то есть сначала в нем будут последовательно записаны ID фреймов, затем – ID элемента, а «неиспользованные» элементы массива сохранят значение **-1** (выбор значения **-1** связан с тем, что ID может иметь значение **0**). Константа **MAX_POSITION_DEPTH** определяет допустимую «глубину вложенности» фреймов. В текущих версиях CODESYS ее значение равно **19**.

Теперь добавим реализацию метода **VariableWritten**. Я не перевожу автоматически созданные комментарии – метод уже был описан в п. 7.5.2.

```
{warning 'добавить реализацию метода'}
(* This method will be called after a variable has been written by the user of the
 visualization. The variable was written using the input field.
 The call to this method will be done from the VISU_TASK so take care when you do work
 that needs synchronization with other tasks.
*)
METHOD VariableWritten : BOOL
VAR_INPUT
   (* A pointer to the variable that was written. "Pointer To Byte" is used here as a
   placeholder for the concrete type of the written variable. If you want to access
   the changed variable, you can cast this pointer to the correct type (eg. a pointer
   to DWORD etc.).
   The information about the type is given in the parameter <see cref="varType"/>
   *)
   pVar  : POINTER TO BYTE;
   (* The type of the written variable *)
   varType       : VisuElems.CmpVisuHandler.Visu_Types;
   (* The maximum size of the variable in bytes (will only be valid if the type is a
   stringtype, otherwise this value will be smaller than 0).
   *)
   iMaxSize      : INT;
   (* A pointer to the client structure that was writing this value*)
   pClient       : POINTER TO VisuElems.VisuElemBase.VisuStructClientData;
END_VAR
VAR
   sIpAddr:                 STRING;
   xIsIpAddr:               BOOL;
   fbClientTagDataHelper:   VisuElems.VisuElemBase.VisuFbClientTagDataHelper;

   piVar:                   POINTER TO INT;
   prVar:                   POINTER TO REAL;
END_VAR
```

Локальные переменные объявлены нами. Функциональный блок **VisuFbClientTagDataHelper** позволяет извлечь из контекста клиента интересную информацию – в частности, IP-адрес клиента web-визуализации. Он будет сохранен в нашу переменную **sIpAddr**. В переменную **xIsIpAddr** будет установлен флаг наличия у клиента IP-адреса. Указатели **piVar** и **prVar**

используются для работы с введенными в визуализации значениями (мы знаем, что в примере вводятся только значения типа **INT** и **REAL**, поэтому двух указателей достаточно).

Код метода будет выглядеть следующим образом:

```
peLastEditVarType^ := varType;

paiLastInputElementId^ := pClient^.Inputdata.InputInfo.aiInputPosition;

CASE varType OF

   VisuElems.CmpVisuHandler.Visu_Types.TYPE_INT:
      piVar := pVar;
      piNewValue^ := piVar^;
      psLastEditVarValue^ := TO_STRING(piVar^);

   VisuElems.CmpVisuHandler.Visu_Types.TYPE_REAL:
      prVar := pVar;
      prNewValue^ := prVar^;
      psLastEditVarValue^ := TO_STRING(prVar^);

END_CASE

fbClientTagDataHelper(pClientData := pClient, xIPv4Valid => xIsIpAddr,
   stIPv4 => sIpAddr);

IF pClient^.GlobalData.ClientType =
   VisuElems.VisuElemBase.Visu_ClientType.WebVisualization THEN

   psClientDefinition^ := CONCAT('Web-visu, IP = ', sIpAddr);

ELSIF pClient^.GlobalData.ClientType =
   VisuElems.VisuElemBase.Visu_ClientType.TargetVisualization THEN

   psClientDefinition^ := 'Target-visu';

END_IF
```

По указателю **peLastEditVarType** записывается тип последней измененной в визуализации переменной. Массив идентификаторов, определяющих элемент визуализации, в котором произошло событие записи, извлекается из контекста клиента и записывается по указателю **paiLastInputElementId**.

В зависимости от типа изменившейся переменной происходит приведение нетипизированного указателя **pVar**, являющегося одним из входов метода, к указателю на соответствующий тип и преобразование значения под этим указателем в строку. Строковое представление последнего записанного в визуализации значения размещается по указателю **psLastEditVarValue**.

Далее мы вызываем экземпляр ФБ **VisuFbClientTagDataHelper**, чтобы определить IP-адрес клиента web-визуализации и записать его в переменную **sIpAddr**. В переменную **xIsIpAddr** будет установлен флаг наличия у клиента IP-адреса.

В операторе **IF** мы проверяем тип клиента визуализации, который изменил значение переменной. Если это клиент web-визуализации, то мы формируем строку с обозначением его типа и IP-адресом, и записываем ее по указателю **psClientDefinition**. Для клиента таргет-визуализации мы просто записываем название его типа.

В программе **VISU_LOGIC**, привязанной к задаче VISU_TASK, объявим экземпляр нашего ФБ и вспомогательные переменные:

```
// Программа привязана к задаче VISU_TASK
PROGRAM VISU_LOGIC
VAR
   // Обработчик записи в переменную из визуализации
   fbEditBoxInputHandler:         EditBoxInputHandler;

   // Команда инициализации
   xInit:                         BOOL;

   // Переменные, привязанные к прямоугольникам
   // Для прямоугольников настроено действие Записать переменную
   iVisuValue:                    INT;
   rVisuValue:                    REAL;

   // Значения, записанные с помощью элементов визуализации
   iNewValue:                     INT;
   rNewValue:                     REAL;

   // Тип клиента визуализации
   sClientDefinition:             STRING;
   // Тип последней переменной, которая была записана из визуализации
   eLastEditVarType:              VisuElems.CmpVisuHandler.Visu_Types;
   // Последнее значение, записанное из визуализации
   sLastEditVarValue:             STRING;
    // Массив с ID элемента (см. экран визуализации - Список элементов - ID)
    // Массив используется по той причине, что элемент может быть размещен в фрейме
    // (или в фрейме, вложенном в другие фреймы)
    // В этом случае в массив сначала записываются ID элементов Фрейм.
    // Последнее значимое (не равное -1) число в массиве определяет ID самого элемента
    aiLastInputElementId:         ARRAY
[0..VisuElems.VisuElemBase.Visu_Selection_Constants.MAX_POSITION_DEPTH] OF INT;
END_VAR
```

Код программы довольно прост – при ее старте мы однократно регистрируем экземпляр нашего ФБ в качестве обработчика событий ввода данных в элементы визуализации и передаем на его входные указатели адреса переменных программы.

```
IF NOT(xInit) THEN

   fbEditBoxInputHandler.peLastEditVarType         := ADR(eLastEditVarType);
   fbEditBoxInputHandler.piNewValue                := ADR(iNewValue);
   fbEditBoxInputHandler.prNewValue                := ADR(rNewValue);
   fbEditBoxInputHandler.psClientDefinition        := ADR(sClientDefinition);
   fbEditBoxInputHandler.psLastEditVarValue        := ADR(sLastEditVarValue);
   fbEditBoxInputHandler.paiLastInputElementId     := ADR(aiLastInputElementId);

   VisuElems.VisuElemBase.Visu_Globals.g_VisuEventManager.SetEditBoxEventHandler
     (fbEditBoxInputHandler);

   xInit := TRUE;

END_IF
```

Проект полностью готов. Давайте загрузим его в контроллер и проверим, как он работает.

7.5.4. Проверка работы примера

Подключитесь к web- или таргет-визуализации контроллера. Вводите новые значения типа **INT** и **REAL** с помощью верхних прямоугольников. Наблюдайте отображение информации о клиенте, который ввел значение, тип этого значения, его строковое представление и ID элемента (см. рис. 7.5.3.1), с помощью которого был произведен ввод.

На базе примера вы можете разработать свою логику обработки ввода новых значений в визуализации, учитывая в ней всю перечисленную выше информацию, а также другую информацию, которую можно получить от метода интерфейса.

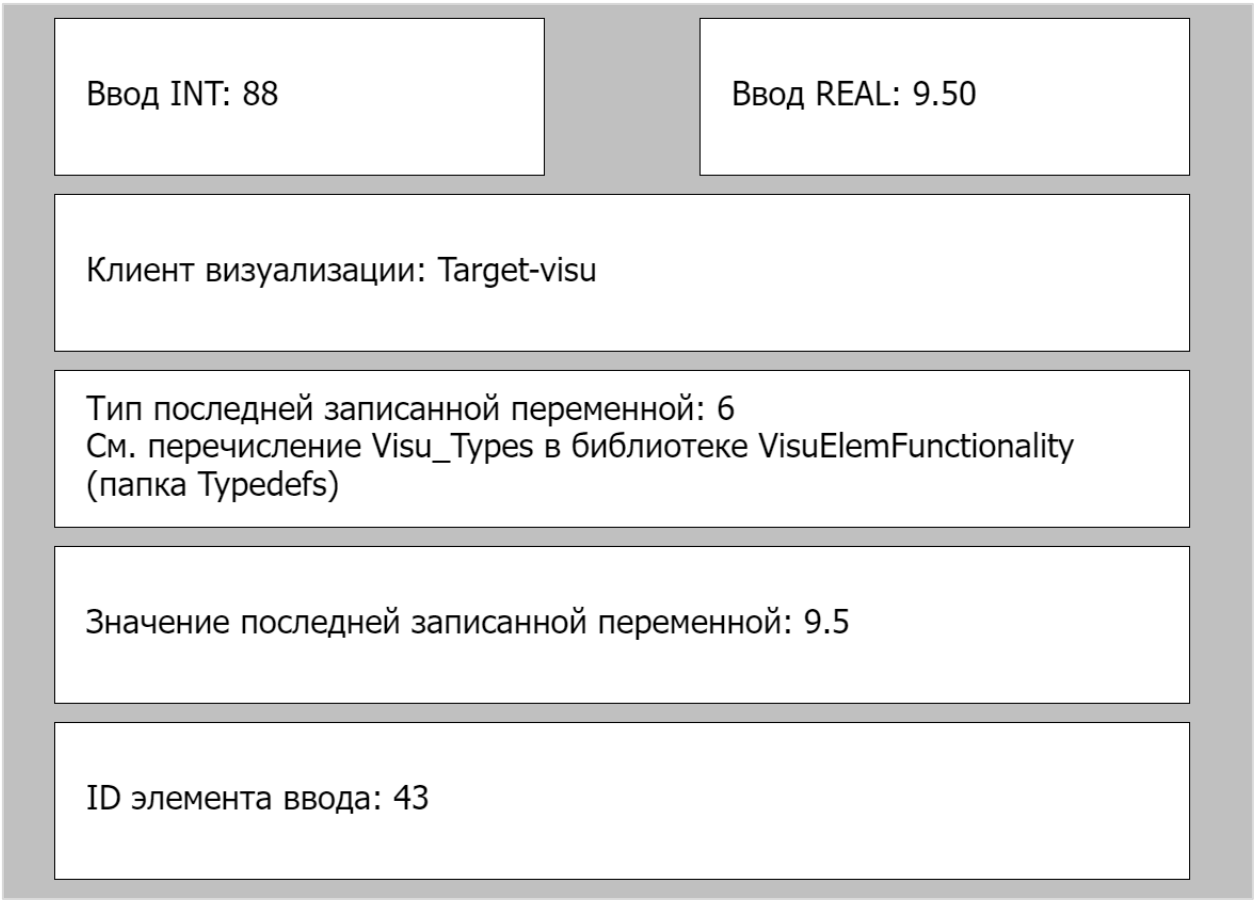


Рисунок 7.5.4.1 – Работа с web-визуализацией примера

7.6. Обработка изменения значения в элементе визуализации (IValueChangedListener)

7.6.1. Основная информация

В п. 7.5 мы рассмотрели интерфейс **IEditBoxInputHandler**, который позволяет детектировать ввод в визуализации нового значения с помощью аппаратной клавиатуры или диалогов ввода (Numpad, Keypad и т. д.). Но, например, этот интерфейс не позволяет отследить изменение значения переменной с помощью ползунка. Кроме того, он предоставляет информацию лишь о новом значении переменной и ее типе – но, например, не позволяет определить, на каком экране визуализации произошел ввод. Чтобы устранить эти недостатки – разработчики CODEYS создали новый интерфес IValueChangedListeter. Надо отметить, что в отличие от остальных интерфейсов, рассмотренных в п. 7, регистрация блока-обработчика, реализующего данный интерфейс, производится не с помощью объекта **VisuElems.VisuElemBase.Visu_Globals.g_VisuEventManager**, а с помощью **VisuElems.VisuElemBase.g_itfValueChangedListenerManager.AddValueChangedListener**.

7.6.2. Обзор интерфейса IValueChangedListeter

Интерфейс IValueChangedListeter входит в состав библиотеки VisuElemBase и содержит единственный метод ValueChanged. Этот метод вызывается при изменении значения с помощью элемента визуализации – причем изменение может быть произведено любым образом (с помощью аппаратной клавиатуры, диалогов ввода, нажатия на горячую клавишу с настроенным действием записи переменной, перемещения ползунка или потенциометра и т. д.). Кроме того, метод вызывается при взаимодействии с элементами визуализации – например, при переключении экрана в фрейме, нажатии на кнопку с управляющей переменной трассировки и т. д.

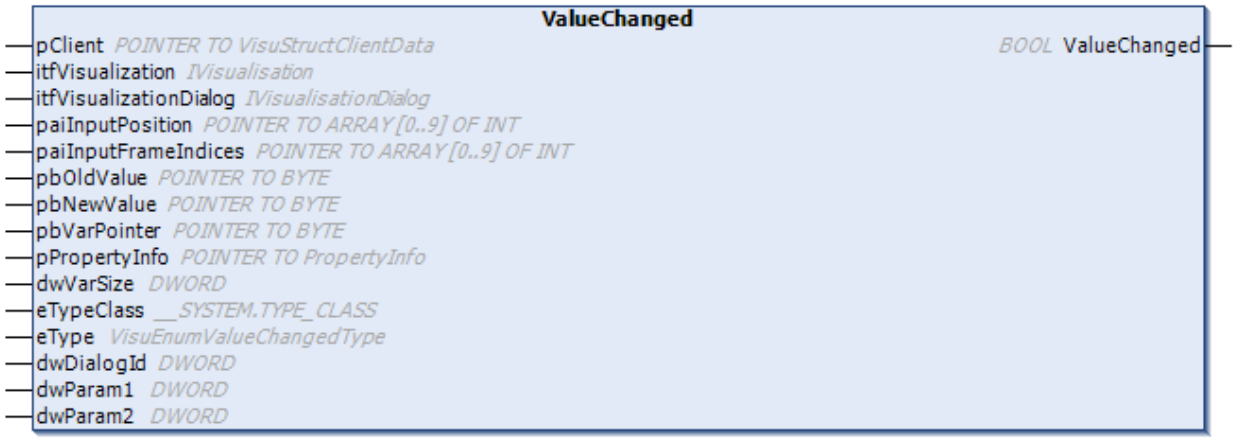


Рисунок 7.6.2.1 – Сигнатура метода **IValueChangedListeter**

Таблица 7.6.2.1 – Описание входов метода **IValueChangedListeter**

Название	Тип	Описание
pClient	POINTER TO VisuElems.VisuElemBase. VisuStructClientData	Указатель на контекст клиента, изменившего значение переменной. Может быть равен **0** при **eType = ConditionVariable.** См. примечание 2 под таблицей
itfVisualization	VisuElems.VisuElemBase. IVisualisation	Экземпляр интерфейса экрана визуализации, в котором произошло изменение значения переменной. Позволяет, например, получить имя экрана
itfVisualizationDialog	VisuElems.VisuElemBase. IVisualisationDialog	Экземпляр интерфейса диалога, в котором произошло изменение значения переменной. Может быть равен **0**, если изменение произошло не в диалоге
paiInputPosition	POINTER TO ARRAY [0..9] OF INT	См. примечание 1 под таблицей
paiInputFrameIndices	POINTER TO ARRAY [0..9] OF INT	См. примечание 1 под таблицей
pbOldValue	POINTER TO BYTE	Указатель на значение переменной до изменения. Если размер переменной превышает 2004 байта (например, для WSTRING(1100)), то ее значение до изменения не сохраняется и **pbOldValue** будет равен **0**
pbNewValue	POINTER TO BYTE	Указатель на значение переменной после изменения
pbVarPointer	POINTER TO BYTE	Указатель на значение переменной. По моим наблюдениям – адрес памяти отличается от **pbNewValue**, но значения под указателями совпадают
pPropertyInfo	POINTER TO VisuElems.VisuElemBase. PropertyInfo	Указатель на структуру, описывающее свойство (если произошло изменение свойства, привязанного к элементу визуализации)
dwVarSize	DWORD	Размер изменившейся переменной в байтах (для строк – значение максимального размера строки, например, 10 байт для WSTRING(4) и т. д.[5])
eTypeClass	__SYSTEM.TYPE_CLASS	Тип изменившейся переменной
eType	VisuElems.VisuElemBase. VisuEnumValueChangedType	Событие, приведшее к изменению переменной. См. примечание 2 под таблицей

[5] Начиная с V3.5 SP17. До этой версии для WSTRING возвращалось число символов, а не байт (CDS-75212), а для STRING – всегда 81 (независимо от ограничения на длину строки, заданное при ее объявлении)

dwDialogId	DWORD	См. примечание 3 под таблицей
dwParam1	DWORD	Только для **eType = KeyEventDown** и **eType = KeyEvendUp**: код нажатой/отпущенной клавиши (см. dwKey)
dwParam2	DWORD	Только для **eType = KeyEventDown** и **eType = KeyEvendUp**: битовая маска клавиш-модификаторов (см. dwModifiers)

Примечания:

1. paiInputPosition и paiInputFrameIndices

paiInputPosition – это указатель на массив, определяющий элемент, в котором произошло изменение переменной (только для элементов, у которых настроено действие **Записать переменную** – например, для ползунка ID определен не будет). Наличие массива связано с тем, что элемент может быть размещен внутри фрейма или «цепочки» вложенных фреймов. Например, представим, что на экране визуализации расположен фрейм с ID = **42**, внутри которого расположен еще один фрейм с ID = **2**, на экране которого размещен элемент с ID = **4**, в котором приозошло событие. Тогда массив будет заполнен следующими значениями: [**42**, **2**, **4**, -1, -1, … , -1] – то есть сначала в нем будут последовательно записаны ID фреймов, затем – ID элемента, а «неиспользованные» элементы массива сохранят значение **-1** (выбор значения **-1** связан с тем, что ID может иметь значение **0**).

В п. 7.3.3 и 7.4.3 мы уже извлекали эту информацию из контекста клиента (**pClient^.Inputdata.InputInfo.aiInputPosition**). В этом методе указатель на нее сразу передается на вход **paiInputPosition**. Можно заметить, что верхняя граница равна **9**, хотя мы уже знаем, что значение константы **MAX_POSITION_DEPTH**, определяющей допустимую «глубину вложенности» фреймов, равно **19**. Я не могу прокомментировать этот момент.

paiInputFrameIndices – это указатель на массив, содержащей индексы экранов фрейма. Например, для рассмотренного выше примера массива **paiInputPosition** ([**42**, **2**, **4**, -1, -1, … , -1]) он может иметь значение значения: [**1**, **0**, -1, -1, -1, … , -1] – то есть:

- в элементе **Фрейм** с ID = **42** был выбран экран с индексом **1**;
- на этом экране располагается элемент **Фрейм** с ID = **2**, в котором был выбран экран с индексом **0**;
- в этом вложенном фрейме прозошло изменение значения переменной с помощью элемента визулизации с ID = **4**.

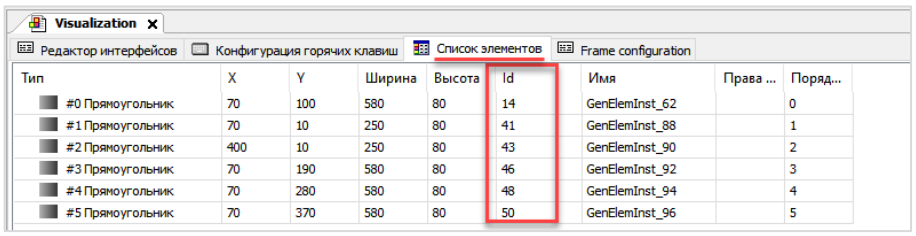


Рисунок 7.6.2.2 – Отображение ID элементов в редакторе визуализации

2. eType

Элементы перечисления **VisuElems.VisuElemBase.VisuEnumValueChangedType** не описаны в документации CODESYS (а само перечисление является скрытым). Тем не менее, по названию элементов можно приблизительно догадаться об источниках соответствующих им событий:

- **Default** – факт изменения значения;
- **OpenDialogPositionInfo** и **CloseDialogPositionInfo** – открытие и закрытие диалога ввода. Например, если вы изменили значение с помощью диалога Numpad, то метод **ValueChanged** будет вызван три раза подряд – сначала с **eType** = **OpenDialogPositionInfo** (открытие диалога ввода), потом **Default** (факт ввода нового значения) и затем **CloseDialogPositionInfo** (закрытие диалога ввода);
- **TableSelectionInfo** – выделение строки в элементе **Таблица**;
- **SelectionAndCaretConfiguration** – изменение положения каретки в элементе **Текстовое поле** (см. одноименную вкладку параметров элемента);
- **KeyEventDown** / **KeyEventUp** – нажатие/отпускание горячей клавиши;
- **ConditionVarible** – у меня нет гипотез, но, напомню, что для этого события контекст клиента отсутствует – т. е. оно возникает не в результате действия оператора;
- **SwitchFrame** – переключение экрана в элементе **Фрейм**;
- **AlarmTableSelectionInfo** и **AlarmTableValidSelectionInfo** – выделение строки в элементе **Таблица тревог (**у меня нет гипотез, чем эти два события отличаются друг от друга);
- **TraceControlVariables** – изменение значения одной из управляющих переменных элемента **Трассировка**;
- **TimeRangePickerControlVariables** – изменение значения одной из управляющих переменных элемента выбора временного диапазона тренда.

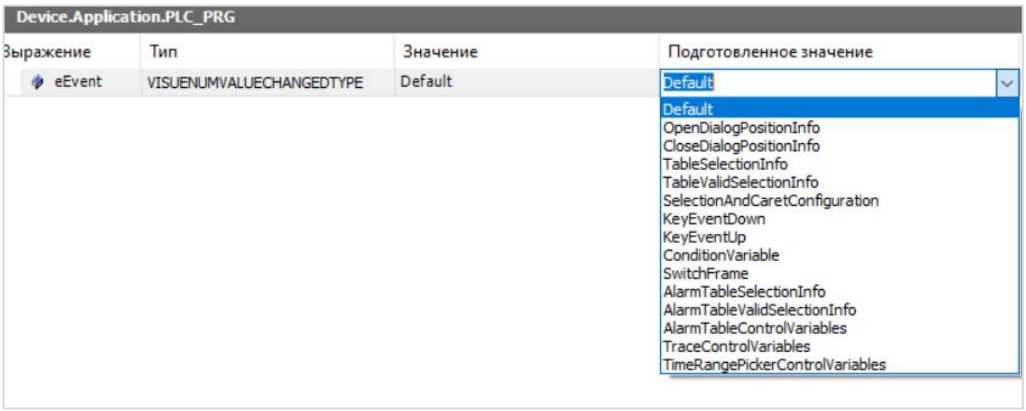


Рисунок 7.6.2.3 – Список элементов перечисления
VisuElems.VisuElemBase.VisuEnumValueChangedType

3. dwDialogId

В документации указано, что этот параметр позволяет определить «*context from which element the dialog was opened*». Мои эксперименты показывают, что при каждом изменении значения переменной с помощью любого диалога ввода (Numpad, Keypad и т. д.) значение, передаваемое на этот вход, увеличивается на 2. В чём именно его смысл – я не знаю.

Выход метода не используется.

Экземпляр ФБ, реализующего данный интерфейс и являющегося обработчиком соответствующего события, должен быть зарегистрирован с помощью вызова метода **VisuElems.VisuElemBase.g_itfValueChangedListenerManager.AddValueChangedListener**.
Единственным входом метода является экземпляр данного интерфейса (соответственно, подойдет и экземпляр ФБ, реализующего данный интерфейс). Метод возвращает **TRUE** в случае успешной регистрации обработчика и **FALSE** – если обработчик данного интерфейса уже был зарегистрирован (для рассматриваемых в п. 7 интерфейсов нельзя зарегистрировать несколько обработчиков).

Как можно заметить – разработчик заранее должен определить, ввод значений в какие элементы визуализации ему необходимо обрабатывать в коде и учесть типы переменных, привязанных к этим элементам.

7.6.3. Обзор и описание примера

Приведенный ниже пример достаточно синтетический, но, тем не менее, позволяет ознакомиться с рассматриваемым в данном пункте функционалом.

На экране визуализации Visualization расположены прямоугольник, к которому привязана переменная программы VISU_LOGIC **iVisuValue** (типа **INT**) и настроено действие **Записать переменную**, и ползунок, к которому привязана переменная **rVisuValue** (типа **REAL**).

В переменные программы **VISU_LOGIC** считывается следующая информация (она же будет отображаться на экране визуализации):

- тип клиента визуализации, который произвел последнее нажатие на любую из кнопок (для клиента web-визуализации – еще и его IP-адрес);
- имя экрана визуализации, на котором произошло изменение значений переменной (соответственно, в рамках примера единственное возможное значение – **Visualization**);
- имя диалога, в котором произошло последнее изменение значения переменной (при изменении значения переменной **iVisuValue** будет отображено **Numpad**);
- код типа последней записанной переменной (да, я снова поленился создать функцию, которое бы конвертировала его в строковое название типа);
- значение последней записанной переменной в строковом виде;
- ID элемента, в котором произошло событие (см. столбец **Id** во вкладке **Список элементов** экрана визуализации). Напомню, что ID заполняется только для элементов с настроенным действием **Записать переменную** – то есть для ползунка будет отображено **-1**.

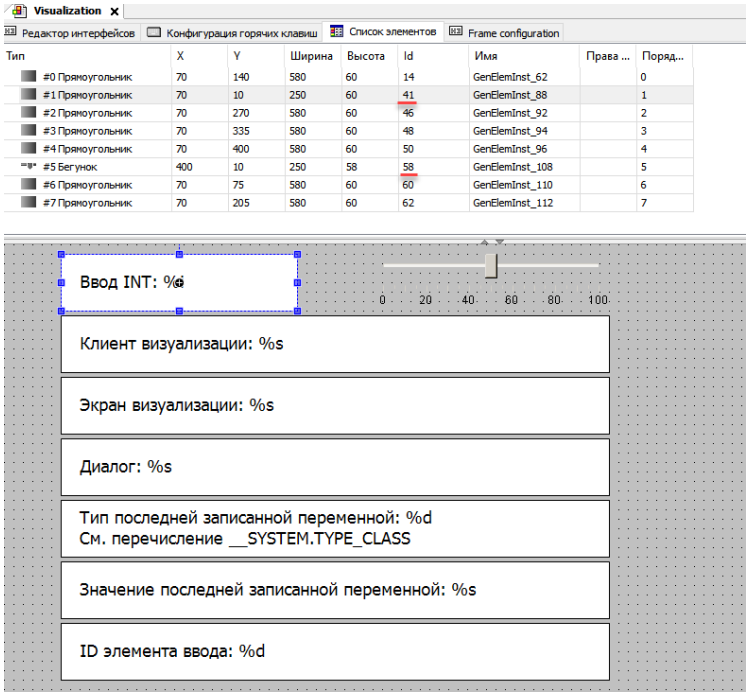


Рисунок 7.6.3.1 – Внешний вид экрана примера

Готовый пример доступен по ссылке: скачать

Для начала создадим ФБ, который будет реализовывать (**IMPLEMENTS**) интерфейс **VisuElems.VisuElemBase.IValueChangedListener**. Назовем его **ValueChangedListener**.

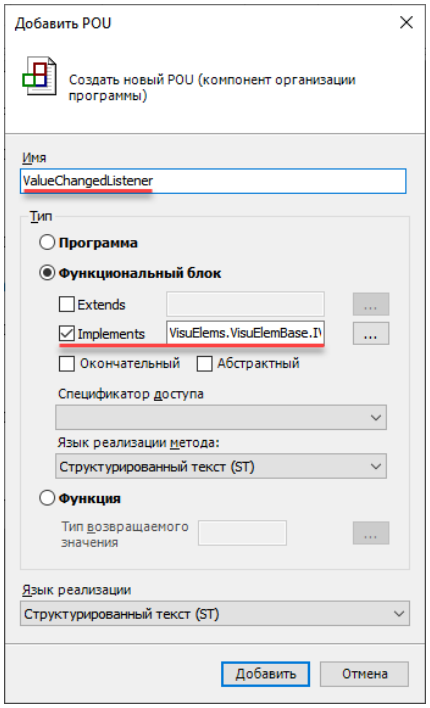


Рисунок 7.6.3.2 – Создание ФБ, реализующего интерфейс **VisuElems.VisuElemBase.IValueChangedListener**

В его автоматически полученном от интерфейса методе **VariableWritten** допишем пространства имен для входов **pClient**, **itfVisualization**, **itfVisualizationDialog**, **pPropertyInfo** и **eType** (последние два не попали на скриншот), чтобы избежать ошибок компиляции:

ValueChangedListener.ValueChanged

```
{warning 'добавить реализацию метода'}
METHOD ValueChanged : BOOL
VAR_INPUT
    (* A pointer to the client structure were the event occurred.
 pClient can be 0 when eType = ConditionVariable
 CHECKED_OMIT*)
    pClient : POINTER TO VisuElems.VisuElemBase.VisuStructClientData;
    (* The visualization from which the value changed event is called.
 Is always set when pClient <> 0
 CHECKED_OMIT*)
    itfVisualization    : VisuElems.VisuElemBase.IVisualisation;
    (* If the value changed event is called from a dialog this parameter is set (<> 0).
 CHECKED_OMIT*)
    itfVisualizationDialog  : VisuElems.VisuElemBase.IVisualisationDialog;
    (* The input position in the visualization is a list of element ids.
 A list is necessary because of element in frames. -1 means not set.
 Example: 41,23,-1,-1,…
 Frame element with id 41 in visualization itfVisualization.GetName(TRUE);
 Element with id 23 in the referenced visualization of the frame
 CHECKED_OMIT*)
```

Рисунок 7.6.3.3 – Добавление пространства имен для переменных

Объявим в области входов ФБ указатели на значения, которые мы хотим получить в нашей программе:

```
FUNCTION_BLOCK ValueChangedListener IMPLEMENTS
VisuElems.VisuElemBase.IValueChangedListener
VAR_INPUT
   peLastEditVarType:             POINTER TO __SYSTEM.TYPE_CLASS;
   piNewValue:                    POINTER TO INT;
   prNewValue:                    POINTER TO REAL;
   psClientDefinition:             POINTER TO STRING;
   psLastEditVarValue:            POINTER TO STRING;
   paiLastInputElementId:         POINTER TO ARRAY [0..9] OF INT;
   psVisuName:                    POINTER TO STRING;
   psDialogName:                  POINTER TO STRING;
END_VAR
```

По указателю **piNewValue** будет размещаться последнее записанное значение типа **INT**, а по указателю **prNewValue** – последнее записанное значение типа **REAL**. По указателю **psLastEditVarValue** будет размещаться последнее из записанных значений любого из типов в строковом виде.

Назначение остальных входов понятно из их названий и рис. 7.6.3.1. Комментарий насчет **pasiLastInputElementId** был приведен после табл. 7.6.2.1.

Теперь добавим реализацию метода **ValueChanged**. Я не перевожу автоматически созданные комментарии – метод уже был описан в п. 7.6.2.

```
{warning 'добавить реализацию метода'}
METHOD ValueChanged : BOOL
VAR_INPUT
   (*
    A pointer to the client structure were the event occurred.
    pClient can be 0 when eType = ConditionVariable
    CHECKED_OMIT
   *)
   pClient               : POINTER TO VisuElems.VisuElemBase.VisuStructClientData;
   (*
   The visualization from which the value changed event is called.
   Is always set when pClient <> 0
   CHECKED_OMIT
   *)
   itfVisualization      : VisuElems.VisuElemBase.IVisualisation;
   (*
   If the value changed event is called from a dialog this parameter is set (<> 0).
   CHECKED_OMIT
   *)
   itfVisualizationDialog : VisuElems.VisuElemBase.IVisualisationDialog;
   (*
   The input position in the visualization is a list of element ids.
   A list is necessary because of element in frames. -1 means not set.
   Example: 41,23,-1,-1,…
   Frame element with id 41 in visualization itfVisualization.GetName(TRUE);
   Element with id 23 in the referenced visualization of the frame
   CHECKED_OMIT
   *)
   paiInputPosition      : POINTER TO ARRAY [0..9] OF INT;
```

```
(*
A list of frame indices for the input position. This information is necessary
to know the referenced visualization.
Example: 1,-1,-1,…
In the frame element the second visualization in the list of frame selection was
set.
CHECKED_OMIT
*)
paiInputFrameIndices   : POINTER TO ARRAY [0..9] OF INT;
(*
A pointer to the old value. The pointer is not necessarily the same than the
pbVarPointer. If a pbVarPointer with a size > 2004 byte is used the old value
cannot be stored. In this case pbOldValue is NULL
CHECKED_OMIT
*)
pbOldValue             : POINTER TO BYTE;
(*
A pointer to the new value. The pointer is not necessarily the same than the
pbVarPointer.
CHECKED_OMIT
*)
pbNewValue             : POINTER TO BYTE;
(* A pointer to the variable which was changed. Can be null if pPropertyInfo is
set.
CHECKED_OMIT
*)
pbVarPointer           : POINTER TO BYTE;
(*
A pointer to the property info of the variable which was changed.
The value is only set when the value changed event comes from a property.
CHECKED_OMIT
*)
pPropertyInfo          : POINTER TO VisuElems.VisuElemBase.PropertyInfo;
(*
The size of the variable which was changed.
CHECKED_OMIT
*)
dwVarSize              : DWORD;
(*
The type of the variable which was changed.
CHECKED_OMIT
*)
eTypeClass             : __SYSTEM.TYPE_CLASS;
(*
The type of the value changed event. This type can be used to filter events.
CHECKED_OMIT
*)
eType                  : VisuElems.VisuElemBase.VisuEnumValueChangedType;
(*
If the value changed event comes from a dialog the dialog id is necessary to
know the context from which element the dialog was opened. This can be necessary
to know the corresponding variables of the dialog.
Normally the following events occur:
Event with type OpenDialogPositionInfo - To know the element where the dialog was
opened.
Event with type Default - ValueChange event for the changed variables
Event with type CloseDialogPositionInfo - To know that the dialog was really
closed.
CHECKED_OMIT
*)
dwDialogId             : DWORD;
(*
If the value changed event comes from a key event dwParam1 contains the key code.
CHECKED_OMIT
*)
dwParam1               : DWORD;
```

```
   (*
   If the value changed event comes from a key event dwParam2 contains the modifier
   code.
   CHECKED_OMIT
   *)
   dwParam2              : DWORD;
END_VAR
VAR
   sIpAddr:                  STRING;
   xIsIpAddr:                BOOL;
   fbClientTagDataHelper:   VisuElems.VisuElemBase.VisuFbClientTagDataHelper;

   piVar:                    POINTER TO INT;
   prVar:                    POINTER TO REAL;
END_VAR
```

Локальные переменные объявлены нами. Функциональный блок **VisuFbClientTagDataHelper** позволяет извлечь из контекста клиента интересную информацию – в частности, IP-адрес клиента web-визуализации. Он будет сохранен в нашу переменную **sIpAddr**. В переменную **xIsIpAddr** будет установлен флаг наличия у клиента IP-адреса. Указатели **piVar** и **prVar** используются для работы с введенными в визуализации значениями (мы знаем, что в примере вводятся только значения типа **INT** и **REAL**, поэтому двух указателей достаточно).

Код метода будет выглядеть следующим образом:

```
peLastEditVarType^ := eTypeClass;

paiLastInputElementId^ := paiInputPosition^;

psVisuName^ := itfVisualization.GetName(TRUE);

IF itfVisualizationDialog <> 0 THEN
   psDialogName^ := itfVisualizationDialog.GetName(TRUE);
ELSE
   psDialogName^ := '';
END_IF

CASE eTypeClass OF

   __SYSTEM.TYPE_CLASS.TYPE_INT:
      piVar := pbNewValue;
      piNewValue^ := piVar^;
      psLastEditVarValue^ := TO_STRING(piVar^);

   __SYSTEM.TYPE_CLASS.TYPE_REAL:
      prVar := pbNewValue;
      prNewValue^ := prVar^;
      psLastEditVarValue^ := TO_STRING(prVar^);

END_CASE

IF pClient <> 0 THEN

   fbClientTagDataHelper(pClientData := pClient, xIPv4Valid => xIsIpAddr,
      stIPv4 => sIpAddr);

   IF pClient^.GlobalData.ClientType =
      VisuElems.VisuElemBase.Visu_ClientType.WebVisualization THEN

      psClientDefinition^ := CONCAT('Web-visu, IP = ', sIpAddr);
```

```
   ELSIF pClient^.GlobalData.ClientType =
      VisuElems.VisuElemBase.Visu_ClientType.TargetVisualization THEN

      psClientDefinition^ := 'Target-visu';

   END_IF

END_IF
```

По указателю **peLastEditVarType** записывается тип последней измененной в визуализации переменной. Массив идентификаторов, определяющих элемент визуализации, в котором произошло событие записи, записывается по указателю **paiLastInputElementId**.

Названия экрана визуализации и диалога визуализации, в которых произошло изменение значения переменной, определяются с помощью вызова метода **GetName** соответствующих экземпляров интерфейсов. Аргумент метода определяет, будет ли включать в себя сформированное имя пространство имен экрана/диалога (например, если экран или диалог размещены в библиотеке); **TRUE** – будет. Обратите внимание, что перед доступом к **itfVisualizationDialog** он проверяется на неравенство нулю – это условие не выполняется в том случае, если изменение значения переменной произошло не в диалоге. В такой ситуации вызывать метод экземпляра интерфейса нельзя, так как это приведет к разыменованию нулевого указателя.

В зависимости от типа изменившейся переменной происходит приведение нетипизированного указателя **pVar**, являющегося одним из входов метода, к указателю на соответствующий тип и преобразование значения под этим указателем в строку. Строковое представление последнего записанного в визуализации значения размещается по указателю **psLastEditVarValue**.

Весь остальной код обернут в условие **IF**, в котором указатель на контекст клиента проверяется на неравенство нулю. Дело в том, что в одном конкретном случае (когда **eType = ConditionVariable**, чтобы это ни значило) он действительно будет равен нулю – и в этом случае разыменовывать указатель нельзя.

Мы вызываем экземпляр ФБ **VisuFbClientTagDataHelper**, чтобы определить IP-адрес клиента web-визуализации и записать его в переменную **sIpAddr**. В переменную **xIsIpAddr** будет установлен флаг наличия у клиента IP-адреса.

Во вложенном операторе **IF** мы проверяем тип клиента визуализации, который изменил значение переменной. Если это клиент web-визуализации, то мы формируем строку с обозначением его типа и IP-адресом, и записываем ее по указателю **psClientDefinition**. Для клиента таргет-визуализации мы просто записываем название его типа.

В программе **VISU_LOGIC**, привязанной к задаче VISU_TASK, объявим экземпляр нашего ФБ и вспомогательные переменные:

```
// Программа привязана к задаче VISU_TASK
PROGRAM VISU_LOGIC
VAR
   // Обработчик записи в переменную из визуализации
   fbValueChangedListener:        ValueChangedListener;

   // Команда инициализации
   xInit:                         BOOL;

   // Переменные, привязанные к прямоугольнику и ползунку
   iVisuValue:                    INT;
   rVisuValue:                    REAL;

   // Значения, записанные с помощью элементов визуализации
   iNewValue:                     INT;
   rNewValue:                     REAL;

   // Тип клиента визуализации
   sClientDefinition:             STRING;
   // Имя визуализации, в которой произошло изменение значения
   sVisuName:                     STRING;
   // Имя диалога, в котором произошло изменение значения
   sDialogName:                   STRING;
   // Тип последней переменной, которая была записана из визуализации
   eLastEditVarType:              __SYSTEM.TYPE_CLASS;
   // Последнее значение, записанное из визуализации
   sLastEditVarValue:             STRING;
   // Массив с ID элемента (см. экран визуализации - Список элементов - ID)
   // Массив используется по той причине, что элемент может быть размещен в фрейме
   // (или в фрейме, вложенном в другие фреймы)
   // В этом случае в массив сначала записываются ID элементов Фрейм.
   // Последнее значимое (не равное -1) число в массиве определяет ID самого элемента
   aiLastInputElementId:   ARRAY [0..9] OF INT;
END_VAR
```

Код программы довольно прост – при ее старте мы однократно регистрируем экземпляр нашего ФБ в качестве обработчика событий ввода данных в элементы визуализации и передаем на его входные указатели адреса переменных программы.

```
IF NOT(xInit) THEN

   fbValueChangedListener.peLastEditVarType      := ADR(eLastEditVarType);
   fbValueChangedListener.piNewValue             := ADR(iNewValue);
   fbValueChangedListener.prNewValue             := ADR(rNewValue);
   fbValueChangedListener.psClientDefinition     := ADR(sClientDefinition)
   fbValueChangedListener.psLastEditVarValue     := ADR(sLastEditVarValue);
   fbValueChangedListener.paiLastInputElementId  := ADR(aiLastInputElementId);
   fbValueChangedListener.psVisuName             := ADR(sVisuName);
   fbValueChangedListener.psDialogName           := ADR(sDialogName);

   VisuElems.VisuElemBase.g_itfValueChangedListenerManager.AddValueChangedListener
     (fbValueChangedListener);

   xInit := TRUE;

END_IF
```

Проект полностью готов. Давайте загрузим его в контроллер и проверим, как он работает.

7.6.4. Проверка работы примера

Подключитесь к web- или таргет-визуализации контроллера. Изменяйте значения типа **INT** и **REAL** с помощью диалога вввода, открываемого при нажатии на прямоугольник, и ползунка. Наблюдайте отображение информации о клиенте, который ввел значение, названия экрана и диалога, в рамках которых это значение было введено, тип значения, его строковое представление и ID элемента (см. рис. 7.6.3.1), с помощью которого был произведен ввод.

На базе примера вы можете разработать свою логику обработки ввода новых значений в визуализации, учитывая в ней всю перечисленную выше информацию, а также другую информацию, которую можно получить от метода интерфейса.

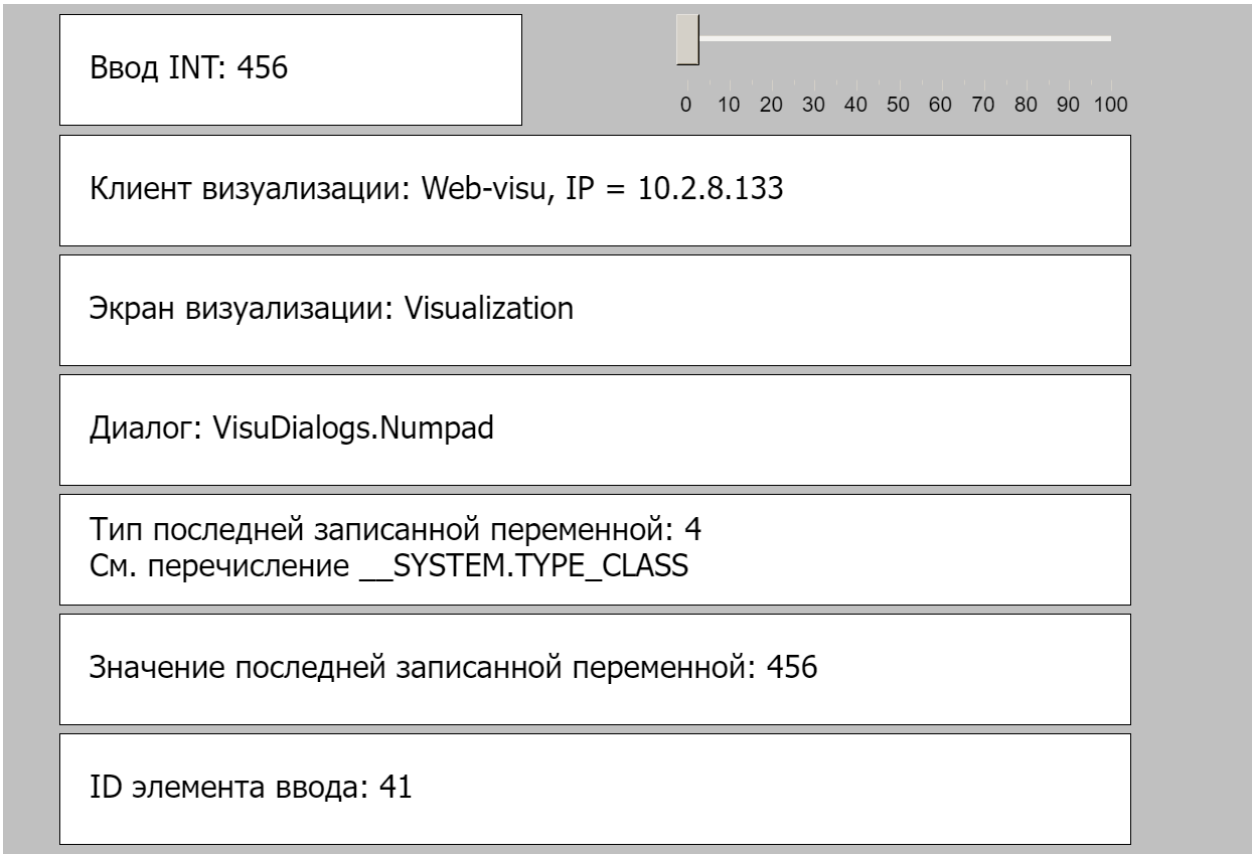


Рисунок 7.6.4.1 – Работа с web-визуализацией примера

Заключение

Вот, собственно, и всё, что я планировал рассказать про работу с визуализацией CODESYS V3. Статья получилась довольно большой, но всё равно она не отражает все возможности, которые представляли бы интерес для разработчика – довольно многие из них попросту скрыты и недокументированы (например, отсутствует полное и подробное описание контекста клиента). Тем не менее, надеюсь, что приведенная информация изложена достаточно доступно и будет полезна тем, кто разрабатывает проекты, включающие сложную визуализацию со специфичекими требованиями.

Если вам интересна тема внутреннего устройства визуализации CODESYS – то я рекомендую ознакомиться с дипломной работой Йонаса Линдхольма Graphical Interface Framework for Control System Environments. Я планирую опубликовать ее перевод в первой половине лета 2023.

Спасибо за внимание.

Список примеров, рассмотренных в документе

Примеры документа созданы в среде **CODESYS V3.5 SP17 Patch 3** с плагином визуализации 4.3.0.0.

Таблица П.1 – Список примеров, рассмотренных в документе

Название и ссылка на пункт документа	Обозначение	Пример из документа	Пример от CODESYS Group
Получение информации о клиентах визуализации	VisuUtils – FbIterateClients	скачать	-
Переключение экрана	VisuUtils – FbChangeVisu	скачать (базовый, дополненный)	-
Открытие диалога и обработка его закрытия	VisuUtils – FbOpenDialogExtended	скачать	-
Передача файлов	VisuUtils - FBTransferFile	скачать	-
Работа с пользователями	VisuUserManagement	скачать	перейти
Обработка событий клиентов визуализации	IClientManagerListener	скачать	перейти
Работа с фреймами	IFrameManager	скачать	перейти
Выбор элемента и имитация нажатия из кода программы	ISelectionManager	скачать	перейти
Обработка нажатия клавиш	IKeyEventHandler	скачать	перейти
Обработка курсора	IMouseEventHandler	скачать	
Обработка ввода значения в элементе визуализации	IEditBoxInputHandler	скачать	
Обработка событий в элементе визуализации	IInputOnElementEventHandler	скачать	-
Обработка изменения значения в элементе визуализации	IValueChangedListener	скачать	перейти
Обработка жестов	Multitouch Handling	-	перейти
Работа с диалогами из кода	Visu Dialog ST	-	перейти