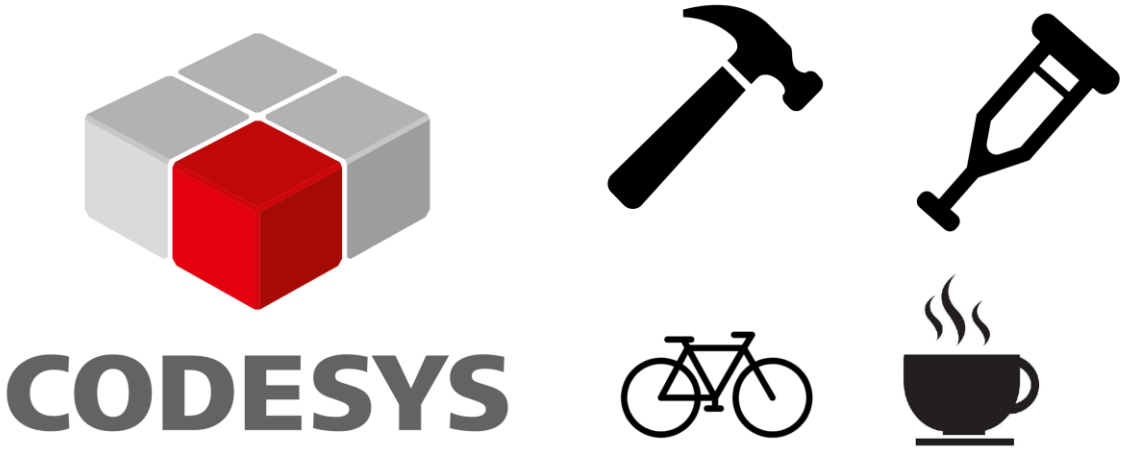
Отладка проектов в CODESYS V3



09.03.2021
версия 1.0

https://oscat.ru/

Оглавление

Введение

«Отладка кода вдвое сложнее, чем его написание. Так что если вы пишете код настолько умно, насколько можете, то вы по определению недостаточно сообразительны, чтобы его отлаживать.»

*Брайн Кернига*н

В процессе разработки программного обеспечения неизбежно наступает тот трагический момент, когда при очередном запуске вашего кода вы встречаетесь с ошибками. Под «ошибкой» подразумевается поведение программы, которое не соответствует техническому заданию. Наличие задокументированного и согласованного технического задания является принципиально важным моментом – если оно отсутствует, то говорить о наличии или отсутствии ошибок в ПО невозможно, так как нет критериев для их определения.

Процедура поиска и устранения ошибок называется *отладкой*. Практически все современные среды разработки предоставляют набор инструментов, которые упрощают отладку – утилиты для просмотра состояния памяти и стека вызовов, профилировщики, логи ошибок и т.д. Кроме того, к настоящему моменту уже написано достаточно много литературы, содержащей описание конкретных методов и рекомендаций (некоторые книги упомянуты в списке литературы).

Повседневная работа авторов этой статьи связана с программируемыми логическими контроллерами (ПЛК). Проекты для ПЛК, естественно, также приходится отлаживать. Мы часто наблюдаем ситуацию, когда разработчик проектов для ПЛК не имеет базовых навыков программирования. В известной степени это вызвано историческими причинами – первые ПЛК предназначались для замены релейных схем и аналоговых регуляторов, и программы для них на графических языках создавали техники, инженеры-технологи и другие специалисты, не связанные с программированием. Со временем требования к системам автоматизации возросли, и сегодня от инженера-программиста ожидают не только знаний дискретной логики, но и умения кодировать сложные алгоритмы, реализовывать нестандартные протоколы обмена, работать с файлами и базами данных, интегрироваться с web-сервисами через API и т.д. При этом в серьезных компаниях круг обязанностей инженера-программиста ограничивается именно разработкой и пусконаладкой ПО (*а иногда даже ПНР проводят отдельные специалисты*) – от него не требуют заниматься проектированием, монтажом, настройкой сетевого оборудования (мы сейчас говорим не про условный коммутатор в шкафу автоматики, а про отдельные системы связи типа SkyEdge и т.д.) и другими делами – что позволяет ему сосредоточиться на задачах, непосредственно связанных с программированием, и со временем развить серьезные компетенции в этой области. Мы всё чаще видим, что инжиниринговые компании и системные интеграторы стали ответственно относиться к специализации своих сотрудников и не пытаются заставить их разбираться во всём, чём только можно – зачастую конкретный человек занимается только конкретным типом (*или несколькими типами, но не всем зоопарком*) ПЛК, разработкой проектов для ПЛК и HMI/SCADA занимаются отдельные сотрудники и т.д. Тем не менее, так происходит не везде – в службе эксплуатации на условном заводе к разработке проектов могут привлечь инженеров КИПиА (чтобы не создавать еще одну должность), а в мелких инжиниринговых компаниях сохраняется ситуация, когда человек

должен быть «универсальным солдатом» – суметь написать ТЗ (часто заказчик проекта не в состоянии этого сделать), подобрать оборудование, спроектировать шкаф, собрать его, написать ПО, провести ПНР на объекте. Можно по-разному относиться к этой ситуации, но так или иначе – пока что она не является такой уж редкостью.

Итак, если разработкой ПО для ПЛК занимается человек без навыков программирования – на что он должен ориентироваться при отладке ПО? Как мы уже говорили – разумеется, есть специализированная литература, но для ее использования всё равно требуется определенный IT-бэкграунд. С другой стороны, специализированной литературы, посвященной отладке проектов именно для ПЛК практически не существует (потому что процедура отладки в данном случае не сильно отличается от, например, отладки ПО для embedded-систем).

В своей статье мы хотели бы попробовать заполнить этот пробел и рассказать о процессе отладки ПО для ПЛК простым языком. Мы поговорим о типах ошибок, которые могут возникнуть при разработке и о том, что делать при их возникновении. Мы расскажем об инструментах отладки, которые имеются в основной для нас среде разработке – **CODESYS V3.5**. Мы приведем примеры конкретных ошибок и расскажем, как происходила их отладка.

Мы не претендуем на какую-то уникальность этого материала – большая часть советов универсальна, и вы найдете их в любой из книг, которые мы привели в списке литературы. Тем не менее, надеемся, что для кого-то из читателей приведенная в статье информация окажется полезной и интересной.

Авторы: Евгений Кислов, Екатерина Чибисова

1. Процедура отладки

Причиной начала отладки является обнаружение в процессе работы программы какой-либо ошибки. Отладка может быть начата как в момент обнаружения ошибки, так и позже (например, из-за гипотезы, что в данный момент ошибка проявляется из-за неготовности одного из программных модулей).

Процедура отладки в своей наиболее полной форме состоит из следующих шагов:

1. Анализ – поиск информации об ошибке в корпоративных и общедоступных ресурсах (*возможно, ваши коллеги уже сталкивались с такой ошибкой?),* поиск всех последствий ошибки (*может быть, их больше, чем кажется на первый взгляд? может быть, вы наблюдаете последствия нескольких ошибок, случившихся одновременно, или одна из ошибок маскирует другую?*), определение типа ошибки, оценка последних изменений, внесенных в проект и являющихся потенциальной причиной ошибки.

2. Проверка воспроизводимости – серия попыток воспроизвести ошибку на другом оборудовании/другой версии прошивки/другом проекте/в других условиях эксплуатации и т.д. В результате вы можете получить новую информацию об ошибке (например, она не воспроизводится на другом аналогичном устройстве – значит, можно выдвинуть гипотезу, что проблема в аппаратной части конкретного прибора), определить набор условий, достаточных для ее воспроизведения, и систематику проявления (например, «*воспроизводится при приблизительно каждой десятой перезагрузке ПЛК*»). В идеальном случае – после этого пункта вы умеете воспроизводить ошибку со 100% повторяемостью.

3. Локализация ошибки – попытка найти конкретную область, в которой она возникает. На этом этапе следует начать аккуратно вырезать фрагменты проекта ПЛК, отключать службы ОС и т.д. Если после одного из «урезаний» ошибка исчезнет – есть вероятность, что вы нашли фрагмент кода, с которым она связана. С другой стороны, в результате подобных операций ошибка может не исчезнуть, а просто потерять видимые проявления (*например, при некорректном доступе к памяти модуль 1 «перезатирал» данные модуля 2, в результате чего происходило деление на 0. После исключения из проекта модуля 2 деления на ноль больше не происходит, но при этом модуль 1 продолжает «перезатирать» чью-то память – просто пока вы не видите внешних проявлений этого*). После этого пункта вы должны иметь минимальный набор аппаратных и программных средств, на котором можно воспроизвести проблему – очевидно, что чем меньше устройств входит в состав системы и чем меньше проект – тем проще отлаживать ошибку.

4. Выдвижение и проверка гипотез о возможных причинах ошибки – в результате одной из них ошибка должна быть обнаружена и устранена. В сложных случаях условия и результаты каждого такого теста должны документироваться и сохраняться (иначе в какой-то момент вы забудете, какие гипотезы уже проверили, а какие нет).

5. Проверка устранения ошибки – вы возвращаетесь к п. 2 и проверяете, что ошибка перестала воспроизводиться в рамках всей системы (со всем подключенным оборудованием, с исходным проектом и т.д.).

6. Проверка других фрагментов ПО, в которых может быть допущена эта ошибка (*например, если ошибка связана с присвоением некорректных аргументов при вызове функции – то следует проверить все остальные вызовы этой функции на предмет корректности аргументов*).

7. Фиксация результатов (добавление информации в корпоративный баг-трекер или wiki, доработка корпоративных стандартов разработки ПО, передача информации коллегам и т.д.)

В зависимости от конкретной ошибки, часть шагов может быть пропущена (например, опытный инженер может сразу понять наиболее вероятную причину конкретной ошибки по особенностям ее проявления и сразу перейти к отработке этой гипотезы).

В некоторых ситуациях самостоятельно провести отладку не получится и вам потребуется прибегнуть к помощи коллег или других людей (например, сотрудников техподдержки производителя вашего оборудования). В этом нет ничего постыдного – но вы должны с уважением относиться к их времени и заранее предоставить максимальное количество информации:

- перед обращением за помощью соберите стенд, на котором удается воспроизвести ошибку, и убедитесь, что владеете всей информацией о нем (включая версии прошивок и проектов, серийные номера и даты изготовления приборов и т.д.);
- вы должны суметь четко сформулировать описание ошибки (и почему считаете, что наблюдаемая ситуация является ошибкой) и какие действия приводят к ее возникновению. Не забывайте, что человек, к которому вы обращаетесь, может быть не в курсе всех нюансов системы и проектов, которые вы изучаете уже на протяжении нескольких дней или даже недель (*в этот момент они становятся вам настолько родными, что уже сложно представить, как кто-то не может по паре фраз понять, что вы вообще имеете в виду*) и ему требуется полное и детализированное описание ситуации с указанием всех фактов, которые вам известны. E-mail с одной-двумя фразами типа «*иногда крашится программа при старте ПЛК помогите пожалуйста*», вероятно, получит самый низкий приоритет и будет рассматриваться только после решения проблем людей, которые более четко и подробно описали свою проблему;
- перед обращением за помощью вы уже должны выдвинуть и отработать некоторые гипотезы и суметь четко рассказать о ваших опытах и их результатах. Вам не следует обижаться, если ваш собеседник попросит повторить всё тоже самое под его присмотром – это не значит, что он вам не доверяет или считает недостаточно компетентным. Это значит лишь то, что он своими глазами хочет увидеть происходящее;
- если вы договорились об удаленном подключении – то должны обеспечить ПК и канал связи с приемлемым быстродействием, а также заранее подготовить всё необходимое оборудование (провода, адаптеры, накопители и т.д.).
- прочитайте статью Саймона Тэтхема (разработчика утилиты PuTTY) «Как эффективно сообщать об ошибках».

2. Типы ошибок

Природа ошибок может быть самой разнообразной, и первое, что должен сделать разработчик, столкнувшись с ошибкой – классифицировать её. В рамках АСУ ТП можно выделить следующие характерные типы ошибок:

- аппаратные неисправности;
- ошибки в прошивке;
- ошибки в системе исполнения/среде программирования;
- ошибки в пользовательском проекте;
- ошибки, вызванные внешними условиями и человеческим фактором;
- ошибки, которые не являются ошибками.

Аппаратные неисправности могут возникать как по вине производителя (например, из-за непропая элементов платы на производстве), так и по вине пользователя (например, случайно подали 220 В на RS-485). **Способ определения аппаратной неисправности:** если проблема не проявляется на другом приборе идентичной модели с той же версией прошивки, тем же пользовательским проектом и находящимся в тех же условиях эксплуатации – то, вероятно, причиной наблюдаемой ошибки является аппаратная неисправность. В этом случае следует связаться с производителем и организовать отправку прибора в сервисный центр или запросить инструкции для проведения самостоятельного ремонта.

Под прошивкой ПЛК подразумевается низкоуровневое (с точки зрения пользователя) ПО, непосредственно управляющее аппаратной частью контроллера. В состав прошивки обычно входит начальный загрузчик, операционная система, драйвера для периферии, дополнительные службы и сервисы, не связанные с основными задачами ПЛК (например, web-конфигуратор, NTP-клиент, FTP-сервер и т.д.). Как и любое другое ПО, прошивка может содержать баги различной степени критичности, которые могут иметь совершенно разные проявления – например, сброс значений энергонезависимых переменных в начальные значения после перезагрузки устройства, неработоспособность конкретного функционала (например, того же NTP-клиента), «зависание» ПЛК на этапе загрузки и т.д. Обычно производители с некоторой периодичностью выпускают новые версии прошивок для своих ПЛК, в которых исправляют существующие ошибки (и, к сожалению, иногда добавляют новые). Если техподдержка производителя ПЛК точно знает или подозревает, что наблюдаемая клиентом проблема связана с конкретной версией прошивки – то она даст рекомендации по ее обновлению до актуальной версии. Часто совет по обновлению прошивки до актуальной является универсальным (примерно как совет «попробуйте выключить и включить» в техподдержке интернет-провайдера) и используется техподдержкой вообще в любой ситуации. Это можно понять – обычно инженер техподдержки предпочитает быть в равных условиях с клиентом, а сама техподдержка обычно использует только самые свежие версии прошивок своих приборов. **Способ определения ошибок в прошивке***:* если проблема не проявляется на этом же приборе, прошитом другой версией прошивки (причем в различных случаях может помочь как обновление прошивки до свежей версии, так и откат до более старой), тем же пользовательским проектом и находящимся в тех же условиях эксплуатации – то, вероятно, причиной наблюдаемой ошибки является ошибка прошивки. В этом случае следует связаться с производителем ПЛК для получения рекомендаций по решению проблемы.

В значительном количестве случаев ПЛК программируются в специализированных средах разработки (IDE) на языках стандарта МЭК 61131-3. Некоторые производители сами разрабатывают IDE для своих ПЛК (например, TIA Portal от Siemens, RSLogix 5000 от Rockwell Automation, КОНГРАФ от МЗТА и т.д.), другие же лицензируют «вендоро-независимые» IDE (CODESYS, ISaGRAF, MasterSCADA 4D). Созданный в IDE проект загружается в контроллер, где его выполнение обеспечивает система исполнения (runtime). Рантайм обычно также включает в себя библиотеки, конфигурационные файлы, сервисы для отладки и другие элементы инфраструктуры, которые упрощают разработку и отладку ПО.

И среда программирования, и система исполнения могут содержать ошибки. Ошибки IDE обычно отловить довольно просто – они проявляются в виде сообщений об ошибках, неактивности отдельных кнопок и т.д. Ошибки рантайма отловить гораздо сложнее и проявляться они могут совсем по-разному – например, такой ошибкой может быть отсутствие возможности подключения к контроллеру, сброс значений энергонезависимых переменных в начальные значения после перезагрузки устройства (*да, мы уже использовали этот пример в пункте про ошибки прошивки – но причиной подобной ошибки могут быть и проблемы рантайма*), неработоспособность конкретного функционала (например, определенных коммуникационных драйверов) и т.д.

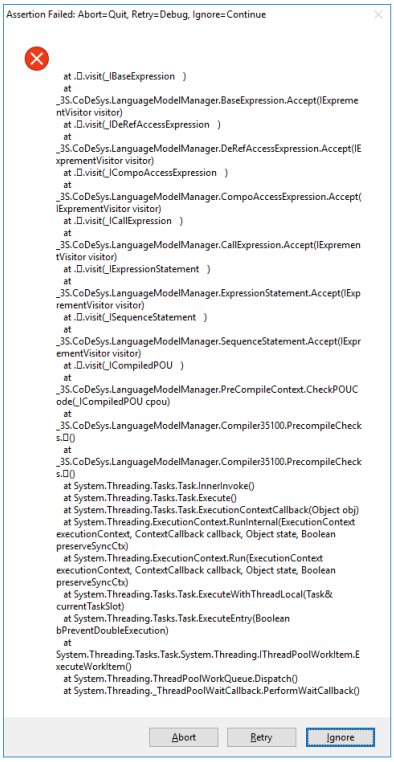


Рис. 2.1. Пример ошибки IDE для CODESYS V3.5

Как и в случае прошивок – обычно производителя ПЛК регулярно выпускают обновления для своих IDE и рантайма. Если вы подозреваете, что наблюдаемая ошибка связана с ними – то вам следует связаться с технической поддержкой производителя и уточнить, известна ли она им и не была ли исправлена в одной из следующих версий ПО. Вы должны понимать, что даже если производитель ПЛК не занимается разработкой своей IDE/рантайма, а использует по лицензии ПО другой компании – то он все равно имеет контакты с лицензиаром (через OEM-техподдержку, доступ к баг-трекеру и т.д.), и может получить информацию о вашей ошибке или же передать информацию о ней разработчикам для последующего исправления.

При разработке проекта для ПЛК пользователь может допустить ошибки. Проявления этих ошибок могут быть самыми разнообразными – спорадическая перезагрузка контроллера, некорректное выполнение алгоритма, срабатывание сторожевого таймера из-за исключений и т.д. **Способ определения ошибок в проекте***:* если проблема не проявляется на этом же приборе с той же версией прошивки, находящимся в тех же условиях эксплуатации, но с другой версией пользовательского проекта (например, более ранней версией или «пустым» тестовым проектом) – то, вероятно, причиной наблюдаемой ошибки являются ошибки в проекте. В этом случае пользователю следует приступить к отладке.

Большинство современных сред разработки предоставляют инструменты, упрощающие процесс отладки. В п. 3 мы расскажем о таких инструментах, имеющихся в среде **CODESYS V3.5**.

Также упростить отладку может грамотный подход к разработке проекта:

- наличие детализированного ТЗ;
- следование корпоративным или иным стандартам разработки ПО;
- регулярное и тщательное тестирование по ходу разработки (когда проект сначала несколько недель пишется «в стол» и только потом запускается на реальном оборудовании – отладка может стать довольно мучительным занятием). Оптимальным, на наш взгляд, вариантом является совмещение модульного тестирования (для отдельных функциональных блоков и других законченных фрагментов приложения) и интеграционного (тестирование всего проекта в рамках системы – с подключением периферии, имитаций условий эксплуатации с помощью генерации помех и т.д.).

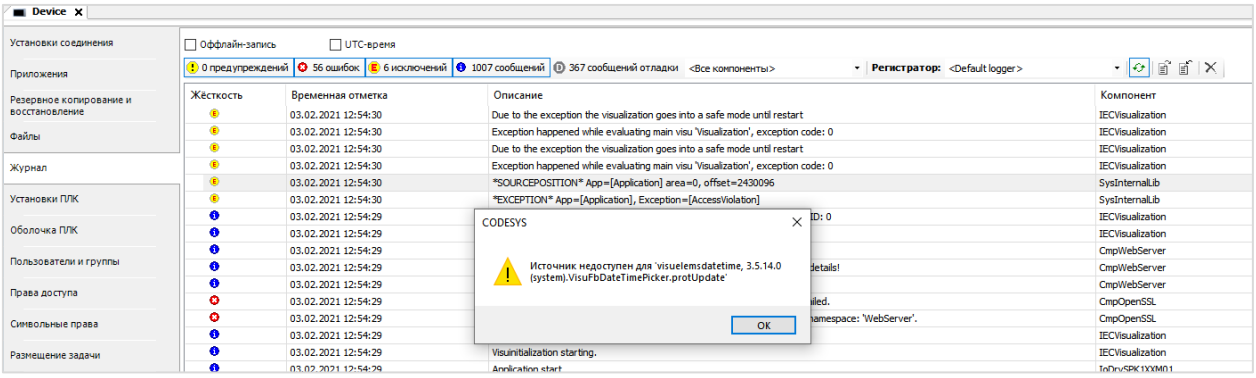


Рис. 2.2. Пример ошибки в проекте CODESYS V3.5 (см. информацию в п. 5.9).

Очень часто условия эксплуатации оборудования на промышленных объектах сильно отличаются от офисных. Эти условия можно разделить на две основные группы:

- условия окружающей среды (температура, влажность, вибрация, механическое воздействие);
- помехи (кондуктивные, импульсные, радиопомехи и т.д.).

Обычно эти вопросы регламентируются отраслевым стандартами. Если требования этих стандартов не выполняются (например, если организация заземления и прокладка сигнальных линий не соответствует требованиям ПУЭ) или на объекте используется оборудование, характеристики которого не соответствуют его условиям – то это может стать причиной появления ошибок в работе приборов (например, периодических ошибок передачи данных во время действия на линию связи электромагнитной помехи).

Кроме того, эксплуатацией оборудования обычно занимается персонал объекта (операторы, дежурные, инженеры КИПиА и т.д.). Некорректные действия персонала также могут являться причиной ошибок. Очевидно, что эксплуатацией оборудования должны заниматься люди с соответствующей квалификацией и прошедшие обучение/инструктаж. Характерный пример из этой статьи на Хабре:

«В пики жары пропадает связь с одним из локальных серверов. На объектах таких серверов много, смонтированы они довольно компактно в техпомещениях, и везде сложности с охлаждением, причём часто используется внешнее принудительное. Ну то есть мощный вентилятор, направленный прямо на стойку. Коллеги называют это модным словом «фрикулинг», но это именно вентилятор, направленный на стойку.

Но случается такое не каждый день по жаре, а примерно только каждый второй. Начинаем разбираться — иногда, как в детективе: оказывается, там в этом же помещении работают два человека. Один специалист знает, что такое стойка, или близко догадывается о таинственной связи мигающих лампочек и вентилятора. Второй специалист — бабушка. Она не знает. И когда жара доходит до максимума, бабушка чувствует thermal threshold, затем берёт и поворачивает вентилятор на себя. Потому что её маленький вентилятор не такой мощный.

Логичное следствие — бабушка охлаждается, стойка перегревается. Дальше по порогу температуры происходит штатный thermal shutdown. А у нас — очередной тикет.»

Способ определения ошибок, связанных с условиями эксплуатации и человеческим фактором*:* если проблема не проявляется на этом же приборе с той же версией прошивки и тем же проектом, но находящемся в других условиях эксплуатации (например, офисных условиях вместо промышленных) – то, вероятно, причиной наблюдаемой ошибки являются особенности внешней среды объекта или человеческий фактор.

Последний из известных нам типов ошибок – это ошибки, которые, как ни странно, не являются ошибками. В некоторых случаях пользователь может принять за ошибку предсказанное поведение ПО. Особенно это характерно для проектов, для которых отсутствует ТЗ – пользователь может считать, что опрос «слишком медленный», но в реальности такой период опроса может просто являться данностью для конкретных slave-устройств или коммуникационных драйверов.

Еще один пример – ошибки в логе контроллера в CODESYS V3.5, приведенные на этом скриншоте:

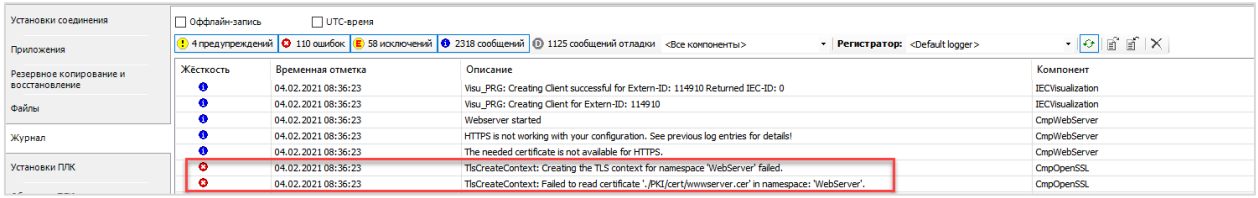


Рис. 2.3. Пример «ошибок» в логе CODESYS V3.5, связанных с отсутствием сертификатов HTTPS

Пользователь может интуитивно воспринять такие сообщения как индикацию каких-то проблем с контроллером или проектом, но они являются просто констатацией факта отсутствия в ПЛК сертификатов HTTPS для сервера web-визуализации. В случае необходимости эти сертификаты могут быть сгенерированы или импортированы в ПЛК, но их отсутствие влияет только на невозможность подключения к web-визуализации по HTTPS.

Зачастую отличить настоящую ошибку от «ошибки» довольно сложно – этот навык приходит лишь с опытом использования конкретных устройств и изучения особенностей их работы.

3. Инструменты отладки в CODESYS V3.5

3.1. Онлайн-подключение к контроллеру

Онлайн-подключение позволяет просматривать и изменять текущие значения переменных в процессе работы проекта, работать с визуализацией контроллера, узнать статус компонентов, добавленных в дереве проекта, изучить лог работы ПЛК и т.д.

Онлайн-подключение автоматически выполняется после загрузки проекта, а также может быть произведено пользователем через команду **Онлайн – Логин**.

Во время онлайн-подключения вкладка каждого POU содержит текущие значения его переменных – они отображаются в области объявления переменных в столбце **Значение** и в области кода рядом с местом использования переменной.

В процессе отладки может быть полезно изменить значение переменных, чтобы проверить, как поведет себя программа. Например, вы хотите понять, не работает ли блок записи в файл в принципе или не удается записать данные в один конкретный файл – соответственно, вам нужно попробовать задать разные значения для переменной, которая содержит путь к файлу.

CODESYS предоставляет два варианта этой операции: запись и фиксирование. В обоих случаях требуется сначала подготовить новое значение переменной – ввести его в столбце **Подготовленное значение** в области объявления или в окне **Подготовка значения**, которое открывается при двойной нажатии ЛКМ на текущее значение переменной в области кода. Подготовленное значение переменной будет отображаться в редакторе кода в угловых скобках справа от текущего[1] значения переменной (см. рис. 3.1.2).

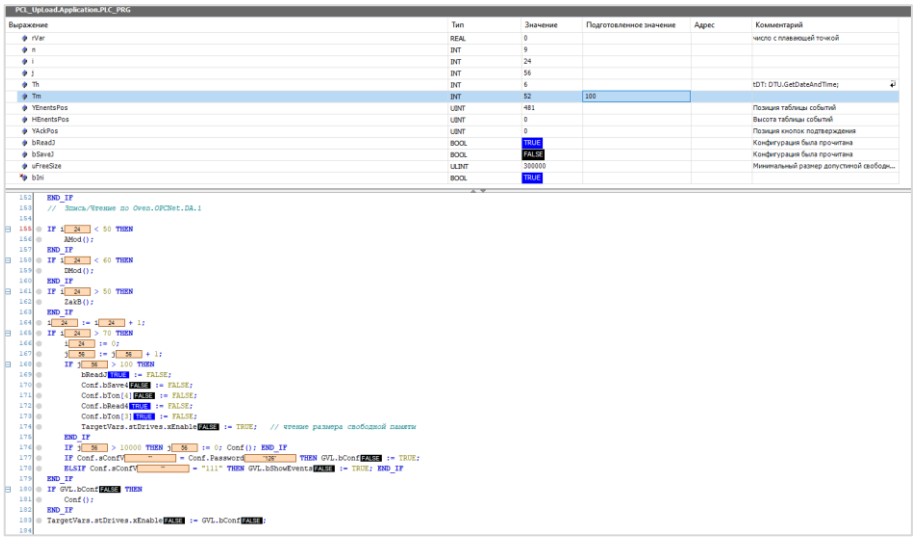


Рис. 3.1.1. Внешний вид программы на языке ST при онлайн-подключении к контроллеру (рисунок хорошо масштабируется)

[1] Под «текущим значением» понимается значение, которое переменная получила после выполнения последнего на данный момент цикла задачи, в котором был вызван соответствующий POU

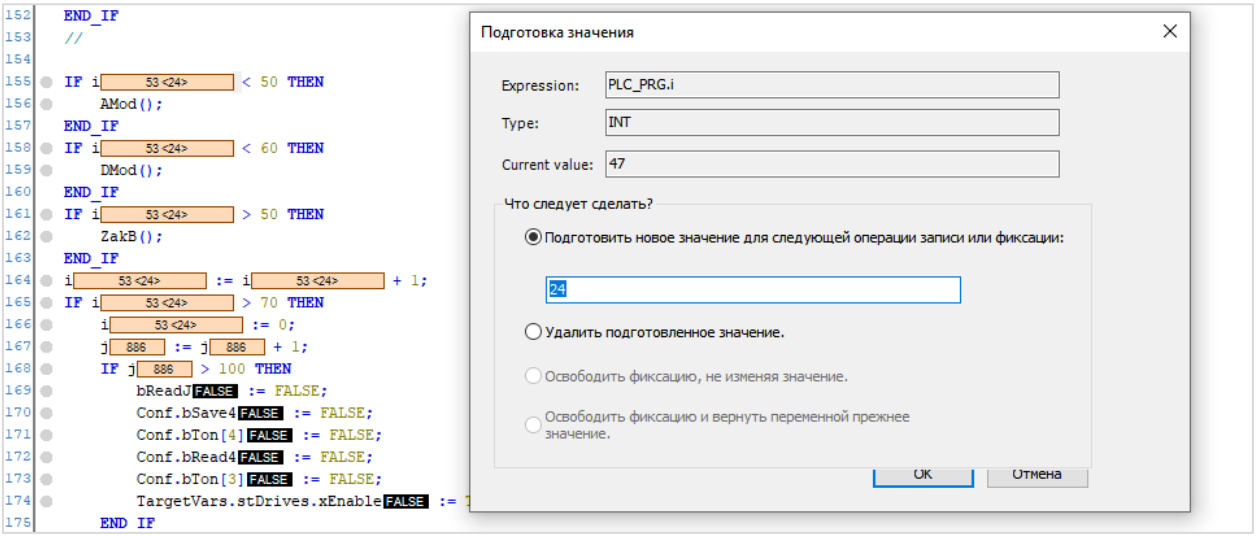


Рис. 3.1.2. Отображение подготовленного значения переменной

Если пользователь подготовил значения для нескольких переменных, то все они будут записаны одновременно. Записать значения можно одним из следующих способов:

- нажатием комбинации горячих клавиш **Ctrl+F7**;
- использованием команды **Отладка – Записать значения**;
- нажатием ПКМ на любую ячейку в области объявления переменных и использования команды **Записать все значения**.

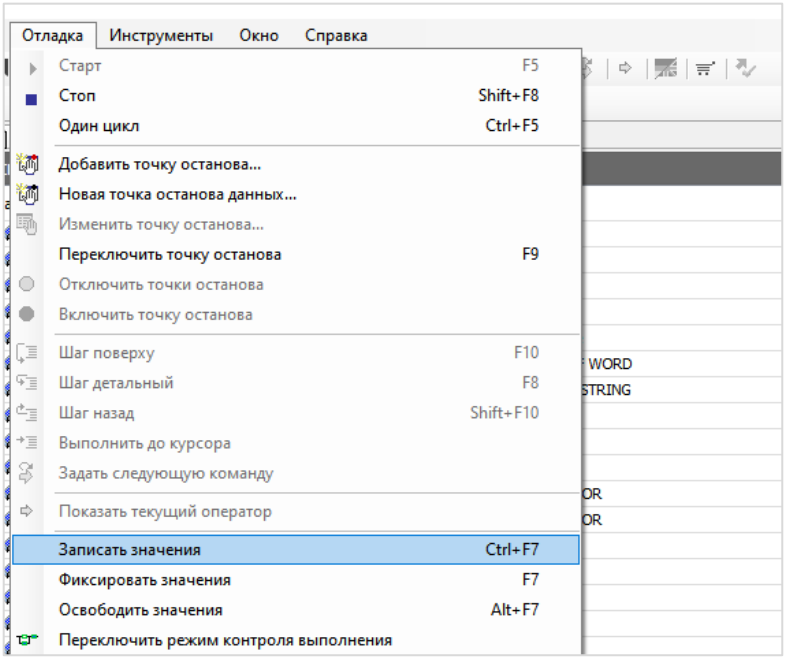


Рис. 3.1.3. Команда **Записать значения** в меню **Отладка**

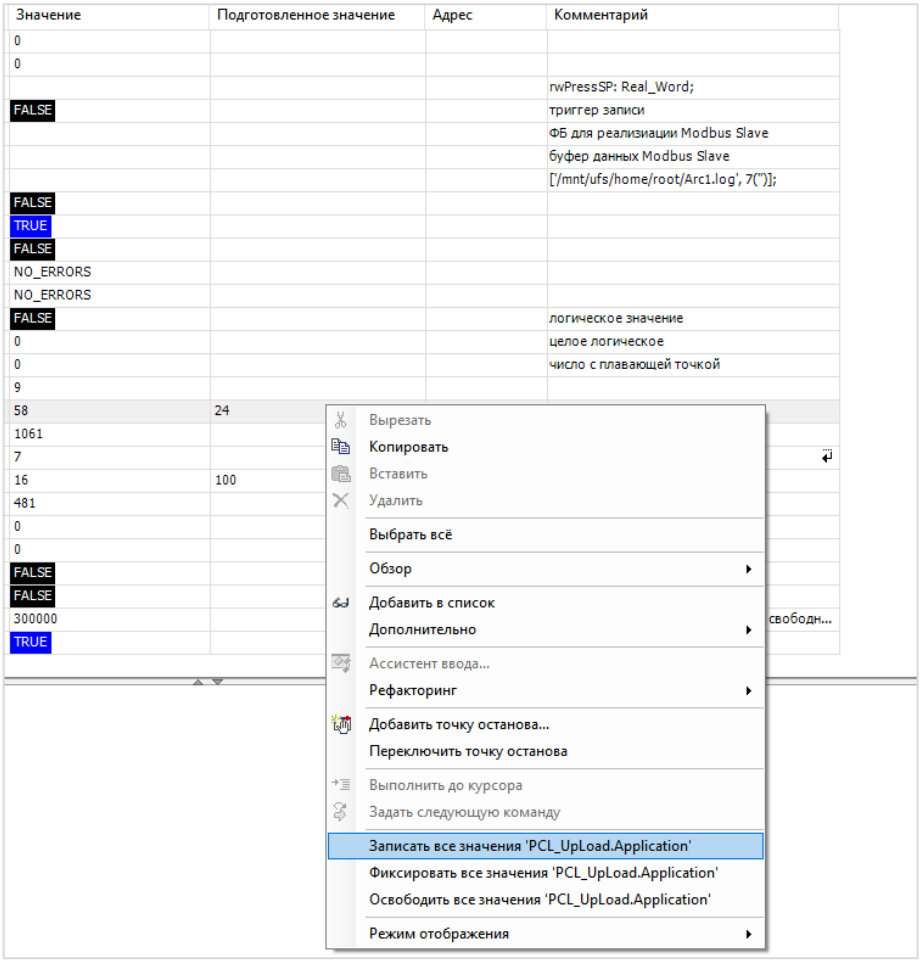


Рис. 3.1.4. Команда **Записать все значения** контекстного меню области объявления

После выполнения записи в выбранные переменные будут однократно записаны подготовленные значения. Фактически запись произойдет перед выполнением ближайшего по времени цикла задачи, в котором используется данный POU. Если в коде выполняется присваивание значений этих переменным – то выполнение команды не произведет никакого эффекта (так как подготовленные значения сразу будут перезаписаны значениями из кода).

В некоторых случаях было бы удобно не просто записать в переменную новое значение, а сделать это только на один цикл ПЛК, после чего переменная получила бы свое предыдущие значение. Например, это было бы очень удобно в тех случаях, когда требуется сгенерировать единичный импульс в переменной типа **BOOL**. В настоящий момент подобный функционал в CODESYS отсутствует, но соответствующее пожелание зафиксировано в баг-трекере.

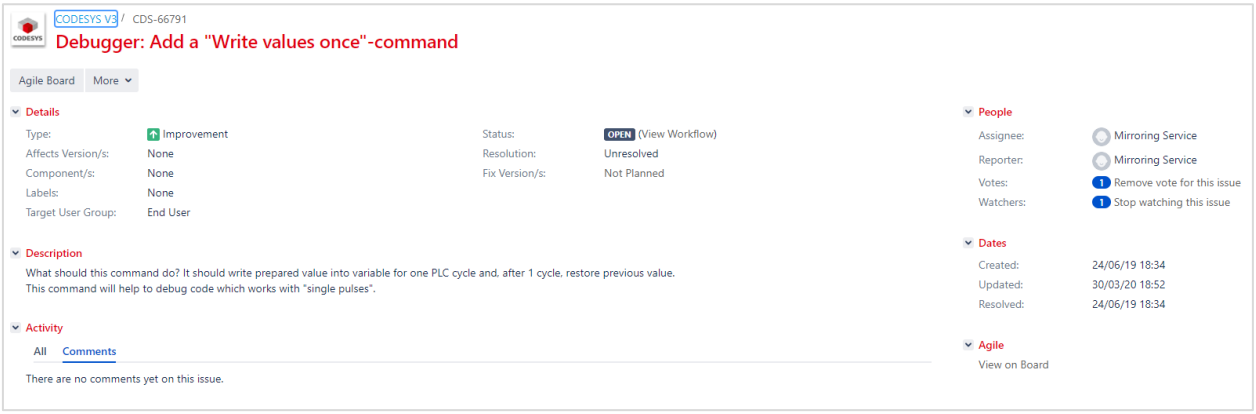


Рис. 3.1.5. Пожелание по добавлению команды **Write values once** в баг-трекере CODESYS

Вторым вариантом записи переменной является фиксирование. В этом случае подготовленные значения не просто однократно записываются в переменные, а присваиваются в каждом цикле ПЛК до и после выполнения программ соответствующей задачи (т.е. сразу после фазы **ReadInputs** и перед фазой **WriteOutputs**). В процессе выполнения программ значения переменных могут меняться в зависимости от их логики.

Смысл фиксирования – эмуляция входных и выходных сигналов ПЛК (аппаратных или сетевых). Предположим, у ПЛК есть дискретные входы, к которым физически ничего не подключено – соответственно, привязанные к ним переменные всегда имеют значения FALSE. Но для отладки может потребоваться понять, что происходит в случае срабатывания дискретного входа – и для эмуляции этого можно фиксировать его значение. В результате после выполнения фазы **ReadInputs** фактическое значение переменной, полученное от драйвера ввода-вывода, будет перезаписано значением, подготовленным пользователем.

Фиксировать подготовленные значения можно одним из следующих способов:

- нажатием горячей клавиши **F7**;
- использованием команды **Отладка – Фиксировать значения;**
- нажатием ПКМ на любую ячейку в области объявления переменных и использования команды **Фиксировать все значения**.

Рядом с текущим значением фиксированной переменной отображается соответствующая пиктограмма:

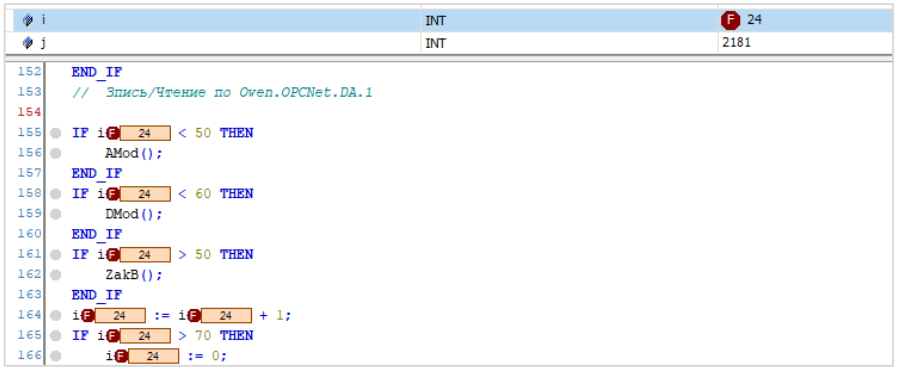


Рис. 3.1.6. Пиктограмма фиксированной переменной

Отменить фиксацию можно одним из следующих способов:

- нажатием комбинации горячих клавиш **Alt+F7**;
- использованием команды **Отладка – Освободить значения**;
- нажатием ПКМ на любую ячейку в области объявления переменных и использования команды **Освободить все значения**;
- двойным нажатием ЛКМ на конкретную фиксированную переменную в редакторе кода или одиночным нажатием ЛКМ на ячейку **Подготовленное значение** этой переменной в области переменной и использованием команды **Освободить фиксацию, не изменяя значение** или **Освободить фиксацию и вернуть переменной прежнее значение**. В отличие от других способов – этот вариант позволяет отменить фиксацию конкретной переменной проекта (а не всех сразу).

Как мы уже упоминали, при онлайн-подключении в области объявления отображаются текущие значения переменных POU. В некоторых случаях удобно наблюдать сразу за значениями переменных разных POU и иметь возможность записывать или фиксировать их. Для этого используется **Список просмотра**, который можно открыть с помощью команды **Вид – Просмотр**. Пользователю доступно 4 списка просмотра (для разных наборов переменных) и список всех зафиксированных в данный момент переменных. Добавить переменную в список просмотра можно в столбце **Выражение** или выбрав переменную в области объявления или области кода и использовав команду контекстного меню **Добавить в список**.

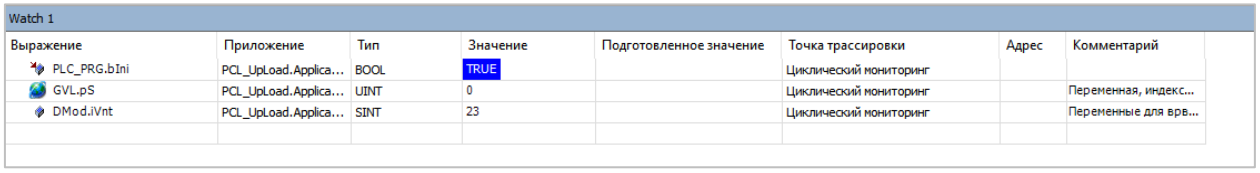


Рис. 3.1.7. Внешний вид списка просмотра

Еще одна функция, которая может упростить отладку – режим контроля выполнения, который включается одноименной командой в меню **Онлайн**. В режиме контроля выполнения текущие значения переменных, обработанных в прошлом цикле ПЛК, выделяются зеленым, а необработанных – белым. Это позволяет легко понять, какие ветки кода исполняются в данный момент времени.

В режиме контроля выполнения время цикла ПЛК увеличивается, а использование точек останова и пошагового выполнения кода является недоступным.

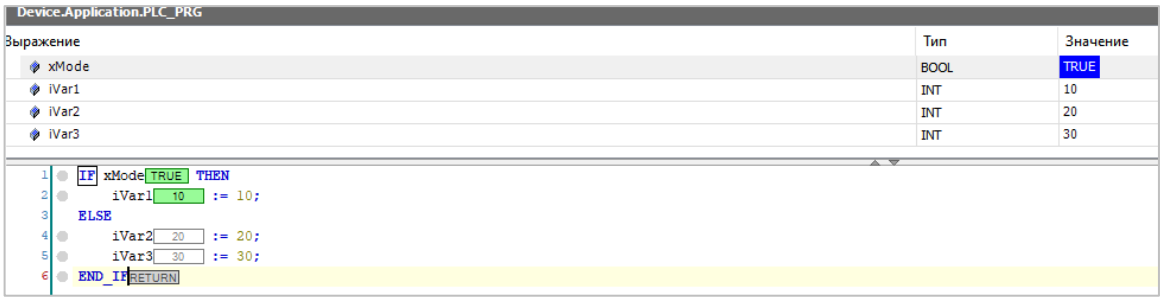


Рис. 3.1.8. Режим контроля выполнения в редакторе ST

3.2. Точки останова

При отладке программы часто требуется понять, выполнялись ли те или иные ветки кода, и если выполнялись – то какие значения имели связанные с ними переменные (например, какие данные попали в буфер COM-порта). Поскольку цикл задачи контроллера составляет единицы миллисекунд, то определить это визуально невозможно. В этом случае следует использовать точки останова.

Точки останова позволяют перевести приложение в состояние **Стоп** при переходе контроллером к выполнению определенных строк кода. В этом «замороженном» состоянии пользователь может изучить текущие значения переменных или использовать команды пошагового выполнения из меню **Отладка**, которые мы рассмотрим позже.

В редактора языка ST рядом со строками, на которых могут быть установлены точки останова, отображаются пиктограммы в виде серых кружков. Для добавления точки останова нужно нажать на такой кружок правой кнопкой мыши и выбрать команду **Добавить точку останова** или использовать одноименную команду из меню **Онлайн**.

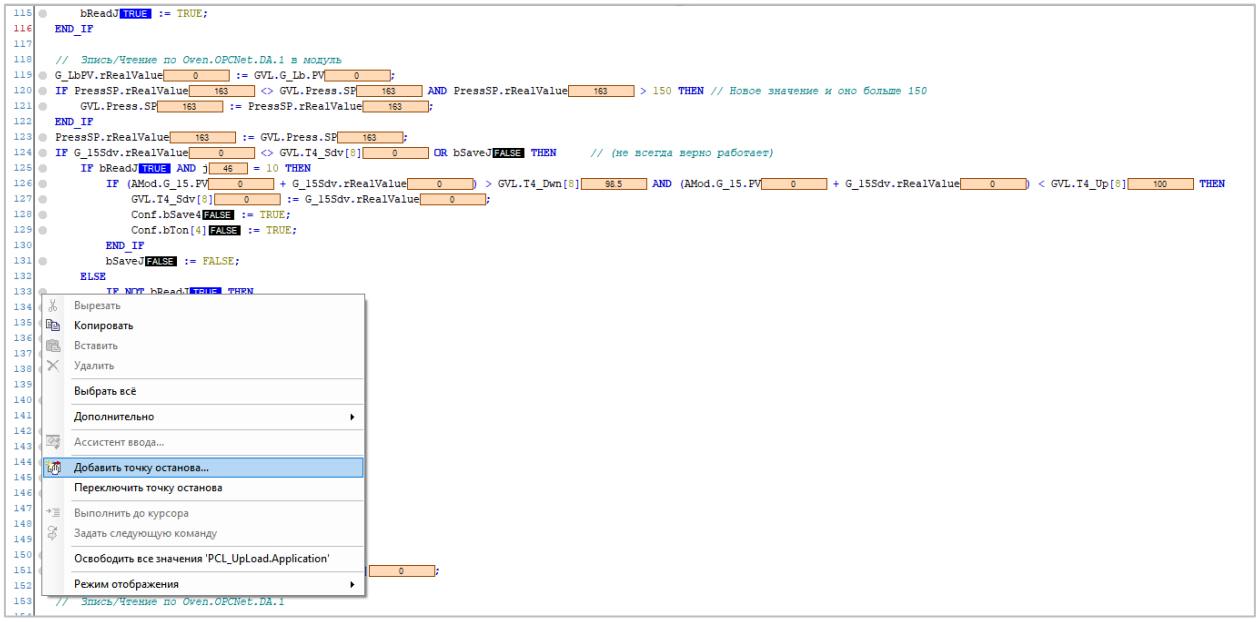


Рис. 3.2.1. Добавление точки останова

После этого откроется окно настройки точки останова.

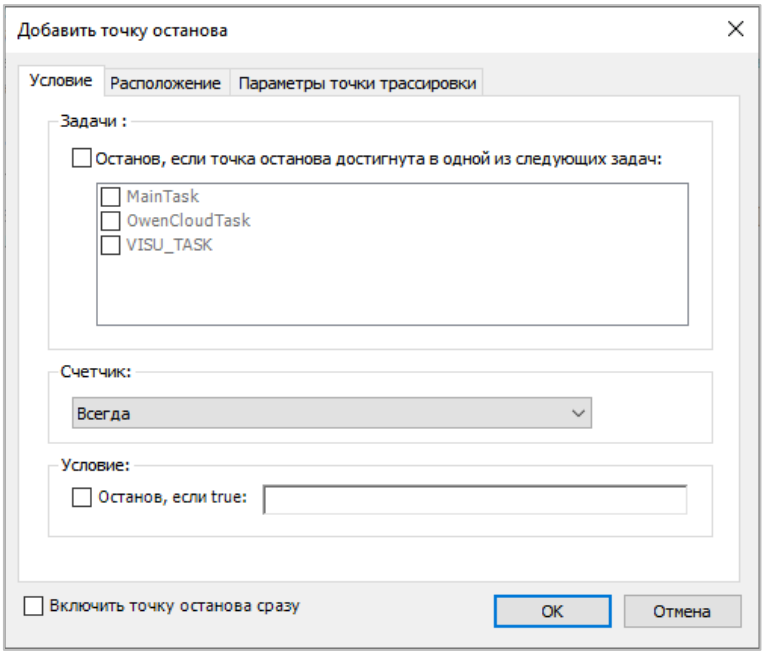


Рис. 3.2.2. Настройки точки останова, вкладка **Условие**

Задачи – список задач, в которых выполнение выбранного ФБ приведет к остановке приложения. Если точка останова устанавливается в программе, то эта опция не обрабатывается.

Счетчик – количество выполнения строки кода, на которой установлена точка останова, которое приведет к остановке приложения. Возможные варианты:

- всегда;
- останов, когда число попаданий равно введенному пользователем значению;
- останов, когда число попаданий кратно введенному пользователем значению;
- останов, когда число попаданий больше или равно введенному пользователем значению.

Условие – логическое выражение (например, *PLC_PRG.xDebugEnabled AND PLC_PRG.wWorkMode = 2*), которое определяет возможность остановки приложения при достижении данной точки останова. Для обработки выражения следует установить галочку **Останов, если TRUE**.

Включить точку останова сразу (*эта опция отображается на всех вкладках настройки точки останова*) – в случае установки галочки данная точка останова будет активирована сразу после ее добавления. Если галочка не установлена, то пользователь должен будет активировать точку останова вручную с помощью команд **Включить** или **Переключить точку останова** (из контекстного меню точки редактора кода – см. рис. 3.2.1 – или одноименных команд меню **Онлайн**).

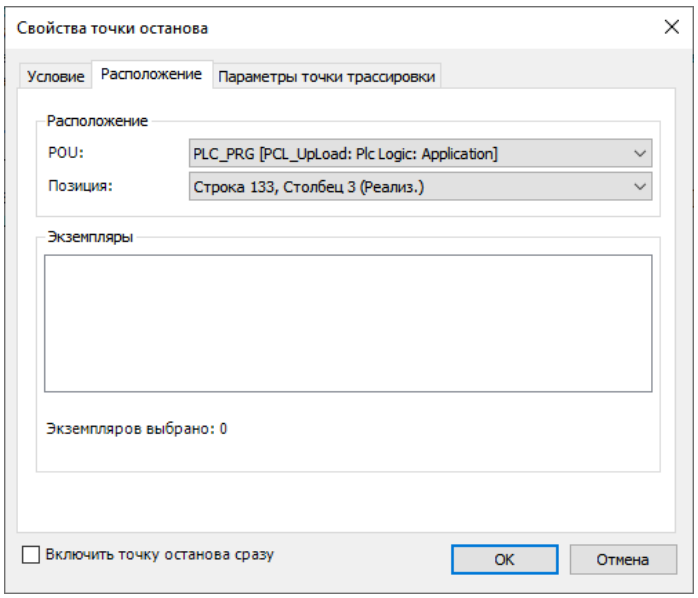


Рис. 3.2.3. Настройки точки останова, вкладка **Расположение**

На данной вкладке отображаются название POU и номер строки, на которой установлена точка останова (они заполняются автоматически). Если точка останова устанавливается внутри ФБ – то можно выбрать конкретный экземпляр блока.

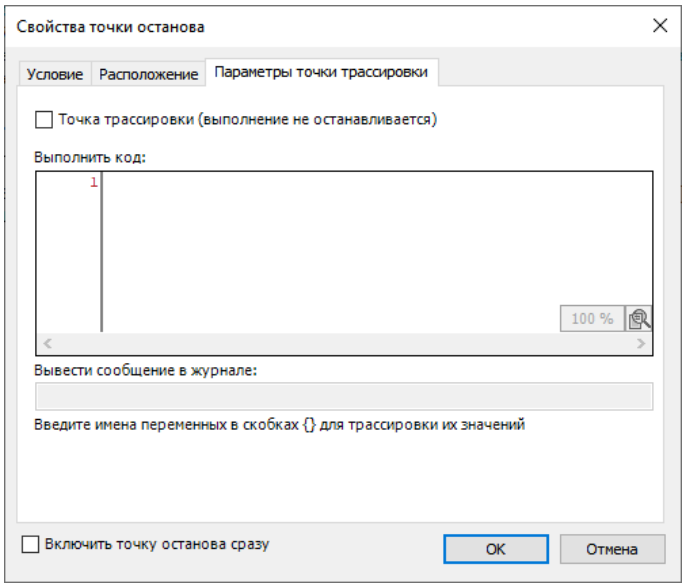


Рис. 3.2.4. Настройки точки останова, вкладка **Параметры точки трассировки**

На данной вкладке пользователь может изменить тип создаваемой точки останова на точку трассировки (с помощью галочка **Точка трассировки**). При достижении точки трассировки приложение не останавливается; вместо этого происходит добавление записи в журнал ПЛК и выполнение кода, добавленного пользователем на данной вкладке.

Добавленная, но не активированная точка останова выглядит следующим образом:

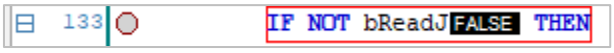


Рис. 3.2.5. Неактивированная точка останова

Активированная точка остановка выглядит вот так:

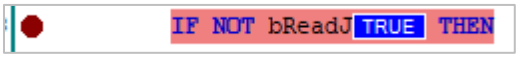


Рис. 3.2.6. Активированная точка останова

Если приложение было остановлена в точке останова, то соответствующая строка выделяется желтым:

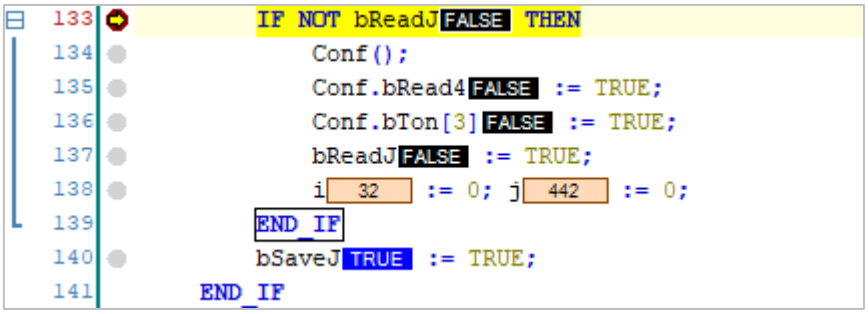


Рис. 3.2.7. Сработавшая точка останова

Для просмотра всех точек остановка проекте используется команда **Вид – Точки останова.**

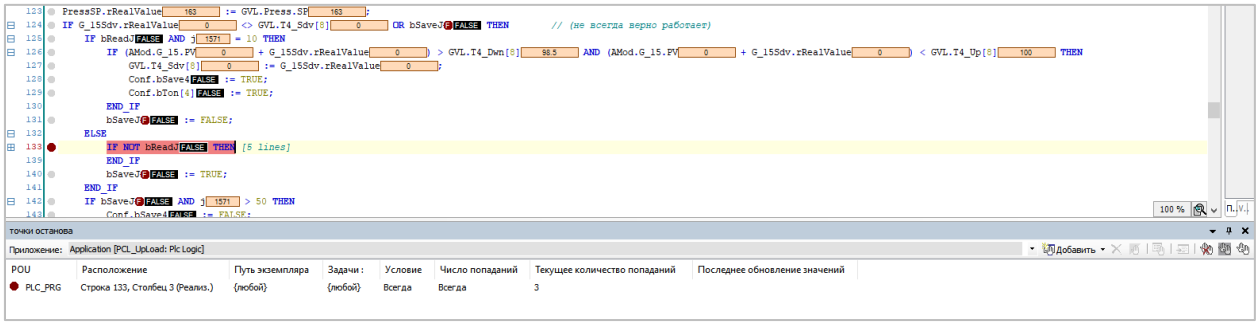


Рис. 3.2.8. Внешний вид панели точек останова

В меню **Отладка** доступны следующие команды для работы с точками останова:

- **Добавить точку останова** – добавляет в проект новую точку останова;
- **Новая точка останова данных** – добавляет в проект новую точку останова данных. Точки останова данных – особый вид точек останова, которые связываются не со строками кода, а с переменными. Приложение переходит в режим **Стоп** в том случае, если значение связанной переменной изменяется. В текущих версиях CODESYS (V3.5 SP16) точки останова данных поддерживаются только в системах исполнения для ОС Windows. Более подробную информацию о них можно найти в справке CODESYS;
- **Изменить точку останова** – открывает меню настроек выбранной точки останова;
- **Переключить точку останова** (F9) – удаляет/возвращает точку останова выбранной строки;
- **Отключить точку останова** – деактивирует (запрещает обработку) выбранной точки останова;
- **Включить точку останова** – активирует (разрешает обработку) выбранной точки останова.

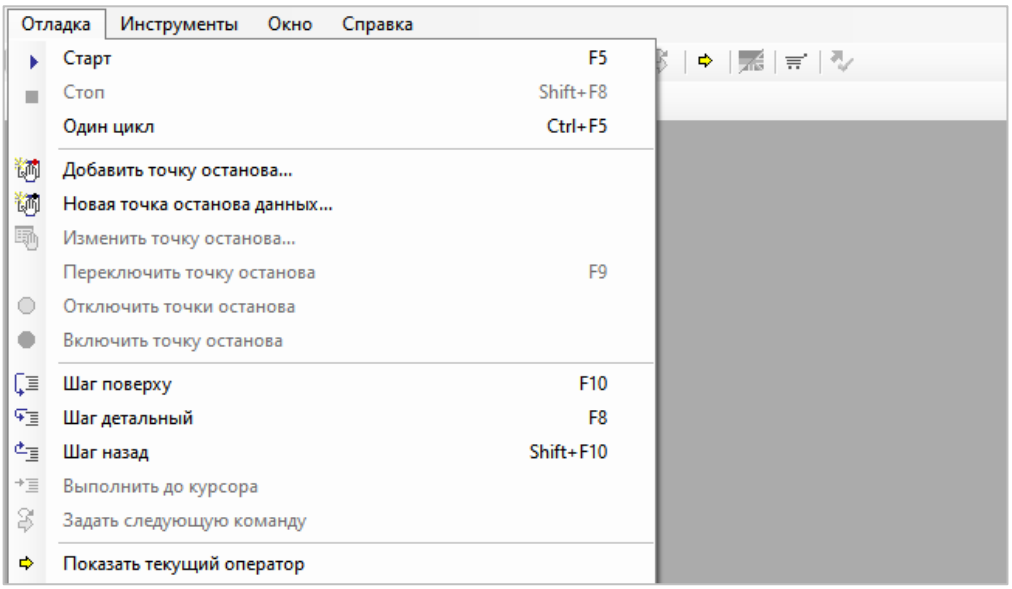


Рис. 3.2.9. Команды для работы с точками останова в меню **Отладка**

Если приложение остановлено в точке останова, то становятся доступны команды пошагового выполнения проекта:

- **Шаг поверху** – выполняется следующее выражение. Если выражение представляет собой вызов POU, то весь этот POU выполняется целиком;
- **Шаг детальный** – выполняется следующее выражение. Если выражение представляет собой вызов POU, то выполняется первое выражение POU, и далее можно производить пошаговый вызов этого POU;
- **Шаг назад** – весь код POU, в котором установлена точна останова, выполняется до конца, и выполнение останавливается на первой строке в POU. Если ранее была выполнена команда **Шаг детальный**, то весь POU, в который был

осуществлен переход, выполняется до конца, и выполнение останавливается на строке вызова этого POU в исходном POU;

- **Выполнить до курсора** – выполняется весь код от текущей точки и до текущей позиции курсора в редакторе кода;
- **Задать следующую команду** – выполняется переход к текущей позиции курсора без выполнения кода, размещенного до этой позиции;
- **Показать текущий оператор** – переход к текущей позиции курсора (к строке, на которой остановилось выполнение кода).

В редких случаях при попытке установки точки останова может появиться следующее сообщение:

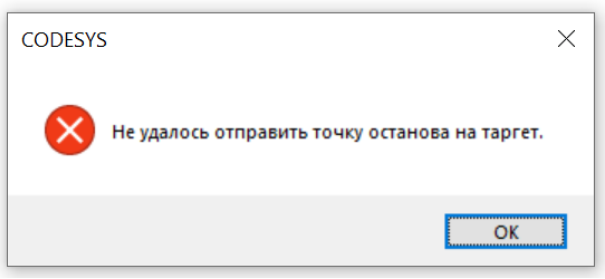


Рис. 3.2.10. Ошибка при добавлении точки останова

Мы встречались с такой проблемой всего в двух пользовательских проектах, и нам не удалось установить точную причину ее возникновения, но гипотеза, которая кажется нам наиболее вероятной – в проекте были ошибки сегментации памяти, из-за чего происходила запись в область памяти, выделенную под отладочный код.

3.3. Журнал ПЛК

Журнал (лог) ПЛК (узел **Device** – вкладка **Журнал**) позволяет посмотреть список сообщений системы исполнения CODESYS. Изучение этих сообщений может помочь понять причины наблюдаемых проблем.

Сообщения подразделяются на 5 категорий:

- **исключения** – сообщения о событиях, которые делают продолжение нормальной работы контроллера невозможным (деление на 0, бесконечный цикл, ошибка сегментации памяти и т.д.). По умолчанию система исполнения после возникновения исключения прекращает выполнения приложения, переводя его в режим **Стоп**. Обработка исключений может производиться пользовательским кодом (**Конфигурация задач – Системные события – обработчик события Exception**, конструкцией **TRY/CATCH/FINALLY** и т.д.) или функционалом, предоставляемым разработчиками контроллера (например, сторожевым таймером ПЛК);
- **ошибки** – сообщения о событиях, при которых контроллер может продолжать свою работу, но часть функционала перестает работать или работает некорректно (например, из-за отсутствия лицензий, несовпадения версий компонентов в среде программирования и системе исполнения и т.д.);
- **предупреждения** – сообщения о событиях, которые связаны с частными ошибками (например, ошибками обмена по конкретному интерфейсу) или особенностями конкретного устройства (например, в логе некоторых панельных контроллеров можно увидеть предупреждение, что они не поддерживают технологию multi-touch);
- **информационные сообщения** – сообщения о версиях используемых компонентов, настройках контроллера и т.д.
- **отладочные сообщения** – сообщения о работе системных компонентов, которые используют разработчики CODESYS и контроллеров во время отладки своего ПО.

Ниже приведен скриншот журнала ПЛК после возникновения исключения, связанного с появлением в коде бесконечного цикла:

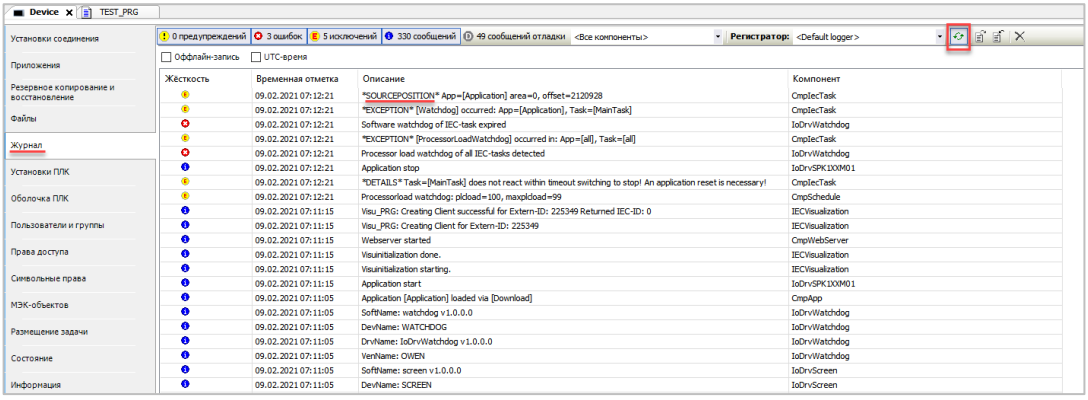


Рис. 3.3.1. Внешний вид журнала ПЛК

В журнале можно увидеть, что произошло после активации в программе кода с бесконечным циклом:

- сообщение (тип: исключение) от компонента **CmpSchedule**, что загрузка процессора ПЛК стала равна 100% при максимально допустимой 99%, что привело к срабатыванию сторожевого таймера планировщика задач;
- сообщение (тип: исключение) от компонента **CmpIecTask**, что задача **MainTask** перестала реагировать на запросы со стороны компонента и сработал таймаут ожидания;
- сообщение (тип: информация) об остановке приложения;
- сообщения (тип: ошибка) от компонента **IoDrvWatchdog** (специализированный компонент контроллеров ОВЕН) о срабатывании сторожевого таймера планировщика задач;
- сообщения (тип: исключение) от компонента **CmpIecTask** о возникновении исключения в задаче **MainTask** с указанием его конкретного расположения (SOURCEPOSITION).

После двойного нажатия на строку с сообщением SOURCEPOSITION будет открыт код POU, который привел к возникновению исключения (соответствующая строка будет выделена желтым):

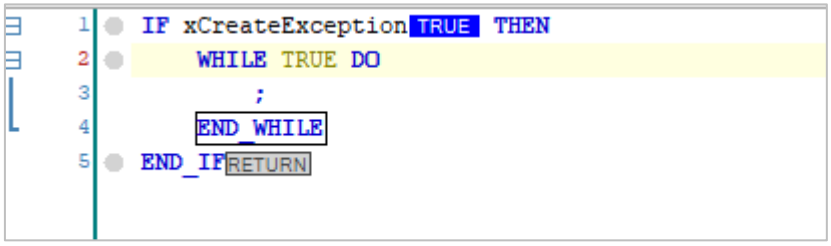


Рис. 3.3.2. Переход к строке, приведшей к исключению, в редакторе кода

В некоторых случаях вместо этого может появиться сообщение «the source in not available» (исходный код недоступен):

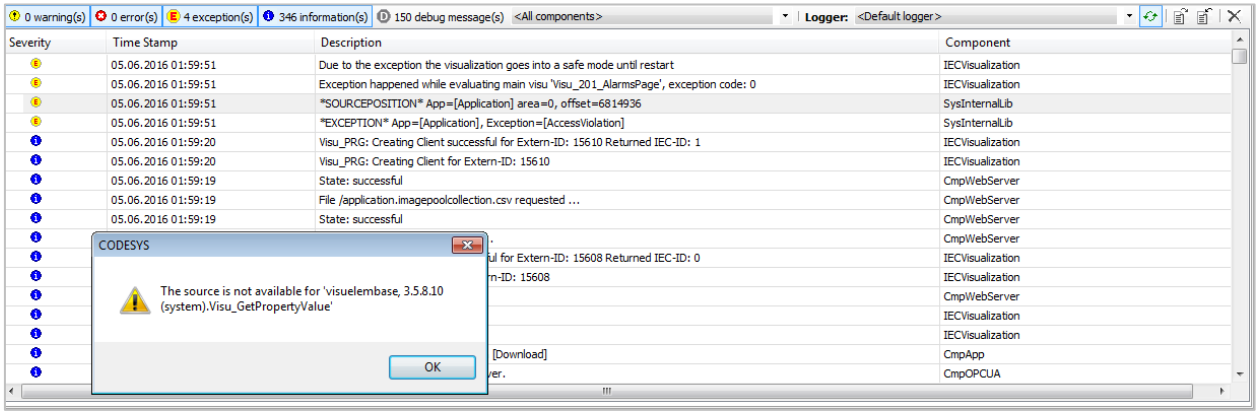
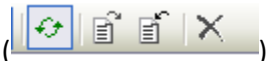


Рис. 3.3.3. Сообщение «the source in not available» при нажатии на строку SOURCEPOSITION

Это означает, что либо исключение произошло в скомпилированной библиотеке (обычно – системной), либо связано с ошибкой сегментацией памяти, которая привела к записи в область памяти, занятой этой библиотекой.

Пользователь может фильтровать сообщения журнала по типу, компоненту и логгеру (всего существует три типа логгеров – логгер сообщений ПЛК, логгер сообщений коммуникационных драйверов и пользовательские логгеры).

Кнопки, расположенные в правой части экрана () позволяют переключить журнал в режим автоматического обновления, а также экспортировать/импортировать/очистить его содержимое.

В контроллерах ОВЕН журнал CODESYS можно также посмотреть в web-конфигураторе (вкладка **ПЛК/Файлы журналов**) и сохранить его в формате .csv.

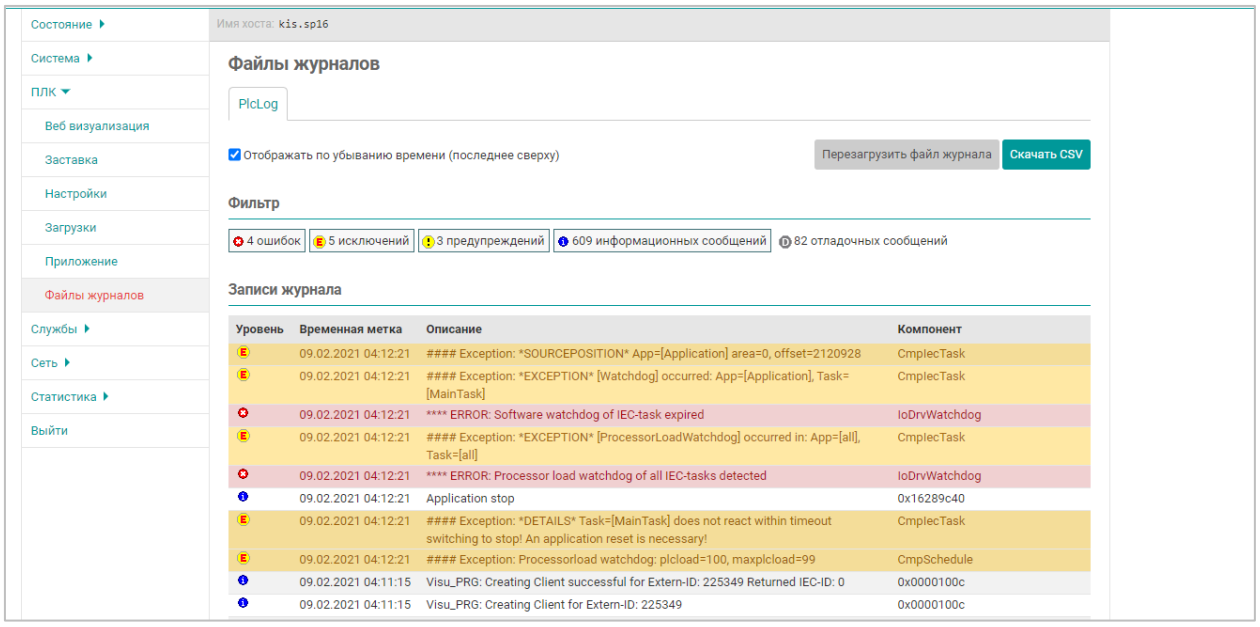


Рис. 3.3.4. Отображение журнала ПЛК в web-конфигураторе контроллеров ОВЕН

Пользователь может выводить сообщения в журнал с помощью библиотеки CmpLog, простым способом через функцию VisuElems.Visu_Output и с помощью точек трассировки (см. рис. 3.2.4).

3.4. Трассировка

Компонент **Трассировка** (**Appilcation – Добавление объекта – Trace**) позволяет оценить изменение значения переменной на заданном интервале времени. При отладке это бывает полезным, чтобы понять, как меняется значение переменной в процессе работы программы и оказывают ли на это влияние те или иные факторы (например, добавление/удаление определенных фрагментов кода).

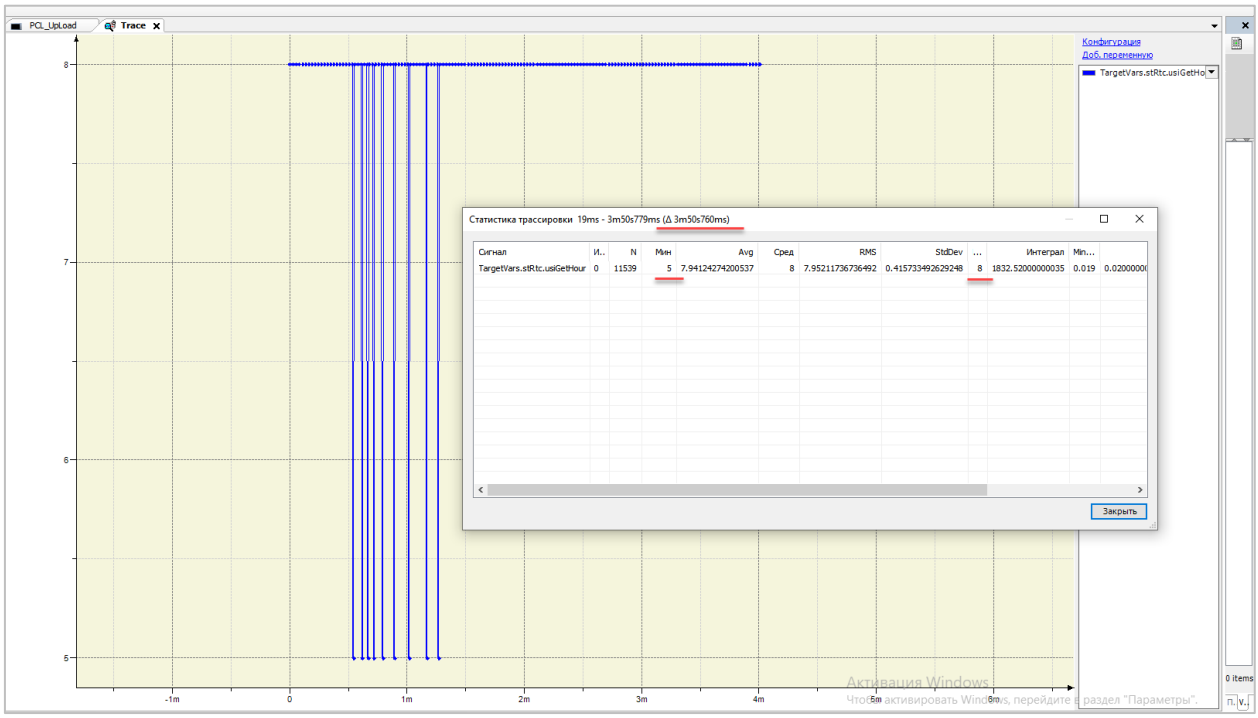


Рис. 3.4.1. Внешний вид трассировки

3.5. Плагин MemoryTools

Плагин **MemoryTools** позволяет просматривать содержимое оперативной памяти, используемой пользовательским приложением, и произвести проверку на корректность работы с ней. Это упрощает отладку приложений, в которых наблюдается ошибка сегментации памяти. Например, вы можете понять, какие данные «заехали» в чужую область и после этого проверить, в каком фрагменте кода это могло произойти.

Плагин можно скачать из CODESYS Store и установить через меню **Инструменты – Менеджер пакетов**. После установки потребуется перезапустить CODESYS.

В результате в CODESYS появятся дополнительные функции:

- Show Memory View и Show Memory Usage в меню **Вид**;
- Check Memory for Active Application в меню **Отладка**.

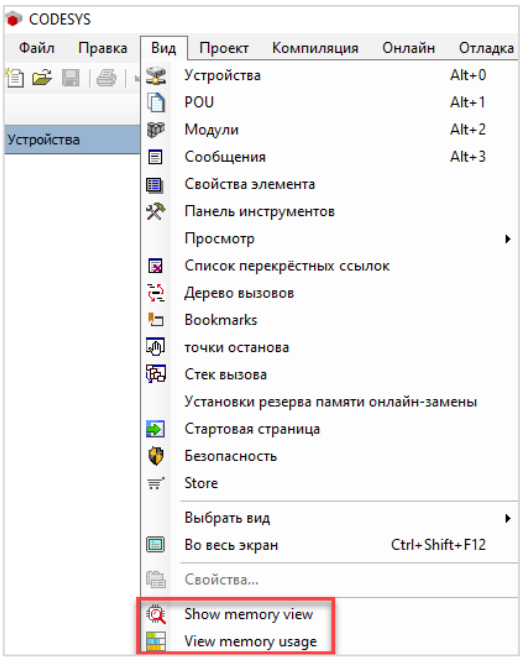


Рис. 3.5.1. Команды плагина **MemoryTools** в меню **Вид**

Команда **Show Memory View** доступна только при онлайн-подключении к контроллеру и позволяет просматривать содержимое оперативной памяти, используемой пользовательским приложением. С помощью кнопки  можно найти адрес памяти определенной переменной. С помощью кнопки  можно сохранить дамп памяти в бинарном виде.

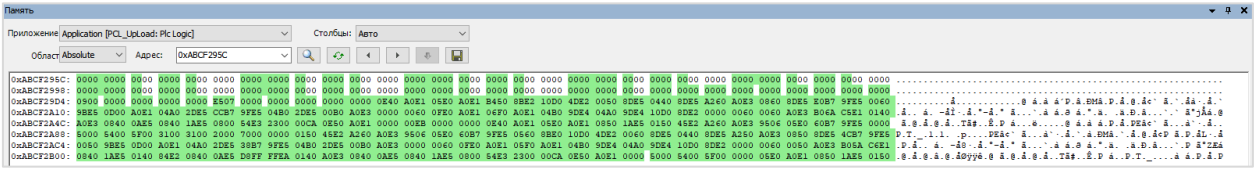


Рис. 3.5.2. Просмотр оперативной памяти с помощью команды **Show memory view**

Команда **Show Memory Usage** позволяет оценить, сколько памяти выделяется под переменные приложения. Подключения к контроллеру для использования команды не требуется – достаточно просто выполнить компиляцию проекта.

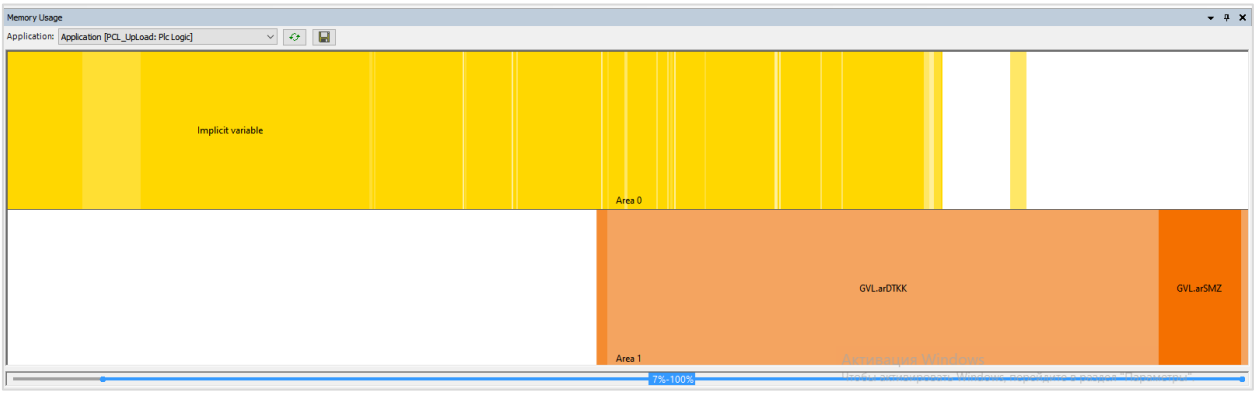


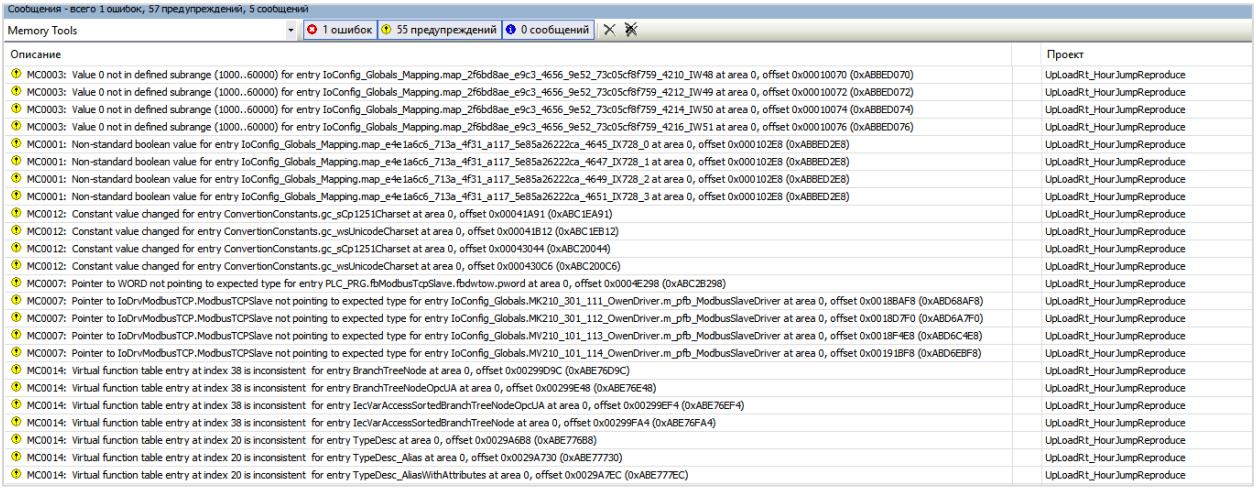
Рис. 3.5.3. Окно команды **Show Memory Usage**

Команда **Check Memory for Active Application** доступна только при онлайн-подключении к контроллеру и позволяет проверить использование памяти приложения на типовые ошибки. Примеры некоторых проверок:

- проверка на то, что переменные типа BOOL имеют значение 0 или 1 (так как переменная типа BOOL занимает в памяти один байт, то при ошибке сегментации в него может быть записано произвольное значение);
- проверка на то, что экземпляры перечислений имеют корректные значения (опять же, при ошибке сегментации в переменную может быть записано число, которое не соответствует ни одному элементу перечисления);
- проверка на то, что значения переменных с ограничением диапазона (subrange) входят в заданный диапазон (так как проверка диапазона производится только на этапе компиляции – то в процессе выполнения программы таким переменным могут быть присвоены значения, выходящие за диапазон);
- проверка на то, что в строках переменных типа STRING и WSTRING имеются терминирующие нули;
- проверка на то, что под указателем находится переменная, тип которой соответствует типу указателя.

Полный список проверок приведен в онлайн-справке CODESYS.

Если требуется исключить какие-то переменные или POU из списка проверяемых – то можно воспользоваться прагмой *{attribute 'memory_check' := 'ignore'}*.


Сообщения - всего 1 ошибок, 57 предупреждений, 5 сообщений

Memory Tools | 1 ошибок | 55 предупреждений | 0 сообщений

Описание	Проект
MC0003: Value 0 not in defined subrange (1000..60000) for entry IoConfig_Globals_Mapping.map_2f6bd8ae_e9c3_4656_9e52_73c05cf8f759_4210_IW48 at area 0, offset 0x00010070 (0xABBED070)	UpLoadRt_HourJumpReproduce
MC0003: Value 0 not in defined subrange (1000..60000) for entry IoConfig_Globals_Mapping.map_2f6bd8ae_e9c3_4656_9e52_73c05cf8f759_4212_IW49 at area 0, offset 0x00010072 (0xABBED072)	UpLoadRt_HourJumpReproduce
MC0003: Value 0 not in defined subrange (1000..60000) for entry IoConfig_Globals_Mapping.map_2f6bd8ae_e9c3_4656_9e52_73c05cf8f759_4214_IW50 at area 0, offset 0x00010074 (0xABBED074)	UpLoadRt_HourJumpReproduce
MC0003: Value 0 not in defined subrange (1000..60000) for entry IoConfig_Globals_Mapping.map_2f6bd8ae_e9c3_4656_9e52_73c05cf8f759_4216_IW51 at area 0, offset 0x00010076 (0xABBED076)	UpLoadRt_HourJumpReproduce
MC0001: Non-standard boolean value for entry IoConfig_Globals_Mapping.map_e4e1a6c6_713a_4f31_a117_5e85a26222ca_4645_IX728_0 at area 0, offset 0x000102E8 (0xABBED2E8)	UpLoadRt_HourJumpReproduce
MC0001: Non-standard boolean value for entry IoConfig_Globals_Mapping.map_e4e1a6c6_713a_4f31_a117_5e85a26222ca_4647_IX728_1 at area 0, offset 0x000102E8 (0xABBED2E8)	UpLoadRt_HourJumpReproduce
MC0001: Non-standard boolean value for entry IoConfig_Globals_Mapping.map_e4e1a6c6_713a_4f31_a117_5e85a26222ca_4649_IX728_2 at area 0, offset 0x000102E8 (0xABBED2E8)	UpLoadRt_HourJumpReproduce
MC0001: Non-standard boolean value for entry IoConfig_Globals_Mapping.map_e4e1a6c6_713a_4f31_a117_5e85a26222ca_4651_IX728_3 at area 0, offset 0x000102E8 (0xABBED2E8)	UpLoadRt_HourJumpReproduce
MC0012: Constant value changed for entry ConvertionConstants.gc_sCp1251Charset at area 0, offset 0x00041A91 (0xABC1EA91)	UpLoadRt_HourJumpReproduce
MC0012: Constant value changed for entry ConvertionConstants.gc_wsUnicodeCharset at area 0, offset 0x00041B12 (0xABC1EB12)	UpLoadRt_HourJumpReproduce
MC0012: Constant value changed for entry ConvertionConstants.gc_sCp1251Charset at area 0, offset 0x00043044 (0xABC20044)	UpLoadRt_HourJumpReproduce
MC0012: Constant value changed for entry ConvertionConstants.gc_wsUnicodeCharset at area 0, offset 0x000430C6 (0xABC200C6)	UpLoadRt_HourJumpReproduce
MC0007: Pointer to WORD not pointing to expected type for entry PLC_PRG.fbModbusTcpSlave.fbdwtow.pword at area 0, offset 0x0004E298 (0xABC2B298)	UpLoadRt_HourJumpReproduce
MC0007: Pointer to IoDrvModbusTCP.ModbusTCPSlave not pointing to expected type for entry IoConfig_Globals.MK210_301_111_OwenDriver.m_pfb_ModbusSlaveDriver at area 0, offset 0x0018BAF8 (0xABD68AF8)	UpLoadRt_HourJumpReproduce
MC0007: Pointer to IoDrvModbusTCP.ModbusTCPSlave not pointing to expected type for entry IoConfig_Globals.MK210_301_112_OwenDriver.m_pfb_ModbusSlaveDriver at area 0, offset 0x0018D7F0 (0xABD6A7F0)	UpLoadRt_HourJumpReproduce
MC0007: Pointer to IoDrvModbusTCP.ModbusTCPSlave not pointing to expected type for entry IoConfig_Globals.MV210_101_113_OwenDriver.m_pfb_ModbusSlaveDriver at area 0, offset 0x0018F4E8 (0xABD6C4E8)	UpLoadRt_HourJumpReproduce
MC0007: Pointer to IoDrvModbusTCP.ModbusTCPSlave not pointing to expected type for entry IoConfig_Globals.MV210_101_114_OwenDriver.m_pfb_ModbusSlaveDriver at area 0, offset 0x00191BF8 (0xABD6EBF8)	UpLoadRt_HourJumpReproduce
MC0014: Virtual function table entry at index 38 is inconsistent for entry BranchTreeNode at area 0, offset 0x00299D9C (0xABE76D9C)	UpLoadRt_HourJumpReproduce
MC0014: Virtual function table entry at index 38 is inconsistent for entry BranchTreeNodeOpcUA at area 0, offset 0x00299E48 (0xABE76E48)	UpLoadRt_HourJumpReproduce
MC0014: Virtual function table entry at index 38 is inconsistent for entry IecVarAccessSortedBranchTreeNodeOpcUA at area 0, offset 0x00299EF4 (0xABE76EF4)	UpLoadRt_HourJumpReproduce
MC0014: Virtual function table entry at index 38 is inconsistent for entry IecVarAccessSortedBranchTreeNode at area 0, offset 0x00299FA4 (0xABE76FA4)	UpLoadRt_HourJumpReproduce
MC0014: Virtual function table entry at index 20 is inconsistent for entry TypeDesc at area 0, offset 0x0029A6B8 (0xABE776B8)	UpLoadRt_HourJumpReproduce
MC0014: Virtual function table entry at index 20 is inconsistent for entry TypeDesc_Alias at area 0, offset 0x0029A730 (0xABE77730)	UpLoadRt_HourJumpReproduce
MC0014: Virtual function table entry at index 20 is inconsistent for entry TypeDesc_AliasWithAttributes at area 0, offset 0x0029A7EC (0xABE777EC)	UpLoadRt_HourJumpReproduce

Рис. 3.5.4. Список сообщений после выполнения команды **Check Memory for Active Application**

3.6. Онлайн-мониторинг Конфигурации задач

При подключении к контроллеру на вкладке **Мониторинг** компонента **Конфигурация задач** отображается информация онлайн-мониторинга задач – например, время выполнения задачи при ее последнем вызове. При отладке это позволяет оценить реальное время выполнения задач (оно может быть больше интервала вызова, установленного пользователем), наличие периодических «выбросов» во временах выполнения и т.д.

Механизм выполнения задач и описание параметров вкладки **Мониторин**г рассмотрены в статье Использование задач в CODESYS V3.

Для сброса значений следует нажать на строку нужной задачи **ПКМ** и выбрать команду **Сброс**. Следует отметить, что в момент старта приложения все тайминги имеют высокие значения (так как помимо пользовательского кода выполняется код инициализации). Поэтому для реалистичной оценки значений следует сбросить их после начала выполнения приложения.

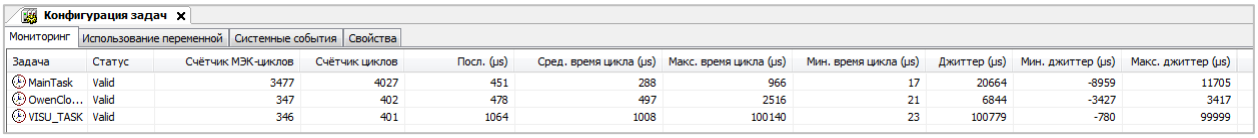


Рис. 3.6.1. Внешний вид вкладки онлайн-мониторинга задач

Для контроллеров ОВЕН информация мониторинга задач также доступна в web-конфигураторе (вкладка **ПЛК/Приложение**).

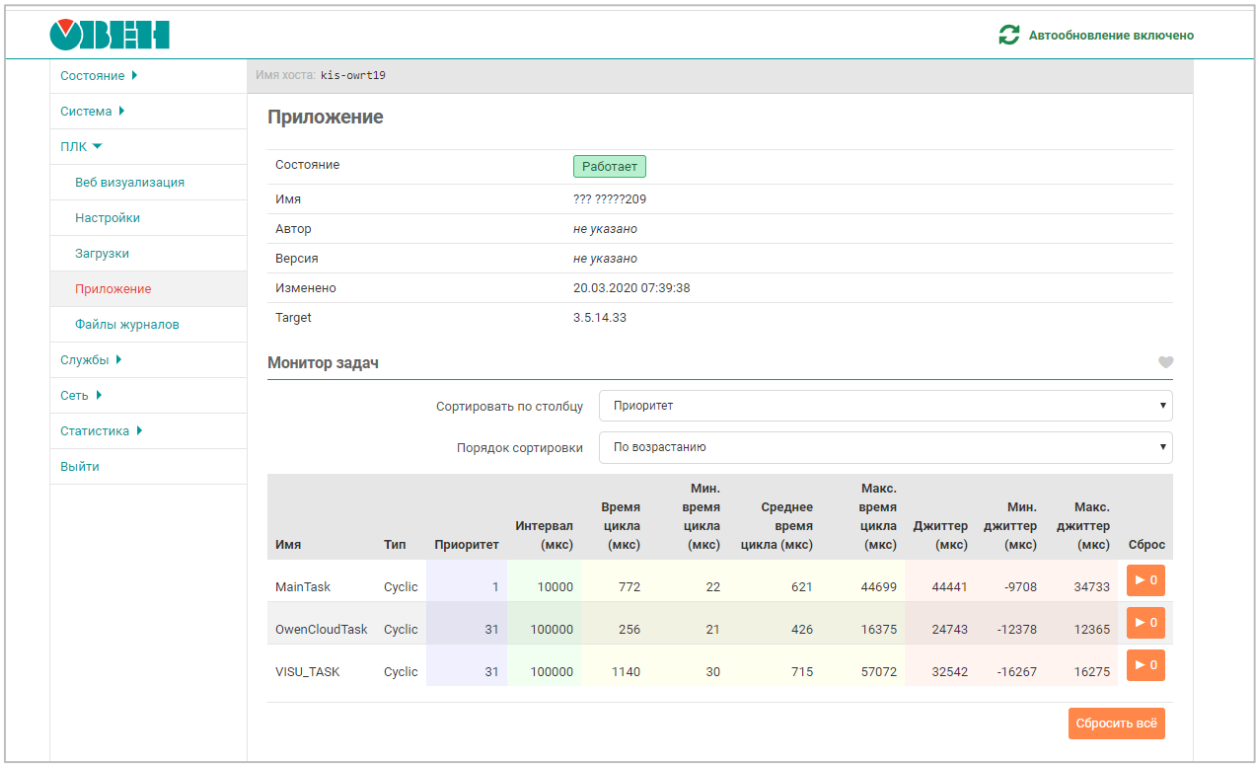


Рис. 3.6.2. Отображение информации мониторинга задач в web-конфигураторе для ПЛК ОВЕН

3.7. Индикация статусов компонентов в дереве проекта

При подключении к контроллеру в дереве проекта рядом с некоторыми узлами (например, узлами коммуникационных компонентов и дополнительных компонентов, разработанных производителями контроллера) отображаются пиктограммы, характеризующие их статус.

Пиктограмма	Описание для коммуникационных компонентов	Описание для остальных компонентов
	На запрос получен корректный ответ	Компонент работает корректно
	Отсутствует лицензия на компонент	
	Ожидание соединения/Отсутствие запросов от мастера	-
	На запрос получен ответ с кодом ошибки	Компонент работает некорректно (например, IP-адрес в компоненте Ethernet отличается от реального IP-адреса контроллера)
	Ответ не получен	Ошибка инициализации компонента (например, не удалось выделить память)

Ниже приведен скриншот для контроллера ОВЕН ПЛК210, на котором видны проблемы с системными узлами. Причина этого – использование некорректных версий таргет-файла и его компонентов. Сообщения о несоответствии версий также можно увидеть в журнале ПЛК. Пиктограмма компонента Ethernet означает, что выбранный в его настройках IP-адрес не соответствует реальному адресу интерфейса (это не мешает работе коммуникационных компонентов Modbus TCP, поэтому их пиктограммы «зеленые»).

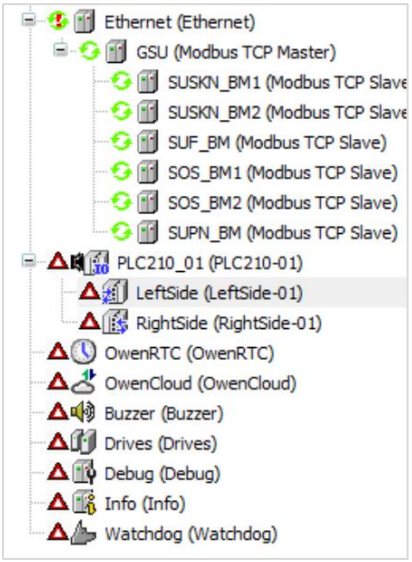


Рис. 3.7.1. Пиктограммы статуса компонентов в дереве проекта

3.8. Дерево и стек вызовов. Список перекрестных ссылок.

В процессе отладки сложных проектов (особенно, если их разрабатывал другой человек) требуется четко понимать структуру приложения – например, как связаны между собой вызовы POU (в таких проектах может быть много слоев абстракции, где POU вызывает POU, который вызывает POU, который… ну, вы поняли).

Помочь разобраться в этом помогают команды меню **Вид**. Для выполнения этих команд не требуется подключения к контроллеру.

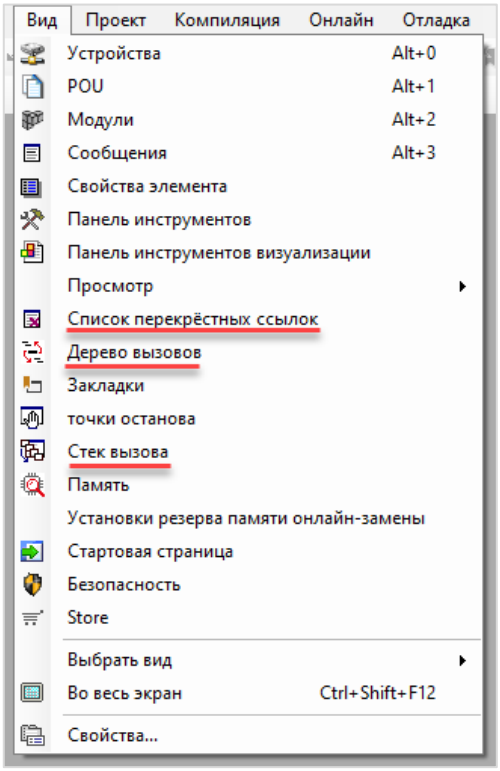


Рис. 3.8.1. Команды меню Вид, используемые для анализа проекта

Команда **Дерево вызовов** позволяет понять, как происходит вызов выбранного POU – от самого POU до задачи, в которой он выполняется. На рис. 3.8.2 приведен скриншот из пользовательского проекта, в котором происходит вызов экземпляра ФБ **OCL.MB_SerialRequest** с названием **MB_SerialRequest_16.** Вызов происходит в POU **WserBuk_2**, который в свою очередь вызывается в POU **W_Buk_1**, который вызывается в программе **PLC_PRG**, связанной с задачей **MainTask**. Понимание последовательности вызовов может быть важным для отладки приложения – вы можете пройти по дереву снизу вверх и проанализировать, на каком слое абстракции могли возникнуть проблемы, ставшие причиной ошибки. Для перехода к POU можно два раза нажать ЛКМ на его имя в дереве вызовов.

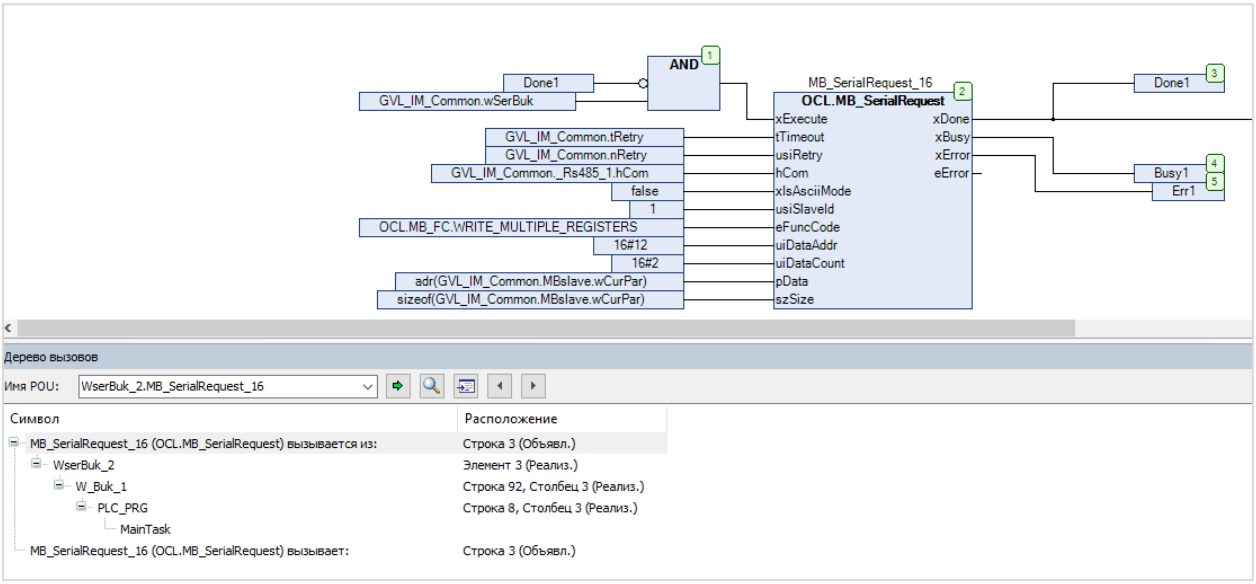


Рис. 3.8.2. Внешний вид панели дерева вызовов

Команда **Стек вызова** используется при онлайн-подключении к контроллеру, когда приложение остановлено в точке останова. Она позволяет понять, в каком именно экземпляре ФБ произошла остановка приложения и отображает полный путь до него.

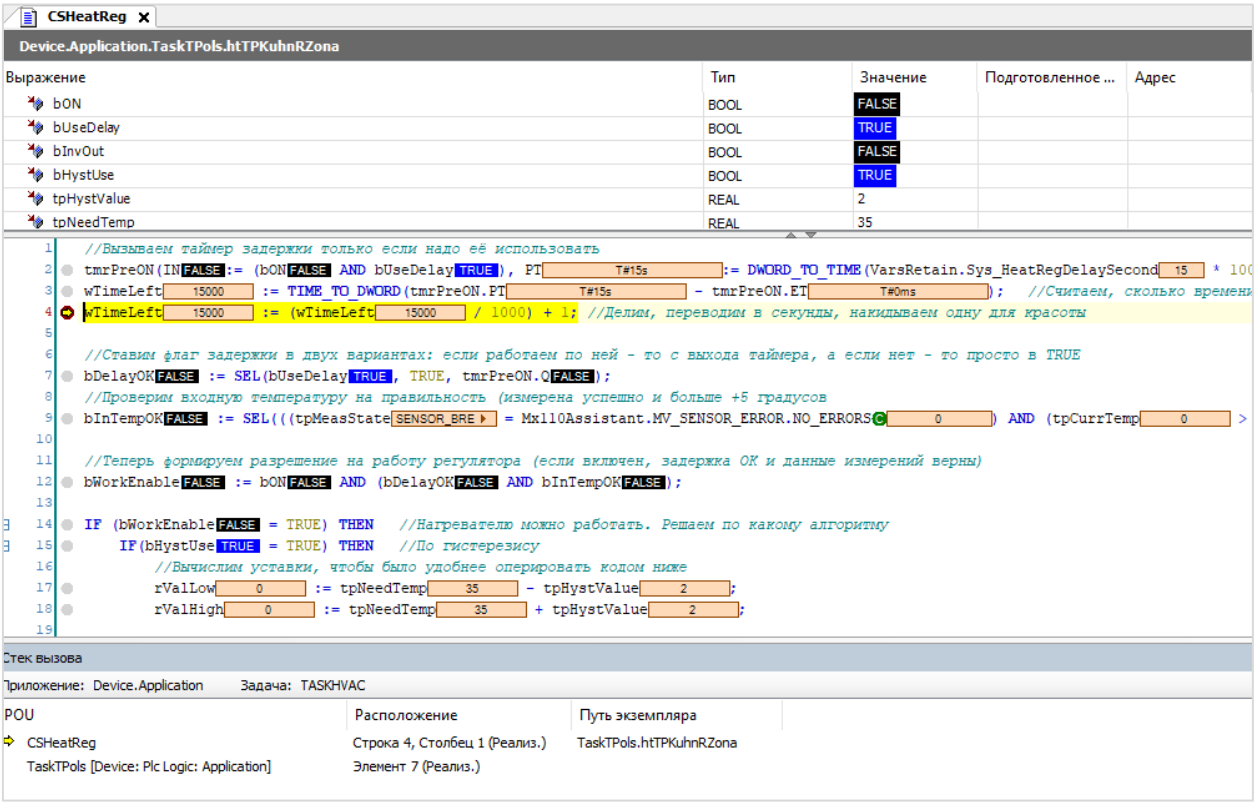


Рис. 3.8.3. Внешний вид панели стека вызова

Команда **Список перекрестных ссылок** не требует подключения к контроллеру. Она позволяет отследить все обращения к выбранной переменной/POU в рамках всего проекта. Это удобно в тех случаях, когда значение переменной меняется в разных фрагментах POU (а для глобальных переменных – даже в разных POU), чтобы, например, отследить, в каком именно фрагменте могла произойти запись некорректного значения.

На рисунке ниже приведен скриншот перекрестных ссылок для ФБ **CSDTrig**. Как можно увидеть – он имеет два экземпляра (**TrigMain** и **TrigStore**), которые объявлены в ФБ **CSIRelay**. Для каждого из экземпляров отображаются список их вызовов и обращений к входам-выходам. Для перехода к конкретному фрагменту нужно два раза нажать ЛКМ на соответствующую строку.

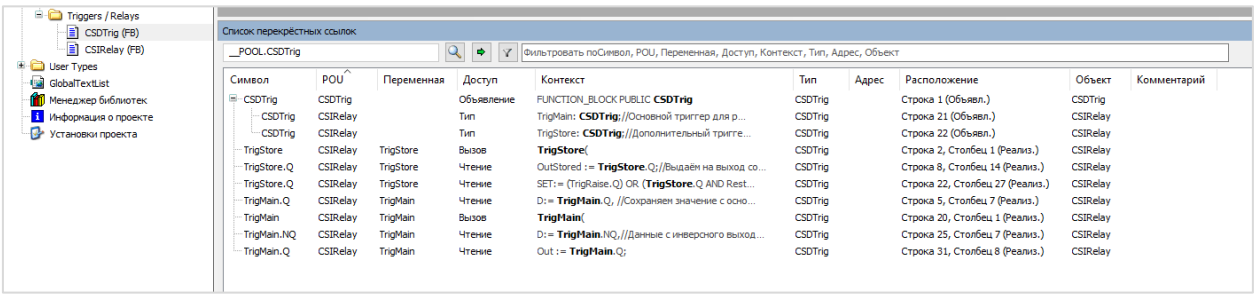


Рис. 3.8.4. Внешний вид панели перекрестных ссылок

3.9. POU для неявных проверок

POU неявных проверок используются для проверки проекта на типовые ошибки – что, безусловно, может быть полезно на этапе отладки. Для их добавления следует нажать ПКМ на узел **Application** и использовать команду **Добавление объекта – POU для неявных проверок**.

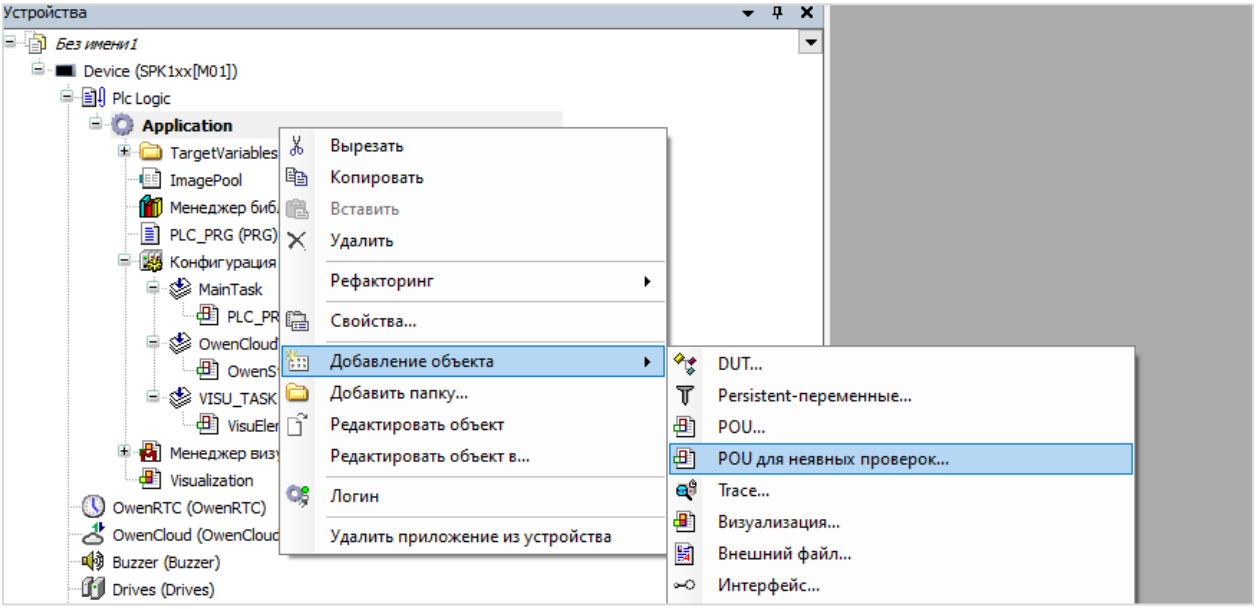


Рис. 3.9.1. Добавление в проект POU для неявных проверок

В появившемся окне следует выбрать нужные POU:

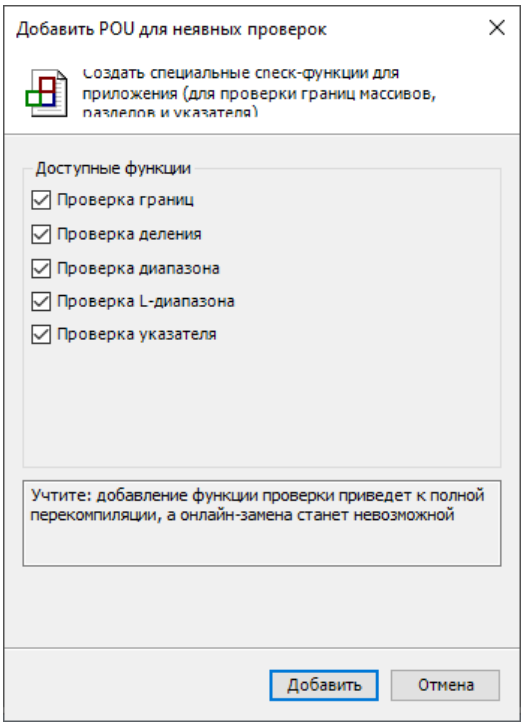


Рис. 3.9.2. Список доступных POU для неявных проверок

Пользователю доступны следующие POU, которые автоматически будут вызываться в соответствующих их условиям фрагментах кода:

- **Проверка границ** (CheckBounds) – проверка индекса при доступе к массиву. При выходе за границы значение индекса в данном вызове фиксируется на нижней или верхней границе массива;
- **Проверка деления** (CheckDivDInt, CheckDivLInt, CheckDivLReal, CheckDivReal) – проверка деления на 0. При детектировании такой операции – соответствующая строка кода не выполняется;
- **Проверка диапазона** (CheckRangeSigned, CheckRangeUnsigned) – проверка значения, записываемого в переменную с ограничением диапазона (subrange) для 32-битных типов. При выходе за диапазон значение переменной в данном вызове фиксируется на нижней или верхней границе диапазона;
- **Проверка L-диапазона** (CheckLRangeSigned, CheckLRangeUnsigned) – проверка значения, записываемого в переменную с ограничением диапазона (subrange) для 64-битных типов. При выходе за диапазон значение переменной в данном вызове фиксируется на нижней или верхней границе диапазона;
- **Проверка указателя** (CheckPointer) – проверка корректности указателя (например, на то, что он является ненулевым и указываем на доступную область памяти). Данная функция не имеет стандартной реализации – пользователь должен самостоятельно реализовать ее для своей конкретной платформы. В качестве альтернативы можно использовать функцию **SysMemIsValidPointer** из библиотеки **SysMem**.

Пользователь может доработать код POU для свой задачи (например, добавить вывод сообщений в журнал ПЛК при определенных условиях).

Как упоминалось выше – POU для неявных проверок автоматически вызываются системой исполнения в нужные моменты времени, так что пользователю не нужно вызывать их в своем коде. Необходимо понимать, что эти POU нельзя использовать в качестве «костыля» для решения проблем в ПО – хотя при их добавлении ошибка может перестать проявляться, разработчик обязан разобраться в причинах ее возникновения и устранить их. Скорее всего, ввод этих проверок не решит проблему, а просто замаскирует ее или уменьшит частоту воспроизведения. Помните о том, что отлаживать ПО на столе всегда проще, чем на реальном объекте с остановившейся производственной линией под шум станков и крики паникующих людей.

Основной смысл использования этих POU – указать на вероятную причину ошибки. Например, если проблема перестала появляться после добавления проверки границ – то, вероятно, она связана с некорректной работой с массивами – и теперь нужно проверить, как осуществляется доступ ко всем массивам проекта.

Поскольку использование POU для неявных проверок увеличивает время цикла ПЛК – то рекомендуется использовать их только на этапе отладки и отключить перед запуском приложения на реальном объекте.

См. также видеопример.

3.10. Дамп памяти

Дамп памяти – это «слепок» текущего состояния приложения, остановленного в точке останова или из-за возникновения исключения. Он может помочь определить причину ошибки при удаленной отладке проекта, когда у разработчика нет доступа к контроллеру – но на объекте присутствует инженер, который может сохранить и прислать ему дамп памяти.

Дамп памяти представляет собой файл с названием

<имя_проекта>.<имя_устройства>.<имя_приложения>.<идентификатор>.core

Дамп памяти может быть создан двумя способами:

- автоматически при возникновении в проекте исключения – в этом случае дамп сохраняется в памяти ПЛК в папке **/PlcLogic/Application**. Контроллер должен поддерживать функцию автоматического создания дампа – она доступна не для всех ПЛК;
- вручную при подключении к контроллеру с помощью выполнения команды **Отладка – Дамп памяти – Создать дамп памяти**. В этом случае дамп будет сохранен на ПК в директории файла проекта. Требуется переслать разработчику файлы формата .core и .compileinfo из этой папки (и, при необходимости, файл проекта – если у разработчика он отсутствует).

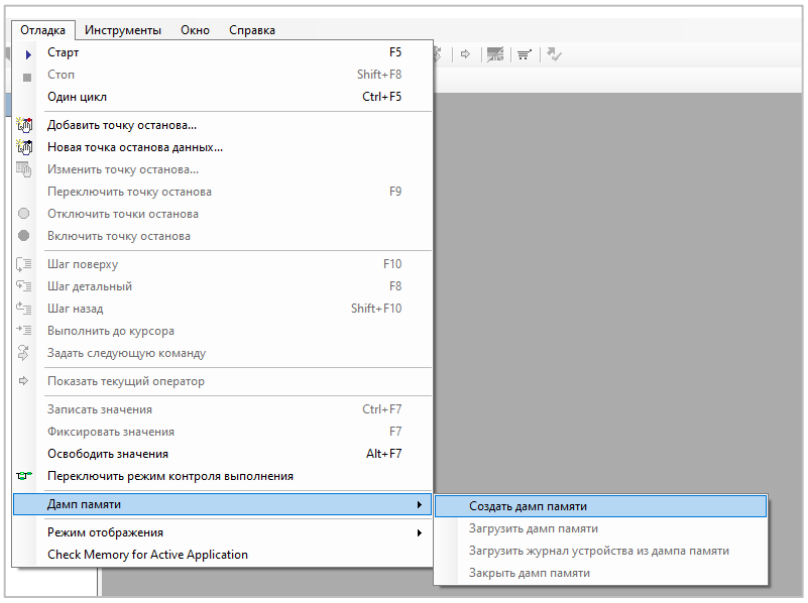


Рис. 3.10.1. Команда создания дампа памяти

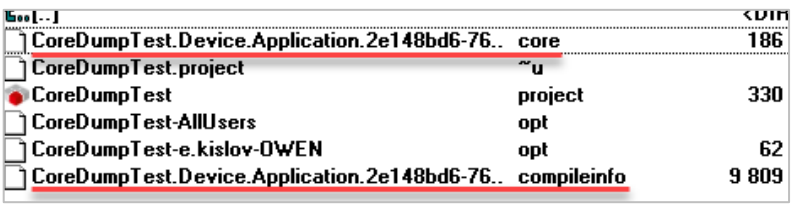


Рис. 3.10.2. Файлы, необходимые для открытия дампа памяти

При наличии проекта (обязательно нужна именно та версия проекта, в которой был создан дамп) и этих файлов можно использовать команду **Отладка – Дамп памяти – Загрузить дамп памяти**. После этого в среде CODESYS будут отображаться значения переменных на момент снятия дампа. Для доступа к журналу ПЛК в момент снятия дампа используется команда **Отладка – Дамп памяти – Загрузить журнал устройства из дампа памяти**. Для прекращения работы с дампом следует использовать команду **Закрыть дамп памяти**.

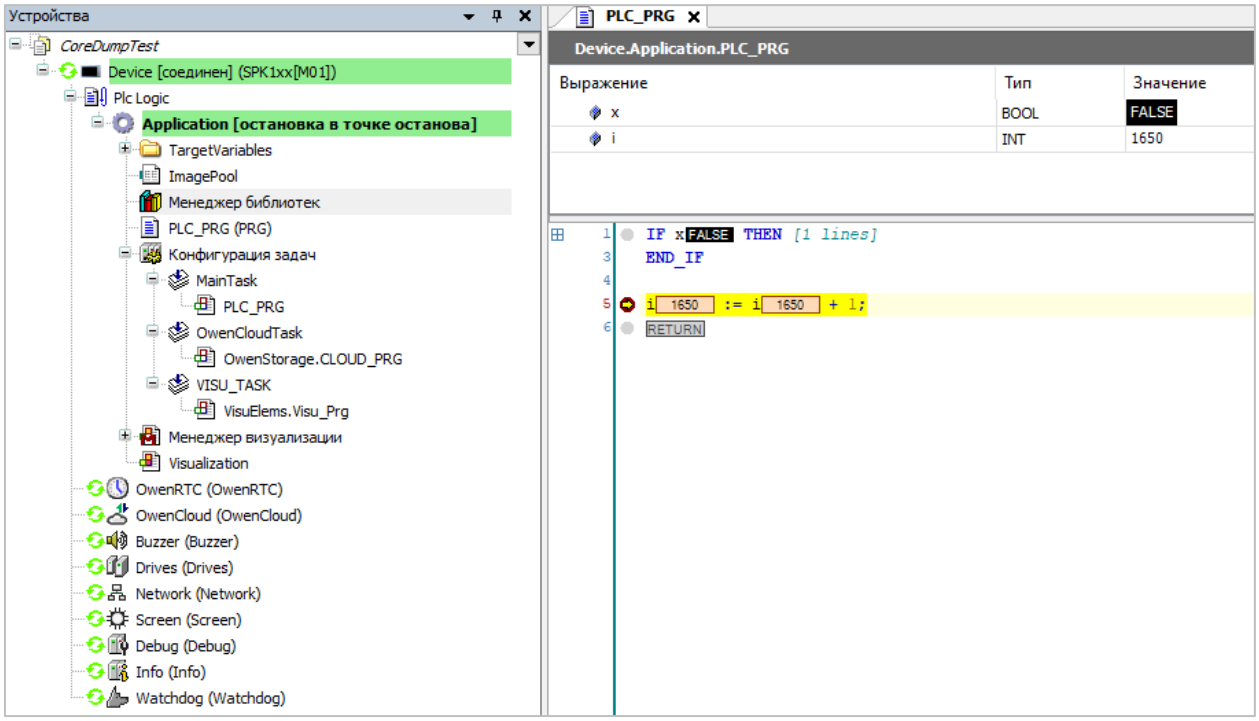


Рис. 3.10.3. Работа с дампом памяти в среде CODESYS

Важно отметить, что при открытии дампа памяти в директории проекта не должно быть дополнительных файлов компиляции, иначе возникнет следующая ошибка:

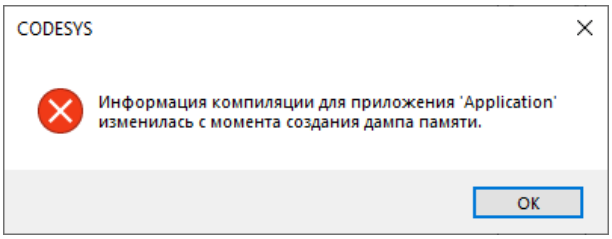


Рис. 3.10.4. Ошибка открытия дампа памяти

Поэтому рекомендуется скопировать файлы .project, .core и .compileinfo в новую папку и после открытия проекта в CODESYS сразу загрузить дамп памяти, не выполняя перед этим команды **Компиляция**, **Перекомпиляция** или **Генерировать код.**

3.11. Отладка проекта на виртуальном контроллере

Виртуальный контроллер **CODESYS Control Win V3** представляет собой полнофункциональную систему исполнения для ОС Windows (SoftPLC). Значительная часть пользовательского приложения может быть отлажена на нём, за исключением:

- функционала некоторых дополнительных компонентов и библиотек, предоставляемых производителем контроллера;
- функционала, связанного с особенностями ОС (например, если вы делаете проект под ПЛК с ОС Linux, то в ряде моментов выполнение проекта на ПЛК и виртуальном контроллера может отличаться).

Если ошибка повторяется на виртуальном контроллере, то это дает вам ценную информацию – она не является аппаратной, не связана с прошивкой ПЛК и факторами внешней среды. Таким образом, проблема либо в вашем проекте, либо в системе исполнения CODESYS.

Для возможности запуска проекта на виртуальном контроллере следует поменять таргет-файл в проекте на **CODESYS Control Win V3** (ПКМ на **Device – Обновить устройство**). Если на вашем ПЛК 64-битная система исполнения, то следует использовать CODESYS Control Win V3 x64 (он входит в дистрибутив 64-битной среды программирования). Версия таргет-файла должна соответствовать версии среды программирования и версии системы исполнения виртуального контроллера, установленной на вашем ПК.

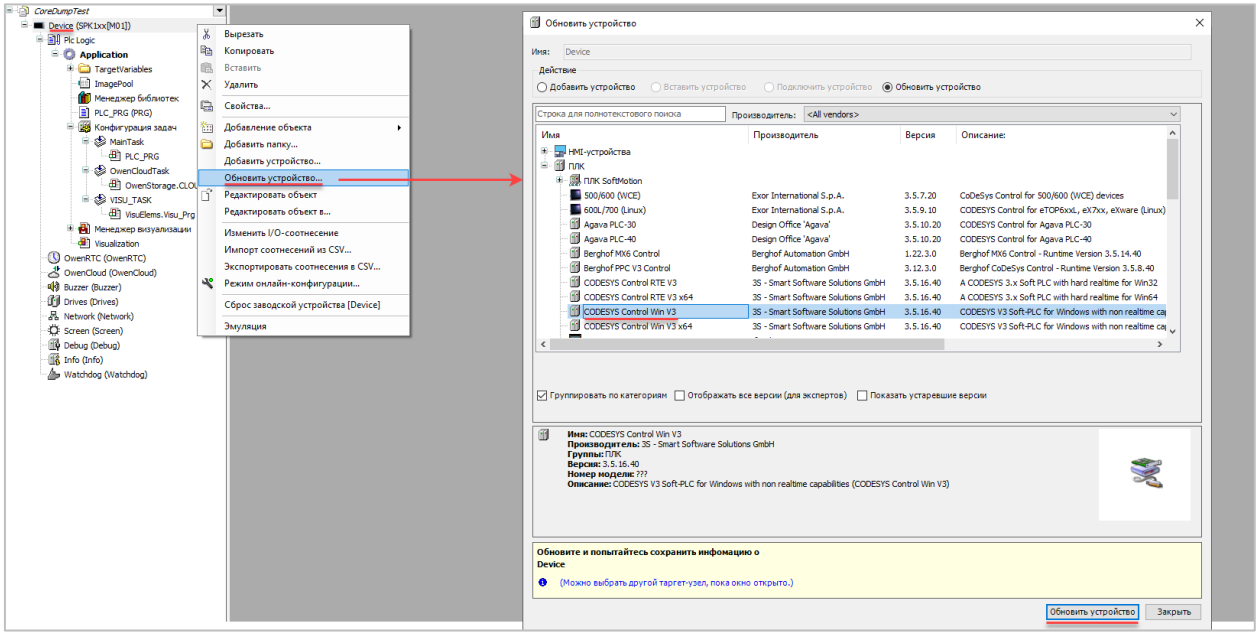


Рис. 3.11.1. Выбор таргет-файла виртуального контроллера

Для запуска виртуального контроллера нужно нажать ПКМ на его иконку в трее Windows и выполнить команду **Start PLC** (*а в окне About можно посмотреть версию системы исполнения*) или запустить его из меню **Пуск** (в этом случае можно запустить несколько экземпляров виртуального контроллера одновременно).

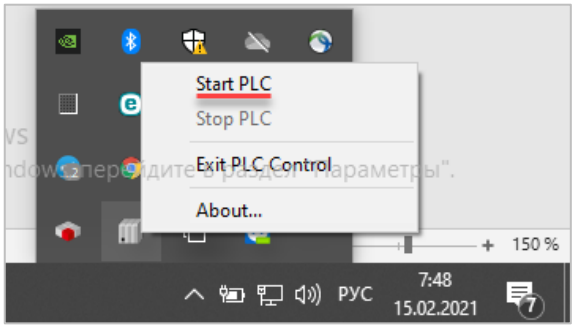


Рис. 3.11.2. Запуск виртуального контроллера из трея Windows

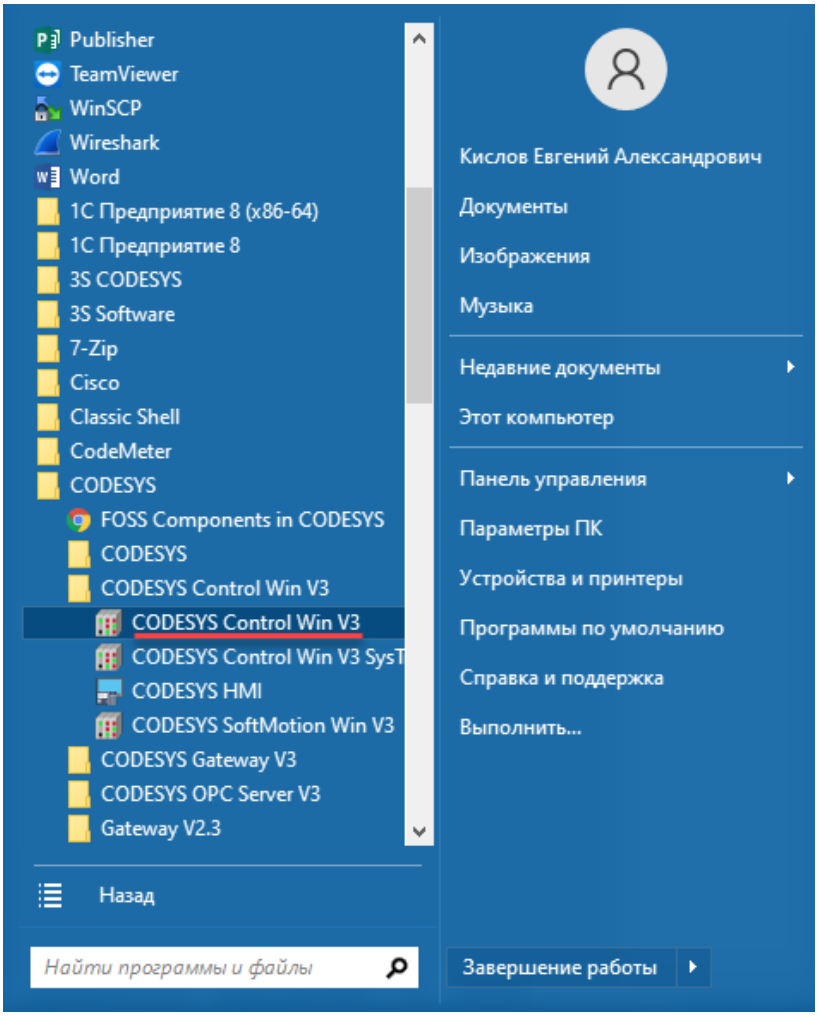


Рис. 3.11.3 Запуск виртуального контроллера из меню **Пуск**

После этого при сканировании сети в CODESYS будет найден контроллер, чье имя совпадает с именем вашего ПК.

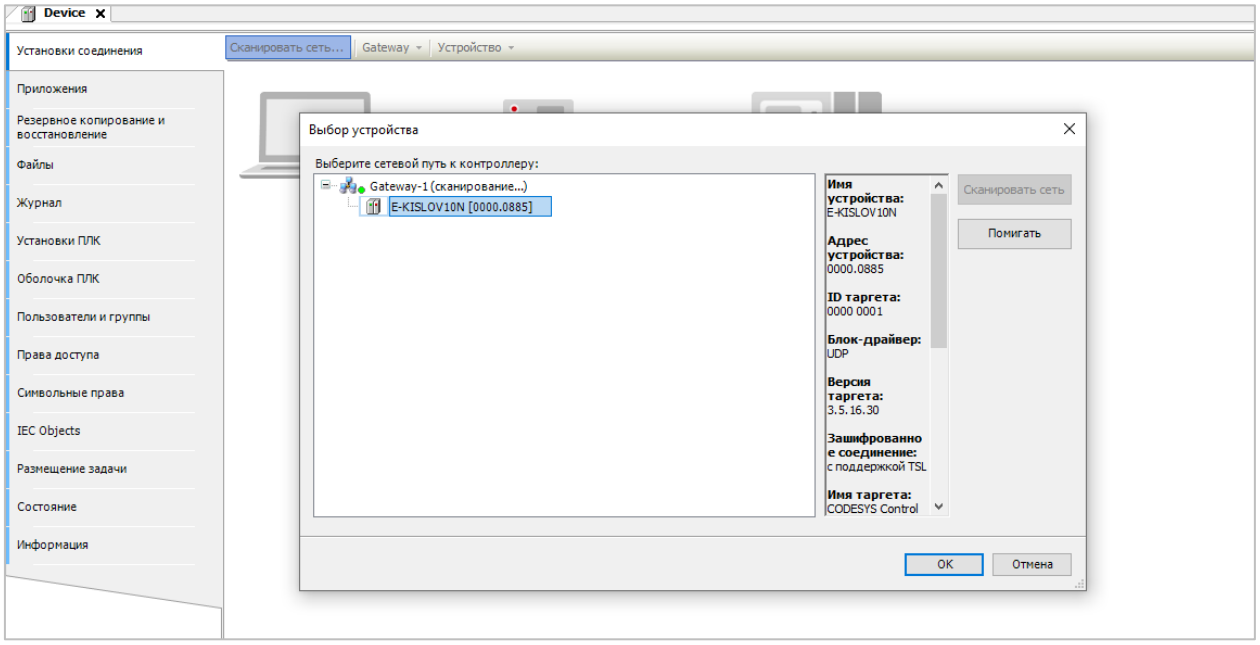


Рис. 3.11.4. Подключение к виртуальному контроллеру

Для загрузки проекта в виртуальный контроллер может потребоваться его адаптация (например, отключение компонентов, разработанных производителем ПЛК).

Виртуальный контроллер является полнофункциональным (*в отличие от симулятора, запускаемого командой* ***Онлайн – Эмуляция****; поэтому не рекомендуется использовать симулятор для отладки, как как это не является показательным*) – он поддерживает web-визуализацию, работу с файлами, коммуникационные драйвера и т.д.

В состав дистрибутива CODESYS входит trial-версия виртуального контроллера, ограниченная двумя часами непрерывной работы (после чего ее можно перезапустить). В случае необходимости можно приобрести лицензию в CODESYS Store.

Также в свежих версиях CODESYS появилась возможность использования виртуального контроллера для ОС Linux.

3.12. Предупреждения компилятора

В процессе разработки проекта программист регулярно сталкивается с ошибками компиляции – например, при ошибках в именах переменных (из-за чего в коде может появиться имя необъявленной переменной), несоответствии типов данных, ошибках в синтаксисе и т.д. Наличие в проекте таких ошибок не позволит загрузить его в контроллер – так что об отладке в этот момент говорить еще не приходится. Обычно исправление ошибок, обнаруженных компилятором, тривиально и не занимает много времени.

Помимо сообщений об ошибках компилятор также выдает предупреждения. Они не мешают загрузке проекта в ПЛК, и поэтому зачастую игнорируются разработчиками. Но при появлении каких-то ошибок в работе проекта стоит вернуться к списку предупреждений и тщательно его изучить – возможно, в нем скрываются какие-то подсказки по поводу происходящей ситуации.

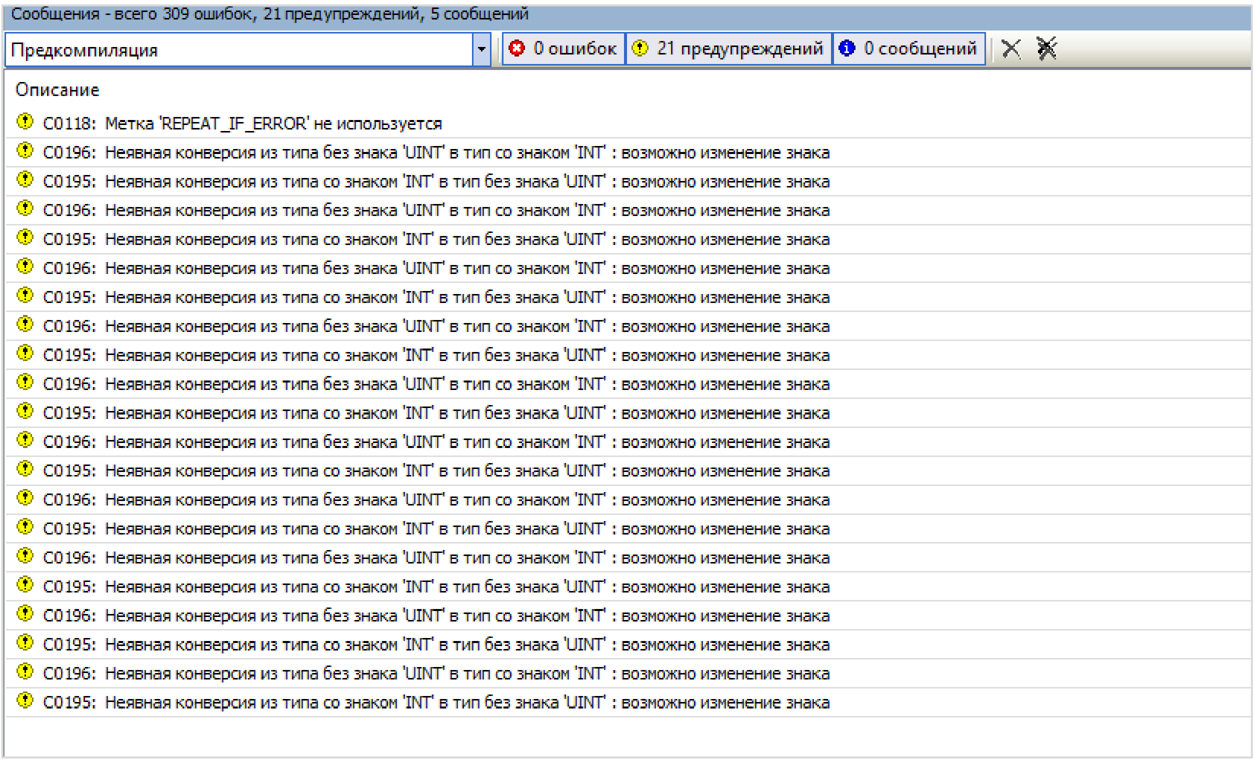


Рис. 3.12.1. Список предупреждений компилятора

Кроме того, для CODESYS доступен статический анализатор. Он является дополнительным платным компонентом, но его урезанная версия (всего с несколькими доступными проверками) встроена в дистрибутив и может использоваться без ограничений. Полная версия включает в себя более полутора сотен проверок. Статический анализатор позволяет повысить качество кода за счет дополнительных проверок и избежать типичных ошибок.

Для включения проверок урезанной версии следует открыть **Установки проекта**, выбрать вкладку **Static Analysis Light** и пометить галочками нужные проверки. Выполнение проверок происходит во время генерации кода (**Компиляция – Генерация кода**). Найденные несоответствия считаются ошибками компиляции и поэтому без их устранения нельзя будет загрузить проект в контроллер.

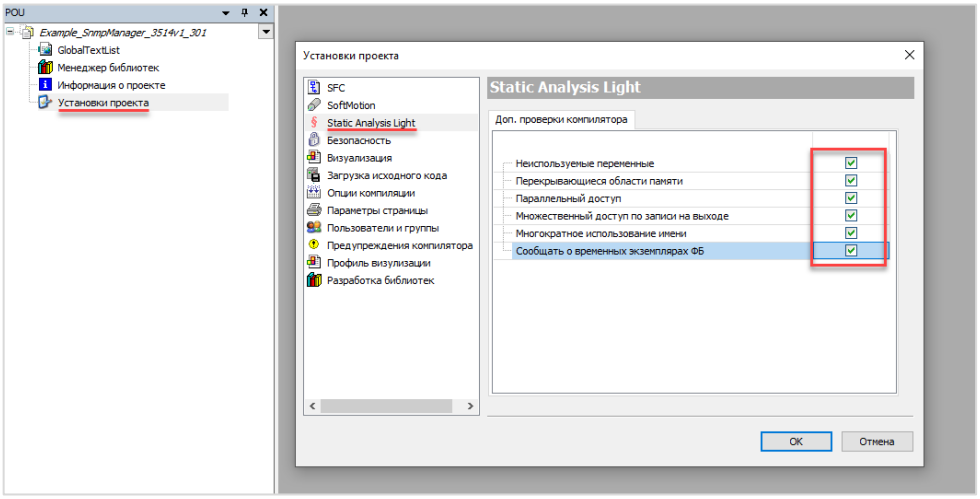
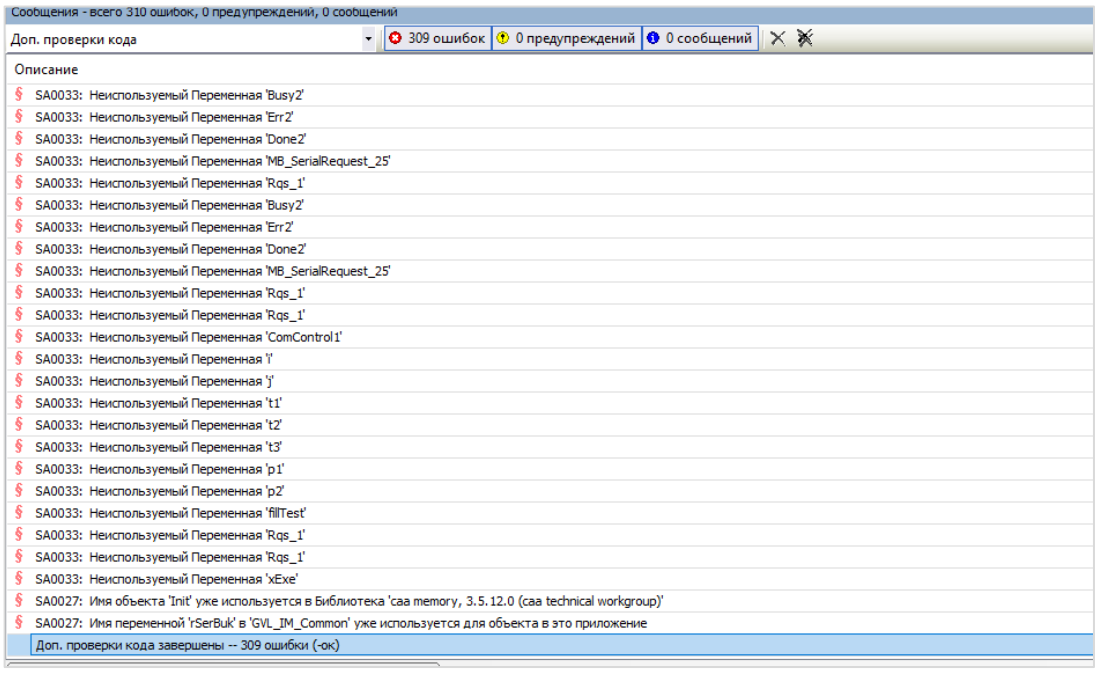


Рис. 3.12.2. Настройки статического анализатора

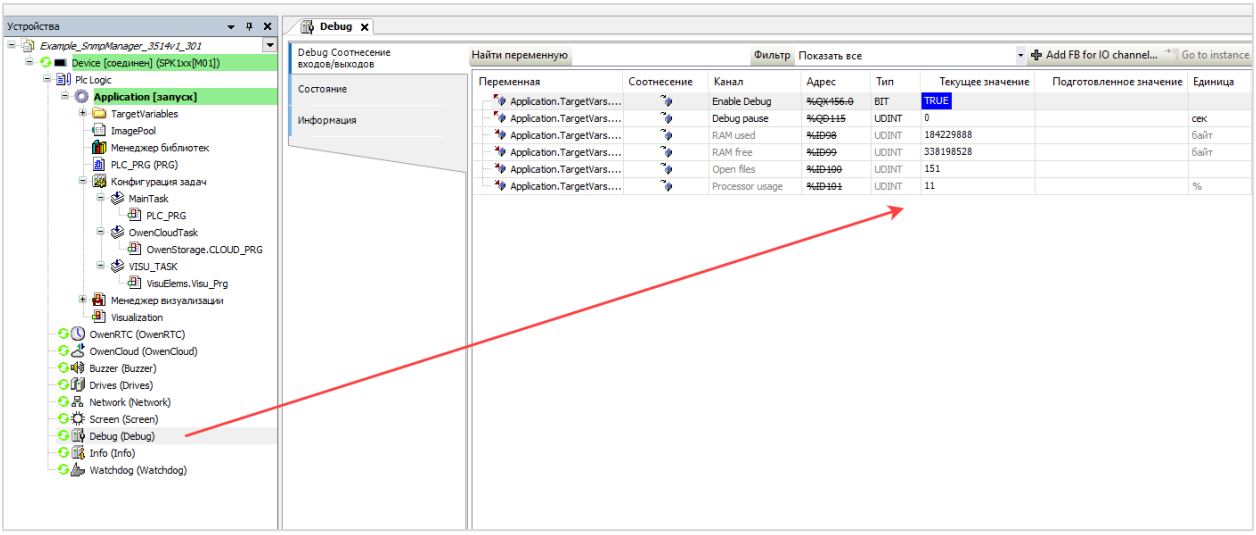


3.12.3. Список сообщений статического анализатора

3.13. Узел Debug (контроллеры ОВЕН)

В таргет-файлах контроллеров ОВЕН доступен специальный узел **Debug**, который позволяет считать в переменные программы отладочную информацию – объем занятой и свободной оперативной памяти, загрузку процессора и число используемых дескрипторов (хэндлов). Анализ этих значений в процессе отладки может помочь определить утечки памяти, аномальную нагрузку на CPU и некорректную работу с хэндлами (например, постоянное открытие новых файлов без закрытия текущих используемых).

Производители других контроллеров могут предоставлять специальные компоненты и библиотеки, выполняющие схожую функцию.



3.12.1. Каналы узла **Debug** в контроллерах ОВЕН

4. Типовые ошибки

Мы регулярно наблюдаем в проектах (как своих, так и чужих) одни и те же ошибки. Они настолько типичны, что обычно опытному разработчику достаточно краткого описания наблюдаемой проблемы, чтобы выдвинуть гипотезу о ее причинах. В этом разделе мы рассмотрим такие типовые ошибки.

4.1. Деление на 0

Большинство ПЛК при делении на 0 формируют исключение, которое приводит к остановке выполнения проекта. Поэтому если в коде происходит деление на переменную, то разработчик должен обеспечить валидацию значения этой переменной.

Рассмотрим следующей пример: пользователь через визуализацию вводит «сырое» значение уставки **uiRawSetpoint** (это значение может попадать в контроллер и другим образом – например, считываться по Modbus, вычитываться из файла рецепта и т.д.), которое в программе делится на коэффициент приведения **uiScaleCoeff** (этот коэффициент может также задаваться через визуализацию, вычисляться в программе ПЛК или формироваться иным способом).

PLC_PRG

```
PROGRAM PLC_PRG
VAR
    xSetSettings:       BOOL;

    uiRawSetpoint:      UINT;
    uiScaleSetpoint:    UINT := 1000;
    uiScaleCoeff:       UINT;
END_VAR
```

```
// ...какой-то код

IF xSetSettings THEN
    uiScaleSetpoint := uiRawSetpoint / uiScaleCoeff;
    // ...какой-то код
END_IF
```

Рис. 4.1.1. Пример кода с потенциальным делением на 0

Если коэффициент приведения по каким-то причинам равен нулю (например, оператор забыл его ввести), то в программе произойдет деление на 0. В журнале ПЛК будет отображаться информация об исключении с характерным описанием **DivisionByZero**. При двойном нажатии на строку SOURCEPOSITION будет открыт фрагмент кода, в котором произошло деление на 0.

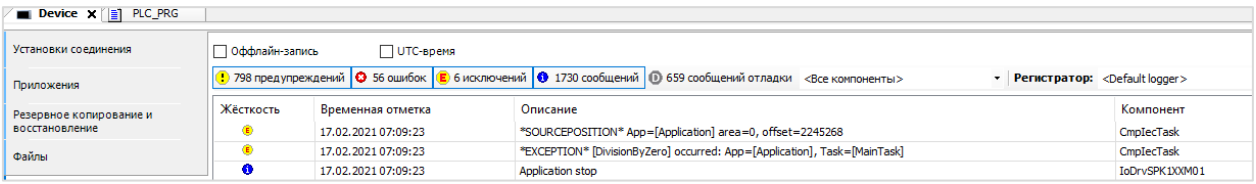


Рис. 4.1.2. Сообщения об исключении при делении на 0 в журнале ПЛК

Избежать деления на 0 можно следующими способами:

- если значение делителя вводится через визуализацию – то можно ограничить диапазон доступных для ввода значений в настройках элемента (рис. 4.1.3);
- перед процедурой деления проверить значение делителя, и если оно равно 0 – не выполнять операцию деления, а, например, вывести в визуализацию сообщение о некорректном значение коэффициента (см. рис. 4.1.4);
- в некоторых случаях можно заменить процедуру деления на процедуру умножения (т.е. оператор вместо коэффициента 100 будет вводить коэффициент 0.01; это потребует перехода к типам данных с плавающей точкой);
- в ряде конкретных случаев (например, в специфических средах, где не поддерживаются условные операторы) можно заменить процедуру деления на процедуру умножения с возведением исходного значения коэффициента в степень -1. В этом случае оператор будет вводить коэффициент привычным ему образом, но разработчик должен убедиться, что в его системе возведения 0 в степень -1 не приводит к исключению (например, в CODESYS V3.5 при использовании стандартного оператора **EXPT** с аргументами (0, -1) будет получен 0 (см. рис. 4.1.4).

Автоматическая проверка деления на 0 возможна с помощью использования POU для неявных проверок.

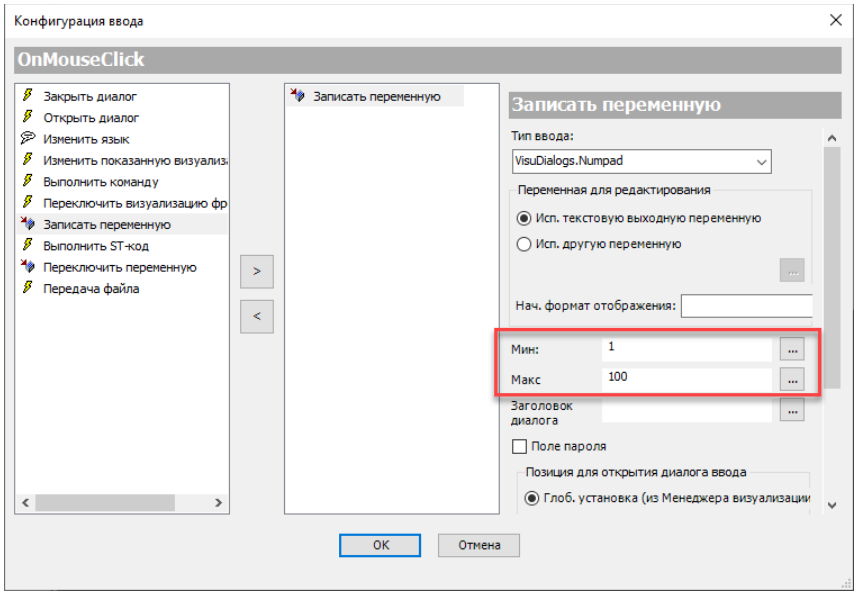


Рис. 4.1.3. Ограничение вводимых в визуализации значений

PLC_PRG

```
PROGRAM PLC_PRG
VAR
    xSetSettings:       BOOL;

    uiRawSetpoint:      UINT;
    uiScaleSetpoint:    UINT := 1000;
    uiScaleCoeff:       UINT;
END_VAR
```

```

// ...какой-то код

IF xSetSettings THEN

    IF uiScaleCoeff <> 0 THEN
        uiScaleSetpoint := uiRawSetpoint / uiScaleCoeff;
    ELSE
        // вывести на визуализацию сообщение об ошибке
    END_IF
    // ...какой-то код
END_IF
```

Рис. 4.1.4. Проверка делителя на неравенство нулю

PLC_PRG

```
PROGRAM PLC_PRG
VAR
    xSetSettings:       BOOL;

    uiRawSetpoint:      UINT;
    uiScaleSetpoint:    UINT := 1000;
    uiScaleCoeff:       UINT;
END_VAR
```

```

// ...какой-то код

IF xSetSettings THEN
    uiScaleSetpoint := uiRawSetpoint * TO_UINT(EXPT(uiScaleCoeff, -1) );
    // ...какой-то код
END_IF
```

Рис. 4.1.5. Рефакторинг кода с целью избежать деления

4.2. Бесконечный цикл

Бесконечный цикл – это цикл, условие выхода из которого никогда не выполняется. В большинстве ПЛК система исполнения обеспечивает основной бесконечный цикл контроллера (*так называемый «суперцикл»; в среде CODESYS V3.5 этому понятию соответствует задача с типом вызова «циклический»*), в котором выполняется пользовательский код. Обычно период вызова задачи определяется разработчиком. Из-за наличия суперцикла практически невозможно представить ситуацию, когда программисту потребовалось бы организовывать свой бесконечный цикл – почти всегда это является ошибкой.

Часто такие ошибки связаны с неправильным пониманием принципа выполнения цикла в ПЛК. Для большинства контроллеров операторы циклов выполняются синхронно по отношению к задаче ПЛК. Обработка задач производится следующим образом: перед новым вызовом задачи происходит чтение связанных с ней входных переменных (чтение значений физических входов ПЛК, копирование данных из приемного буфера коммуникационного интерфейса и т.д.), затем выполняется код задачи, и после этого происходит запись выходных переменных. Таким образом, на время выполнения кода задачи значения ее входных переменных «замораживаются». Если программист рассчитывает, что на условие выхода из цикла могут повлиять какие-то «внешние» данные (например, нажатие на кнопку на экране визуализации или передача команды от другого устройства) – то его ждет жестокое разочарование.

В качестве примера приведем следующий код.

Device | PLC_PRG

```
PROGRAM PLC_PRG
VAR
    xEnable:    BOOL;
    i, j:       INT;
END_VAR
```

```
WHILE xEnable DO

    // какие-то операции...
    i := i + 1;

END_WHILE

j := j + 1;
```

Рис. 4.2.1. Пример бесконечного цикла

Предположим, в программе нужно выполнить определенные операции заранее неизвестное число раз (*например, оператор нажимает кнопку, и пресс должен двигаться до тех пор, пока он ее не отпустит*). В данном случае программист может выбрать цикл WHILE, руководствуясь следующий логикой – пока значение кнопки (переменная **xEnable**) имеет значение **TRUE**, то операции будут выполняться. При отпускании кнопки переменная получит значение **FALSE**, что приведет к выходу из цикла.

Но фактически всё будет происходить иначе (по крайней мере, в CODESYS V3.5). После того, как переменная **xEnable** получит значение **TRUE**, контроллер начнет выполнять тело цикла WHILE. При этом перехода к следующему циклу задачи не произойдет до выхода из цикла WHILE, а выхода из цикла не произойдет, так как для этого нужно считать новое значение переменной **xEnable**, что может произойти только в следующем цикле задачи.

В результате в тот момент, когда переменная **xEnable** примет значение **TRUE**, инкремент переменной **j** перестанет выполняться (приложение «зависло» в бесконечном цикле), а спустя некоторое непродолжительное время (зависящее от настроек системы исполнения) сторожевой таймер ПЛК детектирует бесконечный цикл и сгенерирует исключение. Сообщения, выведенные при этом в журнал ПЛК, мы рассматривали в п. 3.3.

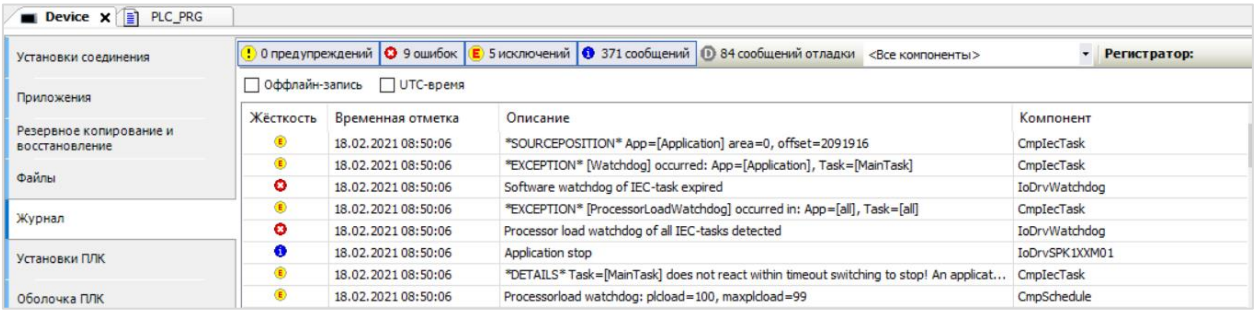


Рис. 4.2.2. Сообщения об исключении из-за бесконечного цикла в журнале ПЛК

Решить проблему очень просто – достаточно вспомнить, что система исполнения сама организовывает суперцикл контроллера, и заменить цикл WHILE на условный оператор IF. В этом случае, если переменная **xEnable** имеет значение **TRUE**, то каждый цикл задачи будет однократно выполнено тело оператора IF. Инкремент переменной **j** будет выполнен в каждом цикле задачи независимо от каких-либо условий. Таким образом, вероятность «зависания» в данном случае исключена.

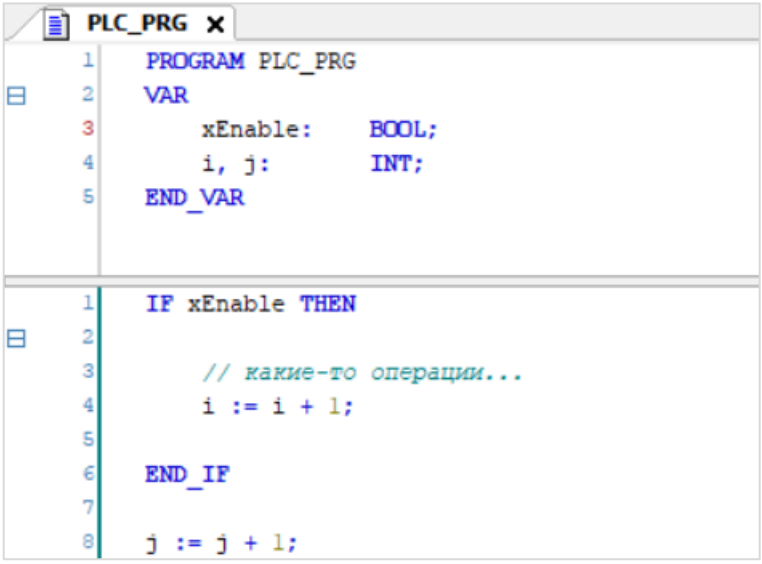


Рис. 4.2.3. Исправленный код

По нашему опыту – необходимость использования операторов WHILE/REPEAT при программировании контроллеров возникает в крайне ограниченном числе случаев.

Еще один пример случайной организации бесконечного цикла – некорректное использование типа переменной для счетчика цикла FOR. Эту ситуацию рассматривает **Игорь Петров** в своей книге **Программируемые контроллеры. Стандартные языки и приемы прикладного проектирования**.

Предположим, разработчику потребовалось написать код инициализации массива из 256 элементов (с индексами **0…255**) некоторым начальным значением. Для решения своей задачи он использовал цикл FOR, а в качестве счетчика цикла – переменную типа **USINT**, которая может принимать значения в диапазоне **0…255**. С точки зрения разработчика всё выглядит логичным – тип переменной соответствует размерности массива. Компилятор CODESYS V3.5 при этом сформирует предупреждение о возможном наличии в коде бесконечного цикла – и если программист его проигнорирует, то при запуске программы и присвоении переменной **xInit** значения **TRUE** получит те же сообщения в журнале ПЛК, что приведены на рис. 4.2.2.

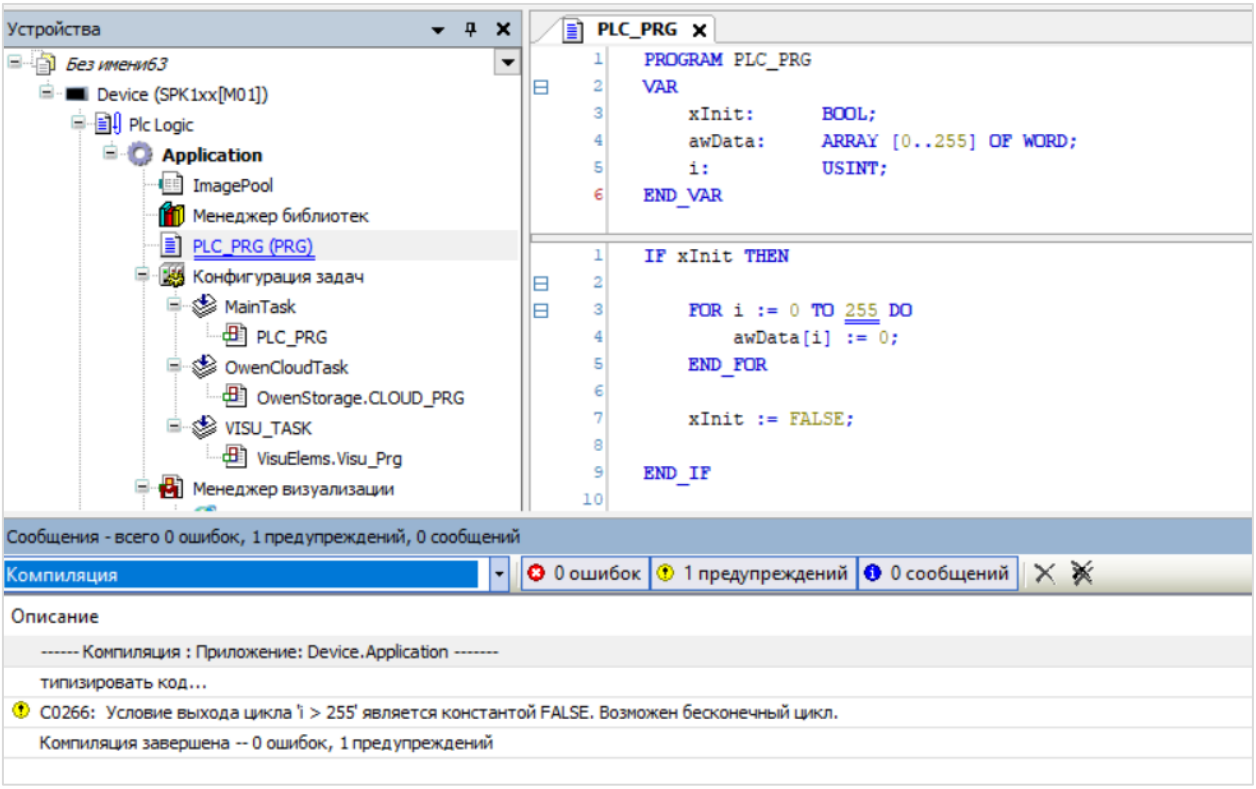


Рис. 4.2.4. Предупреждение компилятора о возможности бесконечного цикла

Это связано с тем, что в стандарте МЭК 61131-3 есть следующая фраза: «*The value of the control variable after completion of the FOR loop is implementation-dependent*» – то есть значение счетчика цикла после выхода из цикла зависит от реализации системы исполнения конкретного ПЛК. В CODESYS V3.5 после выхода из цикла FOR счетчик инкрементируется еще раз – в результате чего происходит переполнение переменной типа **USIN**T (255 + 1 = 0), и цикл начинается заново (так как условие выхода из цикла – это «i > 255», а для типа **USINT** оно никогда не может стать истинным). Для решения проблемы достаточно использовать для переменной-счетчика тип INT или UINT.

Внимательное отношение к циклам может избавить разработчика от множества потенциальных проблем в будущем.

4.3. Ошибка сегментации памяти (access violation)

Ошибка сегментации (англ. Segmentation fault или сокращённо segfault) — это ошибка программного обеспечения, возникающая при попытке обращения к недоступным для записи участкам памяти либо при попытке изменения памяти запрещённым способом. Часто такая ошибка сопровождается сообщением «access violation».

В проектах для CODESYS V3.5 мы наблюдали две основных причины таких ошибок – выход за пределы массива и некорректная работа с указателями (*также она может возникнуть при использовании динамического выделения памяти – впрочем, без указателей тогда тоже не обойдется*).

На рис. 4.3.1. приведен типичный пример ошибки при работе с массивами – массив имеет размерность **1..9**, а в цикле FOR граничное значение счетчика равно **10**. Соответственно, на последней итерации цикла произойдет обращение к **aiData[10]** (*несуществующему элементу массива*), в результате чего будет произведена запись в область памяти за пределами массива. Для ПЛК, который мы использовали для теста, компилятор разместил в следующей за массивом ячейке памяти переменную **xSelfDestruction** – и, соответственно, она приняла значение **TRUE** (*чего вряд ли бы мог ожидать условный разработчик этого кода*).

PLC_PRG

```
PROGRAM PLC_PRG
VAR
    i:                  INT;
    aiData:             ARRAY [1..9] OF INT;
    // Кнопка самоуничтожения. Осторожно!
    xSelfDestruction:   BOOL;
    // Команда инициализации
    xInit:              BOOL;
END_VAR
```

```
IF xInit THEN

    FOR i := 1 TO 10 DO
        aiData[i] := 1;
    END_FOR

    xInit := FALSE;

END_IF
```

Рис. 4.3.1. Ошибка записи в несуществующий элемент массива

В данном случае ошибки можно было бы избежать с помощью использования в качестве границ массива и начального/конечного значения счетчика цикла констант.

```
PLC_PRG
PROGRAM PLC_PRG
VAR
    i:                  INT;
    aiData:             ARRAY [c_iFirstElem..c_iLastElem] OF INT;
    // Кнопка самоуничтожения. Осторожно!
    xSelfDestruction:   BOOL;
    // Команда инициализации
    xInit:              BOOL;
END_VAR
VAR CONSTANT
    c_iFirstElem:       INT := 1;
    c_iLastElem:        INT := 9;
END_VAR

IF xInit THEN

    FOR i := c_iFirstElem TO c_iLastElem DO
        aiData[i] := 1;
    END_FOR

    xInit := FALSE;

END_IF
```

Рис. 4.3.2. Рефакторинг кода с использованием констант

Автоматическая проверка выхода за границы массива возможна с помощью использования POU для неявных проверок.

Использование указателей (POINTER) требует от разработчика соответствующей квалификации и внимательности. Ниже приведен пример ошибки, которая является как раз результатом невнимательности – вместо указателя на переменную **dwValue** в функцию **MemMove** передается само значение переменной. Так как указатели в CODESYS на 32-битных платформах хранятся как значения типа DWORD – то компилятор в этом случае не выдаст сообщений об ошибке (даже предупреждения не выдаст!).

```
PLC_PRG
PROGRAM PLC_PRG
VAR
    xInit:   BOOL;
    dwValue: DWORD := 16#DEADCAFE;
    rValue:  REAL;
END_VAR

IF xInit THEN

    MEM.MemMove(dwValue, ADR(rValue), SIZEOF(rValue) );
    xInit := FALSE;
END_IF
```

Рис. 4.3.3. Пример некорректной работы с указателями

При вызове этого кода произойдет обращение к адресу памяти 16#DEADCAFE. По этому адресу может храниться что угодно – например, какие-то данные проекта. Но может случится и так, что эта память вообще не выделялась под проект и, например, используется системой исполнения.

В нашем случае вызов такого кода привел к исключению. В редакторе появилось следующее окно:

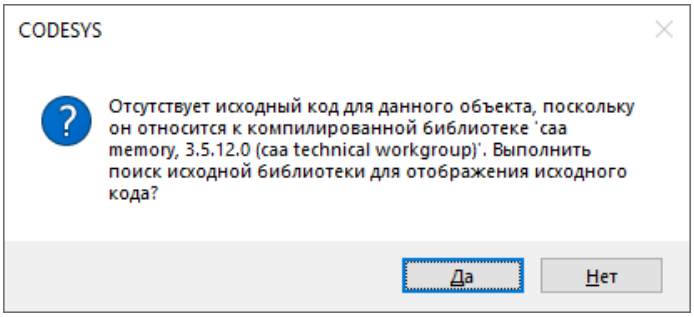


Рис. 4.3.4. Ложное сообщение об исключении в библиотеке, возникшее из-за ошибки сегментации памяти

А в журнале ПЛК можно было увидеть такие сообщения (AccessViolation):

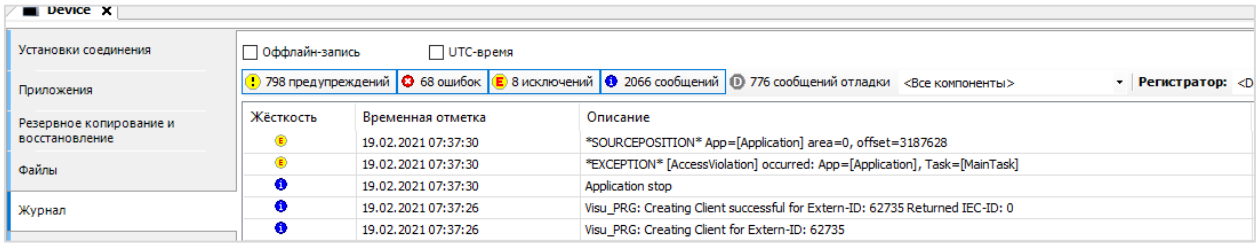


Рис. 4.3.5. Сообщения об исключении AccessViolation в журнале ПЛК

При нажатии на SOURCEPOSITION появилось вот это окно:

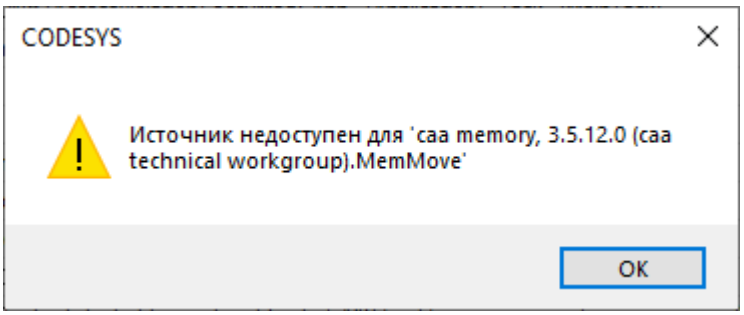


Рис. 4.3.6. Результат нажатия на строку SOURCEPOSITION

В данном случае, вероятно, разработчик довольно быстро бы понял ошибку и исправил ее. Но фактически имя библиотеки могло быть любым – всё зависит от того, к какому адресу памяти бы произошло обращение. В ряде редких случаев такие сообщения свидетельствует об ошибке в одной из системных библиотек, но в большинстве своём они связаны с ошибками при работе с указателями в пользовательском коде. Самая неприятная ситуация – когда происходит перезапись данных самого проекта – тогда исключения может и не произойти, но *что-то пойдет не так*.

Эта ошибка может проявиться не сразу, а спустя месяцы и даже годы после запуска системы (например, происходит перезапись конфигурации для редко используемого режима работы).

Разработчик должен очень внимательно относится к использованию указателей – особенно при арифметических операциях над ними и индексном доступе. Часто вместо написания собственного кода для работы с указателями можно использовать готовые и проверенные временем функции и ФБ из системных библиотек (например, **CAA Memory**).

Отдельный вид головной боли – отладка проектов, где используются указатели для доступа к полям структуры. Рассмотрим, например, такую структуру и код ее обработки (выделение части структуры в массив для последующей передачи в другой ФБ):

DATA

```
1  TYPE DATA :
2  STRUCT
3      x1:     BOOL;
4      x2:     BOOL;
5      x3:     BOOL;
6      x4:     BOOL;
7      by1:    BYTE;
8      dw1:    DWORD;
9  END_STRUCT
10 END_TYPE
```

Рис. 4.3.7. Объявление структуры

PLC_PRG

```
1  PROGRAM PLC_PRG
2  VAR
3      xInit:          BOOL;
4      stData:         DATA;
5      abyData:        ARRAY [0..4] OF BYTE;
6      iTest:          INT;
7  END_VAR
8
```

```
1  stData.by1   := 16#11;
2  stData.dw1   := 16#AABBCCDD;
3
4  IF xInit THEN
5
6      MEM.MemMove(ADR(stData.by1), ADR(abyData), SIZEOF(abyData) );
7      xInit := FALSE;
8
9  END_IF
```

Рис. 4.3.8. Копирование экземпляра структуры в байтовый массив

Разработчик, вероятно, ожидает получить после выполнения этого кода массив значений [16#11, 16#AA, 16#BB, 16#CC, 16#DD]. Но на нашем ПЛК мы получаем следующее:

abyData	ARRAY [0..4] OF BYTE	
abyData[0]	BYTE	16#11
abyData[1]	BYTE	16#00
abyData[2]	BYTE	16#00
abyData[3]	BYTE	16#00
abyData[4]	BYTE	16#DD

Рис. 4.3.9. Результат выполнения кода

Это связано с выравниванием памяти (в структурах данные размещаются не по «соседним» адресам, а с промежутками). См. также информацию об атрибуте pack_mode. В данном конкретном случае можно решить проблему, написав свою функцию для «упаковки» нужных элементов структуры в массив байт.

Еще одна частая ошибка при работе с указателями – разыменование нулевого (неинициализированного) указателя. Например, разработчик использует блок, который возвращает массив указателей (это удобно в тех случаях, когда блок должен возвращать большое количество данных, чтобы избежать лишних операций копирования). Разработчик разыменовывает указатель для получения значения. Но если блок будет отключен (xStart := FALSE;) – то выходы блока (в том числе массив указателей) могут принять нулевые значения.

PLC_PRG

```
1  PROGRAM PLC_PRG
2  VAR
3      xStart:          BOOL := TRUE;
4      fbDataManager:   DataManager;
5
6      dwValue:         DWORD;
7  END_VAR
```

```
1  fbDataManager
2  (
3      xEnable      := xStart,
4      uiValueCount := 1
5  );
6
7  dwValue := fbDataManager.apdwData[1]^;
8
```

Рис. 4.3.10. Пример кода с потенциальным разыменованием нулевого указателями

При попытке разыменования нулевого указателя произойдет исключение (Access Violation). Информация об этом будет выведена в журнал ПЛК. При нажатии на строку SOURCEPOSITION вас «перебросит» к строке 7 программы **PLC_PRG**.

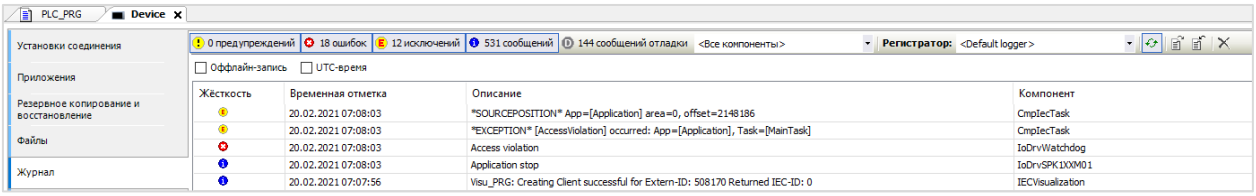


Рис. 4.3.11. Сообщения об исключении AccessViolation в журнале ПЛК

Универсальный совет по работе с указателями – используйте их осторожно и обдуманно. Для отладки в подобных случаях может пригодится плагин MemoryTools. Если вы видите, что ваши переменные получают значения, которые вы нигде им не присваиваете – то, вероятно, причина ошибка связана с некорректным использованием указателя или выходом за границы массива.

4.4. Настройка задач

Для управления вызовом программ в среде CODESYS V3.5 используются задачи. Основные параметры задач – это их приоритет, интервал вызова и настройки сторожевого таймера. Механизм управления задачами может отличаться для разных платформ и операционных систем; особенно сложным он является в многозадачных мультиядерных системах исполнения. При этом в пользовательской документации информация о принципах работы планировщика задач CODESYS практически отсутствует.

Если разработчик будет неосознанного будет менять настройки задач – то это может привести к совершенно различным эффектам (начиная от «торможения» программы и заканчивая возникновением исключений, связанных с критической нагрузкой на CPU).

Ниже приведен пример сомнительной конфигурации задачи **MainTask**, которая в большинстве случаев является основной задачей проекта и задачей цикла шин коммуникационных драйверов – для нее задан низкий приоритет (*в CODESYS самый высокий приоритет – 0, самый низкий – 31; для ПЛК с ОС Linux приоритеты 16…31 соответствуют низкоприоритетным задачам класса SCHED_OTHER*), довольно большой интервал вызова (100 мс – в 5 раз больше значения по умолчанию) и включен сторожевой таймер с таймаутом 40 мс и восприимчивостью 1 (то есть если выполнение хотя бы одного цикла задачи превысит это время – то будет сгенерировано событие от сторожевого таймера). Поскольку приоритет у задачи низкий, то совершенно реальна ситуация, когда она будет вытеснена другими задачами (*если система исполнения является многозадачной*) и не успеет выполниться за это время. Дальнейшее развитие событий зависит от конкретной платформы, но, вероятнее всего, произойдет исключение и приложение будет остановлено. Вполне возможно, что разработчик не будет ожидать такого эффекта; в некоторых случаях периодическое «подтормаживание» низкоприоритетных задач является совершенно приемлемым и не должно приводить к остановке приложения.

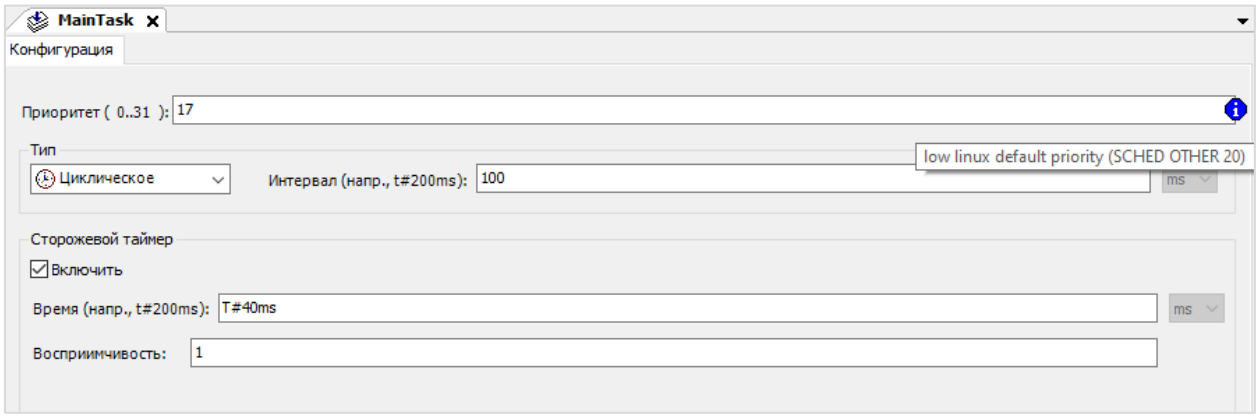


Рис. 4.4.1. Пример сомнительной конфигурации задачи MainTask

Поэтому любые изменения настроек задач должны проводиться осознанно и с полным пониманием возможных побочных эффектов. Некоторую информацию об обработке задач в CODESYS V3.5 можно почерпнуть в этой статье.

4.5. Нарушение условий эксплуатации

По нашему опыту – одной из частых причин «странного» поведения приборов является нарушение условий их эксплуатации. К сожалению, в ряде случаев на реальных объектах можно встретить отступления от технической и рабочей документации, отраслевых стандартов, ПУЭ, спецификаций интерфейсов и протоколов (RS-485, Modbus и т.д.) и здравого смысла.

Приведем просто несколько примеров из нашей практики:

- эксплуатация приборов с рабочим температурным диапазоном 0…50 °С в неотапливаемом шкафу автоматики, размещенном на улице при температуре окружающей среды -30 °С. Формулировка проблемы со стороны пользователя: «прибор иногда зависает и не всегда стартует после подачи питания»;
- сеть RS-485 с протяженностью около километра в условиях типа «заброшенный тоннель» с двумя десятками ответвлений (топология «звезда») Формулировка проблемы со стороны пользователя: «очень медленный обмен, для опроса всех устройств требуется больше минуты-двух»;
- подключение функционального заземления приборов к защитной земле, к которой также подключены преобразователи частоты (которые успешно сливают на неё свои помехи, передающиеся всем остальным устройствам);
- прокладка силовых и сигнальных линий в одном кабель-канале;
- использование телефонного кабеля («лапши») для интерфейса RS-485 при подключении нескольких устройств в шкафу автоматики, в котором также расположено силовое оборудование. Формулировка проблемы со стороны пользователя: «обмен нестабильный, очень много ошибок»;
- и т.п., тысячи было их.

В ответ на замечания о нарушениях мы часто слышим фразу: «*но на других объектах у нас всё сделано так же, и проблем нет*!». Приходится каждый раз приводить пример с человеком, перебегающим дорогу в неположенном месте – если вас не сбили сегодня, то это не значит, что завтра вам тоже повезёт.

Регулярно встречаются проблемы, источником которых является «правильный» кабель (*который был спаян для совсем другого устройства с другой распиновкой COM-порта – и для него, безусловно, этот кабель был правильным*) или «проверенный» адаптер интерфейсов (*иногда разница между USB/TTL и USB/RS-232 кажется кому-то незначительной*).

Все подобные проблемы делают иногда удаленную отладку очень сложной, потому что люди не имеют привычки рассказывать об особенностях монтажа и эксплуатации своих систем (*особенно если чувствуют, что всё сделано «как-то не так»).*

Поэтому если вам требуется помощь в удаленной отладке – мы рекомендуем в первую очередь тщательно проверить, не связана ли проблема с физическим уровнем (кабели, помехи и т.д.) и быть готовым при необходимости предоставить всю нужную информацию по этому вопросу (чертежи, схемы подключения, фотографии и т.д.).

4.6. Отсутствие ТЗ

Довольно часто мы слышим от разработчиков проектов ПЛК фразу «*программа работает неправильно*» (еще чаще – «*этот контроллер работает неправильно*»). На вопрос о том, каковы критерии правильности, обычно следует путанное объяснение про особенности конкретной установки (типа «*должны загореться три зеленые лампочки, а загораются две красные, причем одна из них – оранжевым цветом*»). В принципе, понятно, что человеку легче говорить о том, в чем он разбирается – насосах, системах вентиляции или производственной линии по выпуску резиновых утят. Но когда мы просим абстрагироваться от «физической» составляющей и рассказать про программу (алгоритмы, потоки данных и т.д.), то обычно возникает неловкая пауза. На просьбу показать техническое задание зачастую можно получить в ответ нарисованную от руки принципиальную схему и несколько строк пояснений.

Над рабочим местом одного из наших коллег долгое время висел плакат с известной цитатой: «без внятного ТЗ – результат <…> (непредсказуем)». Сложно не согласиться с этим высказыванием. Если вы не можете сформулировать ожидаемое корректное поведение программы (т.е. какие значения должны получить заданные выходные переменные при изменении значений заданных входных переменных), то вы не вправе называть любое ее поведение некорректным (поскольку некорректное поведение – это поведение, не соответствующее ожидаемому).

Понятно, что в некоторых случаях (*к сожалению*) ТЗ формулируется заказчиком на словах и задокументировать его на этапе отладки уже просто невозможно (банально – нет времени). Но если вы обращаетесь за помощью в отладке – вы должны суметь четко сформулировать свою проблему в терминах программы/проекта (т.е. значений переменных, потоков выполнения и т.д.). Нужно понимать, что посторонний человек может вообще не иметь никакого представления о вашем техпроцессе и никогда не увидит утят-франкенштейнов, которые начали сходить с производственной линии после изменения алгоритма. Он будет ориентироваться на то, что происходит в проекте – и вы должны уметь объяснять, как он устроен, и чем отличаются ожидаемое и фактическое поведение программы.

5. Cool stories: примеры ошибок из нашей практики

В этом разделе мы расскажем о некоторых ошибках, с которыми встречались во время работы, и процессе их отладки. Возможно, в будущей версии статьи мы расширим список историй и добавим в него истории от наших коллег.

5.1. Тонкая красная линия

У пользователя очень «тормозит» панельный контроллер – переключения между экранами занимают до нескольких секунд, обновление данных на экране тоже происходит с секундой задержкой. Открываем присланный проект, смотрим – на первый взгляд, ничего особенного нет, кода не слишком много (полтысячи строк, разбитые на несколько программ), визуализация простая – кнопки, лампочки, поля ввода, прямоугольники, линии и т.д. Загружаем проект, в онлайн-мониторинге задач видим, что время цикла и джиттер для задачи **VISU_TASK** стабильно равны одной-двум секундам.

Удаляем из проекта весь пользовательский код. Проблема остается – значит, вероятно, она связана с визуализацией.

Начинаем последовательно удалять содержимое экранов визуализации. После очистки каждого экрана – заново загружаем проект и следим за временем цикла. В какой-то момент проблема исчезает.

Откатываемся чуть назад (через Ctrl+Z) и начинаем изучать содержимое экрана, который «почистили» последним. После пристального изучения находим линию (элемент **Ломаная**), для которой настроен градиент… Стоп, так разве вообще можно?..

Как оказалось, в **CODESYS V3.5 SP5 Patch 5** так было можно. Визуально, естественно, градиент для линии никак не отображается, но само его наличие производило серьезный эффект на время цикла задачи визуализации. Соответственно, после проверки всех линий проекта (некоторые из них были на экранах, куда мы еще не успели добраться в процессе удаления элементов) – проблема исчезла. В следующих версиях CODESYS возможность включить градиент для элемента **Ломаная** убрали. Вероятно, само наличие галочки было ошибкой среды, связанной с тем, что **Ломаную** можно преобразовать в элемент **Полигон** (для которого градиент имеет смысл). Соответственно, тип ошибки – ошибка в среде программирования.

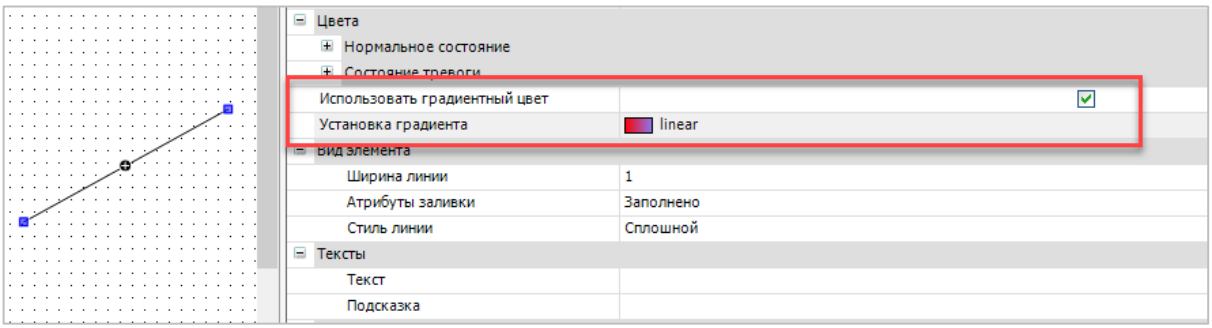


Рис. 5.1.1. Настройки градиента для элемента **Ломаная** в CODESYS V3.5 SP5 Patch 5

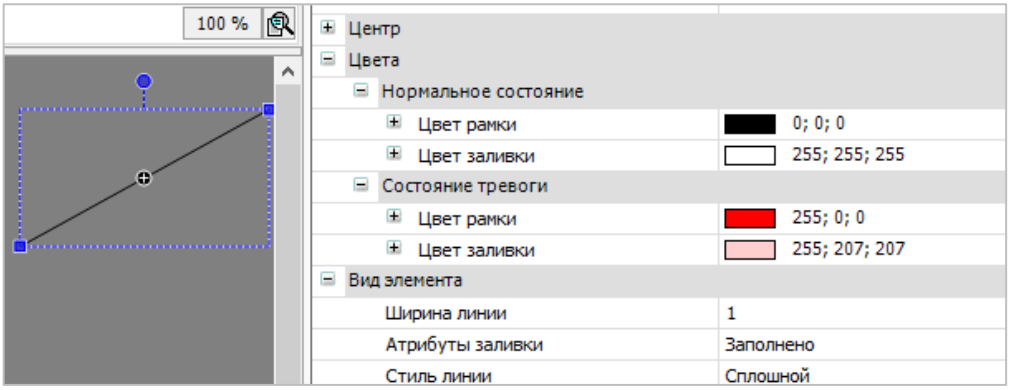


Рис. 5.1.2. В следующих версиях эту галочку убрали

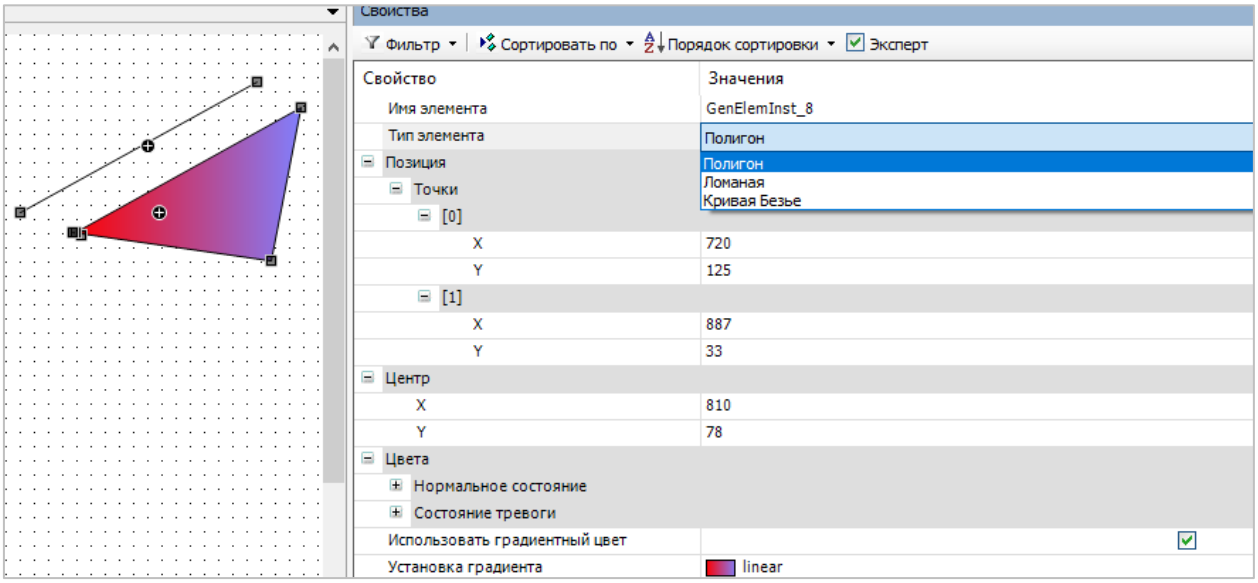


Рис. 5.1.3. Элемент **Ломаная** можно преобразовать в **Полигон** – для него настройка градиента имеет смысл

5.2. Неожиданный ответ

Однажды к нам обратился пользователь, который хотел настроить обмен данными между своим контроллером и сервоприводом Kinco FD422 по протоколу Modbus RTU – и что-то не получалось. Подключаемся по TeamViewer, смотрим на настройки запроса – всё выглядит корректным (код функции, адрес регистра и т.д.).

Более того, в документации на сервопривод приведен конкретный пример запроса чтения – и, судя по настройкам, контроллер должен отправлять именно этот пакет. Но после запроса ПЛК возвращает код ошибки RESPONSE_TIMEOUT – то есть считает, что никакого ответа получено не было.

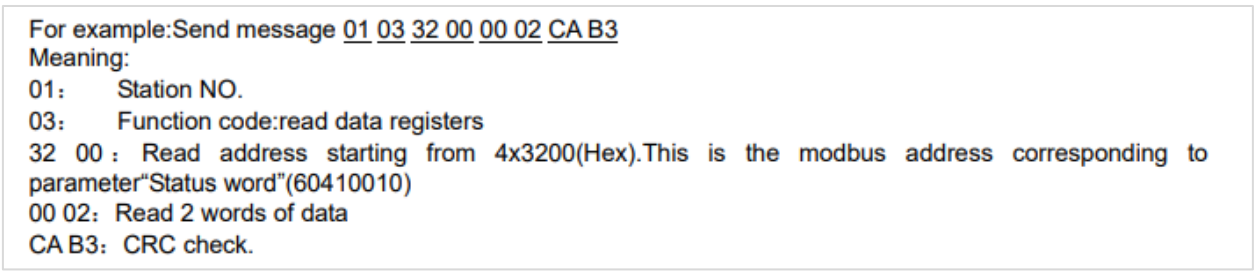

For example:Send message 01 03 32 00 00 02 CA B3
Meaning:
01： Station NO.
03： Function code:read data registers
32 00： Read address starting from 4x3200(Hex).This is the modbus address corresponding to parameter"Status word"(60410010)
00 02：Read 2 words of data
CA B3：CRC check.

Рис. 5.2.1. Фрагмент из документации Kinco FD422

Ладно, решаем исключить из уравнения контроллер – подключаем сервопривод к ПК. Родная сервисная утилита успешно устанавливает связь – значит, на физическом уровне всё нормально. Запускаем проект на виртуальном контроллере CODESYS Control Win V3 – опять RESPONSE_TIMEOUT.

Хорошо, решаем исключить из уравнения CODESYS. У клиента на ПК есть утилита, которая может работать в режиме Modbus RTU Master – и она тоже сообщает нам об отсутствии ответа от slave'а. То есть CODESYS здесь ни при чём, проблема где-то еще.

В этот момент понимаем, что пора уже подключаться к шине сниффером и посмотреть, что вообще происходит на линии связи. Но перед этим решаем сделать последнюю попытку – использовать в качестве мастера наш любимый MasterOPC Universal Modbus Server. Он тоже сообщает нам об ошибке, но при этом:

- текст ошибки гласит, что «размер ответа не соответствует запросу»;
- в логе видим, что сервопривод присылает что-то, напоминающее корректный Modbus-пакет.

Начинаем разбираться. OPC-сервер отправляет именно тот запрос, который был настроен в CODESYS и описан в документации:

01 03 32 00 00 02 CA B3

То есть – используем функцию 0x03 (Read Holding Registers), адрес начального регистра – 0x3200, число считываемых регистров – 0x02.

Вот что отвечает сервопривод:

01 03 **02** 00 31 79 90

Стоп, но он присылает в ответ только 2 байта данных! (то есть один регистр) А где же второй? И вообще, что именно лежит в этих регистрах?

Ищем в документе число 0x3200. Оно встречается всего 3 раза. Понимаем, что в регистре 0x3200 хранится слово состояния, и при этом оно занимает только один регистр (16 бит данных). Регистр 0x3201 (который также считывается в нашем запросе) в документе не упоминается в принципе.

Index	Subindex	Bits	Modbus Address	Command Type	Unit	Descriptions
6041	00	10	0x3200	RO	bitcode	Status byte shows the status of drive bit0：ready to switch on bit1：switch on bit2：operation enable bit3：falt bit4：Voltage Enable bit5：Quick Stop bit6：switch on disable bit7：warning bit8：internal reserved bit9：reserved bit10： target reach bit11： internal limit active bit12： Step.Ach./V=0/Hom.att. bit13： Foll.Err/Res.Hom.Err. bit14： Commutation Found bit15： Referene Found
6060	00	08	0x3500	WO	number	Operation modes: 1 Positioning with position loop 3 Velocity with position loop -3 Velocity loop (immediate velocity mode) -4 Master/slave or pulse/direction control mode 6 Homing 7. CANOPEN based motion interpolation

Рис. 5.2.2. Фрагмент карты регистров Kinco FD422

Тогда зачем вообще мы запрашиваем два регистра?.. А, точно, так ведь было в примере из документации. Но теперь мы знаем, что он не совсем корректный. Меняем запрос так, чтобы считывать только регистр 0x3200:

01 03 32 00 **00 01** 8A B2

Ответ сервопривода не меняется, но теперь OPC считает его корректным (потому что он соответствует запросу). Подключаем привод обратно к контроллеру, проверяем обмен – ожидаемо, всё работает. Проблема решена.

Источником проблемы был пример запроса в документации (может, автор перепутал байты и регистры?) и поведение сервопривода, который возвращал ответ, не соответствующий запросу (логично было бы в этом случае присылать код ошибки 02 – INVALID DATA ADDRESS).

Мораль:

- не стоит слепо верить документации;
- использование сниффера часто сокращает время обнаружения источника проблемы, связанной с обменом.

5.3. Не судите книгу по обложке

Нашему коллеге требовалось настроить обмен с модулем аналогового ввода. Для имитации значения он подключил к первому аналоговому входу датчик, настроил обмен и успешно считал с модуля значение канала – но оно оказалось нулевым (ожидалось что-то похожее на комнатную температуру).

Коллега попросил нас помочь разобраться в ситуации. Его рабочей гипотезой было неверное подключение датчика ко входу. Для начала мы проверили, что опрос настроен корректно – посмотрели на код функции, адрес начального регистра, число регистров. Все это соответствует документации на модуль, более того – контроллер не индицирует ошибок, то есть в ответ на запрос приходит корректный ответ, просто с «нулевыми» данными.

Решаем проверить настройки входа (коллега сказал, что ему не удалось их найти). У модуля есть web-конфигуратор, подключаемся с нему, ищем настройки аналоговых входов. Не находим – есть только настройки аналоговых выходов. Стоп, а разве у этого модуля есть аналоговые выходы?.. В этот момент обращаем внимание, что модель прибора в конфигураторе не соответствует гравировке на корпусе.

Разбираем прибор, смотрим на плату – действительно, перед нами модуль аналоговых выходов… в корпусе модуля аналоговых входов. Всё ясно – инженер, который работал с этими модулями раньше, разобрал их, чтобы изучить схемотехнику, а при сборке перепутал корпуса. Из-за прекрасной унификации приборов – всё отлично собралось (и даже число клемм совпало). Прошивка модулей, видимо, тоже имела определенную универсальность, и у всех модулей была общая карта регистров – просто при запросе данных «отсутствующих» входов/выходов в ответ возвращались нули.

Так может выглядеть ошибка, источником которой является человеческий фактор.

5.4. Помехи в эфире

Звонок с удаленного объекта – периодически зависают панели оператора. Под зависанием клиент понимает мерцание экрана и временную потерю связи. Через какое-то время приборы начинают работать нормально (без перезагрузки или другого воздействия). Долго уточняем детали ситуации в поисках систематики – в какой-то момент узнаем, что о проблемах всегда сообщают по рации (других способов связи на объекте нет). После еще пары вопросов выясняем, что рация работает на частоте 400 МГц – это совпадает с частотой процессора панели. Просим перейти на другую частоту – через какое-то время нам сообщают, что проблема больше не воспроизводится.

Бывает и такое.

5.5. Картинки с выставки

В понедельник начинается выставка, на которой клиент хочет продемонстрировать свое решение на базе панельного контроллера и модулей ввода-вывода. Как обычно, последние изменения в ПО вносятся в пятницу вечером. Новая версия проекта проверяется на работоспособность и отправляется менеджеру, который утром в понедельник с флэшки загружает ее в контроллер.

Через полчаса панический звонок от менеджера – «с экрана пропали картинки». Просим прислать фото – действительно, часть изображений на дисплее отсутствует, хотя в проекте они есть. После перезагрузки контроллера картинки возвращаются на свои законные места. Менеджер облегченно выдыхает. Но через 15 минут ситуация повторяется. Разработчик запускает проект у себя, чтобы попробовать воспроизвести проблему. Менеджер каждые 15 минут перезагружает контроллер и нервно улыбается посетителям выставки.

Тем временем проблема воспроизводится на столе у разработчика – через 15 минут действительно с экрана начинают пропадать графические элементы. Прекрасно, воспроизводимость на лицо. Коротко обсуждаем вариант отката на предыдущую версию проекта – но сначала решаем попробовать разобраться, в чем дело здесь. В журнале ПЛК видим сообщения, что, действительно, некоторые графические файлы не удается «подгрузить». Проверяем, есть ли они в памяти контроллера в нужной директории – убеждаемся, что есть. Смотрим настройки визуализации, проверяем время выполнения задач – всё выглядит корректным. На всякий случай решаем проверить значения в узле Debug – может, загрузка CPU под 100%? Нет, с CPU все нормально – зато видим, что число используемых дескрипторов (хэндлов) постоянно растет.

Это уже проясняет ситуацию – в контроллере есть ограничение на число одновременно используемых дескрипторов (1024), и когда их лимит исчерпан – то, в принципе, может произойти что угодно. Пропадание картинок не выглядит в этом случае чем-то фантастическим. В пользовательской программе создавать дескрипторы приходится не так уж и часто – в основном, это происходит при работе с файлами и сокетами. В этот момент инициативу перехватывает разработчик, который начинает понимать происходящее.

Как оказалось, одним из изменений, произведенных в пятницу вечером, было добавление в проект кода для ведения архивов. Из-за ошибки в программе после записи в файл не происходило его закрытие – в результате каждая запись в файл приводила к использованию нового дескриптора, и вскоре их число заканчивалось, после чего проявлялся эффект пропадания картинок (наверняка, происходило что-то еще, просто это было самым ярким проявлением проблемы). На столе у разработчика это сначала не воспроизвелось, так как проверка основного функционала проекта занимала менее 15 минут.

В итоге код, связанный с архивацией, просто закомментировали (на выставке она, в общем-то, не особо и требовалась), в обед менеджер обновил проект и после этого картинки больше не пропадали. Выставка прошла без эксцессов и все были счастливы.

5.6. Побочный эффект

У пользователя «в какой-то момент пропала web-визуализация». При попытке открыть web-страницу – индикатор загрузки промаргивает, но после этого отображается только белый экран. В журнале ПЛК множество одинаковых сообщений о запуске визуализации:

Фильтр

0 ошибок 0 исключений 0 предупреждений 207 информационных сообщений 0 отладочных сообщений

PlcLog.csv

Уровень	Временная метка	Описание	Компонент
i	05.10.2020 14:57:22	Visuinitialization done.	0x0000100c
i	05.10.2020 14:57:22	Visuinitialization starting.	0x0000100c
i	05.10.2020 14:57:21	Visuinitialization done.	0x0000100c
i	05.10.2020 14:57:21	Visuinitialization starting.	0x0000100c
i	05.10.2020 14:57:21	Visuinitialization done.	0x0000100c
i	05.10.2020 14:57:21	Visuinitialization starting.	0x0000100c
i	05.10.2020 14:57:21	Visuinitialization done.	0x0000100c
i	05.10.2020 14:57:21	Visuinitialization starting.	0x0000100c
i	05.10.2020 14:57:21	Visuinitialization done.	0x0000100c
i	05.10.2020 14:57:21	Visuinitialization starting.	0x0000100c
i	05.10.2020 14:57:21	Visuinitialization done.	0x0000100c
i	05.10.2020 14:57:21	Visuinitialization starting.	0x0000100c
i	05.10.2020 14:57:21	Visuinitialization done.	0x0000100c
i	05.10.2020 14:57:20	Visuinitialization starting.	0x0000100c
i	05.10.2020 14:57:20	Visuinitialization done.	0x0000100c
i	05.10.2020 14:57:20	Visuinitialization starting.	0x0000100c
i	05.10.2020 14:57:20	Visuinitialization done.	0x0000100c
i	05.10.2020 14:57:20	Visuinitialization starting.	0x0000100c
i	05.10.2020 14:57:20	Visuinitialization done.	0x0000100c
i	05.10.2020 14:57:20	Visuinitialization starting.	0x0000100c
i	05.10.2020 14:57:20	Visuinitialization done.	0x0000100c

Рис. 5.6.1. Журнал ПЛК в web-конфигураторе контроллера

Получаем доступ по TeamViewer, подключаемся к контроллеру, удостоверяемся, что при попытке зайти на web-визуализацию виден только белый экран. Еще раз смотрим лог – кроме сообщений с рисунка выше в нем нет ничего интересного.

Проверяем время выполнения **VISU_TASK** – оно совершенно неадекватное, сотни миллисекунд (джиттер составляет секунды). Выдвигаем гипотезу, что проблема на одном из экранов визуализации. Удаляем все элементы со всех экранов – проблема сохраняется. Ладно, вспоминаем одну из похожих историй и переходим в узел Debug – проверяем, нет ли утечки дескрипторов. Утечки нет, но замечаем, что нагрузка CPU уверенно держится на уровне 100%.

Переходим к программе. Для начала убираем вообще весь код, загружаем проект и видим, что проблема исчезла – web-визуализация работает корректно (пользовательские экраны отображаются), время цикла задачи **VISU_TASK** теперь нормальное (десятки миллисекунд).

Отлично, значит, проблема где-то в коде. Возвращаем код (соответственно, проблема опять начинает воспроизводиться) и приступаем к поискам – отключаем друг за другом фрагменты кода (закомментировав нужные строки), загружаем проект и смотрим, не исчезнет ли проблема. После отключения очередной программы проблема исчезает.

Переходим в эту программу и продолжаем уже в ней искать проблемный фрагмент. Через некоторое время он обнаруживается – мы видим, что проблема пропала после того, как мы закомментировали 10 строк кода без вызова каких-то блоков и функций – то есть проблема явно где-то здесь.

Внимательно изучаем этот фрагмент:

```
VAR
   axTest:            ARRAY [0..10]  OF BOOL;
   iCountTest:        INT;
END_VAR

FOR iCountTest :=0 TO 20 BY 1 DO

   axTest[iCountTest] := TRUE;

END_FOR
```

Всё ясно – мы встретились с типичной проблемой некорректного обращения к памяти. Размерность массива **axTest** – **0…10**, а в цикле происходит запись его элементов в диапазоне **0…20** – то есть происходит перезапись «чужой» памяти. Вероятно, в этой области памяти были размещены какие-то данные задачи визуализации – и именно это приводило к наблюдаемым проблемам с web-визуализацией. Если бы в этой области памяти были размещены другие данные – то проявление ошибки могло быть совершенно иным.

Возвращаемся к исходной версии проекта, исправляем верхнюю границу счетчика цикла и проверяем, что проблема больше не воспроизводится. Проверяем все остальные циклы в проекте, чтобы исключить вероятность наличия незамеченной аналогичной проблемы. Оставляем разработчику рекомендации по использованию констант в качестве границ массивов и счетчиков циклов.

Дело закрыто.

5.7. Как выглядела схема?

Коллега просит совета насчет странной ситуации. Он участвует в разработке стенда тестирования для приборов заказчика. Прибор подключается к контроллеру по интерфейсу USB, контроллер конфигурирует его и проводит ряд проверок. Стенд успешно прошел испытания на тестовом полигоне, после чего заказчику была отправлена спецификация оборудования, схемы подключения и ПО для контроллера. Через некоторое время приходит обратная связь от заказчика – тестирование одного прибора проходит корректно, но после его отключения и подключения второго прибора – контроллер уже не может его определить. Проблема решается только перезагрузкой ПЛК.

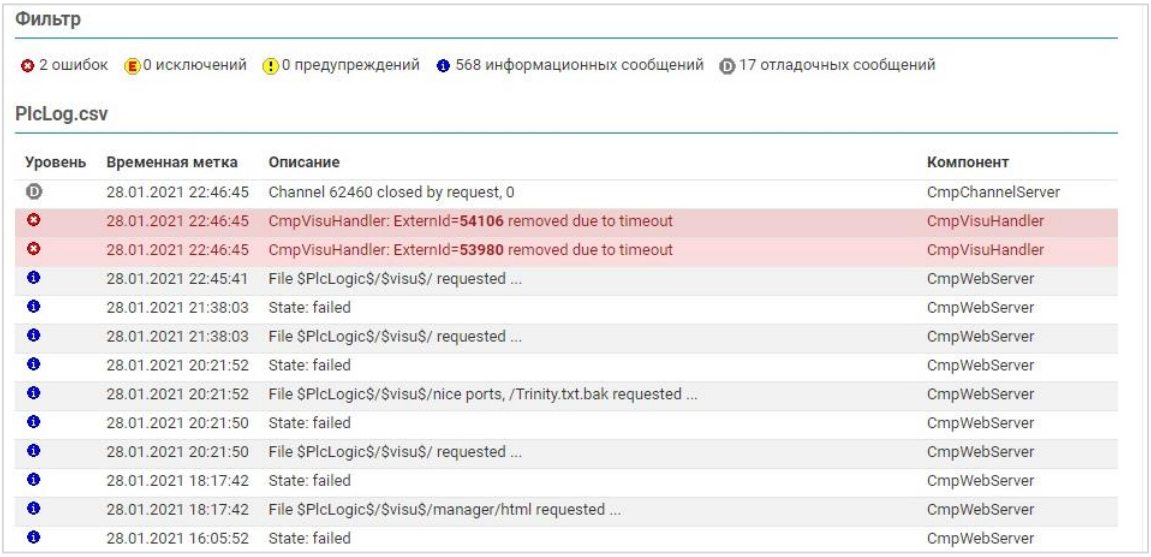
На столе у разработчика проблема не воспроизводится. Начинаем искать отличия между системами разработчика и заказчика – сравниваем модификации контроллеров, аппаратную ревизию, версию прошивки. Всё совпадает. Предлагаем запросить у заказчика фотографии собранного им стенда. Это предложение не вызывает энтузиазма – схема подключения простая, а квалификация инженеров заказчика не подвергается сомнению. Убеждаем, что фотографии все же нужны, чтобы разобраться в причинах проблемы.

Спустя некоторое время получаем фото стенда и видим принципиальные отличия от отправленной заказчику схемы подключения – согласно ей приборы подключались напрямую к контроллеру (и именно так проверял работу стенда разработчик), у заказчика же порт USB контроллера соединен с бытовым USB-хабом, к которому подключается не только тестируемый прибор, но и находящийся рядом ПК (это было сделано, что иметь возможность быть быстро подключиться к прибору с помощью конфигурационного ПО).

Хаб убрали, приборы начали подключать напрямую к контроллеру согласно схеме подключения – и проблема исчезла. Так может выглядеть ошибка, вызванная человеческим фактором.

5.8. Удаленный доступ

У пользователя спустя какое-то время работы контроллера перестает открываться web-визуализация. В присланном скриншоте с логом, на первый взгляд, только сообщение о принудительном отключении клиента визуализации по неизвестным причинам.



Фильтр

2 ошибок | 0 исключений | 0 предупреждений | 568 информационных сообщений | 17 отладочных сообщений

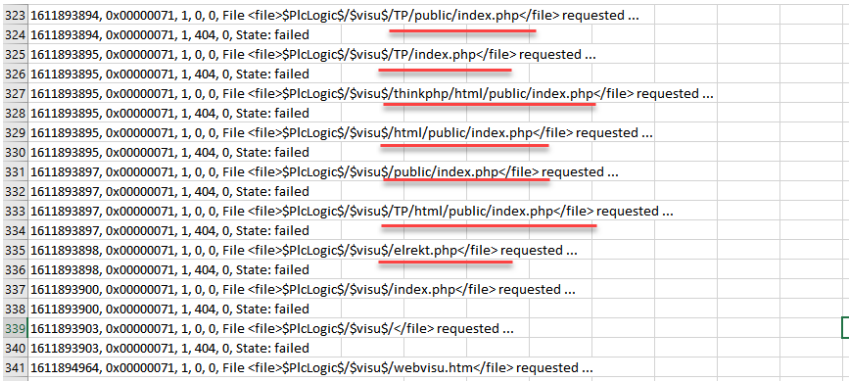
PlcLog.csv

Уровень	Временная метка	Описание	Компонент
	28.01.2021 22:46:45	Channel 62460 closed by request, 0	CmpChannelServer
	28.01.2021 22:46:45	CmpVisuHandler: ExternId=**54106** removed due to timeout	CmpVisuHandler
	28.01.2021 22:46:45	CmpVisuHandler: ExternId=**53980** removed due to timeout	CmpVisuHandler
	28.01.2021 22:45:41	File $PlcLogic$/$visu$/ requested ...	CmpWebServer
	28.01.2021 21:38:03	State: failed	CmpWebServer
	28.01.2021 21:38:03	File $PlcLogic$/$visu$/ requested ...	CmpWebServer
	28.01.2021 20:21:52	State: failed	CmpWebServer
	28.01.2021 20:21:52	File $PlcLogic$/$visu$/nice ports, /Trinity.txt.bak requested ...	CmpWebServer
	28.01.2021 20:21:50	State: failed	CmpWebServer
	28.01.2021 20:21:50	File $PlcLogic$/$visu$/ requested ...	CmpWebServer
	28.01.2021 18:17:42	State: failed	CmpWebServer
	28.01.2021 18:17:42	File $PlcLogic$/$visu$/manager/html requested ...	CmpWebServer
	28.01.2021 16:05:52	State: failed	CmpWebServer

Рис. 5.8.1. Журнал ПЛК в web-конфигураторе контроллера

ПЛК установлен на удаленном объекте, возможности для отладки практически отсутствуют – подключиться к контроллеру из CODESYS нельзя. Есть только удаленный доступ к web-визуализации и web-конфигуратору (на объекте контроллер подключен к роутеру с внешним статическим IP, на котором проброшены нужные порты). При возникновении проблемы контроллер перезагружают из web-конфигуратора, после чего web-визуализация какое-то время работает нормально.

Ловим дежавю, просим предоставить как можно больше информации о ситуации (с какой периодичностью возникает, в какое время суток и т.д.). Спустя какое-то время получаем лог контроллера в текстовом виде. Начинаем его изучать и находим следующие сообщения (*и понимаем, что что-то похожее было и на скриншоте выше, просто раньше не бросалось в глаза*):



```
323 1611893894, 0x00000071, 1, 0, 0, File <file>$PlcLogic$/$visu$/TP/public/index.php</file> requested ...
324 1611893894, 0x00000071, 1, 404, 0, State: failed
325 1611893895, 0x00000071, 1, 0, 0, File <file>$PlcLogic$/$visu$/TP/index.php</file> requested ...
326 1611893895, 0x00000071, 1, 404, 0, State: failed
327 1611893895, 0x00000071, 1, 0, 0, File <file>$PlcLogic$/$visu$/thinkphp/html/public/index.php</file> requested ...
328 1611893895, 0x00000071, 1, 404, 0, State: failed
329 1611893895, 0x00000071, 1, 0, 0, File <file>$PlcLogic$/$visu$/html/public/index.php</file> requested ...
330 1611893895, 0x00000071, 1, 404, 0, State: failed
331 1611893897, 0x00000071, 1, 0, 0, File <file>$PlcLogic$/$visu$/public/index.php</file> requested ...
332 1611893897, 0x00000071, 1, 404, 0, State: failed
333 1611893897, 0x00000071, 1, 0, 0, File <file>$PlcLogic$/$visu$/TP/html/public/index.php</file> requested ...
334 1611893897, 0x00000071, 1, 404, 0, State: failed
335 1611893898, 0x00000071, 1, 0, 0, File <file>$PlcLogic$/$visu$/elrekt.php</file> requested ...
336 1611893898, 0x00000071, 1, 404, 0, State: failed
337 1611893900, 0x00000071, 1, 0, 0, File <file>$PlcLogic$/$visu$/index.php</file> requested ...
338 1611893900, 0x00000071, 1, 404, 0, State: failed
339 1611893903, 0x00000071, 1, 0, 0, File <file>$PlcLogic$/$visu$/</file> requested ...
340 1611893903, 0x00000071, 1, 404, 0, State: failed
341 1611894964, 0x00000071, 1, 0, 0, File <file>$PlcLogic$/$visu$/webvisu.htm</file> requested ...
```

Рис. 5.8.2. Фрагмент журнала ПЛК

Клиент визуализации запрашивает у web-сервера контроллера индексные файлы, которые он ожидает найти в директориях /TP, /thinkphp, /public и других. Но таких директорий в контроллере попросту нет, только если пользователь не создал их самостоятельно (*что кажется маловероятным из-за довольно специфических названий*).

Начинаем гуглить что-то насчет этих директорий и довольно быстро находим несколько статей (вот пример одной из них), где другие люди сталкиваются со схожими проблемами. Суть в следующем – нехорошие люди запускают ботов, которые постоянно сканируют все интернет-ресурсы на наличие известных уязвимостей для последующего взлома.

Вероятно, из-за интенсивности этих запросов web-сервер контроллера (*аппаратные ресурсы которого далеко не безграничны*) не выдерживал и падал. Мы передали пользователю список рекомендаций по решению проблемы – изменение используемых портов на произвольные (*которые не относятся к числу общеизвестных; по умолчанию web-сервер визуализации использовал порт 8080 – стандартный порт для HTTP proxy и Apache Tomcat*), настройка firewall'а на роутере и т.д.

5.9. Времени нет

Пользователь столкнулся со странной проблемой – при загрузке проекта возникает следующее исключение:

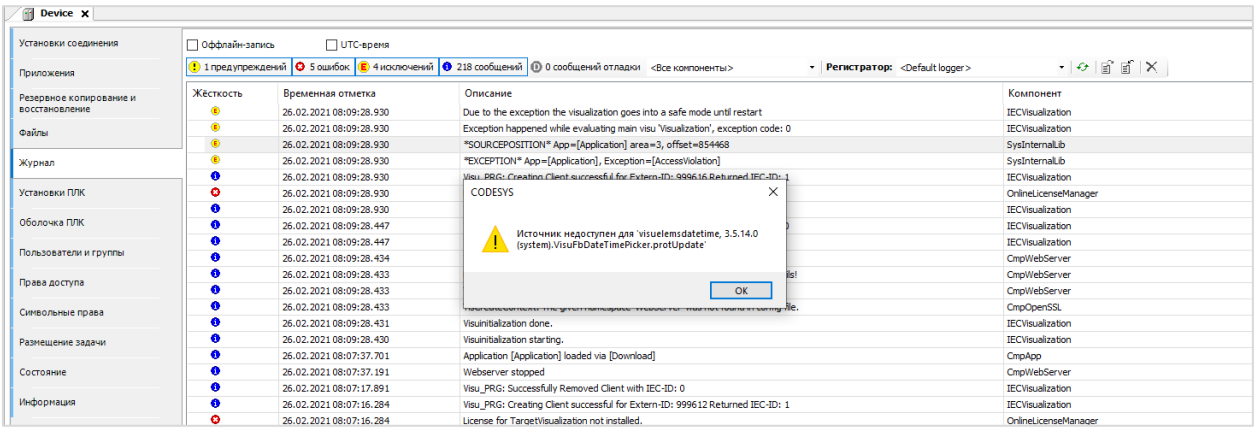


Рис. 5.9.1. Сообщения об исключении AccessViolation в журнале ПЛК

А, ну так это типичная ошибка сегментации памяти!

Запрашиваем проект, воспроизводим у себя, начинаем разбираться. Для начала удаляем из проекта весь код – но проблема остается. Странно, мы ожидали, что в этот момент она исчезнет. Тогда удаляем из проекта визуализацию – и вот после этого проблема пропадает. Еще раз внимательно читаем текст, который возникает при нажатии на SOURCEPOSITION – там что-то про **visuelemdatetime**…

Это похоже на название элемента выбора даты и времени. Возвращаем в проект визуализацию, пробегаемся по ней – но этого элемента не находим. Странно. Ладно, начинаем последовательно удалять содержимое экранов визуализации, чтобы определить проблемный экран. Находим его. Элемента выбора даты и времени на нем не видно. Раскрываем список элементов в редакторе визуализации и только тогда его обнаруживаем – с нулевой шириной и высотой (см. рис. 5.9.2). Понятно – проблема в некорректных настройках элемента. Вероятно, в данном случае ошибку можно отнести к ошибкам IDE – среда либо должна без ошибок обрабатывать такие моменты (например, для элемента выбора даты с такими настройками исключения не возникнет), либо предупреждать разработчика об этой ошибке еще на этапе компиляции.

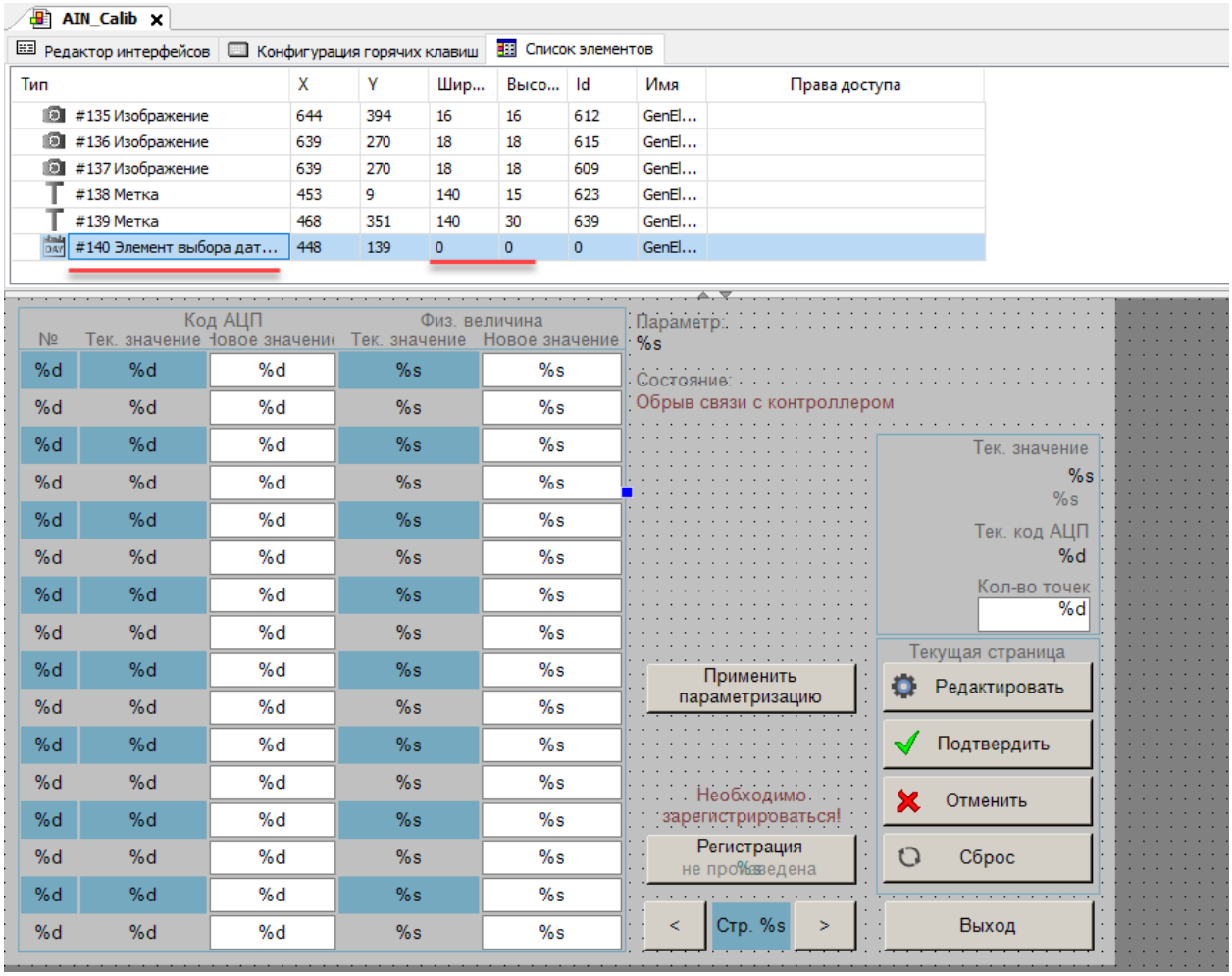


Рис. 5.9.2. Некорректно настроенный элемент выбора даты и времени

5.10. Солнечная энергия

На объекте пользователя периодически повреждаются файлы архивов. Систематику установить не удается – в среднем, раз в пару дней в файлы записывается какой-то «мусор» (*его природу установить не получается – выглядит как произвольный набор байтов*). На столе у разработчика проблему воспроизвести не удается.

Как обычно – запрашиваем у клиента подробную информацию об объекте. В результате выясняем, что объект запитывается от местной солнечной электростанции. После консультаций со схемотехниками формулируется гипотеза, что по цепи питания периодически проходит электромагнитный импульс, который и является причиной проблемы.

Используя эту информацию – проводим эксперимент у себя на столе. Берем лампу мощностью 100 Вт, запитываем ее от той же сети, что и контроллер. ПЛК периодически включает и выключает реле, коммутирующее цепь питания лампы. В результате через непродолжительное время проблема воспроизводится – получаем битый файл архива.

Информацию передали пользователю; через какое-то время на объекте начались работы по перемонтажу оборудования, в результате чего шкаф с контроллером был переставлен в другое место, и проблема пропала. Мы так и не узнали, была ли она связана с питанием, наводками от другого оборудования или чем-то еще.

Список литературы

1. Брайан Керниган, Роб Пайк. Практика программирования

Довольно компактная книга с очень хорошим стилем изложения, один из авторов которой точно не нуждается в представлении. Вопросам отладки посвящена глава 5 – на 20 страницах которой раскрыты все основные моменты.

2. Стив Макконнелл. Совершенный код. Практическое руководство по разработке программного обеспечения

Одна из самых известных книг по разработке ПО. Вопросам отладки посвящена глава 23.

3. David J. Agans. The 9 Indispensable Rules for Finding Even the Most Elusive Software and Hardware Problems

4. Paul Butcher. Debug It!: Find, Repair, and Prevent Bugs in Your Code

Эти две книги целиком посвящены различным аспектам отладки. К сожалению, насколько нам известно, на русском языке они не издавались.

5. Цикл статей «Фольклор программистов и инженеров», опубликованных на Хабре: часть 1, часть 2, часть 3

Подборка историй про специфические ошибки и их отладку – прекрасный образчик того, насколько нестандартные ситуации могут возникнуть в реальной жизни. Мы настоятельно рекомендуем всем читателями ознакомится с ними.

6. Павел Зубарев, Ольга Шаронова. Искусство отладки

Презентация о принципах отладки и основных инструментах, которые могут использоваться в ее ходе.

7. Цикл статей от Елены Сагалаевой (Алёны C++)

Несколько коротких статей на тему отладки. В каждой статье присутствуют ссылки на «байки» и другие статьи.

8. Игорь Петров. Отладка прикладных ПЛК программ в CoDeSys

Цикл статей про отладку ПО в среде CoDeSys V2.3.