|  |  |  |
| --- | --- | --- |
|  |  |  |
| МИНОБРНАУКИ РОССИИ | | |
| Федеральное государственное бюджетное образовательное учреждение  высшего образования  **«МИРЭА – Российский технологический университет»**  **РТУ МИРЭА** | | |
| Институт искусственного интеллекта | | |
| Базовая кафедра №536 – программного обеспечения систем радиоэлектронной аппаратуры | | |

|  |  |  |
| --- | --- | --- |
| **РАБОТА ДОПУЩЕНА К ЗАЩИТЕ** | | |
| Заведующий кафедрой |  | Михеев В.А. |
|  |  | |
| «\_\_\_\_» \_\_\_\_\_\_\_\_\_\_\_ 2025 г. | | |

**ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА**

|  |
| --- |
| по направлению подготовки бакалавра 01.03.02 «Прикладная математика и информатика» |
| на тему: |
| «Имитация процедур управления спецвычислителями  радиолокатора космического базирования» |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Обучающийся |  |  |  | Торкин Данила Андреевич | | |
|  |  | *подпись* |  |  | | |
|  |  |  |  |  |  |  |
| Шифр |  | 21К0563 |  |  |  |  |
| Группа |  | КМБО-02-21 |  |  |  |  |
|  |  |  |  |  |  |  |
| Руководитель  работы |  |  |  | к.т.н., доцент |  | Черниенко А.А. |
|  |  | *подпись* |  |  |  |  |
| Консультант |  |  |  |  |  | Ветюгов С.В. |
|  |  | *подпись* |  | *ученая степень, ученое звание, должность* |  |  |

Москва 2025

АННОТАЦИЯ

В рамках выпускной квалификационной работы …

…

…

Объем пояснительной записки – … страниц, … слов.

Пояснительная записка содержит – … рисунков и … приложений.

В списке использованной литературы – … источников.

Ключевые слова: …

ОГЛАВЛЕНИЕ

[ВВЕДЕНИЕ 5](#_Toc199426090)

[1 АНАЛИЗ ЗАДАЧИ ИМИТАЦИИ ПРОЦЕДУР УПРАВЛЕНИЯ СПЕЦВЫЧИСЛИТЕЛЯМИ РАДИОЛОКАТОРА КОСМИЧЕСКОГО БАЗИРОВАНИЯ 8](#_Toc199426091)

[1.1 Общие принципы работы РСА 8](#_Toc199426092)

[1.2 Вычислительная система РСА 9](#_Toc199426093)

[2 АНАЛИЗ ТРЕБОВАНИЙ 15](#_Toc199426094)

[2.1 Обзор протокола обмена сообщениями УВМ-СВ 15](#_Toc199426095)

[2.2 Выбор технологий и средств разработки 22](#_Toc199426099)

[3 РЕАЛИЗАЦИЯ ПРОГРАММНОГО ИНТЕРФЕЙСА 24](#_Toc199426100)

[3.1 Архитектура разработанного программного комплекса 24](#_Toc199426101)

[3.2 Программный модуль СВ 28](#_Toc199426102)

[3.3 Программный модуль УВМ 39](#_Toc199426108)

[3.4 Пользовательский интерфейс (gui\_app) 61](#_Toc199426115)

[3.5 Сборка и конфигурирование комплекса 68](#_Toc199426120)

[4 ТЕСТИРОВАНИЕ 72](#_Toc199426121)

[4.1 Программа и методика тестирования 72](#_Toc199426122)

[4.2 Результаты тестирования 72](#_Toc199426123)

[4.3 Выводы 73](#_Toc199426124)

[ЗАКЛЮЧЕНИЕ 74](#_Toc199426125)

[СПИСОК ИСПОЛЬЗУЕМОЙ ЛИТЕРАТУРЫ 75](#_Toc199426126)

[ПРИЛОЖЕНИЕ А 76](#_Toc199426127)

[ПРИЛОЖЕНИЕ Б 78](#_Toc199426128)

[ПРИЛОЖЕНИЕ В 79](#_Toc199426129)

[ПРИЛОЖЕНИЕ Г 80](#_Toc199426130)

[ПРИЛОЖЕНИЕ Д 81](#_Toc199426131)

СПИСОК СОКРАЩЕНИЙ

|  |  |  |
| --- | --- | --- |
| СВ | – | Спецвычислитель |
| УВМ | – | Управляющая Вычислительная Машина |
| РСА | – | Радиолокационная Система с синтезированной Апертурой |
|  | – |  |
|  | – |  |
|  | – |  |
|  | – |  |
|  | – |  |

# ВВЕДЕНИЕ

Современные радиолокационные системы с синтезированной апертурой (РСА) космического базирования являются сложными многокомпонентными комплексами, предназначенными для получения высокодетальных изображений земной поверхности. Ключевым элементом таких систем является бортовая вычислительная система, обеспечивающая управление компонентами РСА и обработку получаемых данных. Эффективность и надежность этой системы напрямую зависят от корректного взаимодействия ее основных частей: УВМ и группы бортовых СВ, каждый из которых отвечает за специфические этапы обработки радиолокационной информации.

Процесс разработки и, в особенности, наземной отладки и комплексных испытаний программного обеспечения УВМ, а также верификации протокола обмена данными с СВ, сопряжен со значительными трудностями. Основная проблема заключается в ограниченной доступности или высокой стоимости реальных аппаратных модулей СВ на этапах разработки, что затрудняет всестороннее тестирование управляющей логики, отработку нештатных ситуаций и проверку корректности реализации протокола, особенно при взаимодействии с несколькими вычислителями одновременно. Это может приводить к увеличению сроков и стоимости создания всего комплекса РСА.

В связи с этим, актуальной задачей является создание программных средств, позволяющих имитировать процедуры управления бортовыми спецвычислителями и обмен цифровыми радиолокационными данными между компонентами бортовой вычислительной системы. Разработка такого программного комплекса для имитации является предметом настоящей выпускной квалификационной работы.

Целью данной работы является обеспечение наземной отладки и испытаний вычислительной системы РСА космического базирования в части взаимодействия бортовых спецвычислителей и управляющей вычислительной машины РСА космического базирования.

Для достижения указанной цели были поставлены следующие задачи: анализ требований протокола взаимодействия, разработка архитектуры и кода программного обеспечения (ПО), его отладка и документирование. В качестве исходных данных использовался документ «Протокол взаимодействия бортовых спецвычислителей и управляющей вычислительной машины».

Объектом разработки выступает программный комплекс, моделирующий информационный обмен и процедуры управления между УВМ и группой эмулируемых СВ. Предметом разработки являются алгоритмы и программные модули, обеспечивающие имитацию поведения до четырех экземпляров СВ, реализацию управляющей логики УВМ для взаимодействия с ними по четырем асинхронным каналам сети Ethernet (с поддержкой протокола TCP/IP), а также средства мониторинга и отладки этого взаимодействия.

Новизна работы заключается в создании программного обеспечения, позволяющего имитировать процедуры управления и обмена данными, специфичные для разрабатываемого АО «Концерн «Вега» РСА космического базирования. Практическая значимость заключается в предоставлении инструмента, который позволяет проводить эффективную отладку и тестирование ПО УВМ в условиях, максимально приближенных к реальным, без необходимости использования полной аппаратной конфигурации, что способствует сокращению сроков и затрат на разработку.

Программный комплекс реализован на языке C с использованием стандартных библиотек POSIX для обеспечения многопоточности и сетевого взаимодействия, что гарантирует его работоспособность в операционной системе Linux, включая отечественные защищенные платформы. Для визуализации процесса взаимодействия разработан графический интерфейс пользователя на Qt. Программная документация подготовлена в соответствии с требованиями группы ГОСТ Единой системы программной документации.

Данная пояснительная записка включает введение, четыре основные главы, заключение, список использованной литературы и приложения. В первой главе проводится анализ задачи имитации. Вторая глава посвящена анализу требований к ПО, включая обзор протокола и выбор средств разработки. Третья глава детально описывает архитектуру и реализацию программных модулей. Четвертая глава представляет программу и методику испытаний, а также полученные результаты.

# 1 АНАЛИЗ ЗАДАЧИ ИМИТАЦИИ ПРОЦЕДУР УПРАВЛЕНИЯ СПЕЦВЫЧИСЛИТЕЛЯМИ РАДИОЛОКАТОРА КОСМИЧЕСКОГО БАЗИРОВАНИЯ

## Общие принципы работы РСА

Радиолокационные системы с синтезированной апертурой (РСА) представляют собой класс активных систем дистанционного зондирования, предназначенных для получения радиолокационных изображений земной поверхности или иных объектов. Отличительной особенностью РСА является их способность функционировать в различных метеорологических условиях и независимо от времени суток, благодаря использованию радиоволнового диапазона электромагнитного излучения.

Фундаментальный принцип работы РСА заключается в когерентной обработке последовательности отраженных от зондируемой поверхности радиосигналов, принятых антенной системой локатора в процессе движения его носителя. Для РСА космического базирования носителем выступает искусственный спутник Земли. В процессе орбитального движения аппаратная часть РСА осуществляет периодическое излучение зондирующих электромагнитных импульсов в направлении наблюдаемого участка и прием эхо-сигналов. За счет перемещения носителя вдоль траектории полета, малоапертурная физическая антенна последовательно занимает различные пространственные позиции. Полученные с этих позиций и когерентно накопленные данные подвергаются сложной цифровой обработке. Результатом такой обработки является синтез антенной системы с большой эффективной апертурой, длина которой может достигать нескольких километров, что значительно превышает физические размеры бортовой антенны. Именно синтезированная апертура позволяет достигать высокого пространственного разрешения радиолокационных изображений, сопоставимого, а в некоторых случаях и превосходящего, разрешение оптических систем при неблагоприятных условиях наблюдения.

Размещение РСА на космических аппаратах обеспечивает ряд существенных преимуществ, таких как возможность глобального обзора, регулярность и оперативность получения данных по обширным территориям. Это делает РСА космического базирования эффективным инструментом для решения широкого спектра научно-прикладных задач, включая картографирование, мониторинг использования земельных ресурсов, оценку состояния ледовой обстановки, контроль смещений земной поверхности, обнаружение последствий чрезвычайных ситуаций и решение задач в интересах обороны и безопасности. Конечным продуктом работы РСА являются цифровые радиолокационные изображения (ЦРЛИ) и производные от них информационные продукты, характеризующие отражательные свойства и геометрию наблюдаемой поверхности.

## 1.2 Вычислительная система РСА

Функционирование РСА космического базирования и реализация сложных алгоритмов обработки радиолокационной информации обеспечиваются бортовой вычислительной системой. Ключевыми компонентами этой системы, отвечающими за управление комплексом и специализированную обработку данных, являются Управляющая Вычислительная Машина (УВМ) и группа Спецвычислителей (СВ).

УВМ выступает в роли центрального координирующего узла бортовой аппаратуры РСА. К ее основным функциям относятся: прием и интерпретация команд полетного задания, поступающих от наземного сегмента управления или общей системы управления космического аппарата; формирование и передача управляющих команд и детальных параметров настройки каждому из подключенных СВ для реализации конкретных режимов радиолокационной съемки; осуществление непрерывного мониторинга телеметрической информации о состоянии и работоспособности СВ и других подсистем РСА; а также сбор, первичная буферизация, возможное агрегирование и форматирование потоков данных, получаемых от СВ, для их последующей передачи на Землю по радиоканалу. Таким образом, УВМ обеспечивает общее управление работой РСА и взаимодействие с другими бортовыми и наземными системами.

СВ представляют собой высокопроизводительные бортовые модули, предназначенные для выполнения специфических, вычислительно интенсивных задач цифровой обработки радиолокационных сигналов в реальном или близком к реальному масштабе времени. Их специализация обусловлена необходимостью эффективной реализации конкретных алгоритмов обработки при жестких бортовых ограничениях. Типичная конфигурация РСА предполагает наличие нескольких таких СВ, что позволяет организовать многоканальную или многоэтапную обработку данных. В задачи СВ входит: прием и декодирование управляющих воздействий и параметров от УВМ; непосредственное управление аппаратными компонентами приемо-передающего тракта и антенной системы РСА в процессе зондирования; аналого-цифровое преобразование принятых эхо-сигналов; выполнение алгоритмов цифровой фильтрации, децимации, когерентного накопления и других операций первичной обработки, а также формирование цифровых потоков обработанных радиолокационных данных для их передачи в УВМ.

Взаимодействие между УВМ и группой СВ в бортовой вычислительной системе РСА носит иерархический характер и строго регламентируется протоколом обмена сообщениями. УВМ определяет общую стратегию функционирования РСА и текущий режим съемки, распределяя задачи и параметры между модулями СВ, а также осуществляя контроль их выполнения. СВ, в свою очередь, реализуют непосредственные алгоритмы обработки сигналов и формируют потоки научных или предварительно обработанных данных. Корректная и надежная реализация протокола этого взаимодействия является основополагающим фактором для успешного выполнения целевых задач всем комплексом РСА.

## 1.3 Проблематика отладки и тестирования

Процесс создания и ввода в эксплуатацию бортовой вычислительной системы РСА космического базирования включает строгую последовательность этапов отладки и испытаний программного обеспечения (ПО) ее компонентов. Финальной стадией является установка и проверка работоспособности всего программно-аппаратного комплекса непосредственно в составе изделия верхнего уровня – космического аппарата. Однако до этого момента критически важно обеспечить верификацию как индивидуальной функциональности каждого вычислительного модуля (УВМ и СВ), так и корректности их совместной работы, регламентированной протоколом информационного обмена.

Основная проблематика на этапах наземной отработки заключается в обеспечении надежной и эффективной проверки взаимодействия между УВМ и группой СВ. В условиях параллельной разработки этих компонентов, когда каждый из них может находиться на разной стадии готовности или содержать неустраненные ошибки, проведение прямого интеграционного тестирования часто оказывается неэффективным. При возникновении сбоев в таком «сыром» взаимодействии крайне сложно локализовать источник проблемы: вызвана ли она ошибкой в ПО УВМ, некорректной работой одного из СВ или неточностями в реализации самого протокола.

Попытка отладки непосредственно на полномасштабных аппаратных стендах, включающих реальные модули УВМ и несколько СВ, сопряжена с рядом объективных трудностей. Во-первых, такие стенды являются сложными, дорогостоящими и зачастую уникальными ресурсами, что ограничивает их доступность для разработчиков и не позволяет проводить параллельное тестирование в необходимом объеме. Во-вторых, использование реального оборудования на ранних этапах отладки ПО несет риск его случайного повреждения из-за потенциальных ошибок в управляющих программах. В-третьих, отладка на физическом «железе» не всегда позволяет гибко и полномасштабно воспроизвести все многообразие штатных режимов работы, а также, что особенно важно, смоделировать различные нештатные ситуации и граничные условия, необходимые для проверки робастности системы и алгоритмов отказоустойчивости. Обнаружение и исправление ошибок на поздних стадиях интеграции с реальной аппаратурой неизбежно приводит к существенному увеличению сроков и стоимости разработки всего комплекса РСА.

Для минимизации этих рисков и обеспечения высокого качества разрабатываемого ПО ключевую роль играют имитационные стенды и программные эмуляторы. Перед тем как интегрировать разрабатываемый модуль (например, УВМ) с другими реальными компонентами системы, необходимо провести его всестороннюю проверку с использованием имитаторов смежных систем, работоспособность которых уже подтверждена и верифицирована. В частности, для отладки и испытаний ПО УВМ требуется наличие надежного имитатора, способного адекватно воспроизводить отклик и поведение одного или нескольких СВ в строгом соответствии с протоколом обмена. Такой подход позволяет проводить предварительные испытания в контролируемой и воспроизводимой среде, изолированно проверять работоспособность отдельных модулей и корректность реализации протокола, а также значительно сократить объем и сложность отладочных работ на реальной бортовой аппаратуре.

## 1.4 Постановка цели и задач

Проанализированные в предыдущем подразделе сложности и риски наземной отладки вычислительной системы РСА диктуют необходимость создания специализированного программного инструментария. Такой инструментарий должен позволить проводить всестороннюю проверку и испытания программного обеспечения УВМ в условиях, максимально приближенных к реальным, но без задействования дорогостоящей и не всегда доступной бортовой аппаратуры СВ.

Таким образом, целью настоящей ВКР является разработка программного комплекса, обеспечивающего имитацию процедур управления группой СВ и обмена данными с УВМ, для эффективного решения задач наземной отладки и испытаний.

Для достижения данной цели в ходе ВКР были определены следующие ключевые задачи, подлежащие решению.

Первоочередной задачей являлся детальный анализ требований документа «Протокол взаимодействия бортовых спецвычислителей и управляющей вычислительной машины» для обеспечения точного моделирования предписанных процедур обмена и форматов данных.

Далее следовало спроектировать и разработать архитектуру программного эмулятора СВ, позволяющего имитировать одновременную работу нескольких (до четырех) независимых экземпляров, каждый из которых способен функционировать на выделенном сетевом канале.

Параллельно требовалось разработать архитектуру и реализовать программный модуль УВМ, способный устанавливать и поддерживать асинхронные соединения с группой эмулируемых СВ, отправлять им управляющие команды и корректно обрабатывать получаемые ответы. Важным аспектом являлось обеспечение в эмуляторе СВ возможности гибкой настройки его поведения, включая имитацию различных режимов работы и нештатных ситуаций, что необходимо для полноценного тестирования алгоритмов отказоустойчивости УВМ.

Вся разработка должна была вестись на языке C с использованием стандартных библиотек POSIX для обеспечения переносимости и совместимости с целевыми операционными системами, включая Эльбрус Linux. В качестве дополнительного инструмента для повышения наглядности и удобства анализа процесса взаимодействия, предусматривалась разработка графического интерфейса пользователя (GUI).

Завершающими этапами стали отладка и тестирование разработанного программного комплекса для проверки его функциональности и соответствия поставленным требованиям, а также подготовка комплекта программной документации согласно установленным стандартам.

Успешное решение этих задач позволит создать востребованный программный продукт, способствующий повышению эффективности процесса разработки и испытаний сложных бортовых вычислительных систем.

# 2 АНАЛИЗ ТРЕБОВАНИЙ

## 2.1 Обзор протокола обмена сообщениями УВМ-СВ

### 2.1.1 Назначение и общие принципы протокола

Протокол обмена сообщениями между УВМ и УВМ предназначен для регламентации их информационного взаимодействия в рамках бортовой вычислительной системы РСА. Он определяет набор правил, форматы данных и последовательности обмена, необходимые для выполнения УВМ функций управления группой СВ, а также для получения от СВ результатов обработки и телеметрической информации.

Обмен данными по протоколу носит асинхронный характер. УВМ, как правило, выступает инициатором команд, направляя СВ запросы на выполнение определенных операций, установку режимов работы или передачу параметров. СВ, в свою очередь, обрабатывают эти команды и формируют ответные сообщения, содержащие подтверждения, статусы выполнения или запрошенные данные. Кроме того, протокол предусматривает возможность асинхронной передачи данных от СВ к УВМ, например, при отправке потоков обработанной радиолокационной информации или служебных сообщений о возникновении нештатных ситуаций.

Протокол изначально рассчитан на взаимодействие УВМ с несколькими модулями СВ, что позволяет реализовывать многоканальные конфигурации РСА, где каждый СВ может выполнять свою часть общей задачи по обработке сигналов или отвечать за отдельный сектор обзора. Управление и сбор данных со всей группы СВ координируется централизованно со стороны УВМ.

### 2.1.2 Структура сообщений и типы данных

Фундаментальной единицей информационного обмена в рамках рассматриваемого протокола является сообщение. Каждое сообщение, независимо от его назначения, инкапсулируется в стандартизированный формат, который включает заголовок фиксированного размера и, опционально, тело сообщения переменной длины. Детальная структура сообщения представлена в Таблице 4.1 документа «Протокол взаимодействия бортовых спецвычислителей и управляющей вычислительной машины» (далее – Протокол).

Заголовок сообщения, имеющий общую длину 6 байт, предшествует телу сообщения и несет всю необходимую служебную информацию для его корректной доставки, идентификации и первоначальной обработки принимающей стороной. Поля заголовка располагаются в следующей последовательности:

1. Адрес (Address, 1 байт, uint8): Это поле содержит логический адрес устройства или компонента системы, которому предназначено данное сообщение. Протоколом определены уникальные логические адреса для УВМ и для каждого из СВ, а также для отдельных лучей РСА в соответствующем канале. Это поле обеспечивает маршрутизацию сообщений в многокомпонентной системе. Перечень логических адресов приведен в Таблице 4.2 Протокола.
2. Флаги (Flags, 1 байт, структура битовых полей): Данное поле представляет собой набор из восьми однобитовых флагов, каждый из которых несет специфическую управляющую или информационную нагрузку. Ключевыми флагами, подробно описанными в Таблице 4.3 Протокола, являются:
   * НП (Направление Передачи, бит 0): Определяет направление информационного потока. Значение '0' соответствует передаче от УВМ к СВ, а значение '1' – от СВ к УВМ.
   * НСст8р, НСст9р, НСст10р (биты 1, 2, 3 соответственно): Эти три бита представляют собой старшие разряды (8-й, 9-й и 10-й) полного 11-битного номера сообщения. Использование этих флагов расширяет диапазон нумерации сообщений.
   * Остальные биты (4-7) зарезервированы для будущего использования и должны быть установлены в '0'.
3. Длина тела сообщения (Body Length, 2 байта, uint16): Это поле указывает размер тела сообщения в байтах. Значение может варьироваться от 0 (для сообщений, не содержащих тела) до максимального значения 65522 байт (что соответствует максимальному размеру данных, который может быть инкапсулирован в сообщение, за вычетом размера самого заголовка, если бы тело заполняло все доступное пространство). Данное числовое значение всегда передается и интерпретируется в сетевом порядке байт (Big-Endian), и для корректной работы на платформах с другим порядком байт требуется соответствующее преобразование.
4. Номер сообщения (Message Number, 1 байт, uint8): Содержит младшие 8 бит (разряды 0-7) порядкового номера сообщения. Отправитель нумерует сообщения циклически, начиная с 0. Полный 11-битный номер сообщения формируется путем комбинации этого байта со старшими тремя битами из поля «Флаги». Эта нумерация критически важна для обеспечения контроля последовательности доставки сообщений, обнаружения потерь или дубликатов, а также для механизма подтверждения доставки (если таковой реализуется на более высоком уровне или требуется протоколом для определенных типов обмена).
5. Тип сообщения (Message Type, 1 байт, uint8): Является идентификатором, который однозначно определяет назначение сообщения и, следовательно, структуру и семантику его тела. Каждому типу операции или передаваемых данных в протоколе соответствует уникальный код типа сообщения. Полный перечень определенных типов сообщений и их кодов представлен в Таблице 4.4 Протокола.

Тело сообщения (Message Body) следует непосредственно за заголовком. Его наличие и структура полностью определяются полем «Тип сообщения». Если «Длина тела сообщения» в заголовке равна нулю, тело отсутствует. В противном случае, оно содержит полезную нагрузку – данные, параметры или команды, специфичные для данного типа сообщения. Максимальный размер тела ограничен значением 65522 байт.  
Протокол предусматривает использование в теле сообщений различных фундаментальных и составных типов данных, как указано в примечаниях к Таблице 4.1 Протокола. К ним относятся:

* Целочисленные типы без знака (uint(k)) и со знаком (int(k)) различной разрядности k (например, uint8, int16, uint32).
* Числа с фиксированной точкой без знака (ufixed(k)) и со знаком (fixed(k)), где положение двоичной точки предопределено.
* Комплексные числа, представленные парой действительных и мнимых частей, каждая из которых может быть целым числом (complex int(k)) или числом с фиксированной точкой (complex fixed(k)).
* Однобитовые флаги (bit).
* Массивы перечисленных выше типов данных.
* Структуры, объединяющие поля различных типов.  
  Порядок байт для многобайтовых числовых полей в теле сообщения (например, uint16, int32, компоненты комплексных чисел) также должен соответствовать сетевому порядку (Big-Endian), если иное не оговорено для конкретного типа сообщения.

### 2.1.3 Классификация и основные типы сообщений

Протокол взаимодействия УВМ и СВ регламентирует структурированный и многоэтапный обмен данными, который осуществляется с помощью стандартизированных сообщений. Каждый этап этого обмена включает специфический набор сообщений, предназначенных для выполнения конкретных функций системы. Полный перечень типов сообщений, их идентификаторов и максимальных размеров приведен в Таблице 4.4 документа «Протокол взаимодействия...».

Первоначальным является этап включение и инициализация канала связи, подробно иллюстрированный на Рисунке 3.1 Протокола. После установления физического соединения УВМ, как правило, инициирует обмен на прикладном уровне, отправляя сообщение «Инициализация канала» (тип 128). Это сообщение предназначено для назначения или подтверждения уникального логического адреса (LAK) конкретному экземпляру СВ, что необходимо для его однозначной идентификации в системе. Также оно содержит логический адрес самой УВМ (LAUVM) и служит сигналом для СВ к запуску его внутренних служб, включая счетчики времени и состояния. В ответ на успешную инициализацию СВ передает УВМ сообщение «Подтверждение инициализации канала» (тип 129). Данное сообщение не только подтверждает принятие назначенного LAK, но и предоставляет УВМ важную диагностическую информацию: версии программного обеспечения ПЛИС различных модулей СВ (ВДР, ВОР1, ВОР2), состояние линий передачи (СЛП) и текущее значение счетчика времени наработки СВ (ВСВ).

После успешной инициализации канала следует этап подготовки к сеансу наблюдения, описанный на Рисунке 3.2 Протокола. На данной стадии УВМ осуществляет проверку готовности СВ к выполнению целевых задач. Для этого УВМ может инициировать процедуру самодиагностики СВ командой «Провести контроль» (тип 1), указывая в теле сообщения тип запрашиваемого контроля (ТК). СВ незамедлительно подтверждает получение этой команды сообщением «Подтверждение контроля» (тип 3), возвращая свой LAK, тип запрошенного контроля и текущее значение ВСВ. По завершении процедур самоконтроля, УВМ запрашивает их результаты командой «Выдать результаты контроля» (тип 2), в теле которой указывается вид запрашиваемых результатов (ВРК). В ответ СВ передает сообщение «Результаты контроля» (тип 4), содержащее его LAK, битовую маску результатов самоконтроля (РСК), детализирующую состояние различных узлов и подсистем, время, затраченное на самоконтроль (ВСК), и текущее значение ВСВ. Кроме того, для оценки качества физического канала связи, УВМ может отправить команду «Выдать состояние линии» (тип 6) (с пустым телом), на что СВ отвечает сообщением «Состояние линии» (тип 7), передавая текущее значение счётчика ВСВ.

Следующий критически важный этап – подготовка к сеансу съемки, который может различаться в зависимости от выбранного режима работы РСА (ОР, ДР или ВР), как показано на Рисунках 3.3-3.5 Протокола. На этом этапе УВМ загружает в СВ большой объем конфигурационных параметров, определяющих алгоритмы обработки радиолокационных данных. Ключевыми сообщениями здесь являются команды типа «Принять параметры...». Например, «Принять параметры СО» (тип 160) используется для настройки обзорных режимов (ОР, ВР) и содержит такие параметры, как режим работы РСА (РР), маска бланкирования лучей (БРЛ), пороговые константы (Q0, Q, KNK), коэффициенты взвешивающего фильтра (Weight), длины опор свертки (L1-L3), параметры АРУ и уровень обработки (УО). Для режима детального разрешения (ДР) УВМ отправляет «Принять параметры СДР» (тип 170), содержащее специфичные настройки для синтеза данных с высоким разрешением, включая параметры прореживания, взвешивания, БПФ, а также может включать массив опоры по дальности (HRR), что делает это сообщение потенциально большим. Для режима ОР, использующего третичную цифровую обработку (3ЦО), передается сообщение «Принять параметры 3ЦО» (тип 200) с параметрами диаграмм направленности антенны (DNA), нормализованными константами и информацией о береговой линии.

Также на этом этапе могут загружаться большие массивы опорных сигналов, например, через сообщения «Принять TIME\_REF\_RANGE» (тип 161) или «Принять REF\_AZIMUTH» (тип 201), тело которых может достигать десятков килобайт. Завершает подготовку к съемке передача актуальных «Навигационных данных» (тип 255). Протокол на данном этапе, как правило, не требует от СВ явных подтверждений на каждое принятое конфигурационное сообщение.

Непосредственно сеанс съемки (Рисунки 3.6-3.8 Протокола) инициируется управляющим воздействием со стороны УВМ. В ходе сеанса СВ осуществляет сбор и обработку радиолокационных данных в соответствии с ранее загруженными параметрами и начинает их потоковую передачу в УВМ. Основной поток научных данных от СВ к УВМ передается специализированными сообщениями. К ним относятся, например, «СУБК» (тип 127 – Субкадр Одного Канала), содержащий комплексные отсчеты сигнала после внутриимпульсной свертки, «КО» (тип 137 – Кадр Обработки) для режима ДР, «Строка голограммы СУБК» (тип 8), «Строка радиоголограммы ДР» (тип 18), «Строка К3» (тип 19) или «Строка изображения К4» (тип 20). Именно эти типы сообщений характеризуются потенциально большим объемом тела, который может достигать максимального значения, определенного протоколом (65522 байт), и требуют эффективных механизмов передачи и приема на стороне УВМ. Также во время сеанса съемки СВ может передавать предварительные результаты обнаружения, такие как «НК» (тип 80 – Обнаруженные Неподвижные Контрастные объекты) и «Помеха» (тип 81) при обнаружении активной помехи. В случае возникновения нештатных ситуаций, СВ асинхронно информирует УВМ сообщением «Предупреждение» (тип 254), содержащим код типа критического события (ТКС).

Завершение сеанса съемки (Рисунок 3.9 Протокола) также инициируется УВМ. После этого могут быть выполнены финальные процедуры контроля состояния СВ, аналогичные этапу подготовки к сеансу наблюдения, для оценки его работоспособности после выполнения целевой задачи.

Такая детализированная последовательность этапов и соответствующих им сообщений обеспечивает полный и гибкий цикл управления специализированными вычислителями и получение от них всей необходимой информации для функционирования РСА.

## 2.2 Выбор технологий и средств разработки

Выбор технологического стека для реализации программного комплекса имитации и управления процедурами взаимодействия УВМ и СВ основывался на ряде ключевых требований, предъявляемых к подобным системам, включая производительность, надежность, переносимость и возможность функционирования на целевых отечественных платформах.

В качестве основного языка программирования для разработки модулей эмулятора СВ и управляющей машины УВМ был определен язык C. Этот выбор обусловлен его высокой эффективностью, предоставляемыми возможностями для низкоуровневого управления системными ресурсами и памятью, что является критически важным для приложений, функционирующих в условиях, приближенных к реальному времени. Кроме того, язык C обладает широкой поддержкой и доступностью компиляторов для различных операционных систем, включая ОС Эльбрус Linux, что соответствует требованиям к переносимости и использованию на защищенных платформах.

Целевой операционной системой для функционирования разрабатываемого комплекса выбрана ОС семейства Linux. Данная среда предоставляет богатый набор системных вызовов и API, соответствующих стандарту POSIX, что закладывает основу для создания переносимого программного обеспечения.

Для обеспечения асинхронной обработки нескольких соединений и параллельного выполнения задач в модулях УВМ и СВ была выбрана стандартная библиотека POSIX Threads (Pthreads). Данная библиотека предоставляет стандартизированный интерфейс для создания и управления потоками, а также необходимые механизмы синхронизации (мьютексы, условные переменные), что позволит реализовать требуемую многопоточную архитектуру.

Для сетевого взаимодействия между модулем УВМ и эмулируемыми экземплярами СВ был выбран протокол TCP/IP. Выбор данного протокола продиктован необходимостью обеспечения гарантированной доставки команд управления и критически важных данных, а также контроля целостности информации и сохранения порядка следования сообщений. Для реализации сетевых функций будет использован стандартный интерфейс сокетов Беркли (BSD sockets). Архитектурно модуль ввода-вывода будет спроектирован с возможностью поддержки и других транспортных механизмов, например, обмена через последовательный порт, что обеспечит гибкость решения.

Управление конфигурационными параметрами комплекса, такими как сетевые настройки, логические адреса и параметры имитации различных сценариев работы, решено осуществлять через внешние INI-файлы. Для их парсинга была выбрана легковесная библиотека inih, обеспечивающая простоту интеграции и использования.

Для визуализации процесса взаимодействия и предоставления пользователю средств мониторинга и логгирования предусмотрена разработка отдельного графического интерфейса пользователя (GUI). В качестве инструмента для его создания выбран фреймворк Qt и язык программирования C++, благодаря их кроссплатформенности, обширному набору компонентов интерфейса и развитым средствам для сетевого взаимодействия.

Основными инструментами на этапе разработки будут служить компилятор GCC, система сборки GNU Make и отладчик GDB. Для разработки графического интерфейса будет использоваться среда Qt Creator.

# 3 РЕАЛИЗАЦИЯ ПРОГРАММНОГО ИНТЕРФЕЙСА

## 3.1 Архитектура разработанного программного комплекса

Разработанный в рамках преддипломной практики программный комплекс предназначен для имитации и управления взаимодействием между УВМ и группой СВ. Комплекс состоит из трех основных, независимо запускаемых программных приложений: эмулятора СВ (svm\_app), модуля УВМ (uvm\_app) и графического интерфейса пользователя (gui\_app), взаимодействие которых представлено на Рисунке 1

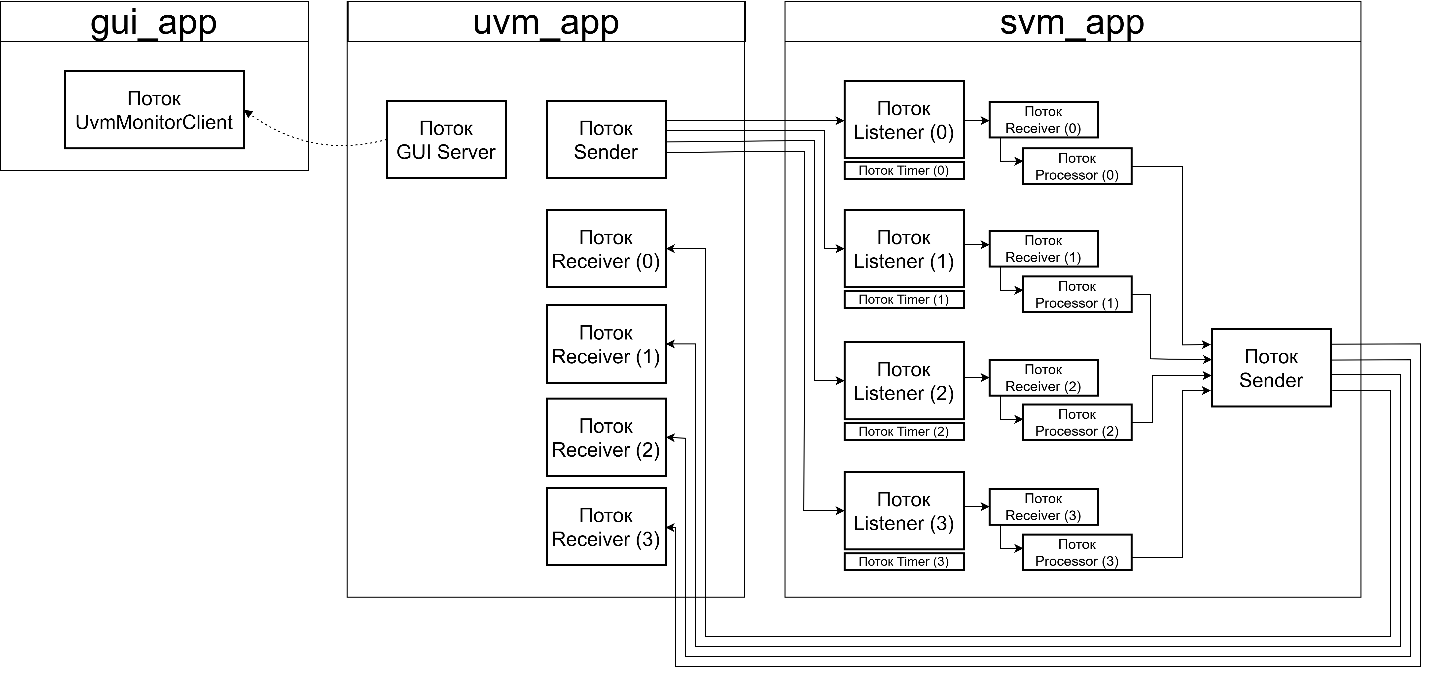


Рисунок 1 – Архитектура программного комплекса

Приложение svm\_app функционирует как единый процесс, архитектура которого построена на многопоточности для обеспечения эмуляции до четырех независимых экземпляров СВ. Каждый такой эмулируемый экземпляр СВ, состояние которого хранится в индивидуальной структуре SvmInstance, идентифицируется уникальным логическим адресом (LAK) и прослушивает выделенный ему сетевой TCP-порт. Для каждого сконфигурированного порта основной поток svm\_app запускает отдельный поток-слушатель (Listener). Задача потока-слушателя – ожидать входящее TCP-соединение от УВМ. При установлении соединения, Listener активирует соответствующий экземпляр SvmInstance и порождает для него три персональных рабочих потока: поток-приемник (Receiver), поток-обработчик (Processor) и персональный поток-таймер (InstanceTimer). Поток-приемник асинхронно считывает команды, поступающие от УВМ по установленному соединению, и помещает их, обернув в структуру QueuedMessage (содержащую идентификатор экземпляра), в персональную потокобезопасную входящую очередь данного экземпляра. Поток-обработчик извлекает команды из этой очереди, вызывает соответствующую функцию-обработчик из набора svm\_handlers (которая реализует логику ответа СВ, включая имитацию сбоев на основе параметров из SvmInstance), и, если протоколом предусмотрен ответ, помещает сформированное ответное сообщение в общую для всех экземпляров потокобезопасную исходящую очередь. Обслуживанием этой исходящей очереди занимается единый общий поток-отправитель (Sender), который извлекает сообщения и направляет их соответствующим подключенным УВМ. Персональный поток-таймер (InstanceTimer) для каждого активного экземпляра СВ обеспечивает периодическое обновление его внутренних счетчиков времени и состояния (например, BCB, счетчики состояния линии связи), доступ к которым синхронизирован индивидуальным мьютексом экземпляра. Такая архитектура позволяет одному процессу svm\_app эффективно и независимо обслуживать несколько сессий взаимодействия, при этом каждый эмулируемый СВ имеет собственные независимые временные характеристики.

Приложение uvm\_app реализует логику Управляющей Вычислительной Машины и также является многопоточным. Его основной поток отвечает за инициализацию, чтение конфигурации (определяющей параметры подключения к каждому из четырех СВ) и установку TCP-соединений. Для каждого успешно установленного соединения с экземпляром СВ запускается индивидуальный поток-приемник (Receiver). Каждый такой Receiver асинхронно ожидает и считывает ответы и данные, поступающие от «своего» СВ, после чего помещает их (в виде структуры UvmResponseMessage, содержащей идентификатор источника) в общую потокобезопасную очередь входящих ответов. Формирование команд для СВ происходит в основном потоке uvm\_app; эти команды (в виде структур UvmRequest, содержащих целевой ID СВ и само сообщение) помещаются в общую потокобезопасную очередь исходящих запросов. Единый общий поток-отправитель (Sender) извлекает запросы из этой очереди и передает их соответствующему СВ по установленному TCP-соединению. Важной частью uvm\_app является поток-сервер GUI (GUI Server), который создает TCP-сервер на предопределенном локальном порту (12345) и ожидает подключения графического интерфейса. При установлении соединения с GUI, а также в процессе работы УВМ (при отправке команд, получении ответов, изменении статуса соединений или обнаружении ошибок), основной поток uvm\_app через специальную функцию транслирует актуальную информацию о состоянии всех взаимодействий в gui\_app. Кроме того, основной поток uvm\_app реализует логику управления протоколом, включая выбор режима работы РСА, и базовые механизмы отказоустойчивости, такие как Keep-Alive для обнаружения «молчащих» СВ и обработка ошибок связи.

Графический интерфейс пользователя gui\_app, разработанный с использованием фреймворка Qt на языке C++, функционирует как TCP-клиент к приложению uvm\_app. Он подключается к uvm\_app, получает от него поток структурированных текстовых сообщений, описывающих события отправки/получения протокольных сообщений и изменения статусов. GUI парсит эту информацию и отображает ее пользователю в реальном времени в виде табличных логов для каждого из четырех СВ, а также выводит их текущий статус, логические адреса и ключевые параметры, такие как счетчик BCB. Это обеспечивает наглядный мониторинг и инструмент для анализа процесса имитации.

Для обеспечения логической организации и переиспользования кода, исходный код всего программного комплекса структурирован по функциональным модулям, представленным отдельными директориями.

Модуль protocol инкапсулирует все определения, специфичные для протокола обмена УВМ-СВ. Он включает детальные структуры заголовков и тел всех типов сообщений, перечисления кодов сообщений и логических адресов, а также все необходимые константы, регламентированные документом протокола. Важной частью этого модуля являются функции-конструкторы (message\_builder), предназначенные для формирования стандартизированных протокольных сообщений с корректно заполненными заголовками и, в случае сообщений от СВ к УВМ, телами. Также в модуль входят утилиты (message\_utils) для выполнения вспомогательных операций над сообщениями, таких как преобразование порядка байт между сетевым и хостовым представлением для многобайтовых полей и извлечение полного номера сообщения из его компонент.

Модуль io предоставляет абстрактный интерфейс ввода-вывода (IOInterface), который унифицирует операции сетевого взаимодействия. Он определяет набор стандартных функций (таких как подключение, прослушивание, прием и передача данных, разрыв соединения) и скрывает детали их реализации для конкретного транспортного протокола. В текущей версии основная реализация интерфейса (io\_ethernet.c) обеспечивает работу по протоколу TCP/IP с использованием сокетов Беркли. Архитектура также включает заготовку для поддержки обмена через последовательный порт (io\_serial.c), что демонстрирует гибкость интерфейса для возможного расширения. Общие функции для отправки и приема полных протокольных сообщений (io\_common.c), работающие поверх абстрактного IO-интерфейса, также являются частью этого модуля.

Модуль config отвечает за считывание и парсинг конфигурационных данных из текстовых файлов формата INI. Для этой цели используется легковесная сторонняя библиотека inih. Данный модуль позволяет гибко настраивать множество параметров работы приложений uvm\_app и svm\_app без необходимости их перекомпиляции. Это включает сетевые адреса и порты для каждого эмулируемого экземпляра СВ, их логические адреса, а также параметры для имитации различных нештатных ситуаций и режимов сбоя на стороне СВ.

Модуль utils содержит набор вспомогательных утилит, используемых в различных частях проекта. Ключевыми компонентами этого модуля являются реализации потокобезопасных очередей (ts\_queue для стандартных протокольных сообщений Message, ts\_queue\_req для структур запросов UvmRequest в uvm\_app, и ts\_uvm\_resp\_queue для структур ответов UvmResponseMessage в uvm\_app, а также ts\_queued\_msg\_queue для структур QueuedMessage в svm\_app). Эти очереди, реализованные с использованием мьютексов и условных переменных Pthreads, являются основой для организации асинхронного и безопасного обмена данными между различными потоками в многопоточных приложениях uvm\_app и svm\_app.

Логика, специфичная для эмулятора СВ, включая управление состоянием экземпляров и обработку команд, сосредоточена в модуле svm. Аналогично, модуль uvm содержит всю специфику реализации УВМ. Наконец, модуль gui содержит весь исходный код Qt-приложения для графического интерфейса пользователя. Подробнее эти модули мы рассмотрим далее.

Такая модульная архитектура и четкое разделение ответственности между компонентами способствуют лучшей читаемости кода, упрощают его тестирование и отладку, а также создают прочную основу для дальнейшего сопровождения и возможного расширения функциональности разработанного программного комплекса.

## 3.2 Программный модуль СВ

Программный модуль svm\_app разработан для обеспечения имитации функционирования группы (до четырех) независимых экземпляров Специализированных Вычислителей (СВ). Ключевой особенностью его архитектуры является реализация всей логики эмуляции в рамках одного процесса операционной системы, что упрощает развертывание и управление на испытательном стенде. Несмотря на работу в едином адресном пространстве, каждый эмулируемый экземпляр СВ функционирует логически изолированно, взаимодействуя с УВМ через свой собственный, уникальный сетевой TCP-порт.

### 3.2.1 Общая архитектура и управление экземплярами svm\_app

Центральным элементом управления состоянием каждого эмулируемого СВ является структура данных SvmInstance. Для каждого потенциального экземпляра (в текущей реализации до четырех) в приложении создается свой объект этой структуры. SvmInstance агрегирует всю информацию, необходимую для индивидуальной имитации поведения конкретного СВ, включая его уникальный идентификатор (ID), назначенный логический адрес (LAK), текущее внутреннее состояние (например, STATE\_NOT\_INITIALIZED, STATE\_INITIALIZED, STATE\_SELF\_TEST), персональные счетчики (такие как счетчик времени наработки BCB и счетчики состояния линии связи), а также флаги и параметры для моделирования различных нештатных ситуаций, считываемые из конфигурационного файла. Кроме того, структура SvmInstance содержит указатель на персональную потокобезопасную входящую очередь сообщений, дескриптор активного клиентского соединения (если оно установлено с УВМ) и мьютекс для синхронизации доступа к своим данным из разных потоков.

Основной поток приложения svm\_app, реализуемый в функции main файла svm\_main.c, выполняет первоначальную настройку и запуск всех компонентов эмулятора. При старте происходит инициализация глобальных ресурсов, таких как общая исходящая очередь сообщений, а также инициализация генератора случайных чисел, используемого в персональных таймерах для эмуляции изменений состояния линии. Затем считывается конфигурация из файла config.ini, на основе которой определяется количество эмулируемых экземпляров СВ и их индивидуальные параметры (порт, LAK, настройки имитации сбоев). Для каждого сконфигурированного экземпляра инициализируется соответствующая структура SvmInstance путем копирования в нее прочитанных из файла настроек.

Важным шагом является запуск отдельных потоков-слушателей (listener\_thread\_func), по одному на каждый TCP-порт, указанный в конфигурации для эмулируемых СВ. Эти потоки отвечают за прием входящих соединений от УВМ. После инициализации слушателей, основной поток запускает один общий сервисный поток: поток-отправитель (sender\_thread\_func), обслуживающий отправку всех ответных сообщений от эмулируемых экземпляров СВ. Завершив инициализацию, основной поток переходит в режим ожидания сигнала завершения работы (например, Ctrl+C), обеспечивая фоновое функционирование всех эмулирующих потоков.

### 3.2.2 Реализация потоков-слушателей (listener\_thread\_func)

Для обеспечения независимого прослушивания каждым эмулируемым экземпляром СВ своего уникального сетевого порта, в svm\_app для каждого сконфигурированного экземпляра создается и запускается отдельный поток-слушатель, выполняющий функцию listener\_thread\_func. При запуске каждому такому потоку передается структура ListenerArgs, содержащая его идентификатор (ID), назначенный порт для прослушивания и логический адрес (LAK), которые были считаны из конфигурационного файла.

В начале своей работы поток-слушатель, используя переданный ему номер порта, создает и инициализирует собственный экземпляр интерфейса ввода-вывода (IOInterface), специализированный для работы с TCP/IP. Затем, с помощью метода listen этого интерфейса, создается серверный сокет, который привязывается к указанному порту и переводится в режим прослушивания входящих соединений. Дескриптор этого слушающего сокета сохраняется в глобальном массиве для возможности его корректного закрытия при общем завершении работы приложения.

Основная логика потока-слушателя заключена в цикле, который продолжается до тех пор, пока установлен глобальный флаг keep\_running. В каждой итерации цикла поток вызывает блокирующую функцию accept на своем слушающем сокете, ожидая попытки подключения со стороны УВМ. При успешном принятии входящего соединения (возвращается дескриптор client\_handle), поток-слушатель приступает к активации соответствующего ему экземпляра СВ, доступ к которому осуществляется через глобальный массив svm\_instances по svm\_id.

Перед активацией и запуском рабочих потоков производится блокировка мьютекса, ассоциированного с данным экземпляром SvmInstance, для обеспечения потокобезопасного изменения его состояния. Проверяется флаг is\_active; если экземпляр по какой-либо причине уже активен, новое соединение отклоняется, и сокет клиента закрывается. В штатной ситуации для неактивного экземпляра происходит его настройка: в структуру SvmInstance сохраняется дескриптор client\_handle и указатель на IOInterface, созданный данным потоком-слушателем (этот интерфейс будет использоваться рабочими потоками для операций send и recv). Также сбрасываются счетчики сообщений и состояния экземпляра, инициализируется его персональная потокобезопасная входящая очередь сообщений (ts\_queued\_msg\_queue) вызовом qmq\_create().

После подготовительных операций поток-слушатель запускает три дочерних рабочих потока для обслуживания установленного соединения: поток-приемник (receiver\_thread\_func), поток-обработчик (processor\_thread\_func) и персональный поток-таймер (svm\_instance\_timer\_thread\_func), отвечающий за обновление счетчиков конкретного экземпляра СВ. Обоим потокам в качестве аргумента передается указатель на структуру SvmInstance данного экземпляра, что обеспечивает им доступ ко всему его состоянию и ресурсам. После успешного запуска рабочих потоков, флаг instance→is\_active устанавливается в true, и мьютекс экземпляра освобождается. С этого момента экземпляр СВ считается активным и готовым к обмену сообщениями с УВМ.

Запустив рабочие потоки, поток-слушатель переходит в состояние ожидания их завершения, последовательно вызывая pthread\_join сначала для потока-приемника, затем для потока-обработчика, и, наконец, для персонального потока-таймера. Перед ожиданием завершения потока-таймера, поток-слушатель устанавливает специальный флаг (personal\_timer\_keep\_running в структуре SvmInstance) в false, сигнализируя таймеру о необходимости корректного завершения своего цикла. Завершение этих потоков сигнализирует об окончании клиентской сессии (например, УВМ разорвал соединение, или сработала имитация отключения на стороне СВ). После возврата из pthread\_join, поток-слушатель снова захватывает мьютекс экземпляра, выполняет очистку ресурсов: закрывает клиентский сокет client\_handle, уничтожает персональную входящую очередь вызовом qmq\_destroy(), сбрасывает флаг is\_active в false и обнуляет идентификаторы завершившихся рабочих потоков, включая идентификатор потока-таймера.. После этого поток-слушатель освобождает мьютекс и возвращается к началу своего основного цикла, снова вызывая accept для ожидания нового подключения на своем порту.

Если в процессе ожидания на accept или при другой операции глобальный флаг keep\_running становится false (например, по сигналу Ctrl+C), цикл потока-слушателя прерывается. Перед завершением поток-слушатель закрывает свой слушающий сокет и используемый им IOInterface, освобождая занятые сетевые ресурсы.

### 3.2.3 Реализация рабочих потоков экземпляра СВ (Receiver, Processor и InstanceTimer)

Для каждого установленного TCP-соединения с УВМ и соответствующего ему активного экземпляра СВ (SvmInstance) поток-слушатель (listener\_thread\_func) порождает три специализированных персональных рабочих потока: поток-приемник (receiver\_thread\_func), поток-обработчик (processor\_thread\_func) и персональный поток-таймер (svm\_instance\_timer\_thread\_func). Эти три потока функционируют согласованно, обеспечивая полный цикл обработки команд от УВМ, формирование ответов и имитацию внутренней жизнедеятельности эмулируемого СВ.

Поток-приемник (receiver\_thread\_func) получает при запуске указатель на структуру SvmInstance, с которой он будет работать. Его основная задача – непрерывное чтение данных из сетевого сокета, связанного с данным экземпляром (instance→client\_handle), используя функции модуля ввода-вывода (instance→io\_handle). В основном цикле, который продолжается, пока установлен общий флаг работы приложения keep\_running и данный экземпляр СВ помечен как активный (instance→is\_active), поток вызывает функцию receive\_protocol\_message. В случае успешного приема полного протокольного сообщения (Message), оно упаковывается в структуру QueuedMessage, к которой добавляется идентификатор текущего экземпляра СВ (instance→id). Эта QueuedMessage затем помещается в персональную входящую потокобезопасную очередь instance→incoming\_queue с помощью вызова qmq\_enqueue(). Если в процессе чтения из сокета происходит ошибка или обнаруживается, что соединение было закрыто удаленной стороной (УВМ), поток-приемник корректно обрабатывает эту ситуацию: он вызывает функцию qmq\_shutdown() для своей входящей очереди, чтобы сигнализировать потоку-обработчику об отсутствии новых сообщений, и после этого завершает свое выполнение. Обработка прерываний системных вызовов (EINTR) также предусмотрена для обеспечения стабильной работы.

Поток-обработчик (processor\_thread\_func), также получающий указатель на SvmInstance при старте, отвечает за логическую обработку команд, полученных от УВМ. Он работает в цикле, извлекая сообщения (QueuedMessage) из персональной входящей очереди instance→incoming\_queue с помощью блокирующего вызова qmq\_dequeue(). Если очередь пуста, поток ожидает поступления новых сообщений. Если qmq\_dequeue() возвращает неуспех, это означает, что очередь была закрыта (потоком-приемником или при общем завершении приложения) и в ней больше нет данных, в этом случае поток-обработчик корректно завершает свою работу. После успешного извлечения QueuedMessage, поток-обработчик определяет тип полученного протокольного сообщения (queued\_msg.message.header.message\_type) и находит соответствующую функцию-обработчик в глобальном массиве message\_handlers. Если обработчик для данного типа сообщения найден, он вызывается с передачей указателя на текущий SvmInstance и указателя на само сообщение (&queued\_msg.message).

Функции-обработчики, реализованные в модуле svm\_handlers.c, содержат всю специфическую логику ответа СВ на команду УВМ. Они могут модифицировать состояние SvmInstance (например, instance→current\_state, инкрементировать instance→message\_counter), получать значения счетчиков (например, get\_instance\_bcb\_counter(instance)) и, что важно для тестирования, активировать имитацию различных нештатных ситуаций (ошибку контроля, задержку ответа, отправку предупреждения) на основе флагов, хранящихся в instance (например, instance→simulate\_control\_failure).  
Если функция-обработчик сформировала ответное сообщение (возвращает указатель на динамически выделенную структуру Message), поток-обработчик упаковывает его в новую QueuedMessage (снова добавляя instance→id для корректной маршрутизации на стороне общего отправителя), освобождает память, выделенную под исходный ответ функцией-обработчиком, и помещает QueuedMessage с ответом в общую для всех экземпляров СВ исходящую очередь svm\_outgoing\_queue. Если для полученного типа сообщения не найден обработчик, или если обработчик не предполагает ответа (возвращает NULL), никаких действий по отправке не предпринимается, и соответствующая информация логируется.

Персональный поток-таймер (svm\_instance\_timer\_thread\_func) создается для каждого активного экземпляра SvmInstance и отвечает за имитацию внутренних периодических процессов и обновление счетчиков времени и состояния именно этого экземпляра. При запуске поток получает указатель на свой SvmInstance. Он работает в цикле, управляемом флагом instance→personal\_timer\_keep\_running (устанавливаемым потоком-слушателем) и общим флагом приложения keep\_running. В каждой итерации поток приостанавливается на предопределенный интервал (например, 50 миллисекунд для обновления счетчика времени наработки BCB) с помощью функции usleep(). После пробуждения поток-таймер захватывает индивидуальный мьютекс instance→instance\_mutex для потокобезопасного доступа к данным своего экземпляра. Он инкрементирует счетчик instance→bcb\_counter. После обновления счетчика мьютекс экземпляра освобождается. Поток-таймер продолжает свою работу до тех пор, пока один из управляющих флагов не будет сброшен, после чего он корректно завершается. Поток-слушатель дожидается его завершения с помощью pthread\_join.

Такая организация с тремя персональными потоками на каждый экземпляр СВ позволяет изолировать логику приема, обработки команд и обновления состояния для каждого эмулируемого вычислителя, обеспечивая их независимое и асинхронное функционирование в рамках единого процесса svm\_app.

### 3.2.4 Реализация общего сервисного потока СВ (Sender)

Помимо потоков, персонально обслуживающих каждый активный экземпляр СВ, в архитектуре svm\_app присутствует один общий сервисный поток-отправитель (sender\_thread\_func). Этот поток запускается один раз при старте приложения и функционирует на протяжении всего времени его работы, отвечая за централизованную отправку всех ответных сообщений от эмулируемых СВ к соответствующим УВМ.

Общий поток-отправитель работает в цикле, извлекая сообщения типа QueuedMessage из единой для всех экземпляров потокобезопасной исходящей очереди svm\_outgoing\_queue. Каждая структура QueuedMessage содержит как само протокольное сообщение Message для отправки, так и идентификатор (instance\_id) того экземпляра СВ, который сформировал этот ответ. Используя instance\_id, поток-отправитель обращается к глобальному массиву svm\_instances для получения указателя на соответствующий объект SvmInstance. Из этого объекта извлекаются актуальный дескриптор клиентского соединения (client\_handle) и указатель на объект интерфейса ввода-вывода (io\_handle), который был создан потоком-слушателем для данного экземпляра при установлении TCP-соединения.

Перед отправкой производится проверка флага instance→is\_active; если экземпляр СВ по какой-либо причине уже неактивен (например, УВМ разорвал соединение, или сработала имитация отключения), сообщение для него отбрасывается. В противном случае, для отправки протокольного сообщения используется функция send\_protocol\_message из модуля io\_common, которая инкапсулирует детали сетевого взаимодействия.

Также особенностью данного потока является реализация механизма имитации принудительного разрыва соединения со стороны СВ, настраиваемого через конфигурационный файл. Для каждого экземпляра SvmInstance в его структуре хранятся параметр disconnect\_after\_messages и счетчик уже отправленных им сообщений messages\_sent\_count. Перед каждой фактической отправкой ответного сообщения этот счетчик для соответствующего экземпляра инкрементируется. Если количество отправленных сообщений данным экземпляром достигает или превышает значение disconnect\_after\_messages (и это значение сконфигурировано как положительное число), поток-отправитель инициирует процедуру имитации отключения для этого конкретного экземпляра СВ. В этом случае он помечает данный SvmInstance как неактивный (устанавливая флаг is\_active в false), вызывает системную функцию shutdown() для client\_handle этого экземпляра (для корректного разрыва TCP-соединения с точки зрения УВМ) и закрывает персональную входящую очередь сообщений данного экземпляра (qmq\_shutdown()), чтобы сигнализировать его потоку-обработчику (processor\_thread\_func) о необходимости штатного завершения. Обработка ошибок, которые могут быть возвращены функцией send\_protocol\_message (например, при разрыве соединения со стороны УВМ), также приводит к деактивации соответствующего экземпляра СВ и инициированию закрытия связанных с ним ресурсов.

### 3.2.5 Обработка сообщений и имитация поведения СВ (svm\_handlers.c)

Логика реакции эмулируемых экземпляров СВ на команды, поступающие от УВМ, а также реализация имитации их различного поведения, включая нештатные ситуации, сосредоточена в модуле svm\_handlers.c. Основой механизма обработки является диспетчеризация входящих сообщений на основе их типа.

При инициализации приложения svm\_app вызывается функция init\_message\_handlers(), которая заполняет глобальный массив указателей на функции message\_handlers. Индексом этого массива служит числовой код типа сообщения (от 0 до 255), а значением – указатель на соответствующую функцию-обработчик, ответственную за данный тип сообщения. Когда поток-обработчик (processor\_thread\_func) экземпляра СВ извлекает очередное сообщение из своей входящей очереди, он использует поле message\_type из заголовка полученного сообщения для определения и вызова нужной функции из массива message\_handlers.

Каждая функция-обработчик, например handle\_init\_channel\_message или handle\_vydat\_rezultaty\_kontrolya\_message, имеет стандартизированную сигнатуру, принимая в качестве аргументов указатель на структуру SvmInstance (соответствующую текущему эмулируемому СВ) и указатель на полученное от УВМ сообщение. Передача указателя на SvmInstance является ключевым моментом, так как это позволяет обработчику оперировать состоянием и параметрами конкретного экземпляра. Обработчик может считывать и модифицировать поля SvmInstance, такие как current\_state (текущее состояние СВ), assigned\_lak (его логический адрес), использовать персональный счетчик исходящих сообщений message\_counter или запрашивать значения других счетчиков, например, BCB, через функцию get\_instance\_bcb\_counter(instance). Доступ к полям SvmInstance, изменяемым из нескольких потоков (например, current\_state или счетчики, обновляемые потоком-таймером), защищается персональным мьютексом данного экземпляра. По результатам обработки команда УВМ функция-обработчик может сформировать ответное сообщение, для чего динамически выделяется память под структуру Message. Указатель на это сообщение (или NULL, если ответ не предусмотрен протоколом или имитируется его отсутствие) возвращается в вызвавший поток-обработчик, который затем помещает его в общую исходящую очередь, предварительно освободив память, если ответ был сформирован.

Реализация логики протокола в обработчиках точно следует спецификациям. Например, при получении сообщения «Инициализация канала», функция handle\_init\_channel\_message считывает из тела запроса предлагаемый УВМ логический адрес, сверяет его с уже назначенным данному экземпляру instance→assigned\_lak (который был установлен из конфигурационного файла при запуске потока-слушателя), и формирует ответное сообщение «Подтверждение инициализации канала». В тело этого ответа помещаются подтвержденный LAK экземпляра, актуальные версии прошивок (имитируемые значения) и текущее значение счетчика BCB, полученное от потока-таймера. После этого внутреннее состояние экземпляра instance→current\_state переводится в STATE\_INITIALIZED. Аналогично, обработчик handle\_vydat\_rezultaty\_kontrolya\_message имитирует сбор данных о самоконтроле и формирует ответ, содержащий поле RSK (Результаты Самоконтроля) и ВСК (Время Самоконтроля).

Важной особенностью модуля svm\_handlers.c является реализация механизмов имитации нештатного поведения СВ. При инициализации каждого SvmInstance из конфигурационного файла считываются флаги и параметры, такие как simulate\_control\_failure, simulate\_response\_timeout, send\_warning\_on\_confirm и warning\_tks. Функции-обработчики проверяют значения этих полей для текущего instance и соответствующим образом модифицируют свое поведение. Например, если установлен флаг instance→simulate\_control\_failure, то handle\_vydat\_rezultaty\_kontrolya\_message запишет в поле RSK ответного сообщения код, имитирующий ошибку. Если установлен instance→simulate\_response\_timeout, то обработчик handle\_provesti\_kontrol\_message сначала установит внутренний флаг instance→user\_flag1 (сигнализирующий последующим обработчикам этого экземпляра о необходимости прекратить ответы), а затем внесет значительную задержку (sleep()) перед отправкой своего ответа. Последующие обработчики, проверяя instance→user\_flag1, будут возвращать NULL, имитируя «зависание» СВ. Если же установлен флаг instance→send\_warning\_on\_confirm, то handle\_init\_channel\_message вместо «Подтверждения инициализации» сформирует и вернет сообщение «Предупреждение» с кодом события instance→warning\_tks. Такая гибкость позволяет целенаправленно моделировать различные сбои и проверять корректность реакции на них со стороны УВМ.

## 3.3 Программный модуль УВМ

Программный модуль uvm\_app представляет собой реализацию логики УВМ в рамках разработанного программного комплекса. Его основная задача – управление группой эмулируемых СВ, обмен с ними данными в соответствии с заданным протоколом, а также обеспечение взаимодействия с графическим интерфейсом пользователя (GUI) для мониторинга и отладки. Модуль uvm\_app написан на языке C с использованием стандартных библиотек POSIX для многопоточности и сетевого взаимодействия по протоколу TCP/IP.

### 3.3.1 Общая архитектура и управление соединениями uvm\_app

Программный модуль uvm\_app, реализующий логику УВМ, построен на основе многопоточной архитектуры, обеспечивающей асинхронное взаимодействие с несколькими эмулируемыми СВ. Координацию всех процессов осуществляет основной поток приложения (main), который отвечает за инициализацию системы, установку и поддержку TCP-соединений с экземплярами СВ, управление основным циклом протокольного взаимодействия и запуск вспомогательных сервисных потоков.

Центральным элементом для управления взаимодействием с каждым эмулируемым СВ выступает структура данных UvmSvmLink. Приложение создает и поддерживает индивидуальный объект этой структуры для каждого из четырех потенциальных каналов связи с СВ. Данная структура агрегирует всю необходимую информацию для управления конкретным соединением и отслеживания его состояния. В нее входит уникальный идентификатор связанного экземпляра СВ, указатель на объект абстрактного интерфейса ввода-вывода (IOInterface), сконфигурированный для работы по протоколу TCP/IP, и активный дескриптор установленного TCP-сокета. Важными полями являются текущий общий статус TCP-соединения, хранящийся в перечислении UvmLinkStatus (например, UVM\_LINK\_ACTIVE или UVM\_LINK\_FAILED), и логический адрес (LAK), который УВМ ожидает от данного экземпляра СВ согласно конфигурационным данным. Для реализации сложного многоэтапного протокола подготовки СВ к работе, структура UvmSvmLink также содержит поле prep\_state (типа PreparationState), отражающее текущее состояние автомата управления протоколом для данного линка. Для контроля таймаутов и корректной нумерации сообщений предусмотрены поля для хранения времени отправки последней команды (last\_command\_sent\_time), типа этой команды (last\_sent\_prep\_cmd\_type) и индивидуального счетчика сообщений для команд этапа подготовки (current\_preparation\_msg\_num). Кроме того, структура включает поля для логирования последних отправленных и полученных сообщений, времени последней сетевой активности, а также флаги и сопутствующую информацию для фиксации специфических ошибок протокола или связи, таких как несоответствие LAK, сбой внутреннего контроля СВ или получение предупреждающих сообщений.

Инициализация uvm\_app начинается со считывания параметров из конфигурационного файла config.ini. На этом этапе определяются IP-адрес целевого хоста (где функционирует приложение svm\_app) и индивидуальные настройки для каждого эмулируемого экземпляра СВ, включая номер TCP-порта для подключения и его ожидаемый логический адрес (LAK). После загрузки конфигурации основной поток последовательно пытается установить TCP-соединения с каждым сконфигурированным СВ. Для этого для каждого экземпляра динамически создается и настраивается объект IOInterface (используя реализацию io\_ethernet.c), после чего вызывается его метод connect(). В случае успешного установления соединения, соответствующий объект UvmSvmLink переводится в статус UVM\_LINK\_ACTIVE, в нем сохраняется дескриптор активного сокета, и его внутренний автомат протокольной подготовки инициализируется начальным состоянием PREP\_STATE\_READY\_TO\_SEND\_INIT\_CHANNEL.

Для обеспечения асинхронной обработки данных и выполнения фоновых задач в uvm\_app создаются и запускаются несколько типов потоков. Для каждого успешно установленного TCP-соединения с экземпляром СВ порождается индивидуальный поток-приемник, выполняющий функцию uvm\_receiver\_thread\_func. Этот поток отвечает за непрерывное чтение входящих данных (ответов и асинхронных сообщений) от своего, выделенного ему СВ. Централизованную отправку всех исходящих команд от УВМ ко всем активным экземплярам СВ осуществляет единый общий поток-отправитель, реализуемый функцией uvm\_sender\_thread\_func. Наконец, для обеспечения связи с внешним средством мониторинга запускается общий поток-сервер GUI (gui\_server\_thread), который создает TCP-сервер на предопределенном локальном порту и ожидает подключения графического интерфейса пользователя.

Обмен данными между основным потоком и сервисными потоками (отправителем и приемниками) организован через две общие потокобезопасные очереди. Команды, сформированные основным потоком для отправки конкретным СВ, помещаются в виде структур UvmRequest в исходящую очередь uvm\_outgoing\_request\_queue, откуда их извлекает поток-отправитель. Сообщения, полученные потоками-приемниками от различных СВ, упаковываются в структуры UvmResponseMessage (включающие идентификатор СВ-источника) и помещаются во входящую очередь uvm\_incoming\_response\_queue, из которой их асинхронно читает и обрабатывает основной поток. Такая архитектура позволяет uvm\_app эффективно и параллельно управлять взаимодействием с несколькими экземплярами СВ, оперативно реагировать на входящие события и обеспечивать непрерывный поток данных для системы мониторинга.

### 3.3.2 Реализация потоков-приемников от СВ (uvm\_receiver\_thread\_func).

Для каждого успешно установленного TCP-соединения с эмулируемым экземпляром СВ основной поток uvm\_app порождает отдельный, выделенный поток-приемник. Функциональность этого потока реализуется функцией uvm\_receiver\_thread\_func, которая при запуске получает в качестве аргумента указатель на соответствующую структуру UvmSvmLink, содержащую всю информацию о данном канале связи, включая активный дескриптор сокета и указатель на объект интерфейса ввода-вывода (IOInterface).

Основная задача потока-приемника заключается в непрерывном асинхронном чтении данных, поступающих от связанного с ним экземпляра СВ. Поток функционирует в цикле, который продолжается до тех пор, пока установлен глобальный флаг работы приложения uvm\_keep\_running и пока статус соответствующего UvmSvmLink указывает на активное соединение (например, UVM\_LINK\_ACTIVE). В каждой итерации цикла поток-приемник вызывает функцию receive\_protocol\_message из модуля io\_common. Эта функция осуществляет блокирующее чтение из сокета до тех пор, пока не будет получено полное протокольное сообщение (заголовок и тело) или не произойдет ошибка.

В случае успешного приема (receive\_protocol\_message возвращает 0), полученное сообщение Message упаковывается в структуру UvmResponseMessage. К этой структуре добавляется идентификатор source\_svm\_id, соответствующий экземпляру СВ, от которого пришло данное сообщение (этот ID берется из UvmSvmLink, переданного потоку при старте). Затем UvmResponseMessage помещается в общую для всего приложения uvm\_app потокобезопасную очередь входящих ответов uvm\_incoming\_response\_queue с помощью функции uvq\_enqueue(). После этого основной поток uvm\_app сможет извлечь это сообщение из очереди для дальнейшей обработки и обновления состояния соответствующего UvmSvmLink. При каждом успешном получении сообщения также обновляется поле last\_activity\_time в структуре UvmSvmLink данного канала, что используется для механизма Keep-Alive.

Поток-приемник также обрабатывает ситуации разрыва соединения или ошибки чтения. Если receive\_protocol\_message возвращает 1 (соединение закрыто удаленной стороной, т.е. СВ), поток-приемник логирует это событие, после чего корректно завершает свою работу. Если возвращается -1 (ошибка чтения), это также приводит к завершению потока. В обоих случаях, перед завершением, поток-приемник может (если это предусмотрено логикой) изменить статус UvmSvmLink на UVM\_LINK\_INACTIVE или UVM\_LINK\_FAILED (с использованием мьютекса uvm\_links\_mutex) и уведомить графический интерфейс о разрыве связи. Если receive\_protocol\_message возвращает -2 (что в текущей реализации io\_common.c может означать таймаут внутреннего poll или прерывание EINTR, не приведшее к ошибке), поток-приемник делает короткую паузу и продолжает цикл чтения, проверяя флаги uvm\_keep\_running и актуальный статус соединения.

Таким образом, наличие индивидуального потока-приемника для каждого канала связи с СВ позволяет uvm\_app асинхронно и независимо обрабатывать входящие потоки данных от всех подключенных вычислителей, не блокируя основной поток приложения или взаимодействие с другими СВ. Завершение работы потока-приемника происходит либо при общем завершении приложения uvm\_app (когда uvm\_keep\_running становится false), либо при разрыве или ошибке соответствующего TCP-соединения.

### 3.3.3 Реализация общего потока-отправителя команд (uvm\_sender\_thread\_func)

В отличие от потоков-приемников, создаваемых для каждого активного соединения индивидуально, отправка команд от УВМ всем эмулируемым экземплярам СВ централизована и обслуживается единым общим потоком-отправителем. Этот поток реализуется функцией uvm\_sender\_thread\_func и запускается один раз при старте приложения uvm\_app, функционируя до его завершения.

Основной задачей потока-отправителя является извлечение запросов на отправку сообщений из общей потокобезопасной очереди uvm\_outgoing\_request\_queue и их последующая передача целевым СВ через соответствующие TCP-соединения. В эту очередь основной поток uvm\_app помещает структуры UvmRequest, каждая из которых содержит идентификатор целевого экземпляра СВ (target\_svm\_id) и полностью сформированное протокольное сообщение Message, предназначенное для отправки.

Поток-отправитель работает в цикле, в каждой итерации которого он пытается извлечь очередной запрос UvmRequest из очереди uvm\_outgoing\_request\_queue с помощью блокирующего вызова queue\_req\_dequeue(). Если очередь пуста, поток ожидает появления новых запросов. Если извлечение неуспешно по причине закрытия очереди (что происходит при общем завершении работы uvm\_app), поток-отправитель корректно завершает свой цикл и работу.

После успешного извлечения запроса, поток-отправитель использует target\_svm\_id для доступа к соответствующей структуре UvmSvmLink из глобального массива svm\_links. Перед отправкой он проверяет текущий статус данного соединения (link→status). Команда отправляется только в том случае, если статус соединения UVM\_LINK\_ACTIVE. Это предотвращает попытки отправки данных по неактивным или сбойным каналам связи. Для фактической отправки сообщения используется функция send\_protocol\_message из модуля io\_common, которой передаются дескриптор клиентского сокета (link→connection\_handle) и указатель на объект интерфейса ввода-вывода (link→io\_handle), хранящиеся в структуре UvmSvmLink.

В случае ошибки при вызове send\_protocol\_message (например, если TCP-соединение было неожиданно разорвано СВ), поток-отправитель обрабатывает эту ситуацию. Ошибка логируется, и статус соответствующего UvmSvmLink изменяется на UVM\_LINK\_FAILED (с использованием мьютекса uvm\_links\_mutex для синхронизации доступа). Это также приводит к генерации и отправке соответствующего события EVENT в графический интерфейс пользователя, информируя о сбое канала связи. Попыток повторной отправки команды в случае сетевой ошибки данный поток не предпринимает, полагаясь на общую логику управления соединениями в основном потоке uvm\_app.

Для синхронизации с основным потоком uvm\_app, особенно в сценариях, где требуется дождаться завершения отправки группы команд перед переходом к следующему этапу протокола, используется механизм счетчика ожидающих отправки сообщений (uvm\_outstanding\_sends) и условной переменной (uvm\_all\_sent\_cond). При помещении запроса типа UVM\_REQ\_SEND\_MESSAGE в исходящую очередь, основной поток инкрементирует этот счетчик. Поток-отправитель, после фактической обработки (успешной или неуспешной отправки) такого запроса, декрементирует счетчик. Если счетчик достигает нуля, поток-отправитель сигнализирует условной переменной, что позволяет основному потоку, ожидающему на этой переменной, продолжить выполнение.

Таким образом, общий поток-отправитель обеспечивает асинхронную и упорядоченную отправку команд всем активным СВ, инкапсулирует логику физической передачи данных и участвует в механизмах обработки ошибок связи и синхронизации с основным потоком приложения.

### 3.3.4 Реализация основного потока uvm\_app

Основной поток приложения uvm\_app, выполняющийся в функции main, является центральным координирующим звеном всей управляющей логики. После завершения этапов инициализации, загрузки конфигурации, установки TCP-соединений с эмулируемыми экземплярами СВ и запуска всех необходимых сервисных потоков (отправителя, приемников для каждого СВ и сервера GUI), основной поток переходит в непрерывный цикл управления. Этот цикл продолжается до тех пор, пока установлен глобальный флаг uvm\_keep\_running, который может быть сброшен, например, по сигналу завершения приложения (Ctrl+C).

#### 3.3.4.1 Организация основного цикла и обработка входящих сообщений от СВ

Основной цикл управления в uvm\_app построен таким образом, чтобы итеративно выполнять несколько ключевых задач: инициировать отправку команд специализированным вычислителям в соответствии с их текущим состоянием протокольной подготовки, обрабатывать входящие сообщения (ответы и асинхронные данные) от СВ, а также контролировать таймауты ожидания ответов и общую активность каналов связи (Keep-Alive). Для обеспечения отзывчивости и предотвращения излишней загрузки процессора при отсутствии событий, в конце каждой итерации цикла, если не было выполнено никаких значимых действий, предусматривается короткая пауза с помощью функции usleep().

Центральным механизмом получения данных от СВ является общая потокобезопасная очередь входящих сообщений uvm\_incoming\_response\_queue. Потоки-приемники (uvm\_receiver\_thread\_func), каждый из которых обслуживает свое соединение с СВ, помещают в эту очередь полученные протокольные сообщения, предварительно упаковав их в структуру UvmResponseMessage. Эта структура содержит само сообщение Message и идентификатор source\_svm\_id экземпляра СВ, от которого оно поступило.

В каждой итерации основного цикла производится попытка извлечения очередного UvmResponseMessage из uvm\_incoming\_response\_queue с помощью неблокирующего или кратковременно блокирующего вызова функции uvq\_dequeue(). Если сообщение успешно извлечено, основной поток первым делом выполняет преобразование порядка байт для полей полученного сообщения Message из сетевого в хостовый с помощью функции message\_to\_host\_byte\_order(). Это обеспечивает корректную интерпретацию многобайтовых полей, таких как длина тела сообщения и числовые данные в его теле, на платформе УВМ. Также извлекается полный порядковый номер сообщения функцией get\_full\_message\_number().

После этого, используя source\_svm\_id из UvmResponseMessage, основной поток получает доступ к соответствующей структуре UvmSvmLink из глобального массива svm\_links (предварительно захватив мьютекс uvm\_links\_mutex для обеспечения потокобезопасности). В этой структуре немедленно обновляется поле last\_activity\_time временной меткой текущего момента, что критически важно для механизма контроля активности канала (Keep-Alive).

Затем формируется и передается в графический интерфейс пользователя (GUI) стандартизированное текстовое IPC-сообщение типа RECV. Это сообщение информирует GUI о факте приема данных, указывая идентификатор СВ источника, тип и номер полученного протокольного сообщения, логический адрес (LAK) отправителя (из заголовка сообщения), а также вычисленный полный вес сообщения (заголовок + тело) и извлеченные из тела ключевые параметры (например, счетчик BCB, результаты контроля RSK, параметры состояния линии KLA, SLA, KSA или код события TKS для сообщений типа «Предупреждение»). Дальнейшая детальная обработка полученного сообщения и принятие решений по управлению протоколом происходит в последующих блоках логики основного цикла, в частности, при управлении машиной состояний для этапа подготовки СВ.

#### 3.3.4.2 Этап «Подготовка к сеансу наблюдения»

Первичным этапом взаимодействия УВМ с каждым экземпляром СВ является «Подготовка к сеансу наблюдения». Этот этап включает последовательный обмен несколькими парами команд «запрос-ответ» для инициализации канала связи и проверки готовности СВ. Для корректной и асинхронной координации этого процесса для каждого активного соединения с СВ (представленного структурой UvmSvmLink) в uvm\_app реализована машина состояний. Текущее состояние каждого СВ на данном этапе отслеживается в поле link→prep\_state, принадлежащем перечислению PreparationState. Это перечисление включает состояния, отражающие как готовность УВМ к отправке очередной команды (PREP\_STATE\_READY\_TO\_SEND\_...), так и ожидание УВМ конкретного ответного сообщения от СВ (PREP\_STATE\_AWAITING\_...\_REPLY), а также терминальные состояния (PREP\_STATE\_PREPARATION\_COMPLETE, PREP\_STATE\_FAILED).

Отправка команд подготовки осуществляется в рамках основного цикла uvm\_app. В каждой итерации, после захвата мьютекса uvm\_links\_mutex, происходит обход всех активных UvmSvmLink. Если экземпляр СВ находится в состоянии, предполагающем отправку очередной команды подготовки (например, PREP\_STATE\_READY\_TO\_SEND\_INIT\_CHANNEL после успешного установления TCP-соединения, или PREP\_STATE\_READY\_TO\_SEND\_PROVESTI\_KONTROL после получения корректного ответа «Подтверждение инициализации канала»), основной поток формирует соответствующее протокольное сообщение. Для этого используются функции-конструкторы из модуля message\_builder (например, create\_init\_channel\_message()). В качестве порядкового номера сообщения используется значение из поля link→current\_preparation\_msg\_num данного экземпляра СВ. Сформированное сообщение упаковывается в структуру UvmRequest (с указанием target\_svm\_id) и помещается в общую исходящую очередь uvm\_outgoing\_request\_queue с помощью функции send\_uvm\_request(). В случае успешной постановки запроса в очередь, в структуре UvmSvmLink обновляются поля last\_command\_sent\_time (текущее время) и last\_sent\_prep\_cmd\_type (тип только что отправленной команды), а состояние link→prep\_state переводится в соответствующее состояние ожидания ответного сообщения (например, из PREP\_STATE\_READY\_TO\_SEND\_INIT\_CHANNEL в PREP\_STATE\_AWAITING\_CONFIRM\_INIT\_REPLY). Если постановка запроса в очередь по какой-либо причине не удалась, состояние подготовки данного СВ немедленно переводится в PREP\_STATE\_FAILED, а общий статус соединения – в UVM\_LINK\_FAILED, с последующим уведомлением GUI.

Обработка ответов на команды подготовки происходит также в основном цикле uvm\_app, после извлечения сообщения UvmResponseMessage из входящей очереди uvm\_incoming\_response\_queue (как описано в 3.3.4.1). После первичной обработки и отправки RECV уведомления в GUI, основной поток анализирует тип полученного сообщения в контексте текущего состояния link\_resp→prep\_state для СВ-источника.  
Если тип сообщения соответствует ожидаемому для текущего состояния ожидания (например, получено MESSAGE\_TYPE\_CONFIRM\_INIT, когда prep\_state был PREP\_STATE\_AWAITING\_CONFIRM\_INIT\_REPLY), производится детальный анализ содержимого тела ответного сообщения. Например, для «Подтверждения инициализации канала» проверяется совпадение логического адреса (LAK) в теле ответа с ожидаемым link\_resp→assigned\_lak. Для «Результатов контроля» анализируется поле РСК (Результат самоконтроля). В случае если ответ корректен и все проверки пройдены успешно, счетчик команд link\_resp→current\_preparation\_msg\_num инкрементируется (подготавливая номер для следующего запроса УВМ этому СВ), и состояние link\_resp→prep\_state переводится в следующее состояние готовности к отправке (например, из PREP\_STATE\_AWAITING\_CONFIRM\_INIT\_REPLY в PREP\_STATE\_READY\_TO\_SEND\_PROVESTI\_KONTROL).  
Если же в ожидаемом ответном сообщении обнаруживается ошибка (например, несоответствие LAK, или код ошибки в поле РСК), то состояние подготовки link\_resp→prep\_state переводится в PREP\_STATE\_FAILED. Общий статус соединения link\_resp→status может быть изменен на UVM\_LINK\_FAILED (в случае критической ошибки, как LAK mismatch) или UVM\_LINK\_WARNING (в случае некритической ошибки контроля). Соответствующее событие (EVENT) об ошибке и изменении статуса также транслируется в GUI.

Контроль таймаутов для команд подготовки является неотъемлемой частью управления протоколом. В каждой итерации основного цикла, после обработки входящих сообщений, производится проверка таймаутов для всех СВ, находящихся в одном из состояний ожидания ответа (PREP\_STATE\_AWAITING\_...\_REPLY). Для каждого такого СВ сравнивается текущее время с временем отправки последней команды (link\_timeout→last\_command\_sent\_time). Если разница превышает предопределенный для данной команды таймаут (например, 5 секунд для «Подтверждения инициализации» или 8 секунд для «Результатов контроля»), считается, что СВ не ответил вовремя. В этом случае его состояние подготовки link\_timeout→prep\_state принудительно переводится в PREP\_STATE\_FAILED, общий статус соединения link\_timeout→status устанавливается в UVM\_LINK\_FAILED, фиксируется флаг response\_timeout\_detected, и в GUI отправляются события, информирующие о таймауте ответа на конкретную команду и об изменении общего статуса канала связи.

Этап «Подготовки к сеансу наблюдения» считается успешно завершенным для конкретного экземпляра СВ, когда он последовательно проходит все необходимые шаги обмена командами и его состояние prep\_state достигает значения PREP\_STATE\_PREPARATION\_COMPLETE. После этого данный СВ готов к получению команд следующего этапа – «Подготовки к сеансу съемки».

#### 3.3.4.3 Этап «Подготовка к сеансу съемки»

После того как экземпляр СВ успешно прошел все шаги этапа «Подготовки к сеансу наблюдения» и его состояние prep\_state в структуре UvmSvmLink установлено в PREP\_STATE\_PREPARATION\_COMPLETE, УВМ готова инициировать следующий этап – «Подготовку к сеансу съемки». Этот переход также координируется основным потоком uvm\_app в рамках его главного цикла управления.

В Блоке 1 основного цикла (отвечающем за отправку команд), когда для очередного экземпляра СВ link→prep\_state оказывается равным PREP\_STATE\_PREPARATION\_COMPLETE, и при этом общий статус TCP-соединения (link→status) является UVM\_LINK\_ACTIVE или UVM\_LINK\_WARNING (позволяя отправку команд даже при наличии ранее зафиксированных некритических ошибок контроля), УВМ приступает к отправке конфигурационных параметров, необходимых для предстоящего сеанса съемки.

В отличие от команд предыдущего этапа, сообщения, передающие параметры съемки, как правило, не требуют немедленного ответного сообщения от СВ для подтверждения их получения или обработки на прикладном уровне протокола. Поэтому УВМ формирует и отправляет всю последовательность необходимых конфигурационных сообщений для данного СВ единой «пачкой», не ожидая индивидуальных ответов после каждой команды.

Набор отправляемых сообщений зависит от выбранного режима работы РСА (mode), который определяется при запуске uvm\_app (например, из аргументов командной строки) и является общим для всех СВ в текущем сеансе.

Для режима детального разрешения (ДР) отправляются сообщения «Принять параметры СДР» (тип 170) и «Принять параметры ЦДР» (тип 210).

Для обзорного режима (ОР) передаются сообщения «Принять параметры СО» (тип 160), «Принять параметры 3ЦО» (тип 200), а также могут загружаться массивы опорных сигналов через сообщения «Принять TIME\_REF\_RANGE» (тип 161) и «Принять Reper» (тип 162).

Для режима высокого разрешения (ВР) также используются «Принять параметры СО» и «Принять параметры 3ЦО».

Завершает пачку конфигурационных команд передача актуальных «Навигационных данных» (тип 255), необходимых для всех режимов работы.

Каждое из этих сообщений формируется с использованием соответствующих функций-конструкторов из модуля message\_builder, тело сообщения заполняется предопределенными или демонстрационными данными, и порядковый номер сообщения инкрементально присваивается из поля link→current\_preparation\_msg\_num данного СВ. Сформированные запросы UvmRequest последовательно помещаются в общую исходящую очередь uvm\_outgoing\_request\_queue для последующей отправки потоком-отправителем.

После успешной постановки в очередь всей пачки команд «Подготовки к сеансу съемки» для данного СВ, его состояние link→prep\_state переводится в новое значение, например, PREP\_STATE\_SHOOTING\_PARAMS\_SENT. Это новое состояние сигнализирует о том, что конфигурация для съемки передана, и данный СВ теперь ожидает либо начала непосредственно сеанса съемки, либо других управляющих команд. Изменение состояния на PREP\_STATE\_SHOOTING\_PARAMS\_SENT также предотвращает повторную отправку параметров съемки на последующих итерациях основного цикла uvm\_app для этого экземпляра СВ. Счетчик сообщений link→current\_preparation\_msg\_num обновляется, отражая последний использованный номер.

Таким образом, uvm\_app обеспечивает индивидуальный и асинхронный переход каждого экземпляра СВ к этапу конфигурирования для съемки, как только он подтверждает свою полную готовность после прохождения всех процедур начальной подготовки и самоконтроля.

#### 3.3.4.4 Обработка асинхронных сообщений от СВ

Помимо ответных сообщений, поступающих от СВ в рамках строго регламентированных процедур обмена (таких как «Подготовка к сеансу наблюдения»), протокол взаимодействия предусматривает возможность асинхронной передачи данных от СВ к УВМ. Такие сообщения могут информировать УВМ о различных событиях, результатах обработки или возникновении нештатных ситуаций и не являются прямым ответом на какую-либо конкретную команду УВМ.

Обработка всех входящих сообщений, включая асинхронные, происходит в основном цикле uvm\_app, после их извлечения из общей очереди uvm\_incoming\_response\_queue. Когда основной поток получает сообщение UvmResponseMessage, он сначала определяет его тип (msg\_resp→header.message\_type). Если тип полученного сообщения не соответствует ожидаемому ответу для текущего состояния подготовки link\_resp→prep\_state данного СВ, или если СВ уже находится в состоянии, не предполагающем ожидания специфического ответа (например, PREP\_STATE\_PREPARATION\_COMPLETE или PREP\_STATE\_SHOOTING\_PARAMS\_SENT), то такое сообщение рассматривается как асинхронное и обрабатывается соответствующим образом.

Наиболее важным типом асинхронного сообщения от СВ, обработка которого реализована в текущей версии комплекса, является сообщение «Предупреждение» (тип 254). При получении данного сообщения УВМ выполняет последовательность действий, направленных на фиксацию события и информирование пользователя. В первую очередь, из тела сообщения извлекается код типа критического события (ТКС) и, если они присутствуют, сопутствующие параметры этого события. Эта информация немедленно логируется в консольный вывод приложения uvm\_app с указанием идентификатора СВ, сгенерировавшего предупреждение. Одновременно с этим формируется и передается в графический интерфейс пользователя (GUI) специальное IPC-сообщение типа EVENT с категорией Warning, в детали которого включается полученный код ТКС и другая релевантная информация из сообщения СВ. Для обеспечения возможности последующего анализа и отслеживания состояния, в соответствующей структуре UvmSvmLink обновляются поля last\_warning\_tks (код последнего полученного ТКС) и last\_warning\_time (временная метка получения предупреждения). Поскольку получение «Предупреждения» свидетельствует о потенциальной проблеме или нештатной ситуации на стороне СВ, общий статус соединения link\_resp→status для данного СВ переводится в состояние UVM\_LINK\_WARNING, если он до этого момента находился в состоянии UVM\_LINK\_ACTIVE. Это изменение общего статуса канала также незамедлительно транслируется в GUI посредством отправки соответствующего EVENT сообщения с типом LinkStatus. Важно отметить, что состояние протокольной подготовки link\_resp→prep\_state при получении асинхронного «Предупреждения», как правило, не изменяется, так как УВМ может все еще ожидать завершения ранее отправленной основной команды или уже находиться в состоянии готовности к следующим операциям в рамках основного протокола.

В перспективе, при дальнейшей реализации функционала сеанса съемки, к категории асинхронных сообщений будут также относиться потоки научных и телеметрических данных, передаваемых от СВ к УВМ. К таким сообщениям относятся, например, «СУБК» (Субкадр Одного Канала, тип 127), «КО» (Кадр Обработки, тип 137), «Строка голограммы СУБК» (тип 8), а также сообщения о результатах обнаружения неподвижных контрастных объектов («НК», тип 80) или активных помех («Помеха», тип 81). Логика их обработки в uvm\_app будет включать корректный прием, валидацию формата, отображение факта получения в GUI и, для полнофункциональной системы, сохранение или передачу этих данных для последующего анализа и использования.

Таким образом, архитектура основного цикла uvm\_app предусматривает механизмы для корректной обработки не только ожидаемых ответов в рамках установленной машины состояний протокола, но и спонтанных асинхронных сообщений от СВ. Это обеспечивает полноту мониторинга взаимодействия, своевременное информирование оператора о нештатных ситуациях через GUI и позволяет УВМ адекватно реагировать на различные события в системе.

### 3.3.5 Реализация механизмов отказоустойчивости и обработки событий

Для обеспечения стабильной и надежной работы в условиях возможных сбоев связи или некорректного поведения эмулируемых СВ, в программном модуле УВМ (uvm\_app) реализован комплекс механизмов отказоустойчивости и обработки нештатных ситуаций. Эти механизмы охватывают различные уровни взаимодействия, начиная от установки TCP-соединения и заканчивая анализом содержания протокольных сообщений.

#### 3.3.5.1 Обработка ошибок TCP-соединений

На начальном этапе работы, при попытке установить TCP-соединение с каждым из сконфигурированных экземпляров СВ, uvm\_app обрабатывает возможные ошибки подключения. Если соединение с каким-либо СВ не может быть установлено (например, указан неверный порт, или процесс svm\_app для данного экземпляра не запущен), соответствующий UvmSvmLink помечается статусом UVM\_LINK\_FAILED, информация об ошибке логируется, и УВМ продолжает работу с остальными доступными СВ.

В процессе обмена данными, ошибки, возникающие на уровне TCP/IP (например, разрыв соединения, ошибки функций send или recv), также отслеживаются. Потоки-приемники (uvm\_receiver\_thread\_func) и общий поток-отправитель (uvm\_sender\_thread\_func) при обнаружении таких ошибок изменяют статус соответствующего UvmSvmLink на UVM\_LINK\_FAILED или UVM\_LINK\_INACTIVE (в случае штатного закрытия соединения удаленной стороной). Эти изменения статуса незамедлительно транслируются в графический интерфейс пользователя (GUI) посредством отправки EVENT сообщения с типом LinkStatus, что позволяет отслеживать состояние каждого канала связи в реальном времени.

#### 3.3.5.2 Контроль таймаутов ответов на команды

Для команд УВМ, предполагающих обязательный ответ от СВ в рамках этапа «Подготовки к сеансу наблюдения», реализован механизм контроля таймаутов. Как описано в разделе 3.3.4.2, для каждого СВ, находящегося в состоянии ожидания ответа (PREP\_STATE\_AWAITING\_...\_REPLY), основной поток uvm\_app отслеживает время, прошедшее с момента отправки команды (link→last\_command\_sent\_time). Если это время превышает предопределенный для данной команды интервал ожидания, фиксируется таймаут. В этом случае состояние подготовки СВ (link→prep\_state) принудительно переводится в PREP\_STATE\_FAILED, общий статус соединения (link→status) устанавливается в UVM\_LINK\_FAILED, и в структуре UvmSvmLink выставляется флаг response\_timeout\_detected. GUI информируется об этом событии через отправку EVENT сообщения с типом ResponseTimeout, указывающего, на какую команду не был получен ответ, а также EVENT сообщения LinkStatus с новым статусом канала.

#### 3.3.5.3 Механизм Keep-Alive

Для обнаружения «молчащих» СВ, то есть тех, с которыми установлено TCP-соединение, но от которых длительное время не поступает никаких данных (ни ожидаемых ответов, ни асинхронных сообщений), в uvm\_app реализован механизм Keep-Alive на прикладном уровне. Основной поток периодически проверяет поле last\_activity\_time в структуре UvmSvmLink для каждого активного соединения. Это поле обновляется каждый раз при успешном получении любого сообщения от соответствующего СВ. Если с момента последней активности проходит время, превышающее заданный таймаут Keep-Alive (например, 15 секунд, значение которого может настраиваться через конфигурационный файл), УВМ считает данный канал связи неисправным. Статус UvmSvmLink изменяется на UVM\_LINK\_FAILED, в его структуре устанавливается флаг timeout\_detected, инициируется закрытие TCP-сокета (shutdown()), и в GUI отправляются соответствующие EVENT сообщения (KeepAliveTimeout и LinkStatus). Этот механизм позволяет своевременно выявлять и изолировать не отвечающие СВ, не дожидаясь системных TCP-таймаутов.

#### 3.3.5.4 Обработка ошибок на уровне протокола

Помимо ошибок связи, uvm\_app выполняет проверки корректности полученных от СВ сообщений на соответствие протоколу.  
При получении сообщения «Подтверждение инициализации канала» (тип 129) проверяется совпадение логического адреса (LAK), указанного в теле сообщения, с ожидаемым LAK для данного экземпляра СВ (link→assigned\_lak). В случае несоответствия, это расценивается как критическая ошибка протокола: статус UvmSvmLink устанавливается в UVM\_LINK\_FAILED, выставляется флаг lak\_mismatch\_detected, и в GUI отправляется EVENT с типом LAKMismatch. Дальнейшее взаимодействие с таким СВ на этапе подготовки прекращается. При получении сообщения «Результаты контроля» (тип 4) анализируется поле РСК (Результат самоконтроля). Если значение РСК указывает на наличие сбоев во внутренних тестах СВ (т.е. РСК не равен коду полного успеха, например, 0x3F), то статус UvmSvmLink переводится в UVM\_LINK\_WARNING, устанавливается флаг control\_failure\_detected, и в GUI отправляется EVENT с типом ControlFail, содержащий код ошибки РСК. При этом, если ошибка контроля не считается критической для продолжения работы, УВМ может продолжить взаимодействие с данным СВ, но его «проблемный» статус будет виден оператору. Асинхронные сообщения «Предупреждение» (тип 254) от СВ, сигнализирующие о внутренних критических событиях на стороне вычислителя, также приводят к установке статуса UVM\_LINK\_WARNING для соответствующего канала и передаче информации о коде события (ТКС) в GUI через EVENT с типом Warning.

### 3.3.6 Реализация потока-сервера для GUI (gui\_server\_thread)

Для обеспечения возможности мониторинга и визуализации процесса взаимодействия между УВМ и эмулируемыми Специализированными Вычислителями (СВ) в реальном времени, приложение uvm\_app включает функциональность TCP-сервера, предназначенного для связи с графическим интерфейсом пользователя (gui\_app). Эта функциональность реализована в отдельном потоке, выполняющем функцию gui\_server\_thread.

Поток gui\_server\_thread запускается один раз при старте uvm\_app. Его основная задача – создание серверного TCP-сокета, прослушивание его на предопределенном локальном порту (например, 12345) и ожидание входящего подключения от приложения gui\_app. Архитектура сервера рассчитана на обслуживание одного одновременного подключения от GUI; при поступлении нового запроса на соединение, если предыдущее соединение с GUI еще активно, оно принудительно закрывается, и устанавливается новое.

После успешного принятия входящего соединения от gui\_app (с помощью системного вызова accept), дескриптор клиентского сокета GUI сохраняется в глобальной переменной gui\_client\_fd (доступ к которой синхронизируется мьютексом gui\_socket\_mutex). Первым действием после установления связи является отправка клиенту GUI начального состояния всех каналов связи с экземплярами СВ. Эта информация включает текущий статус (UvmLinkStatus) каждого UvmSvmLink, назначенный ему логический адрес (LAK), а также, если имеются, данные о последних отправленных и полученных протокольных сообщениях и зафиксированных событиях или ошибках. Это позволяет gui\_app сразу после подключения отобразить актуальную картину взаимодействия.

Дальнейшая передача данных в GUI осуществляется асинхронно из различных частей основного потока uvm\_app через вызов вспомогательной функции send\_to\_gui\_socket(). Эта функция принимает в качестве аргумента текстовую строку, содержащую информацию о событии, и отправляет ее по активному соединению с gui\_client\_fd. Формат передаваемых строк стандартизирован для упрощения парсинга на стороне gui\_app и включает следующие основные типы IPC-сообщений:

1. SENT;SVM\_ID:X;Type:Y;Num:Z;LAK:0xL;Weight:W: Информирует об отправке УВМ протокольного сообщения экземпляру СВ с идентификатором X. Указываются тип сообщения Y, его полный номер Z, логический адрес получателя L и общий вес сообщения W в байтах.
2. RECV;SVM\_ID:X;Type:Y;Num:Z;LAK:0xL;BCB:0xB;Weight:W;Details:текст: Информирует о получении УВМ протокольного сообщения от экземпляра СВ X. Указываются тип Y, номер Z, логический адрес отправителя (СВ) L (из заголовка сообщения), опционально счетчик времени наработки СВ B (BCB, если присутствует в теле), общий вес сообщения W и строка Details с ключевыми параметрами из тела ответного сообщения (например, РСК, ТКС, КЛА и т.д.).
3. EVENT;SVM\_ID:X;Type:ИмяСобытия;Details:текст: Информирует о возникновении различных событий, связанных с каналом связи СВ X. ИмяСобытия может быть, например, LinkStatus (с указанием нового статуса и LAK в Details), LAKMismatch, ControlFail (с кодом РСК), ResponseTimeout (с указанием команды, на которую не получен ответ), KeepAliveTimeout или Warning (с кодом ТКС).

Функция send\_to\_gui\_socket() обеспечивает добавление символа новой строки к каждому сообщению для корректной обработки на стороне GUI и обрабатывает возможные ошибки отправки, включая разрыв соединения со стороны клиента GUI. В случае разрыва, дескриптор gui\_client\_fd сбрасывается, и поток gui\_server\_thread возвращается в состояние ожидания нового подключения.

Поток gui\_server\_thread продолжает свою работу до тех пор, пока установлен глобальный флаг uvm\_keep\_running. При получении сигнала на завершение работы uvm\_app, слушающий сокет GUI и активное клиентское соединение (если оно есть) корректно закрываются, после чего поток завершается. Такая реализация обеспечивает непрерывную передачу актуальной информации о состоянии системы УВМ-СВ в приложение мониторинга.

## 3.4 Пользовательский интерфейс (gui\_app)

Для наглядного мониторинга процесса взаимодействия между УВМ и эмулируемыми Специализированными Вычислителями (СВ), а также для анализа возникающих событий и состояний, в рамках программного комплекса разработан графический интерфейс пользователя (GUI). Приложение gui\_app создано с использованием кроссплатформенного фреймворка Qt и языка программирования C++, что обеспечивает его переносимость.

### 3.4.1 Назначение и общая архитектура GUI

Основным назначением графического интерфейса gui\_app является предоставление оператору или разработчику удобного инструмента для наблюдения в реальном времени за информационным обменом между uvm\_app и всеми эмулируемыми экземплярами svm\_app. GUI позволяет отслеживать последовательность отправляемых и принимаемых протокольных сообщений, текущий статус каждого канала связи с СВ, ключевые параметры, передаваемые в сообщениях (такие как счетчик времени наработки BCB), а также оперативно получать информацию о возникающих ошибках, нештатных ситуациях или предупреждениях. Дополнительно, интерфейс предоставляет функцию сохранения детальных логов взаимодействия для последующего анализа.

Архитектурно gui\_app реализован как клиентское TCP/IP приложение. Оно состоит из двух основных логических компонентов: модуля сетевого взаимодействия (UvmMonitorClient) и модуля визуального представления (MainWindow).

Модуль UvmMonitorClient отвечает за установление и поддержание TCP-соединения с серверным потоком gui\_server\_thread приложения uvm\_app (по умолчанию на localhost:12345). После успешного подключения UvmMonitorClient асинхронно принимает поток текстовых IPC-строк от uvm\_app, каждая из которых описывает определенное событие (отправка сообщения, получение сообщения, изменение статуса соединения, ошибка). UvmMonitorClient производит парсинг этих строк, извлекая все необходимые данные: идентификатор СВ, временную метку, тип события, тип и номер протокольного сообщения, логические адреса, вес сообщения, значение BCB и другие детали. Распарсенная информация затем передается в модуль MainWindow посредством механизма сигналов и слотов Qt.

Модуль MainWindow является основным компонентом, формирующим графический интерфейс, видимый пользователю. Он получает данные от UvmMonitorClient и динамически обновляет соответствующие визуальные элементы для отображения актуальной информации по каждому из четырех эмулируемых СВ.

Такое разделение на сетевой модуль и модуль представления позволяет изолировать логику сетевого обмена от логики отображения, что упрощает разработку и поддержку приложения.

### 3.4.2 Реализация сетевого взаимодействия с uvm\_app (UvmMonitorClient)

Сетевое взаимодействие графического интерфейса пользователя (gui\_app) с управляющим приложением uvm\_app реализовано в классе UvmMonitorClient. Этот класс инкапсулирует всю логику установления TCP-соединения, приема и предварительной обработки данных, поступающих от uvm\_app. UvmMonitorClient функционирует как TCP-клиент, подключаясь к серверу, запущенному в потоке gui\_server\_thread приложения uvm\_app на предопределенном IP-адресе и порту (по умолчанию localhost:12345).

При инициализации UvmMonitorClient создает объект класса QTcpSocket для осуществления сетевых операций. Предусмотрен механизм автоматического переподключения в случае разрыва связи или неудачной первоначальной попытки соединения, реализованный с помощью объекта QTimer. Этот таймер периодически пытается восстановить соединение с uvm\_app.

После успешного установления TCP-соединения, UvmMonitorClient переходит в режим асинхронного чтения данных из сокета. Для этого используется механизм сигналов и слотов Qt: при поступлении данных в сокет генерируется сигнал readyRead(), который обрабатывается соответствующим слотом в UvmMonitorClient. Входящие данные накапливаются во внутреннем буфере (QByteArray). Как только в буфере обнаруживается один или несколько полных IPC-пакетов (строк, завершающихся символом новой строки \n, как их передает uvm\_app), они извлекаются для дальнейшей обработки.

Ключевой функцией UvmMonitorClient является parseData(), которая отвечает за разбор каждой полученной IPC-строки. Строка имеет текстовый формат с полями, разделенными точкой с запятой (;), и парами «ключ:значение» для отдельных параметров. Функция parseData() выполняет следующие действия:

1. Разделяет строку на компоненты.
2. Определяет основной тип события IPC (SENT, RECV, EVENT).
3. Извлекает обязательное поле SVM\_ID, указывающее, к какому эмулируемому экземпляру СВ относится данное событие.
4. Для событий SENT и RECV извлекаются поля: Type (числовой код типа протокольного сообщения), Num (полный номер сообщения), LAK (логический адрес, участвующий в обмене), опционально BCB (счетчик времени наработки СВ) и Weight (полный вес сообщения в байтах). Тип сообщения преобразуется в его строковое имя с помощью вспомогательной функции getMessageNameByType().
5. Для событий EVENT извлекаются поля Type (строковое имя события, например, LinkStatus, ControlFail) и Details (текстовое описание деталей события, например, новый статус канала или код ошибки).

Все извлеченные и преобразованные данные, включая временную метку получения события, упаковываются в аргументы и передаются далее через сигнал Qt newMessageOrEvent(). Если событие касается изменения общего статуса соединения с конкретным СВ (например, при получении EVENT с типом LinkStatus), дополнительно генерируется сигнал svmLinkStatusChanged(), передающий ID СВ, его новый статус и назначенный LAK.

UvmMonitorClient также отслеживает общее состояние TCP-соединения с uvm\_app и информирует об этом другие компоненты GUI через сигнал connectionStatusChanged(), передавая флаг успешного подключения и текстовое сообщение о статусе. Это позволяет главному окну приложения отображать актуальную информацию о доступности сервера uvm\_app.

### 3.4.3 Визуальное представление и элементы интерфейса (MainWindow)

Основное окно приложения gui\_app, реализованное классом MainWindow, отвечает за визуализацию всей информации, поступающей от uvm\_app через UvmMonitorClient. Интерфейс спроектирован таким образом, чтобы предоставить пользователю наглядное и структурированное представление о состоянии взаимодействия с каждым из четырех эмулируемых экземпляров СВ.

Для каждого экземпляра СВ (с идентификаторами от 0 до 3) в главном окне выделена отдельная панель (QGroupBox). Каждая такая панель содержит следующие ключевые элементы для отображения информации:

1. Индикатор статуса соединения (QLabel): Отображает текущий общий статус TCP-канала связи между УВМ и данным СВ (например, «INACTIVE», «CONNECTING», «ACTIVE», «FAILED», «WARNING»). Текст и цвет фона этого индикатора динамически изменяются в зависимости от статуса, полученного через сигнал svmLinkStatusChanged() от UvmMonitorClient. Например, активное состояние подсвечивается зеленым цветом, ошибки – красным, предупреждения – желтым.
2. Отображение логического адреса (QLabel): Показывает назначенный логический адрес (LAK) для данного экземпляра СВ, как только эта информация становится известна из событий LinkStatus.
3. Индикатор ошибок/событий (QLabel): Текстовое поле, предназначенное для вывода краткой информации о последних значимых событиях или ошибках, связанных с данным СВ (например, «ControlFail: RSK=0x3E», «ResponseTimeout», «LAKMismatch»). Это поле обновляется при получении соответствующих EVENT сообщений от uvm\_app. При отсутствии ошибок отображается статус «OK».
4. Таблица лога сообщений (QTableWidget): Основной элемент для детального протоколирования взаимодействия. Каждая таблица имеет несколько столбцов, включая:
   1. «Время»: Временная метка события (чч:мм:сс.zzz).
   2. «Напр/Соб.»: Направление передачи сообщения («SENT» для УВМ→СВ, «RECV» для СВ→УВМ) или тип события («EVENT»).
   3. «LAK SVM»: Логический адрес СВ, участвующего в обмене (для SENT – LAK получателя, для RECV – LAK отправителя).
   4. «BCB»: Значение счетчика BCB, если оно присутствует в сообщении.
   5. «Вес, Б»: Полный вес протокольного сообщения в байтах (заголовок + тело).
   6. «Тип сообщ.»: Числовой код типа протокольного сообщения.
   7. «Имя сообщ.»: Текстовое наименование типа сообщения или имя события.
   8. «Номер сообщ.»: Полный порядковый номер сообщения.
   9. «Детали»: Дополнительная текстовая информация (например, ключевые поля из тела сообщения для RECV или параметры события для EVENT).

Класс MainWindow содержит слоты, подключенные к сигналам newMessageOrEvent() и svmLinkStatusChanged() от UvmMonitorClient. При поступлении сигнала newMessageOrEvent(), соответствующий слот определяет svmId и добавляет новую строку с полученными данными в QTableWidget нужного экземпляра СВ. При этом обеспечивается автоматическая прокрутка таблицы к последней записи и ограничение максимального количества строк в логе для предотвращения излишнего потребления памяти. При поступлении сигнала svmLinkStatusChanged() обновляются соответствующие индикаторы статуса и LAK.

Инициализация таблиц логов (initTableWidget()) включает настройку количества и заголовков столбцов, их начальной ширины, а также установку режимов запрета редактирования и выбора строк целиком для удобства пользователя.

Помимо панелей для каждого СВ, в главном окне присутствует общая кнопка «Сохранить все логи», инициирующая процедуру сохранения содержимого всех таблиц логов в текстовые файлы. Строка состояния (QStatusBar) в нижней части окна используется для отображения общего статуса подключения к uvm\_app.

Такая организация интерфейса позволяет одновременно отслеживать состояние всех каналов УВМ-СВ и детально анализировать историю обмена сообщениями для каждого из них.

### 3.4.4 Реализация функции сохранения логов

Для обеспечения возможности последующего анализа и документирования процесса взаимодействия между УВМ и СВ, в графическом интерфейсе gui\_app предусмотрена функция сохранения накопленных логов сообщений. Эта функциональность инициируется пользователем нажатием кнопки «Сохранить все логи», расположенной в основной части окна приложения.

При активации данной функции (через слот MainWindow::onSaveLogAllClicked()) пользователю сначала предлагается выбрать директорию на локальном диске для сохранения файлов логов с помощью стандартного диалогового окна QFileDialog::getExistingDirectory(). Если пользователь выбирает директорию и подтверждает свой выбор, приложение последовательно обрабатывает данные для каждого из четырех эмулируемых экземпляров СВ.

Для каждого экземпляра СВ, для которого в соответствующей таблице QTableWidget накоплены записи (т.е., таблица не пуста), создается отдельный текстовый файл. Имя файла формируется таким образом, чтобы обеспечить его уникальность и информативность, например, по шаблону svm\_<ID>\_lak\_<LAK>\_log\_<ДАТА\_ВРЕМЯ>.txt, где <ID> – это идентификатор экземпляра (0-3), <LAK> – его текущий назначенный логический адрес (если известен, иначе «N/A»), а <ДАТА\_ВРЕМЯ> – текущая дата и время сохранения для предотвращения перезаписи файлов.

Процесс сохранения для каждого файла включает следующие шаги, реализованные в функции MainWindow::saveTableLogToFile():

1. Открытие (или создание) текстового файла в выбранной директории в режиме записи (QIODevice::WriteOnly | QIODevice::Text) с принудительной перезаписью, если файл с таким именем уже существует (QIODevice::Truncate), и установкой кодировки UTF-8 для корректной записи символов.
2. Запись в файл информационного заголовка, включающего идентификатор СВ и его LAK.
3. Запись заголовков всех столбцов таблицы лога, аналогично тому, как они отображаются в GUI. Для удобочитаемости столбцы в текстовом файле разделяются символами табуляции или другой комбинацией разделителей.
4. Последовательный обход всех строк (rowCount()) текущей таблицы лога QTableWidget. Для каждой строки происходит итерация по всем ее столбцам (columnCount()).
5. Содержимое каждой ячейки (QTableWidgetItem::text()) считывается. Символы перевода строки внутри ячейки заменяются на пробелы, чтобы каждая запись лога в файле занимала одну строку.
6. Сформированная строка данных для текущей записи лога записывается в файл, после чего добавляется символ новой строки.
7. После обработки всех строк таблицы файл закрывается.

В случае возникновения ошибок при открытии или записи файла для какого-либо из СВ (например, из-за отсутствия прав на запись в выбранную директорию), соответствующее предупреждение выводится в отладочную консоль приложения gui\_app и, возможно, в строку состояния главного окна. После успешного сохранения всех логов (или попытки сохранения) в строке состояния выводится сообщение о завершении операции с указанием директории.

Данная функция позволяет пользователю в любой момент времени сохранить полную историю взаимодействия для всех или выбранных СВ, что является важным инструментом для отладки, анализа нештатных ситуаций и подготовки отчетной документации.

## 3.5 Сборка и конфигурирование комплекса

Разработанный программный комплекс, предназначенный для имитации и управления взаимодействием между УВМ и группой СВ, состоит из трех независимых, но тесно взаимодействующих программных приложений: эмулятора СВ (svm\_app), модуля УВМ (uvm\_app) и графического интерфейса пользователя (gui\_app). Для обеспечения их корректной совместной работы и функционирования в соответствии с заданными сценариями, требуется предварительная сборка каждого компонента из исходных кодов и последующая настройка ключевых параметров их взаимодействия через единый конфигурационный файл.

Процесс сборки для модулей svm\_app и uvm\_app, реализованных на языке программирования C, стандартизирован с использованием системы сборки GNU Make и компилятора GCC. Для каждого из этих приложений подготовлен соответствующий Makefile, который определяет последовательность компиляции исходных файлов в объектные модули и их последующую компоновку в исполняемый файл. В процессе сборки учитываются необходимые зависимости от библиотек, в частности, указывается флаг -pthread для подключения поддержки многопоточности, реализуемой стандартной библиотекой POSIX Threads, и, при необходимости, флаг -lrt для компоновки с библиотекой функций реального времени. В результате выполнения команд make для каждого модуля генерируются исполняемые файлы svm\_app и uvm\_app.  
Сборка графического интерфейса пользователя gui\_app, написанного на языке C++ с использованием фреймворка Qt, осуществляется средствами системы qmake, предоставляемой Qt. Файл проекта gui.pro содержит перечень всех исходных и заголовочных файлов, описаний пользовательского интерфейса (файлов .ui), а также декларацию используемых модулей Qt, таких как core, gui, widgets и network. Сборку можно производить как из командной строки, так и с использованием интегрированной среды разработки Qt Creator, которая также служит основным инструментом для разработки и отладки данного компонента. Результатом сборки является исполняемый файл gui\_app.

Центральным элементом, определяющим параметры функционирования и взаимодействия приложений svm\_app и uvm\_app, является текстовый конфигурационный файл config.ini. Данный файл структурирован в стандартном INI-формате, предполагающем наличие именованных секций (заключаемых в квадратные скобки) и пар «ключ=значение» внутри каждой секции. Считывание и парсинг этого файла при запуске C-приложений реализованы в модуле config с использованием легковесной сторонней библиотеки inih, что обеспечивает простоту интеграции и надежность извлечения конфигурационных данных.

Содержимое файла config.ini определяет как общие параметры связи, так и индивидуальные настройки для каждого эмулируемого экземпляра СВ. В секции [communication] задаются параметры, релевантные для модуля uvm\_app: interface\_type указывает на основной транспортный протокол (в текущей реализации «ethernet»), а uvm\_keepalive\_timeout\_sec определяет интервал в секундах для механизма контроля активности TCP-соединений (Keep-Alive).

Сетевые координаты для установления соединений также задаются в config.ini. Секция [ethernet\_uvm\_target] содержит параметр target\_ip, который указывает IP-адрес хоста, на котором предполагается работа приложения svm\_app и, соответственно, прослушивание портов его эмулируемыми экземплярами СВ. Этот адрес используется модулем uvm\_app для инициирования TCP-подключений.

Индивидуальные настройки для каждого из четырех эмулируемых экземпляров СВ описываются в отдельных секциях с именами вида [settings\_svmN], где N представляет собой идентификатор экземпляра (от 0 до 3). Каждая такая секция содержит, во-первых, port – уникальный номер TCP-порта, который данный экземпляр svm\_app будет использовать для прослушивания входящих соединений от uvm\_app. Во-вторых, параметр lak определяет логический адрес (LAK), назначаемый данному экземпляру СВ, который используется для его идентификации в протокольном обмене.

Кроме того, секции [settings\_svmN] позволяют гибко конфигурировать имитацию различного поведения и нештатных ситуаций на стороне эмулятора svm\_app. Параметр simulate\_control\_failure (логического типа true/false) активирует имитацию ошибки внутреннего самоконтроля СВ. Числовой параметр disconnect\_after\_messages задает количество сообщений, после отправки которого данный экземпляр СВ принудительно разорвет TCP-соединение с УВМ (значение -1 отключает эту функцию). Флаг simulate\_response\_timeout включает режим имитации «зависания» СВ, при котором он перестает отвечать на определенные команды УВМ. Для более тонкой настройки реакции СВ на этапе инициализации предусмотрены флаг send\_warning\_on\_confirm и числовой параметр warning\_tks. Если send\_warning\_on\_confirm установлен в true, то вместо штатного сообщения «Подтверждение инициализации канала» СВ отправит сообщение «Предупреждение» с кодом типа критического события, указанным в warning\_tks. Файл config.ini также включает зарезервированную секцию [serial] для возможной будущей поддержки обмена данными через последовательный порт.

Приложение gui\_app, в свою очередь, не использует config.ini для своей работы. Параметры его подключения к uvm\_app (IP-адрес и порт сервера GUI, по умолчанию localhost:12345) заданы непосредственно в его исходном коде, либо предполагается, что uvm\_app всегда запускает свой GUI-сервер на известном порту.

Такая система конфигурации через внешний файл config.ini обеспечивает значительную гибкость при настройке и тестировании всего программного комплекса, позволяя изменять сетевые параметры, логические адреса и сценарии поведения эмулируемых СВ без необходимости перекомпиляции основных программных модулей. Это существенно упрощает отладку, проведение комплексных испытаний и адаптацию комплекса к различным условиям.

# 4 ТЕСТИРОВАНИЕ

## 4.1 Программа и методика тестирования

Текст раздела Текст раздела Текст раздела Текст раздела Текст раздела Текст раздела Текст раздела Текст раздела Текст раздела Текст раздела Текст раздела Текст раздела (см. рис. 1) …

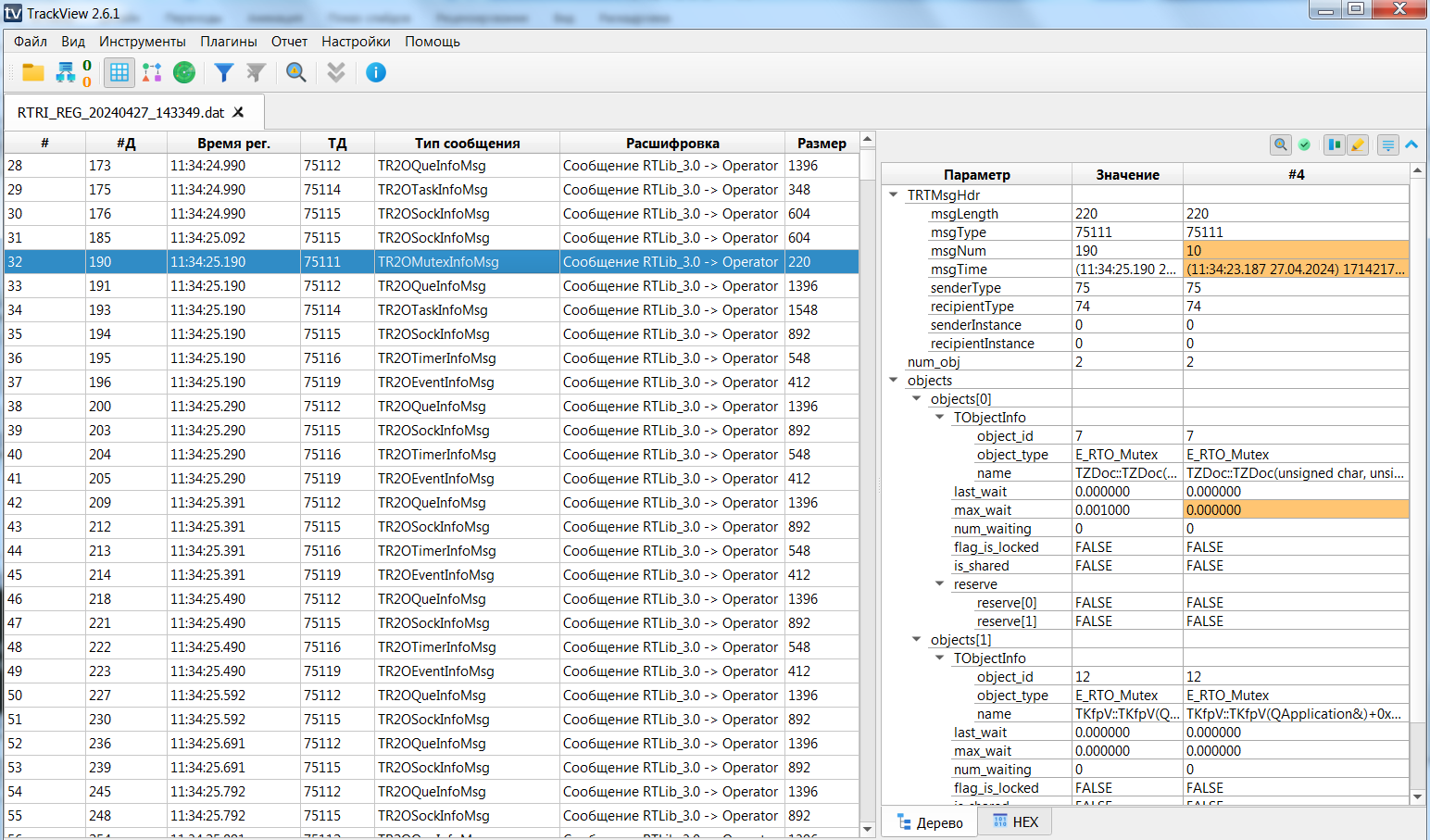


Рисунок 1 – Подрисуночная подпись

Текст раздела Текст раздела Текст раздела Текст раздела Текст раздела Текст раздела Текст раздела Текст раздела Текст раздела Текст раздела Текст раздела Текст раздела ссылка на табл. 1 …

## 4.2 Результаты тестирования

Таблица 1 – Название таблицы

|  |  |  |
| --- | --- | --- |
| **Объект** | **Команда подписки** | **Расшифровка подписки** |
| Мьютекс | E\_RTO\_SUB\_WAITGT | ожидание разблокировки мьютекса дольше N миллисекунд |
| Поток | E\_RTO\_SUB\_THR\_CREATED | поток был создан (запущен) |

Продолжение таблицы 1

|  |  |  |
| --- | --- | --- |
| **Объект** | **Команда подписки** | **Расшифровка подписки** |
|  | E\_RTO\_SUB\_THR\_STOP | потоку был выставлен флаг на завершение |
| E\_RTO\_SUB\_THR\_STOPPED | поток завершился |
| Сокет | E\_RTO\_SUB\_WRITE | запись дольше N миллисекунд при наличии свободного места |
| E\_RTO\_SUB\_READ | чтение дольше N миллисекунд при наличии данных |
| Очередь | E\_RTO\_SUB\_READ | чтение дольше N миллисекунд при наличии данных |
| E\_RTO\_SUB\_WRITE | запись дольше N миллисекунд при наличии свободного места |
| E\_RTO\_SUB\_READ\_EMPTY | чтение дольше N миллисекунд при отсутствии данных |
| E\_RTO\_SUB\_WRITE\_FULL | запись дольше N миллисекунд при отсутствии свободного места |

## 4.3 Выводы

Текст раздела Текст раздела Текст раздела Текст раздела Текст раздела Текст раздела Текст раздела Текст раздела Текст раздела Текст раздела Текст раздела Текст раздела …

# ЗАКЛЮЧЕНИЕ

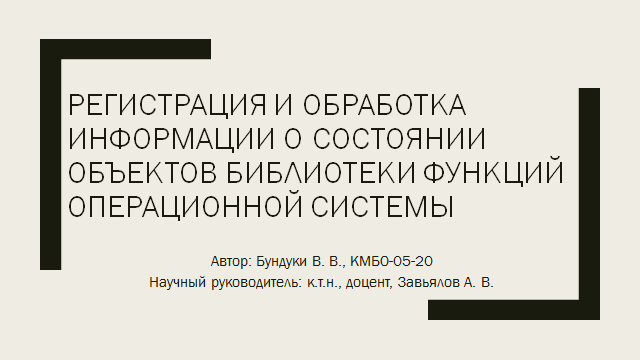
Текст заключения

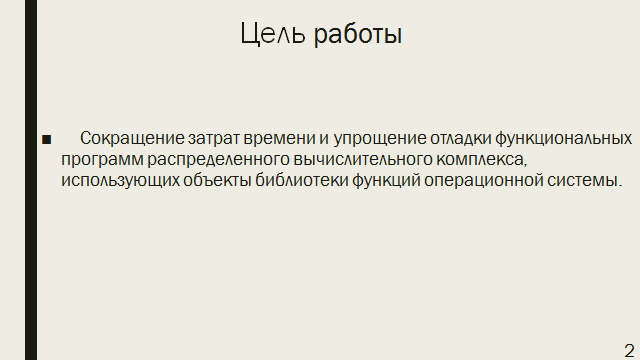
# СПИСОК ИСПОЛЬЗУЕМОЙ ЛИТЕРАТУРЫ

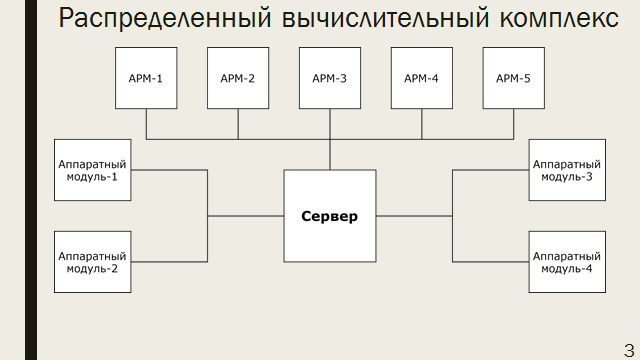
1. Страуструп, Б. Язык программирования C++ (стандарт C++11). Краткий курс. / Б. Страуструп. — М.: Бином, 2019. — 176 с.
2. Буч Г., Рамбо Д., Якобсон И. Язык UML. Руководство пользователя. 2-е изд.: Пер. с англ. Мухин Н. — М.: ДМК Пресс, 2006. — 496 с.
3. Шаблон архитектуры BCE [Электронный ресурс]. — URL: https://www.ndpsoftware.com/OpenUpBasic/openup\_basic/guidances/concepts/entity\_control\_boundary\_pattern,\_uF-QYEAhEdq\_UJTvM1DM2Q.html (дата обращения: 30.04.2025).
4. Документация Qt [Электронный ресурс] — URL: https://doc.qt.io/qt‑5/reference-overview.html (дата обращения: 03.05.2024).
5. Документация Lua [Электронный ресурс]. — URL: <https://www.lua.org/manual/5.4/> (дата обращения 10.05.2025).

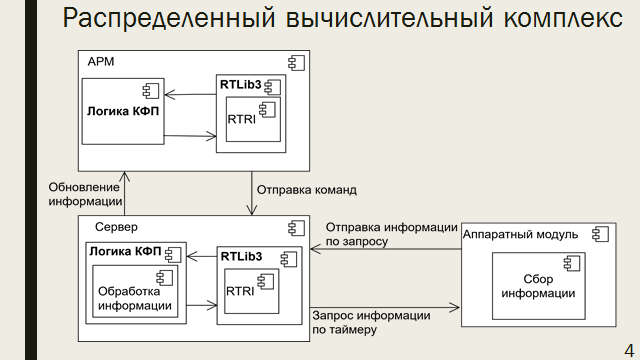
ПРИЛОЖЕНИЕ А

ГРАФИЧЕСКИЕ МАТЕРИАЛЫ









ПРИЛОЖЕНИЕ Б

ЗАДАНИЕ НА ВЫПОЛНЕНИЕ ВКР

<<< после печати заменить лист экземпляром задания >>>

<<< номера страниц проставить вручную, черной ручкой >>>

ПРИЛОЖЕНИЕ В

ОТЗЫВ РУКОВОДИТЕЛЯ

<<< после печати заменить лист экземпляром отзыва >>>

<<< номера страниц проставить вручную, черной ручкой >>>

ПРИЛОЖЕНИЕ Г

АКТ ВНЕДРЕНИЯ

<<< после печати заменить лист экземпляром акта >>>

<<< номера страниц проставить вручную, черной ручкой >>>

ПРИЛОЖЕНИЕ Д

ИСХОДНЫЙ КОД И ДИСТРИБУТИВ

Место крепления конверта   
для компакт-диска

**«УТВЕРЖДАЮ»**

Директор по НИОКР

АО «Концерн «Вега»

\_\_\_\_\_\_\_\_\_\_\_\_\_ Мекекечко В.В.

«\_\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 2025 г.

**АКТ ВНЕДРЕНИЯ**

Настоящий акт выдан в том, что результаты выпускной квалификационной работы Торкина Д. А. на тему «Имитация процедур управления спецвычислителями радиолокатора космического базирования» внедрены и используются в разработках АО «Концерн «Вега».

!!! при необходимости консультант добавляет информацию о степени завершенности работы, качестве результатов, готовности к использованию

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
| Начальник отдела №40 |  |  |  | Четыркин Д.Ю. |
|  |  | *подпись* |  |  |
| Консультант |  |  |  | Ветюгов С.В. |
|  |  | *подпись* |  |  |