**ОБЩАЯ ИДЕЯ**

Протокол реализует механизм удалённого вызова функций (Remote Procedure Call, RPC) поверх канального уровня с проверкой целостности и поддержкой нескольких типов сообщений.

Транспортный уровень отвечает за упаковку/распаковку сообщений с именами функций и аргументами, а канальный уровень обеспечивает доставку пакетов с контролем ошибок.

**1. Канальный уровень**

Формат кадра (фрейма):

* SOF (0xFA) — стартовый байт фрейма.
* LEN (2 байта) — общая длина (от SOD до EOF включительно).
* HDR\_CRC (1 байт) — контрольная сумма заголовка (SOF + LEN).
* SOD (0xFB) — стартовый байт полезных данных.
* PAYLOAD — транспортное сообщение.
* PKT\_CRC (1 байт) — CRC8 (SOD + PAYLOAD).
* EOF (0xFE) — стоповый байт.

Таким образом обеспечивается целостность как заголовка, так и содержимого.

**2. Транспортный уровень**

Формат сообщения:

* TYPE — тип сообщения:
* MSG\_REQ (0x0B) — запрос.
* MSG\_STREAM (0x0C) — потоковое сообщение без ответа.
* MSG\_RESP (0x16) — ответ.
* MSG\_ERR (0x21) — ошибка.
* SEQ — порядковый номер (для сопоставления запрос ↔ ответ).
* NAME — текстовое имя вызываемой функции (минимум 1 символ, максимум MAX\_FUNC\_NAME\_LEN).
* 0x00 — терминатор имени.
* ARGS — аргументы вызова в бинарной форме.

**ПОЛНАЯ АРХИТЕКТУРА RPC ФРЕЙМВОРКА**

|  |  |
| --- | --- |
| **Application Layer**  examples/\*.c (e.g. ping\_pong.c) | Пользовательский код. Регистрирует RPC-функции (rpc\_register()), вызывает удаленные функции (rpc\_request(), rpc\_stream()) и реализует бизнес-логику. |
| **Public API**  (rpc.h, rpc.c) | Высокоуровневый интерфейс для приложений: инициализация, регистрация функций, выполнение RPC-вызовов. |
| **Transport Layer**  (rpc\_transport.c, rpc\_transport.h) | Управление запросами и ответами, управление ожиданием (waiter management), пул worker-ов, диспетчеризация вызовов на зарегистрированные функции. |
| **Link Layer**  (rpc\_link.h, rpc\_link.c) | Формирование кадров (фреймов), расчет CRC, кодирование/декодирование пакетов, парсинг. |
| **Physical Layer**  (rpc\_phy.h + platform implementation) | Аппаратная абстракция (для UART, SPI и т.д.). |
| **OS ABSTRACTION LAYER**  (rpc\_osal.h + platform implementation) | Абстракции для потоков (threads), очередей (queues), мьютексов (mutexes), семафоров (semaphores) (для POSIX, FreeRTOS, …). |
| **Operating System**  (Linux, FreeRTOS, Windows) | Базовая операционная система, предоставляющая ядро и планировщик задач. |

**РЕАЛИЗАЦИЯ OS ABSTRACTION LAYER В RPC ФРЕЙМВОРКЕ**

OS Abstraction Layer (OSAL) — это уровень абстракции операционной системы, предназначенный для того, чтобы RPC-фреймворк мог работать на разных платформах (например, Linux/POSIX, RTOS для микроконтроллеров) без изменения основной логики.

Все интерфейсы определены в заголовочном файле rpc\_osal.h, а конкретная реализация должна быть размещена в папке platform/ (например, platform/linux/rpc\_osal\_linux.c).

**Основные задачи OSAL:**

1. **Абстракция потоков (threads)**

* Создание и управление потоками.

1. **Очереди сообщений (queues)**

* Используются для взаимодействия между слоями (например, Link Layer ↔ Transport Layer).

1. **Семафоры (binary semaphores)**

* Обеспечивают синхронизацию между потоками.
* Используются для реализации ожидания ответа на RPC-запросы.

1. **Мьютексы (mutexes)**

* Используются для защиты общих ресурсов (например, таблицы зарегистрированных функций).

1. **Вспомогательные функции**

* os\_delay\_ms(ms) — задержка выполнения на указанное количество миллисекунд.

**Примеры реализаций:**

* **POSIX - совместимые системы.** Реализация на основе pthreads, POSIX очередей сообщений и семафоров для Linux и других UNIX-подобных систем.
* **FreeRTOS.** Реализация для встроенных систем на основе FreeRTOS API с использованием задач, очередей и семафоров RTOS.
* **Windows.** Реализация для Windows на основе WinAPI threads, events и critical sections.

**РЕАЛИЗАЦИЯ PHYSICAL LAYER В RPC ФРЕЙМВОРКЕ**

Physical Layer (PHY) — это базовый уровень RPC-фреймворка, который обеспечивает передачу и приём «сырых» байтов данных по выбранному каналу связи.

Этот уровень является абстрактным: интерфейсы заданы в rpc\_phy.h, а конкретная реализация должна быть предоставлена для каждой аппаратной платформы и размещена в папке platform/ (например, platform/linux/rpc\_phy\_linux.c).

**Основные задачи PHY:**

1. **Инициализация канала связи**

* Настраивает физический интерфейс (например, UART, SPI).
* Вызывается первым, перед использованием любых функций PHY.

1. **Передача данных (TX)**

* Функция rpc\_phy\_send() отправляет массив байтов.
* Может быть блокирующей (ожидает завершения передачи) или неблокирующей — зависит от реализации.

1. **Приём данных (RX)**

* Функция rpc\_phy\_receive() получает массив байтов.
* Обычно является блокирующей, ожидая появления данных в канале.
* Используется RX-потоком из Link Layer для побайтной подачи данных в парсер.

1. **Деинициализация**

* rpc\_phy\_deinit() закрывает соединение и освобождает ресурсы (например, close() в POSIX или отключение UART в MCU).

**Интерфейс (rpc\_phy.h)**

* int rpc\_phy\_init(void); — инициализация канала.
* int rpc\_phy\_send(const uint8\_t \*data, size\_t len); — передача данных.
* int rpc\_phy\_receive(uint8\_t \*data, size\_t len); — приём данных.
* void rpc\_phy\_deinit(void); — деинициализация.

**Примеры реализации**

* Linux (POSIX): используется файловый интерфейс (open, read, write).
* Микроконтроллер (STM32, FreeRTOS): может использоваться драйвер HAL\_UART\_Transmit и HAL\_UART\_Receive, либо DMA для асинхронной работы.

**РЕАЛИЗАЦИЯ LINK LAYER В RPC ФРЕЙМВОРКЕ**

Link Layer представляет собой критически важный компонент системы, ответственный за обеспечение надежной передачи данных между узлами. Слой реализует функции обрамления пакетов, контроля целостности и управления потоком данных между transport и physical уровнями. Модуль полностью опирается на OSAL. Благодаря этому код не зависит от конкретной ОС (Linux, FreeRTOS и др.).

**Основные задачи Link Layer:**

1. **Парсинг входящих данных (RX)**

* Используется конечный автомат, который последовательно обрабатывает входящие байты.
* Проверяются: маркер начала кадра (SOF), длина пакета, CRC заголовка, маркер начала данных (SOD), CRC полезной нагрузки, маркер конца кадра (EOF).
* При успешной сборке кадра полезная нагрузка передаётся в очередь qLinkToTrans для транспортного уровня.

1. **Формирование исходящих кадров (TX)**

* Из полезной нагрузки собирается полный кадр: SOF, длина, CRC заголовка, SOD, данные (payload), CRC пакета, EOF.
* Готовый кадр отправляется в PHY через rpc\_phy\_send().

1. **Управление потоками RX/TX**

* Поток RX (приём) читает байты из PHY и передаёт их в парсер (rpc\_link\_feed\_bytes).
* Поток TX (передача) получает сообщения из очереди qTransToLink и формирует кадры через rpc\_link\_build\_frame.

**Структура модуля:**

* Парсер (rpc\_link\_feed\_bytes). Реализован как конечный автомат с состояниями: WAIT\_SOF → READ\_LEN1 → READ\_LEN2 → READ\_HDRCRC → WAIT\_SOD → READ\_PAYLOAD → READ\_PKTCRC → WAIT\_EOF.
* Функция сборки кадров (rpc\_link\_build\_frame). Добавляет служебные поля (заголовок, CRC, EOF) и вызывает PHY для отправки.
* Очереди:
  + qLinkToTrans — передача полезной нагрузки из Link Layer в Transport Layer.
  + qTransToLink — исходящие данные от Transport Layer, которые нужно упаковать и отправить.
* Потоки:
  + ThreadRX — постоянно читает байты из PHY и передаёт их в парсер.
  + ThreadTX — получает сообщения из Transport Layer и упаковывает их в кадры для PHY.

**РЕАЛИЗАЦИЯ TRANSPORT LAYER В RPC ФРЕЙМВОРКЕ**

Transport Layer представляет собой центральный компонент RPC системы, ответственный за управление удаленными вызовами, обработку запросов и ответов, а также координацию работы между различными слоями системы. Этот уровень обеспечивает надежную доставку сообщений и управление жизненным циклом RPC вызовов.

**Ключевые компоненты:**

1. **Система регистрации функций.** Централизованный реестр зарегистрированных функций с поддержкой:

* Динамической регистрации обработчиков
* Поиска функций по имени
* Потокобезопасного доступа к реестру

1. **Механизм ожидания ответов (Waiters).** Таблица активных запросов, ожидающих ответов:

* Уникальные sequence numbers для идентификации запросов
* Семафоры для синхронизации ожидания ответов
* Таймауты обработки запросов

1. **Система очередей сообщений.** Набор очередей для межслойного взаимодействия:

* qTransToLink - исходящие сообщения к Link Layer
* qLinkToTrans - входящие сообщения от Link Layer
* qRpcRequests - внутренняя очередь запросов к worker-потокам

**Отправка REQUEST-сообщений**

Отправитель вызывает функцию rpc\_request(). Она обеспечивает отправку запроса на удаленную сторону и ожидание ответа с заданным таймаутом, реализуя классическую модель запрос-ответ. Реализация поддерживает параллельные вызовы rpc\_request() из разных потоков.

**Алгоритм работы:**

1. **Выделение ресурсов отслеживания.** Для каждого вызова выделяется структура "ожидателя" (waiter), которая служит для отслеживания состояния конкретного запроса. Эта структура содержит порядковый номер запроса, информацию о буфере ответа, его емкости, а также механизмы синхронизации для ожидания ответа.
2. **Формирование сообщения.** Создается сообщение, содержащее всю необходимую информацию для удаленного вызова. Сообщение включает последовательный номер для идентификации, имя вызываемой функции, аргументы вызова, а также служебную информацию для обеспечения надежной доставки.
3. **Отправка через транспортный уровень.** Сформированное сообщение помещается в очередь для отправки на Link-уровень. Этот этап обеспечивает разделение ответственности между слоями системы и поддерживает асинхронную отправку данных.
4. **Ожидание ответа.** После отправки запроса функция переходит в состояние ожидания ответа. Используется семафор с таймаутом, который активируется при получении ответа или по истечении заданного времени ожидания.
5. **Обработка результата.** По завершении ожидания функция анализирует результат операции. В случае успеха данные ответа уже находятся в предоставленном буфере, а в переменную длины записывается фактический размер полученных данных. При ошибке возвращается соответствующий код.
6. **Освобождение ресурсов.** Независимо от результата выполнения, функция гарантирует освобождение всех выделенных ресурсов, включая структуру отслеживания запроса.

**Прием и обработка сообщений**

Система приема сообщений представляет собой многоуровневый конвейер обработки входящих данных, обеспечивающий надежное распределение и выполнение удаленных вызовов. Механизм построен по принципу разделения ответственности между специализированными компонентами.

**Архитектура обработки:**

* **Транспортный уровень (ThreadTrans).** Основной цикл обработки запускается в потоке транспортного уровня, который ожидает поступления сообщений из qLinkToTrans (link-слоя). Каждое полученное сообщение передается на разбор и дальнейшую диспетчеризацию.
* **Демультиплексирование сообщений.** Входящие сообщения разделяются на две принципиально разные категории:
  + Ответы на запросы (RESPONSE/ERROR) - завершают ожидающие вызовы
  + Входящие запросы (REQUEST/STREAM) - инициируют выполнение операций

**Обработка ответов (RESPONSE/ERROR):**

* **Поиск ожидающего запроса**. Для каждого ответа система ищет соответствующий ожидающий запрос по уникальному последовательному номеру. Этот механизм обеспечивает корреляцию запросов и ответов в условиях параллельной работы.
* **Копирование данных ответа.** При успешном нахождении ожидающей структуры (waiter), данные ответа копируются в предоставленный пользователем буфер. Выполняется проверка на переполнение буфера, гарантирующая целостность данных.
* **Уведомление о завершении.** После обработки ответа система активирует семафор ожидания, что разблокирует поток, ожидающий результата вызова. Этот механизм обеспечивает минимальную задержку между получением ответа и пробуждением ожидающего потока.

**Обработка входящих запросов (REQUEST/STREAM):**

* **Формирование задачи выполнения.** Входящие запросы преобразуются в опеределенную структуру, содержащую всю необходимую информацию для выполнения: имя функции, аргументы, тип запроса.
* **Постановка в очередь задач.** Сформированные задачи помещаются в очередь для обработки worker-потоками (qRpcRequests).

**Worker-потоки обработки:**

* **Пул обработчиков.** Система создает конфигурируемое количество worker-потоков, которые параллельно обрабатывают входящие запросы. Каждый worker работает в бесконечном цикле, извлекая задачи из очереди (qRpcRequests). Если система сконфигурирована на один воркер, то запросы будут обрабатываться по очереди.
* **Поиск и вызов обработчиков.** Для каждого запроса worker ищет зарегистрированную функцию-обработчик по имени. Поддерживается как синхронный вызов для запросов типа REQUEST, так и выполнение без ответа для STREAM-сообщений.
* **Формирование ответов.** После выполнения обработчика worker формирует ответное сообщение, содержащее результат выполнения или код ошибки. Ответ отправляется в очередь qTransToLink (в link-уровень).

**Отправка STREAM-сообщений**

Функция rpc\_trans\_stream() реализует механизм асинхронной отправки сообщений по принципу "fire-and-forget" (отправил и забыл). Этот тип сообщения предназначен для случаев, где не требуется подтверждение доставки или ответ от удаленной стороны. Используется для уведомлений, логов, телеметрии.

**Алгоритм работы:**

* Формирование сообщения
* Отправка в транспортный уровень. Сформированное сообщение помещается в очередь для отправки на link-уровень (qTransToLink).

**ПОРТИРОВАНИЕ НА НОВЫЕ ПЛАТФОРМЫ**

Система разработана с соблюдением принципов модульности и абстракции, что обеспечивает прямое портирование на различные программно-аппаратные платформы. Архитектура четко разделяет общую логику RPC от платформо-зависимых реализаций.

**Процесс портирования:**

* **Реализация физического уровня (rpc\_phy.h).** Требуется реализация функций низкоуровневой передачи данных, обеспечивающих базовые операции отправки и приема байтовых потоков. Интерфейс абстрагирует конкретные транспортные механизмы, позволяя использовать различные среды коммуникации.
* **Реализация OS abstraction layer (rpc\_osal.h).** Необходимо предоставить реализации примитивов операционной системы, включая механизмы многопоточности, синхронизации, межпроцессного взаимодействия и управления временем. Слой обеспечивает единообразный API поверх различных операционных систем.

**Текущая поддержка платформ:**

* **Реализация для Linux.** Система включает готовую реализацию для POSIX-совместимых систем, расположенную в директории platform/linux. Реализация состоит из двух ключевых компонентов:
* **Физический уровень** - основан на именованных каналах (FIFO), обеспечивающих межпроцессное взаимодействие в рамках одной системы.
* **OS abstraction layer** - использует POSIX API для предоставления необходимых сервисов: pthreads для многопоточности, POSIX mutex/semaphore для синхронизации, delays и стандартные механизмы работы с очередями.

**ВЫЯВЛЕННЫЕ НЕДОСТАТКИ ТЕКУЩЕГО ПРОТОКОЛА**

1. **Ограничения идентификаторов запросов**

Проблема: порядковый номер (SEQ) имеет размер 1 байт → максимум 255 активных запросов.

Риск: может оказаться недостаточным для высоконагруженных или многопоточных систем.

1. **Неэффективное использование идентификаторов функций**

Проблема: функция идентифицируется строковым именем. Это увеличивает размер пакета и требует строковых операций.

Риск: вероятность ошибок (опечатки в имени), низкая эффективность на микроконтроллерах с ограниченными ресурсами.

1. **Отсутствие механизмов безопасности**

Проблема: протокол не предусматривает аутентификацию и шифрование.

Риск: любой узел может инициировать RPC-вызовы или подменять сообщения.

1. **Отсутствие гарантий доставки**

Проблема: отсутствует механизм подтверждения доставки и повторной отправки.

Следствие: потерянные пакеты приводят к "немым" ошибкам и таймаутам.

1. **Нет управления версиями протокола**

Проблема: в пакете отсутствует поле версии протокола.

Риск: возможна несовместимость при взаимодействии устройств с разными реализациями.

1. **Нет поддержки больших сообщений (payload)**

Проблема: полезная нагрузка ограничена максимальным размером кадра.

Риск: невозможно передавать файлы конфигурации, прошивки или изображения.

**ПРЕДЛАГАЕМЫЕ УЛУЧШЕНИЯ**

1. **Увеличение емкости идентификаторов запросов**

Решение: расширить SEQ до 2 байт (65535 активных запросов).

1. **Оптимизация идентификаторов функций**

Решение: ввести числовые ID функций.

* На обеих сторонах поддерживается таблица соответствия {ID ↔ handler}.
* Уменьшение размера пакетов.
* Ускорение парсинга и устранение ошибок от опечаток.

1. **Добавление механизмов безопасности**

Решение: ввести опциональные поля безопасности в заголовок:

* Message Authentication Code (MAC) — для проверки целостности и аутентичности сообщений (например, HMAC-SHA256).
* Опциональное шифрование payload — для защиты конфиденциальности данных.

1. **Гарантии доставки**

Решение: добавить механизм подтверждения приёма:

* Ввести ACK/NACK пакеты.
* Реализовать автоматическую повторную отправку при таймауте.

1. **Управление версиями протокола**

Решение: добавить поле версии в заголовок пакета, что обеспечит совместимость между устройствами.

1. **Поддержка больших payload (фрагментация)**

Решение: ввести механизм фрагментации:

* Флаг "фрагментировано" в заголовке.
* Номер фрагмента и общее количество фрагментов.
* Сборка сообщения на приёмной стороне.