МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«Национальный исследовательский ядерный университет «МИФИ»
Обнинский институт атомной энергетики –
филиал федерального государственного автономного образовательного учреждения высшего образования «Национальный исследовательский ядерный университет «МИФИ»
(ИАТЭ НИЯУ МИФИ)

Отделение Интеллектуальные кибернетические системы
Направление подготовки Информатика и вычислительная техника

Научно-исследовательская работа

Анализ кодогенераторов для CANopen

Студент группы ИВТ-Б22 _____ Карасев Н. А.

Руководитель
инженер-программист _____ Жильцов Д. И.

Обнинск, 2025 г

РЕФЕРАТ

Работа 43 стр., 0 табл., 0 рис., 12 ист.

Ключевые слова: CAN, CANOPEN, RUST

Написать нормальный реферат в конце

СОДЕРЖАНИЕ

ТЕРМИНЫ И ОПРЕДЕЛЕНИЯ

В настоящем отчете о НИР применяют следующие термины с соответствующими определениями:

Controller Area Network (CAN) — шина обмена сообщениями.

CANopen — протокол связи на основе CAN-шины.

PDO (Process Data Object) — объект CANopen для передачи процессных данных в реальном времени; как правило, это короткие сообщения с минимальными накладными расходами, предназначенные для циклического или событийного обмена.

SDO (Service Data Object) — объект CANopen для конфигурации и диагностики устройства; обеспечивает чтение и запись параметров словаря объектов и доступ к сервисной информации.

OD (Object Dictionary, словарь объектов) — структурированный набор параметров, команд и диагностических данных узла CANopen, организованный как адресуемые записи, к которым обращаются стандартными механизмами протокола.

EDS (Electronic Data Sheet) — файл стандартизированного описания словаря объектов устройства CANopen, используемый конфигураторами и сервисными утилитами для автоматической настройки и интеграции.

DCF (Device Configuration File) — файл конфигурации устройства CANopen, представляющий собой EDS с добавленными (или изменёнными) параметрами конкретной установки/проекта, применяемый для развёртывания одинаковых настроек.

TOML — конфигурационный формат, является более удобным для ручной конфигурации аналогом более известного json.

ПЕРЕЧЕНЬ СОКРАЩЕНИЙ И ОБОЗНАЧЕНИЙ

В настоящем отчете о НИР применяют следующие сокращения и обозначения:

CAN — Controller Area Network

SDO — Service Data Objects

PDO — Process Data Objects

OD — Object Dictionary

EDS — Electronic Data Sheet

DCF — Device Configuration File

ВВЕДЕНИЕ

Прежде чем перейти к рассмотрению столь узкой темы, необходимо объяснить, что представляет собой шина CAN.

Например, при разработке электронных устройств нередко возникает ситуация, когда отдельные компоненты (например, датчик) разнесены: датчик находится в одном месте, а блок обработки сигналов — в другом. В таком случае самым простым решением будет соединить их проводами. Однако этот подход не всегда оптимален: рано или поздно возникнет проблема повсеместного разрастания проводов и кабелей. Значительные затраты материалов на проводку — не самая большая трудность; куда сложнее затем обслуживать такую систему, поскольку крайне трудно разобраться среди десятков и сотен различных проводов. Объединение части кабелей в жгуты лишь частично решает проблему. Следующим логическим шагом является переход от соединений «точка-точка» к шинной архитектуре, при которой сигналы между устройствами передаются по общей линии.

Однако при переходе к шине возникает новая задача: как сделать так, чтобы устройство принимало только те сигналы, которые предназначены именно ему? Можно ввести временное мультиплексирование, но это не подходит для систем реального времени, где на счету каждая секунда или чрезвычайно высока цена ошибки. Один из способов обеспечить обмен данными между десятками электронных блоков в автомобиле или промышленной установке по одной общей шине — так, чтобы это было надёжно, детерминировано по времени и устойчиво к помехам — состоит во внедрении шины CAN.

Шина CAN (Controller Area Network) — это система связи, используемая в транспортных средствах (например, автомобилях), которая позволяет электронным блокам управления (ЭБУ) обмениваться данными друг с другом без участия главного компьютера. Например, шина CAN обеспечивает быстрый и надёжный обмен информацией между тормозной системой и двигателем вашего автомобиля. [1]

Для решения вышеописанных проблем CAN предлагает простое и в то же время мощное решение — задание каждому сообщению своего идентификатора. В такой системе:

Каждый узел обрабатывает только то сообщение, которое предназначено именно ему.

Арбитраж происходит без разрушения кадра — при одновременной передаче сигналов от нескольких узлов побеждает то сообщение, идентификатор которого обладает более высоким приоритетом.

CAN самостоятельно контролирует корректность данных на уровне канала.

Однако CAN — лишь шина: она обеспечивает только транспорт для коротких сообщений и не определяет их содержимое. Для этого требуется надстроечный протокол на более высоком уровне абстракции. Эту роль выполняет CANopen.

Стандарт CANopen полезен тем, что обеспечивает готовую к использованию совместимость между устройствами (узлами), например промышленным оборудованием. Кроме того, он предоставляет стандартные методы конфигурирования устройств — в том числе и после установки. [2]

CANopen задаёт общий прикладной каркас: определяет, как устройства описывают свои параметры, как ими управлять, как передавать «процессные» данные, как диагностировать аварии и как сеть в целом функционирует от включения питания до штатной работы.

Протокол CANopen характеризуется шестью ключевыми особенностями [2]:

1. **Три модели взаимодействия узлов.** Master/slave, client/server и producer/consumer. Модель master/slave нужна там, где один узел («master», или управляющий узел) инициирует сетевые действия и управляет жизненным циклом других узлов («slave»): запускает, останавливает, сбрасывает. Модель client/server характерна для запросно-ответного обмена: один узел выступает клиентом, который читает или пишет параметр, другой — сервером, который обслуживает запрос. Наконец, модель producer/consumer описывает потоковую публикацию данных: один узел производит (producer) сообщения с измерениями/состояниями, а несколько потребителей (consumers) их принимают, не требуя явной адресации или подтверждения для каждого получателя.

2. **Коммуникационные объекты и связанные с ними протоколы CANopen.** В CANopen принято говорить, что обмен строится вокруг communication objects — стандартных типов сообщений, у которых есть ясная роль. Вот самые широкоиспользуемые из них:

   – SDO (Service Data Objects) — «сервисный» канал, используемый в первую очередь для конфигурации и диагностики: прочитать параметр, записать параметр, получить сведения об ошибках, задать режим работы.

   – PDO (Process Data Objects) — наоборот, канал для оперативных данных «процесса» в реальном времени: короткие сообщения, минимальные накладные расходы, рассчитанные на регулярный обмен командами и обратной связью.

   – NMT (Network Management) — механизм управления сетью и состояниями устройств, отвечает за то, в каком режиме находится узел и можно ли ему «доверять» процессный обмен.

   – SYNC (Synchronization) — используется для синхронизации действий узлов, например, для синхронной выборки данных или привязки отправки PDO к «такту».

   – TIME (Time Stamp) — предназначено для передачи сетевой временной метки.

   – EMCY (Emergency) — применяется узлом для немедленной сигнализации об ошибке. EMCY имеет высокий приоритет на шине за счёт низкого идентификатора.

   – HEARTBEAT — периодическая рассылка текущего NMT-состояния узлом.

   Основными являются SDO и PDO. Говоря по-простому: SDO предназначен для настройки и проверки, PDO — для работы.

3. **CANopen — автомат.** CANopen представляет собой формализованную модель состояний узла с управлением через NMT. Для CANopen устройство не просто «подключено к CAN», оно находится в одном из

определённых состояний (например, состояние конфигурирования, нормальной работы или остановки). Это важно, потому что многие действия допустимы не всегда: как правило, конфигурацию выполняют, когда устройство ещё не участвует в процессном обмене, а затем переводят узел в рабочее состояние. В этой модели управляющий узел может переводить другие узлы между состояниями — например, выполнить reset. В результате сеть функционирует более предсказуемо: запуск системы — это не «каждый стартует как хочет», а воспроизводимый сценарий.

4. **Object Dictionary (OD), словарь объектов устройства.** Это ключевая «семантическая база» CANopen: каждый узел содержит таблицу параметров, команд и диагностических полей, организованную как адресуемые записи. Именно OD определяет, что означает конфигурация устройства и как к ней обращаться. Практический смысл прост: вместо того чтобы каждый производитель придумывал свой способ настройки, CANopen устанавливает правило: «все параметры лежат в словаре, и доступ к ним осуществляется стандартным способом». Доступ к OD чаще всего идёт через SDO: клиент считывает или задаёт конкретные элементы словаря. Таким образом, OD и SDO концептуально связаны: OD — модель данных, SDO — стандартный инструмент доступа.

5. **EDS (Electronic Data Sheet), электронная спецификация словаря объектов.** Если OD — это содержимое внутри устройства, то EDS — формализованное описание этого содержимого «снаружи» в стандартном файловом формате. Его ценность проявляется в инструментах: сервисные утилиты и конфигураторы могут загрузить EDS и понять, какие параметры существуют, какие типы данных и прочее. Это снижает зависимость от ручной интеграции и облегчает обслуживание.

6. **DCF (Device Configuration File), профили устройств.** Предположим, завод приобрёл сервомотор ServoMotor3000 для интеграции в конвейер. При этом специалист редактирует EDS устройства, добавляя специфичные для интеграции данные — например, задавая битрейт

и идентификатор узла. Модифицированный EDS можно экспортировать как файл конфигурации устройства (DCF).

Логичным продолжением разговора о CANopen становится вопрос о практической реализации: какие программные средства позволяют «оживить» описанные концепции в коде и связать их с реальной CAN-шиной. На этом этапе фокус смещается от модели протокола к инженерному инструментарию — драйверам и библиотекам для работы с CAN, а также к стеку или фреймворку, который берёт на себя логику CANopen либо предоставляет удобные примитивы для её построения.

Исторически основная часть библиотек и стеков для CAN и CANopen реализована на C (C++). Причина тривиальна: C долгое время доминировал в embedded-разработке, где CAN наиболее распространён; этот язык обеспечивает предсказуемость по ресурсам, простоту портирования на микроконтроллеры и хорошую совместимость с существующими драйверами и RTOS. Поэтому именно в C-экосистеме накоплен максимальный объём «полевого» опыта — от базового доступа к CAN-интерфейсу до полноценных CANopen-стеков, профилей устройств и средств конфигурирования.

Таким образом, рассматриваемая область давно «закрыта» с точки зрения существующих решений. Однако сама языковая база, на которой держится эта экосистема, заметно устарела: C остаётся эффективным и привычным, но его модель безопасности и контроля ошибок всё хуже соответствует современным требованиям к надёжности, поддерживаемости и устойчивости системного кода.

Сегодня Rust всё чаще рассматривается как естественный преемник C для задач низкоуровневого программирования. Этот язык сохраняет возможность работать близко к аппаратуре и управлять ресурсами, при этом предлагая более строгие гарантии корректности. Однако фреймворков и библиотек для CAN/CANopen на Rust значительно меньше, чем их аналогов на C, и в среднем они находятся на более ранней стадии зрелости. Для многих проектов характерны ограниченная полнота реализации, менее стабильные API, а также меньшая проверенность в долговременных промышленных системах. Иными словами, экосистема Rust в данной области остаётся фрагментированной и во многом незаполненной. Именно этот вакуум и определяет моти-

вацию дальнейшей работы.

Далее будут рассмотрены конкретные решения для работы с CAN и CANopen, их архитектурные подходы, зрелость и границы применимости.

1 Методология исследования

Данная глава посвящена методологии исследования, направленного на анализ существующих CANopen-стеков, реализованных на языке Rust. В ней сформулированы цель и задачи работы, обоснован выбор объектов исследования, определены критерии сравнительного анализа выбранных реализаций, а также объяснена актуальность проведения такого анализа в контексте задач дипломной работы.

1.1 Цели и задачи исследования

Основная цель данного исследования заключается в комплексном анализе существующих реализаций протокольного стека CANopen на Rust, с тем чтобы выявить их архитектурные особенности, степень функциональной полноты и ограничения. Эта цель продиктована текущей ситуацией в экосистеме Rust: по состоянию на 2025 год отсутствует полноценный и зрелый CANopen-стек, широко принятый сообществом разработчиков [3]. Вместо этого существует несколько отдельных проектов, находящихся на разных стадиях разработки и ориентированных на различные применения. В подобных условиях важно определить, какие аспекты уже реализованы в этих проектах, а какие остаются пробелами, требующими внимания.

Для достижения поставленной цели в работе решаются следующие задачи:

1. Формирование корпуса проектов: отбор открытых Rust-проектов, заявляющих поддержку CANopen либо предназначенных для работы с CAN-сетями, используемыми в CANopen.

2. Сбор данных по проектам на дату среза (репозиторий, документация, активность разработки, наличие тестов и примеров).

3. Анализ заявленного функционала по документации и исходному коду.

4. Сопоставление проектов по набору критериев и фиксация результатов.

5. Формулирование выводов о применимости проектов и выявление функциональных пробелов экосистемы.

1.2 Объект и предмет исследования

Объект исследования — программные реализации стека протоколов CANopen, разработанные на языке Rust. Данный объект охватывает программные средства, предназначенные для организации обмена данными по протоколу CANopen в системах с шиной CAN, включая как встроенные устройства, так и пользовательские приложения.

Предмет исследования — архитектурные и функциональные особенности таких реализаций, а также методологические подходы к их анализу и сравнению. Иными словами, в фокусе находятся конкретные свойства и характеристики Rust-реализаций CANopen (их структура, поддерживаемые возможности, степень соответствия стандарту, и т.п.).

В рамках данной работы объектами исследования являются три открытых программных проекта на языке Rust, реализующие поддержку протокола CANopen либо связанные с его практическим использованием: Zencan, OZE-CanOpen и Ican.

Выбор указанных проектов обусловлен их репрезентативностью для текущего состояния экосистемы CANopen на Rust и соответствием целям исследования. Zencan и OZE-CanOpen представляют собой наиболее заметные на момент написания работы реализации CANopen в экосистеме Rust, ориентированные на различные сценарии применения: создание встроенных CANopen-узлов и разработку мастер-узлов или анализаторов сети соответственно. Эти проекты обладают открытым исходным кодом, публичной документацией и активным обсуждением в сообществе, что делает возможным их детальный анализ и сопоставление.

В качестве дополнительного объекта исследования включён проект Ican, который не является полноценным CANopen-стеком, но представляет иной класс решений — инженерные инструменты для диагностики, мониторинга и интерпретации CAN трафика в целом и CANopen в частности. Его рассмотрение позволяет расширить контекст анализа и учесть практический аспект работы с CANopen-сетями, выходящий за рамки реализации протокольного

стека как такового.

Таким образом, выбранные проекты в совокупности охватывают ключевые роли в экосистеме CANopen: реализацию узлов, поддержку мастер-функциональности и инструментарий диагностики, что обеспечивает содержательную полноту сравнительного анализа.

1.3 Критерии анализа CANopen-стеков

Для систематического анализа выбранных реализаций были определены ключевые критерии, охватывающие как соответствие требованиям протокола CANopen, так и общие показатели качества программных библиотек:

- **Назначение и область применения.** Данный критерий отражает исходное позиционирование проекта, предполагаемые сценарии его использования и целевую среду эксплуатации. Особое внимание уделяется заявленным ограничениям и предположениям о среде выполнения, а также реализации основных сценариев применения: диагностика состояния сети, конфигурация устройств, передача процессных данных в реальном времени (реалтайм), мониторинг трафика.

- **Поддерживаемые механизмы CANopen.** Проверяется, какие функциональные возможности стандарта CANopen реализованы в каждом стеке. В частности, рассматривается поддержка основных типов сообщений (NMT, PDO/SDO, SYNC, …). Критерий позволяет оценить полноту реализации протокола.

- **Архитектура и программная реализация.** Данный критерий включает анализ внутренних архитектурных решений и технических особенностей реализации. Архитектурный анализ позволяет понять, насколько каждый проект приспособлен к целевому применению.

- **Интеграция с транспортом.** Критерий указывает на реализованные способы применения. Изучается поддержка SocketCAN, embedded-hal, конкретных драйверов.

- **Зрелость и стабильность проекта.** По этому критерию оценивается стадия развития каждого стека, степень его испытанности и го-

товности к промышленному использованию. Рассматриваются дата последнего коммита/релиза, наличие CI, тестов, примеров, полнота README и документации.

Перечисленные критерии были выбраны таким образом, чтобы охватить как соответствие функциональным требованиям протокола, так и качества, значимые с точки зрения программной инженерии и практического использования.

1.4 Актуальность и значимость проведённого обзора

Поскольку в экосистеме Rust отсутствует единый общепризнанный стандарт де-факто для CANopen, систематический обзор текущих проектов является своевременным и востребованным.

Следует подчеркнуть, что сам по себе обзор существующих CANopen-стеков на Rust обладает научно-практической ценностью, учитывая новизну темы. На момент написания работы подобных обзоров в открытом доступе немного. Таким образом, результаты проведённого анализа будут полезны не только в рамках данного дипломного проекта, но и более широкому кругу специалистов, интересующихся использованием Rust в системах промышленной автоматики. Анализ, основанный на чётких критериях, обеспечивает объективную картину состояния дел и тем самым служит прочной основой для дальнейших исследований и разработок в области CANopen на языке Rust.

2 Zencan

2.1 Назначение и область применения

Zencan — это открытый проект на Rust, реализующий стек CANopen, то есть набор компонентов для создания и управления узлами CANopen в Rust-среде [4] [3]. Он ориентирован на встроенные системы с минимальными требованиями к окружению (`no_std`, без динамической памяти) и предлагает средства генерации кода, утилиты и клиентские библиотеки для взаимодействия с узлами.

Если классифицировать функциональность Zencan по основным прикладным сценариям [3], можно выделить следующие направления:

- **Диагностика.** Хотя главной задачей фреймворка не является диагностика, в нём реализованы некоторые соответствующие механизмы. Фреймворк позволяет сканировать все устройства на шине и считывать их метаданные, а также способен анализировать сигналы HEARTBEAT для первичной оценки состояния сети в реальном времени. Отмечено также несколько нереализованных возможностей, которые планируются к внедрению [3].

- **Конфигурация.** Это основная задача Zencan, поскольку изначально фреймворк задумывался для чтения конфигурации из TOML-файла и генерации кода для узлов.

- **Реалтайм.** Фреймворк способен работать с PDO и решать задачи в реальном времени, однако не гарантирует жёсткого соблюдения временных ограничений; впрочем, этого сложно добиться, поскольку во многом точность зависит от других факторов (например, драйвера CAN и нагрузки на шину).

- **Мониторинг.** Полноценная функциональность мониторинга не реализована. Zencan имеет собственную утилиту CLI, которая предоставляет вышеописанный функционал [5], в том числе то, что наиболее интересно в нашем контексте: отслеживание HEARTBEAT и считывание метаданных устройств на шине — этого в целом достаточно

для базового мониторинга. Однако, например, средств для логирования или построения графиков в утилите не предусмотрено.

2.2 Поддерживаемые механизмы CANopen

– **SDO.** Реализован, однако в будущем возможны изменения. Этого достаточно для сценария базового мониторинга, но необходимо быть осторожным.

– **PDO.** Реализован.

– **NMT.** Поддерживается на уровне, достаточном для управления жизненным циклом узла.

– **SYNC.** Реализован частично.

– **TIME.** Не заявлен.

– **EMCY.** Не заявлен.

– **HEARTBEAT.** Реализован полностью.

2.3 Архитектура

Ключевая архитектурная особенность Zencan состоит в том, что первичным артефактом конфигурации выступает пользовательский TOML-файл; на его основе в процессе сборки генерируются статические структуры данных узла, включая представление OD и служебные структуры. Такой подход выбран не случайно — автор отмечает, что его не устраивал стандарт EDS по двум основным причинам:

1. EDS показался автору избыточным и трудным для ручного редактирования.
2. Для Zencan потребовалось добавить «служебные» настройки, и это было сделано напрямую в конфигурационном файле.

Далее Zencan предоставляет пользователю конструктор `Node` — тип данных, инкапсулирующий в себе всё поведение узла, подключённого к шине CAN. Это ядро, реализующее поведение CANopen-устройства на шине: оно

знает, как принимать и отвечать на типовые CANopen-сообщения, как читать и писать значения OD, как отсылать PDO, как функционировать в состояниях NMT и т.д. При этом, согласно философии решения, оно не привязано к конкретному оборудованию и не навязывает конкретного runtime: разработчик сам решает, где и как запускать это протокольное ядро — в суперцикле, в задаче RTOS или в асинхронной задаче. Главное, что ядро сделано так, чтобы его было легко встроить: приложение само принимает CAN-кадры (в любом удобном контексте) и периодически предоставляет протокольному ядро возможность обработать накопившиеся данные и при необходимости отправить ответы.

Хотя экземпляр Node должен создавать пользователь, фреймворк самостоятельно генерирует все необходимые служебные данные [6], а именно:

– «почтовый ящик» для приёма входящих сообщений;

– служебное состояние узла;

– OD.

Единственное, что пользователь обязан указать при создании конкретного Node, — это ID узла. Далее пользователь задаёт прикладное поведение узла, а именно:

1. Приём CAN-кадров: необходимо определить, как преобразовать их в структуру NODE_MBOX.

2. Периодический вызов обработки: нужно вызывать метод обработки с определённой частотой.

3. Прикладную задачу: собственно, логику работы узла.

Для удобства эксплуатации устройства к стеку добавлен клиентский слой для Linux. Он работает через SocketCAN и предоставляет утилиты для типовых задач: найти устройство, назначить ему Node-ID (через LSS), загрузить конфигурацию OD, запустить NMT, контролировать HEARTBEAT/телеметрию.

2.4 Интеграция с транспортом

В Zencan можно выделить три способа подключения:

1. **Linux через SocketCAN/tokio.** При запуске клиентского приложения или утилиты на ПК (или на встроенной системе с Linux) шина доступна как интерфейс `can0`/`vcan0`. В этом случае Zencan предоставляет готовый адаптер, который открывает интерфейс SocketCAN и возвращает объекты `sender/receiver` для работы с CAN-кадрами. Этот вариант используется в первую очередь в проектах zencan-client и zencan-cli.

2. **Embedded (MCU, без привязки к конкретному драйверу).** Используется при разработке прошивки CANopen-узла на микроконтроллере. В этом случае разработчику необходимо реализовать приём/передачу CAN-кадров и связать их с API узла.

3. **Виртуальный транспорт.** Используется для запуска программ без реального железа. Благодаря тому, что Zencan опирается на простые абстракции приёма/передачи CAN-кадров (трейты для sync/async), пользователь может написать адаптер, реализующий эти интерфейсы и перенаправляющий кадры куда угодно: например, в структуру `VecDeque`, через каналы `mpsc`, в лог-файл (с последующим воспроизведением), либо между двумя узлами в памяти.

2.5 Зрелость и поддерживаемость

На момент подготовки текста репозиторий Zencan [4] демонстрировал умеренную активность: 27 звёзд, 8 форков, около 180 коммитов, 12 открытых issue и 1 открытый PR по состоянию на 26.12.2025. Проект является «живым», но пока не представляет собой крупную экосистему с широким внешним вкладом.

Опубликованные пакеты имеют версии 0.0.x, что означает нестабилизированный API и возможные частые несовместимые изменения. Например, zencan-client версии 0.0.1 датирован 29 октября 2025 года.

Автор отмечает, что проект находится на стадии прототипа и возможны значительные изменения API [4].

У проекта имеется отличная документация, покрывающая большую часть функциональности.

2.6 Вывод

Zencan представляет собой ранний, но концептуально цельный CANopen-стек на Rust, ориентированный на сферу встроенных устройств (`no_std`, отсутствие динамического выделения памяти, статическая модель данных, явный контроль транспорта и планировщика со стороны приложения [6]). Сильная сторона Zencan — путь «конфигурация (TOML) $\rightarrow$ генерация OD/типов $\rightarrow$ узел как вызываемое протокольное ядро»: это снижает порог входа для создания собственных CANopen-устройств и делает интеграцию предсказуемой [3].

С практической точки зрения проект уже охватывает базовый контур управления и конфигурирования: LSS/NMT, SDO-сервер, PDO mapping явно заявлены, а работоспособные сценарии (LSS FastScan, назначение Node-ID, загрузка конфигурации, запуск NMT, мониторинг состояния «last seen») продемонстрированы на примере CLI [7]. Сам проект маркируется как прототип с недостающей функциональностью и ожидаемой нестабильностью API [4]. Кроме того, полнота охвата «всего CANopen» по состоянию на версию 0.0.1 подтверждена лишь частично (в частности, детали режимов SDO и наличие ряда сервисов уровня EMCY/TIME/SYNC не отражены явно в публичной документации) [6].

Следовательно, Zencan целесообразно использовать как объект исследования и инженерную базу для прототипов и экспериментов в экосистеме Rust (особенно в случаях, когда важны `no_std` и предсказуемость использования ресурсов). Однако в роли «готового промышленного CANopen-стека» Zencan нуждается в дополнительной верификации по ключевым аспектам стандарта и оценке рисков, связанных с ранней стадией развития и возможными изменениями API [4].

3 OZE-CanOpen

3.1 Назначение и область применения

OZE-CanOpen — это открытый проект (компания Ozon Tech) на Rust, реализующий базовый стек протоколов CANopen, предназначенный в первую очередь для пассивного мониторинга шины, а также для работы в роли мастер-узла CANopen в среде ПК [8]. В отличие от полноценных реализаций узлов, OZE-CanOpen не содержит собственной OD и сосредоточен на обмене сообщениями и их разборе. Проект на данный момент ориентирован на запуск в среде Linux (через SocketCAN) с использованием экосистемы `std`/Tokio, хотя в дорожной карте заявлена поддержка `no_std` для встроенных систем (MCU) [8]. Библиотека также предоставляет привязки для Python, что упрощает интеграцию в скрипты и приложения на Python.

В своей нише OZE-CanOpen выделяется фокусом на инструменты отладки CANopen-сетей. Открыто заявлено, что проект находится в стадии разработки и не является завершённым решением.

Если рассмотреть возможности OZE-CanOpen с точки зрения прикладных сценариев, можно выделить следующие направления:

– **Диагностика.** Хотя основной задачей OZE-CanOpen не является автоматизированная диагностика, библиотека предоставляет средства для просмотра и анализа трафика на CAN-шине. Библиотека умеет парсить все типы CANopen-сообщений по их COB-ID (включая SDO, PDO, NMT, HEARTBEAT и др.), идентифицируя тип сообщения и Node-ID. Это позволяет, к примеру, отслеживать HEARTBEAT от узлов и оперативно видеть их смену состояний. С помощью утилит на базе OZE-CanOpen можно фильтровать сообщения по типам, ID узлов и содержимому, отображать только последние значения сигналов и интервалы между ними, что удобно для наблюдения за PDO и другими периодическими данными. Тем не менее специфические механизмы активного сканирования устройств на шине (например, поиск всех Node-ID или опрос объекта Identity) в библиотеке не реализованы.

– **Конфигурация.** Работа с конфигурацией узлов рассматривается как одна из предполагаемых областей применения OZE-CanOpen, хотя выполняется она не автоматически, а вручную через SDO. Библиотека предоставляет SDO-клиент для каждого узла, что позволяет программно читать и записывать параметры OD удалённых устройств. По сути, зная нужные индексы и подындексы, пользователь может отправлять SDO-запросы на чтение/запись конфигурационных параметров. Однако отсутствуют высокоуровневые средства для загрузки/выгрузки конфигураций из EDS/DCF — все операции конфигурирования выполняются «вручную» или посредством пользовательского кода.

– **Реалтайм.** OZE-CanOpen способен обрабатывать PDO-трафик в реальном времени и поддерживает синхронизацию по SYNC. Благодаря асинхронной многопоточной архитектуре на базе Tokio, при работе на ПК библиотека может параллельно принимать и передавать сообщения, что помогает не терять кадры даже при высокой нагрузке шины. Также реализованы серверные роли для NMT и SYNC, позволяющие, например, периодически рассылать SYNC или команды управления состоянием узлов. Однако жёстких гарантий по временным задержкам и детерминизму OZE-CanOpen не даёт — точность соблюдения временных требований зависит от драйвера CAN, нагрузки системы и т.п.

– **Мониторинг.** Функциональность мониторинга в OZE-CanOpen реализована преимущественно через внешние утилиты. В экосистему проекта входит графическая утилита oze-canopen-viewer [9], использующая библиотеку: она позволяет в реальном времени наблюдать за шиной CANopen, отображая типы сообщений, значения данных и загрузку шины. Через Viewer доступен базовый мониторинг состояния узлов (например, по HEARTBEAT) и обмена данными, достаточный для отладки и наблюдения. В самой библиотеке нет встроенного логирования на диск или построения графиков, но пользователь при необходимости может реализовать эти функции самостоятельно либо вос-

пользоваться возможностями Wireshark/SocketCAN для сохранения дампов. В целом, OZE-CanOpen предоставляет необходимый минимум для ручного мониторинга: захват пакетов, разбор их по протоколу CANopen и фильтрацию/выборку интересующих сообщений.

3.2 Поддерживаемые механизмы CANopen

– **SDO.** Поддерживается на уровне SDO-клиента. OZE-CanOpen позволяет инициировать SDO Upload/Download-запросы к узлам и обрабатывать ответы. Реализация рассчитана на один одновременный запрос на узел (дополнительные запросы блокируются с помощью асинхронного мьютекса). SDO-сервер (обработка входящих запросов к OD) отсутствует ввиду отсутствия собственной OD. В будущих версиях возможны изменения API по SDO, учитывая раннюю стадию проекта.

– **PDO.** Реализована обработка PDO-сообщений. Библиотека различает все стандартные типы PDO (1–4, в направлениях Tx/Rx) по COB-ID и может принимать/отправлять их как обычные CAN-фреймы. Специфической логики для разбора содержимого PDO нет (OD не хранится), но совместно с описанием PDO (например, по EDS вне библиотеки) разработчик может интерпретировать данные. Отправку PDO от имени узла библиотека напрямую не выполняет (так как не моделирует узел-источник), однако мастер-устройство может с её помощью рассылать PDO при необходимости, формируя нужные кадры.

– **NMT.** Поддерживается на уровне мастера. OZE-CanOpen может рассылать команды NMT (Start, Stop, Reset) по шине — это можно сделать, отправив сырой CAN-кадр. Также библиотека отслеживает HEARTBEA
от узлов (непосредственно распознаёт их как особый тип сообщения) и определяет текущее состояние NMT-узлов. Жизненный цикл узла (изменение его состояния) не эмулируется — библиотека выступает только инициатором NMT-команд и наблюдателем состояний.

– **SYNC.** Реализован. OZE-CanOpen включает механизм SYNC-производителя
предусмотрена возможность периодически посылать SYNC-кадры с

заданным интервалом. Приём и распознавание входящего SYNC (COB-ID 0x80) также поддерживается парсером, хотя специальной обработки (например, уведомления приложению о наступлении цикла) библиотека не предоставляет — это остаётся на усмотрение пользователя. В целом базовая функциональность SYNC (отправка и идентификация) присутствует, расширенные возможности (например, управление временем рассылки в привязке к задачам) не документированы.

– **TIME.** Не заявлен в документации и при просмотре кода не обнаружены явные обработчики.

– **EMCY.** Отдельно не заявлен. Библиотека может распознать аварийные сообщения (COB-ID 0x80 + Node-ID) лишь как неизвестный тип, поскольку явной обработки EMCY в текущем коде нет. Соответственно, прямой поддержки сервиса аварийных сообщений нет — они проходят через OZE-CanOpen как обычные CAN-фреймы, которые пользователь может перехватить и обработать вручную.

– **HEARTBEAT.** Поддерживается полностью. OZE-CanOpen распознаёт HEARTBEAT-сообщения (COB-ID 0x700 + Node-ID) и определяет состояние узла по содержимому байта состояния. Эти данные могут использоваться, например, для отслеживания «живости» узлов и времени с последнего отклика. Кроме того, в утилите Viewer реализовано отображение состояния узлов и времени получения последнего HEARTBEAT.

3.3 Архитектура

Одной из ключевых архитектурных черт OZE-CanOpen является отказ от хранения OD внутри библиотеки. В качестве «источника истины» о данных устройств выступают внешние описания (EDS/DCF) или знания пользователя. Этот подход выбран для упрощения дизайна: OZE-CanOpen не навязывает структуру OD, а сосредоточен на пересылке пакетов. В результате библиотека получилась легковесной, но ответственность за интерпретацию значений PDO/SDO лежит на приложении или внешних инструментах.

Внутри OZE-CanOpen реализована асинхронная многопоточная модель на базе Tokio. При инициализации создаётся объект интерфейса, который запускает фоновые задачи: асинхронные задачи (tasks) для приёма и передачи кадров (параллельные приёмник/передатчик), а также монитор интерфейса. Это обеспечивает неблокирующую обработку — приём сообщений с шины не задерживается их отправкой, и наоборот. Пользователю предоставляется высокоуровневый API: например, функция `canopen::start(\enquote{vcan0}, So` открывает канал, запускает необходимые задачи и возвращает объект Interface и набор управляющих обработчиков. Через объект Interface можно получить, например, SDO-клиент для конкретного узла (`interface.get_sdo_client(node_` и затем выполнять с ним операции (чтение/запись) буквально в несколько строк кода. Библиотека сама синхронизирует доступ к SDO-клиентам (используется `Mutex<...>` для последовательного выполнения запросов).

Важно, что OZE-CanOpen не зависит от конкретного аппаратного CAN-контроллера: взаимодействие с шиной абстрагировано через стандартный SocketCAN (на ПК). В перспективе (после добавления поддержки `no_std`) планируется поддержка пользовательских драйверов CAN на микроконтроллерах [8] — архитектура изначально заложена модульной. Уже сейчас доступны привязки к Python. Таким образом, архитектура OZE-CanOpen нацелена на гибкость: ядро — это протокольное ядро обработки CANopen-протокола, который можно вызывать из разных сред (Rust-приложение, Python-скрипт, GUI и т.д.).

Отдельно стоит отметить, что на базе библиотеки создано приложение oze-canopen-viewer. Его наличие демонстрирует правильность архитектурного подхода: благодаря выделению ядра поверх него удалось довольно быстро написать GUI-программу для визуализации CANopen-трафика. Viewer использует библиотечные возможности парсинга и фильтрации, дополняя их графическим интерфейсом и такими компонентами, как график загрузки шины, удобные фильтры и т.п. Это подтверждает расширяемость архитектуры OZE-CanOpen для прикладных задач.

3.4 Интеграция с транспортом

В текущей версии OZE-CanOpen поддерживает два способа подключения к CAN-шине и ещё один планируется:

1. **Linux (SocketCAN).** Основной вариант — запуск на ПК под управлением Linux, где CAN-интерфейс представлен как сетевое устройство (например, `can0` или виртуальный `vcan0`). Библиотека напрямую работает с драйвером `socketcan` на Rust, что позволяет слушать и отправлять сообщения на указанном интерфейсе.

2. **Виртуальная шина.** Для целей тестирования и отладки OZE-CanOpen можно использовать виртуальную CAN-шину. Например, разработчики сами используют `vcan0` в тестах. В Viewer также реализована работа с виртуальным интерфейсом для отладки без реального оборудования.

3. **Embedded (no_std) — планируется.**

3.5 Зрелость и поддерживаемость

По состоянию на конец 2025 года проект OZE-CanOpen находится на ранней стадии развития. Репозиторий относительно мал: порядка 7 коммитов, 12 звёзд, 3 форка; нет открытых issue, имеется один Pull Request по состоянию на 26.12.2025. Активность разработки невысока — последний значимый коммит датирован мартом 2025 года. Пакет опубликован на Crates.io в версии 0.1.0 (16 марта 2025), то есть API ещё не стабилизирован и может существенно меняться в последующих выпусках. Авторы прямо помечают проект как «прототип» и ожидают возможного изменения API.

Тем не менее OZE-CanOpen — проект под эгидой довольно крупной компании, что даёт шанс на дальнейшую поддержку. У проекта уже есть сопутствующая утилита (Viewer) и небольшое сообщество пользователей, заинтересованных в CANopen на Rust. Документация к библиотеке представлена в основном в README и примерах кода; дополнительная информация содержится в исходниках и комментариях.

Внешних контрибьюторов пока мало, основной вклад — от команды Ozon Tech.

3.6 Вывод

OZE-CanOpen — ранний, но перспективный CANopen-стек для Rust, нацеленный скорее на задачи интеграции и отладки, чем на реализацию полноценных узлов устройств. Его сильная сторона — упрощённая архитектура без лишних надстроек: разработчик получает прямой контроль над обменом CANopen-сообщениями, может относительно легко встроить библиотеку в свои инструменты (CLI, GUI, скрипты) и использовать асинхронные возможности Tokio для одновременного обслуживания нескольких узлов. OZE-CanOpen особенно полезен там, где нужен «сниффер» CANopen или легковесный мастер-ориентированный стек — например, для написания утилит настройки оборудования, тестирования устройств, мониторинга сетей. Концепция «библиотека + Viewer» снижает порог входа для исследования шины CANopen: можно сразу видеть результат работы кода.

В то же время для промышленного применения OZE-CanOpen пока не дотягивает: отсутствует собственная OD (а значит, из коробки нельзя реализовать полноценное устройство-ведомый узел), не реализованы некоторые сервисы стандарта (EMCY, TIME, LSS) и не гарантируется стабильность API. Полнота соответствия спецификации CANopen требует проверки — например, обработка сегментированных SDO, корректность таймаутов, обработка ошибочных ситуаций покрыты библиотекой лишь на базовом уровне. Таким образом, сейчас OZE-CanOpen целесообразно рассматривать как инструмент для специалистов и энтузиастов — для лабораторных и прототипных задач, а также как объект изучения опыта реализации CANopen на современном языке (Rust). При планировании использования в конечных продуктах следует учитывать риски ранней стадии: возможные изменения API, ограниченную функциональность и зависимость от дальнейшей поддержки разработчиков. В случае активного развития и выхода проекта на стабильную версию он сможет занять свою нишу как лёгкий CANopen-стек для Rust, дополняющий существующие решения на C.

4 Ican

4.1 Назначение и область применения

Ican — это набор утилит и библиотек на Rust для работы с CAN-шиной, включающий возможности по поддержке протокола CANopen [10]. Проект позиционируется как «современные инструменты CAN» для инженеров-разработчик Основная форма распространения — консольное приложение `ican`, предоставляющее функциональность, аналогичную утилитам из пакета can-utils (*candump*, *cansniffer*, *cansend*), но расширенную с учётом специфики CANopen. Так, Ican может не только отображать и отправлять CAN-кадры, но и декодировать CANopen-фреймы при наличии описания узла (EDS-файла).

Проект ориентирован прежде всего на среду Linux: по умолчанию используется драйвер SocketCAN (можно явно указывать `driver://...`, но для SocketCAN достаточно имени интерфейса). Таким образом, Ican удобно применять на ПК для отладки и тестирования сетей CAN/CANopen — например, подключив компьютер через CAN-адаптер к промышленной сети или используя виртуальную шину. В архитектуру также заложена поддержка других источников CAN-данных (альтернативные драйверы, файлы логов и т.д.) за счёт абстрактного указания драйвера в URI-формате, однако на практике основным и единственным реализованным драйвером является SocketCAN.

В своей нише Ican — достаточно уникальный инструмент. Если OZE-CanOpen и аналогичные проекты представляют собой скорее библиотеки-стеки для встраивания в прошивки, то Ican — это именно пользовательское приложение для интерактивной работы. Автор в своём блоге отмечает, что существующие средства его не устраивали: например, *cansniffer* не умеет работать с расширенными кадрами, не отображает напрямую значения объектов CANopen и требует держать в уме расшифровку PDO/SDO [11]. Ican призван устранить эти недостатки, предоставив инженеру «всё в одном»: и sniffer, и отправитель кадров, и декодер CANopen. Проект начат в 2022 году и, вероятно, уже не развивается, поэтому не предназначен для промышленного применения без доработок.

Если распределить возможности Ican по прикладным сценариям, можно

выделить следующие направления:

- **Диагностика.** Ican изначально разработан для облегчения диагностики сетей CAN/CANopen. Утилита позволяет выводить в терминал все принимаемые кадры (`dump`-режим), а также запускать мониторинг в стиле *cansniffer* (`monitor`-режим), при котором на экране отображаются только изменяющиеся сигналы, обновляясь в реальном времени. Главное преимущество — понимание протокола CANopen: Ican распознаёт типы сообщений (PDO, SDO, HEARTBEAT и т.п.) и, зная структуру OD устройства (из EDS-файла), способен декодировать полезные данные прямо в человекочитаемом формате. Например, вместо сырого 8-байтового HEX видно, что пришёл PDO с определёнными параметрами (температура, скорость и т.д.). Это значительно упрощает диагностику устройств: фактически Ican выполняет роль «онлайн-декодера» CANopen, подобно тому как Wireshark декодирует протоколы верхнего уровня. Кроме того, Ican отслеживает HEARTBEAT-сообщения (определяя состояние узлов) и может помочь выявить, какие узлы активны и в каком состоянии. Автоматического сканирования узлов (например, LSS FastScan) в Ican нет, но имея EDS устройства, пользователь получает достаточную информацию для диагностики его поведения на шине.

- **Конфигурация.** В текущей функциональности Ican не предоставляет высокоуровневых средств конфигурирования CANopen-устройств. Отсутствуют команды типа «прочитать весь OD» или «загрузить конфигурацию из файла».

- **Реалтайм.** Ican — утилита пользовательского пространства — не предназначен для жёсткого управления в реальном времени, однако обладает возможностями для работы с периодическими процессами. Во-первых, благодаря асинхронной архитектуре Ican эффективно обрабатывает поступающий поток кадров, практически в реальном времени отображая изменения. Во-вторых, утилита поддерживает генерацию периодических сообщений: опцией `-r <freq>` можно задавать частоту отправки кадра, что позволяет, например, имитировать дат-

чик, посылающий PDO с заданной периодичностью, или генерировать SYNC-кадры через равные интервалы. Это важно для тестирования поведения устройств в динамических условиях.

– **Мониторинг.** Средства мониторинга шины в Ican можно считать хорошо развитыми для консольного инструмента. Мониторинг в данном контексте подразумевает непрерывное наблюдение за состоянием узлов и обменом данными. `monitor`-режим утилиты выводит на экран обновляемый перечень сообщений (обычно сгруппированных по COB-ID), показывая последние значения и отмечая изменения — аналогично утилите *cansniffer*. При наличии EDS Ican может сразу отображать значения отдельных объектов, входящих в PDO, обновляя их в реальном времени. Также отображаются полученные SDO-ответы, что позволяет отслеживать процесс последовательной передачи данных (например, загрузку конфигурации). Графического или веб-интерфейса у Ican нет — мониторинг осуществляется в текстовом консольном окне. Логирование сообщений автоматически не выполняется, но при необходимости пользователь может перенаправить вывод в файл. Таким образом, Ican обеспечивает интерактивный мониторинг, удобный для разработчика: все ключевые события на шине видны сразу, с возможностью интерпретации, но без излишеств (никаких встроенных графиков, GUI и т.п.).

4.2 Поддерживаемые механизмы CANopen

– **SDO.** Поддерживается частично — в основном на уровне клиента и декодера. В Ican реализован разбор SDO-сообщений (исходящих запросов и входящих ответов): утилита распознаёт команды SDO Download/Up
по COB-ID (0x600/0x580) и отображает направление передачи и сырые данные. Более того, при использовании EDS утилита способна интерпретировать индекс и подындекс запрашиваемого объекта, теоретически даже выводить название параметра. Однако полноценный SDO-стек (например, сегментация блочных передач, хранение OD, обработка входящих запросов) в Ican отсутствует — утилита не вы-

ступает SDO-сервером. Основная возможность для пользователя — отправлять SDO-запросы вручную и видеть результаты. Этого достаточно для минимально необходимых операций: например, чтения отдельных параметров или их записи для настройки узла. Механизмы вроде подтверждения размеров блоков или повторной попытки при таймауте зависят от реализации SocketCAN и не контролируются утилитой вручную. В целом SDO-компонент в Ican присутствует и является одним из ключевых элементов, но реализован лишь в объёме, необходимом для мониторинга и простых операций.

– **PDO.** Поддержка PDO в Ican сводится к возможности декодировать и отображать их содержимое. Утилита различает все стандартные PDO (1-4, Tx и Rx) по идентификатору. В связке с EDS она умеет получить карту PDO — т.е. узнать, какие объекты и с каким размером входят в конкретный PDO [11], – и на основе этой информации расшифровать байты PDO в набор значений переменных. Например, TPDO1 устройства может содержать два 16-битных параметра; Ican, зная это из EDS, при получении такого PDO разделит 8 байт на две части, преобразует их в целые числа и покажет каждый параметр отдельно. Это значительно облегчает анализ обмена процессными данными. Отправлять PDO (эмулируя поведение устройства) Ican напрямую не умеет, хотя пользователь всегда может воспользоваться командой `send` для посылки любого кадра с произвольным CAN ID. В утилите нет логики формирования PDO-пакетов на основе локальных данных, так как она не хранит OD.

– **NMT.** Поддерживается в пассивном режиме. Ican отслеживает HEARTBEAT (COB-ID 0x700) и определяет состояние узла (например, Operational или Pre-Op) — это видно при декодировании HEARTBEAT, где выводится текущее состояние NMT-узла. Таким образом, утилита способна мониторить сетевое состояние всех узлов (при условии настройки HEARTBEAT-сообщений). Что касается активного управления NMT: специализированной команды для отправки NMT не предусмотрено. Ican никак не интерпретирует команду NMT на своём уровне (не под-

тверждает её выполнение, кроме как по последующему изменению HEARTBEAT). В целом минимально необходимая поддержка NMT имеется — можно увидеть состояния узлов и при необходимости отправить команды, но она не обёрнута в удобный интерфейс и требует от пользователя соответствующих знаний.

– **SYNC.** Реализована базовая поддержка. Ican распознаёт SYNC-кадры (0x080) как отдельный тип CANopen-фрейма. При мониторинге приход SYNC может отображаться, хотя сам по себе много информации (кроме факта синхронизации) не несёт. Важнее другое: утилита умеет отправлять SYNC-кадры периодически по расписанию. Таким образом, Ican может выступать в роли SYNC-производителя на время отладки, синхронизируя работу узлов. Глубокой интеграции с приложениями (например, вызова колбэка при получении SYNC) не реализовано — Ican остаётся внешним инструментом, а не частью прошивки. Однако для проверки реакций устройств на SYNC-сигналы или организации синхронного сбора данных во время тестирования эта возможность ценна.

– **TIME.** Не поддерживается.

– **EMCY.** Явной поддержки аварийных сообщений не видно. Ican распознаёт кадры EMCY лишь как обычные CAN-фреймы, поскольку их COB-ID (0x080 + NodeID) не перечислен в парсере. Например, сообщение с идентификатором 0x081 Ican ошибочно трактует как Sync с NodeID = 1 (есть такой нюанс в текущей реализации парсера). Это можно считать недоработкой. Таким образом, расшифровку кода и регистра ошибки Ican не выполняет. Тем не менее само наличие сообщений EMCY в дампе утилита не пропустит — пользователь их увидит (пусть и как необработанные кадры) и при необходимости сможет интерпретировать вручную.

– **HEARTBEAT.** Полностью поддерживается (в части мониторинга). Ican декодирует HEARTBEAT-пакеты, выделяя из них Node-ID и состояние NMT-узла. При непрерывном мониторинге можно видеть, что

узел, например, отправляет HEARTBEAT каждые N мс и находится в состоянии Operational. Если узел пропадает (HEARTBEAT перестаёт поступать), это также будет очевидно из вывода (показатель перестанет обновляться). Таким образом, утилита выполняет функцию базового менеджера живучести узлов.

4.3 Архитектура

Архитектурно Ican сочетает в себе черты системной утилиты и библиотечных компонентов. Сердцем проекта является протокольное ядро обработки CAN/CANopen, написанное на Rust с использованием асинхронных возможностей. Применяются Tokio и адаптированная библиотека `tokio-socketcan` для неблокирующего приёма/отправки CAN-фреймов. Особенность реализации — автор переработал `socketcan` для совместимости с трейтами `embedded-hal` для CAN. Это означает, что логика чтения/записи кадров в Ican написана абстрактно: она могла бы работать и с другим источником, реализующим стандартный трейт (например, с аппаратным CAN-контроллером на MCU через HAL-драйвер). Фактически, в кодовой базе Ican CAN-интерфейс является опцией: драйвер выбирается строкой подключения (URI). Например, указание `socketcan://vcan0` подключает модуль для работы с SocketCAN. Такая архитектура делает проект потенциально переносимым и расширяемым — можно реализовать поддержку, скажем, CANAL (CAN over serial) или PCAN, добавив соответствующий модуль, не меняя остальной код.

Важным компонентом архитектуры является парсер CANopen. В Ican введена структура (enum) `CanOpenFrame`, описывающая высокоуровневое представление CANopen-сообщения: Sync, HEARTBEAT (с состоянием NMT), Pdo (с типом PDO и данными), Sdo (с типом SDO и данными) и т.д. Реализована функция `parse(frame)`, которая из сырого CAN-фрейма получает пару (NodeID, CanOpenFrame) либо возвращает ошибку, если идентификатор не относится к CANopen. Этот парсер инкапсулирует знания о разметке адресного пространства CANopen и позволяет остальному коду оперировать понятиями уровня CANopen, а не голыми идентификаторами.

Отдельно реализован модуль для парсинга EDS. Автор вынес его в отдельный пакет: этот модуль читает EDS-файл (формат INI) и формирует струк-

туру данных, содержащую описание OD устройства — список всех объектов, их типы, допустимые диапазоны, карты PDO и пр. В архитектуре Ican структура EDS используется для двух задач:

– Получение карт PDO (объекты 0x1600/0x1A00) и подготовка декодеров PDO. Специальный класс `PdoDecoder` в Ican на основе карты знает, как разложить байтовый массив PDO на отдельные значения.

– Потенциально — для отображения содержимого OD по запросам SDO, т.е. чтобы выводить не просто «SDO response for index 0xABCD», а указывать название параметра (эта часть описана не подробно).

Упомянутые компоненты объединены в консольном приложении. В функции `main` реализованы разбор аргументов (интерфейс, команда — dump/send/monitor, запуск асинхронных задач по приёму кадров и, при необходимости, их периодической отправке, а также логика отображения данных. Для режима `monitor`, по всей видимости, используется обновление консоли. Хотя конкретная реализация интерфейса пользователя в коде GitHub не описана подробно, можно предположить, что монитор строится в виде таблицы: в строках — COB-ID (или имя объекта), в столбцах — текущее значение и, возможно, время последнего изменения. Такой дизайн часто применяется в утилитах для CAN (например, *cansniffer*).

Архитектура Ican не предусматривает какого-либо внешнего API (по крайней мере, пока). В отличие от OZE-CanOpen, который, будучи библиотекой, может быть включён в стороннюю программу, Ican задуман как самостоятельное приложение. Однако части его кода (парсер, EDS-читатель) потенциально могут быть использованы и отдельно, если выделить их в отдельные библиотеки. На текущий момент проект распространяется исходниками на GitHub и устанавливается из них (`cargo install --path .`), что говорит о том, что Ican ещё не оформлен как отдельный пакет на Crates.io, а его API крайне нестабильно.

4.4 Интеграция с транспортом

Возможности интеграции Ican с различными транспортными средами формально заложены, но практически ограничены. Как уже отмечалось, ос-

новной транспорт — SocketCAN. При запуске утилиты достаточно указать интерфейс (например, `vcan0` или `can0`) и команду, и Ican подключится к соответствующему сокету ОС для обмена кадрами. Внутри используется `socketcan-h` модифицированная библиотека, которая адаптирует вызовы CAN к трейту `embedded-hal`. Это сделано для того, чтобы теоретически можно было заменить источник данных.

Ican поддерживает указание альтернативного драйвера через синтаксис `driver://opts`. Например, можно было бы написать `ican pcan://...` для работы с аппаратным адаптером PCAN без SocketCAN (в настоящее время такой драйвер не реализован — это концепция на будущее). Также можно представить драйвер, читающий данные из лог-файла (для проигрывания записанного трафика) — архитектура позволяет это сделать, реализовав трейт `CANInterface` для нужного источника. Однако «из коробки» подобных драйверов нет.

Для встроенных систем Ican напрямую не предназначен. Несмотря на использование `embedded-hal`, утилита зависит от Tokio и `std`. Тем не менее ядро (парсинг CANopen) теоретически могло бы работать на микроконтроллере, если обеспечить ему поток CAN-кадров. Автор не заявлял планов портирования в `no_std`; скорее всего, концепция применения иная: Ican используется на уровне ПК или ноутбука, подключенного к отлаживаемой системе. Таким образом, интеграция с транспортом остаётся в рамках «plug-and-play» через стандартные CAN-интерфейсы.

Для имитации среды или тестирования без реального CAN Ican прекрасно работает с виртуальными интерфейсами (vcan). Автор активно использует `vcan0` в примерах и тестах, а также упоминает, что написал симулятор CANopen-устройств, с которым Ican может взаимодействовать [12]. То есть интеграция утилиты в тестовые стенды возможна: можно поднять несколько виртуальных узлов (с помощью дополнительного ПО) и с помощью Ican осуществлять с ними обмен, проверяя логику.

4.5 Зрелость и поддерживаемость

Проект Ican, начатый в 2022 году, пока находится ещё на стадии разработки одним основным разработчиком. На GitHub репозиторий имеет всего

8 звёзд и 0 форков, что свидетельствует о его небольшой известности. История коммитов (80 коммитов на момент обзора) показывает развитие функциональности в 2022-2023 годах (например, добавление парсинга PDO, обработки ошибок EDS и т.п.); последние коммиты датированы концом 2023 года. Открыто 10 issue, отражающих планы и проблемы (например, предложения новых возможностей, сообщения об ошибках). Ни одного релиза ещё не публиковалось — пользователи устанавливают утилиту из исходников. Это означает, что автор не считает проект завершённым или стабильным для широкого распространения.

Формальной документации нет; роль руководства выполняет README с примерами использования и записи в блоге (достаточно информативные, но не оформленные как справочник).

4.6 Вывод

Ican представляет собой экспериментальный, но очень полезный инструмент для разработчиков, работающих с CANopen. В отличие от традиционных «стеков» для встроенных устройств, Ican решает другую задачу — он помогает наблюдать за сетью CANopen и управлять ею с уровня рабочего места разработчика. Можно сказать, что Ican — это аналог диагностического сканера, сочетающего функции анализатора протокола и генератора сообщений.

Однако проект Ican находится на ранней стадии развития и, вероятно, на данный момент уже не развивается.

Практическое применение Ican на текущий момент — лабораторные и полевые испытания. Например, разработчик может использовать Ican, чтобы подключиться к прототипу устройства и в реальном времени отслеживать, какие PDO оно шлёт и что содержится внутри них (с помощью EDS, если он доступен), или чтобы быстро отправить команду на смену состояния без написания отдельной программы. Для постоянного мониторинга на объекте (в составе конечного продукта) Ican вряд ли предназначен, но как вспомогательный инструмент в арсенале инженера он весьма ценен. В сравнении с громоздкими коммерческими анализаторами (типа CANopen Magic) или комбинацией нескольких утилит Ican предлагает легковесное и скриптуемое ре-

шение с открытым исходным кодом.

Подводя итог, Ican заполняет определённую нишу: это ориентированный на CANopen отладочный инструмент, созданный разработчиком для разработчиков. В перспективе, если проект будет развиваться, он может стать основой для целого набора open-source инструментов (эмуляторов, мастер-конфигураторов и т.д.) вокруг CANopen, что принесёт пользу сообществу. Пока же использовать Ican следует с осторожностью, тщательно проверяя получаемые результаты и учитывая, что ответственность за корректность некоторых аспектов лежит на пользователе.

ЗАКЛЮЧЕНИЕ

Проведённый обзор трёх проектов (Zencan, OZE-CanOpen и Ican) позволяет сделать содержательные выводы не только о каждом из решений по отдельности, но и о текущем состоянии экосистемы CANopen на языке Rust в целом. В отличие от экосистемы на C/C++, где существует несколько зрелых и широко применяемых стеков, экосистема Rust на конец 2025 года выглядит фрагментированной: отдельные проекты закрывают разные инженерные роли (узел/мастер/инструментарий), при этом между ними отсутствует устойчивый общий «каркас» в виде де-факто стандарта реализации, тестового контура совместимости и стабильных API.

Zencan следует рассматривать как наиболее амбициозную попытку приблизиться к полноценному стеку CANopen для встроенных устройств. Принципиальная сильная сторона Zencan заключается не столько в текущей полноте реализации протокола, сколько в архитектурной ставке на предсказуемость и «встроенность»: `no_std`, отказ от динамической памяти, статическая модель данных и проектирование узла как явно вызываемого протокольного ядра, который не навязывает конкретный runtime. Такой выбор снижает интеграционные риски в типичной embedded-среде и в перспективе может обеспечить корректную эксплуатацию в системах с жёсткими ограничениями по ресурсам. Одновременно данная стратегия делает особенно важными две вещи: во-первых, строгая дисциплина совместимости со стандартом CANopen (включая редко используемые, но критичные сервисы), а во-вторых, наличие воспроизводимого контура верификации (наборов тестов, проверок сценариев, совместимости с эталонными узлами). Иначе преимущества Rust (безопасность памяти, управляемость ошибок) не конвертируются в инженерную надёжность протокольного стека как системного компонента.

OZE-CanOpen демонстрирует другой вектор развития: вместо «встроенного узла» проект фактически решает задачи мастер-логики и наблюдаемости обмена в пользовательском пространстве. Асинхронная архитектура, ориентированность на Linux/SocketCAN и наличие Viewer формируют практичный инструментальный слой, который удобно использовать для исследований, прототипирования и разработки тестовой инфраструктуры. Однако прин-

ципиальное ограничение OZE-CanOpen (отсутствие собственной модели OD и, как следствие, невозможность естественным образом выступать в роли полноценного CANopen-устройства) означает, что проект не конкурирует с embedded-стеками напрямую; он занимает иную нишу. В терминах экосистемы это важно: появление мастер- и сниффер- решений на Rust снимает часть инфраструктурной нагрузки (инструменты отладки, конфигурирование, интеграционные проверки), но само по себе не приближает Rust к наличию универсального CANopen-стека для устройств, пока не будет решён вопрос канонической, удобной и проверяемой модели OD.

Ican, в свою очередь, подчёркивает третью составляющую экосистемы, часто недооценённую в протокольных разработках: инженерный опыт при диагностике и интерпретации трафика. Ключевая идея Ican — сделать обмен CANopen «читаемым» для разработчика за счёт декодирования данных по EDS и предоставления удобных режимов мониторинга — показывает, что в реальных проектах ценность стека определяется не только поддержкой формальных сервисов CANopen, но и наличием доступного инструментария наблюдения. При этом Ican не является стеком CANopen в строгом смысле и не претендует на полноту протокольной реализации; скорее, он демонстрирует требования к инструментам сопровождения, которые должны существовать рядом со стеком.

В целом можно констатировать, что CANopen на Rust находится на стадии раннего становления: имеются перспективные архитектурные идеи (в первую очередь в embedded-направлении) и полезные инструменты для мониторинга и отладки, но отсутствует зрелая, согласованная и верифицированная реализация, сопоставимая по полноте и устойчивости с классическими решениями на C. Вместе с тем рассмотренные проекты показывают, что ключевые элементы будущей экосистемы уже проявились: Zencan задаёт направление на безопасный embedded-стек, OZE-CanOpen формирует инфраструктуру для мастер-режима и наблюдаемости, Ican фиксирует ожидания к диагностическому инструментарию.

В совокупности рассмотренные решения не могут быть использованы в работе как готовая основа, поскольку находятся на ранней стадии зрелости и не обеспечивают одновременно полноту ключевых сервисов CANopen и

предсказуемость поведения в широком спектре эксплуатационных сценариев. Кроме того, существующие инструменты и библиотеки, как правило, фиксируют пользовательское взаимодействие на уровне внутренней модели данных CANopen (OD, SDO/PDO, типы и кодировки протокола), которая удобна для унификации и транспорта, но недостаточно выразительна как прикладная абстракция. На практике разработчику и инженеру сопровождения важнее оперировать доменными понятиями конкретного оборудования (например, «скорость», «температура», «режим работы», «состояние привода»), а не протокольными индексами, подындексами и примитивными типами представления данных. Это противоречие между протокольной моделью и прикладной семантикой, а также ограниченная зрелость существующих реализаций, в совокупности приводят к однозначному выводу о необходимости разработки собственного решения в рамках дипломного проекта, ориентированного на более высокоуровневые и предметно-ориентированные интерфейсы поверх CANopen.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. *Falch M.* CAN Bus Explained - A Simple Intro / CSS Electronics. — 01/2025. — URL: https://www.csselectronics.com/pages/can-bus-simple-intro-tutorial (visited on 12/24/2025).
2. *Falch M.* CANopen Explained - A Simple Intro / CSS Electronics. — 02/2025. — URL: https://www.csselectronics.com/pages/canopen-tutorial-simple-intro (visited on 12/24/2025).
3. *McBride J.* Introducing Zencan. — 06.2025. — URL: https://jeffmcbride.net/blog/2025/06/05/introducing-zencan (дата обр. 25.12.2025).
4. *McBride J.* zencan. — 2025. — URL: https://github.com/mcbridejc/zencan (дата обр. 25.12.2025) ; GitHub repository.
5. *zencan developers*. zencan_cli Documentation. — 2025. — URL: https://docs.rs/zencan-cli/latest/zencan_cli/ (дата обр. 25.12.2025) ; API documentation on docs.rs.
6. *zencan developers*. zencan_node Documentation. — 2025. — URL: https://docs.rs/zencan-node/latest/zencan_node/index.html (дата обр. 25.12.2025) ; API documentation on docs.rs.
7. *McBride J.* can-io-firmware. — 2025. — URL: https://github.com/mcbridejc/can-io-firmware (дата обр. 25.12.2025) ; GitHub repository.
8. *Ozon Tech*. oze□canopen. — 2025. — URL: https://github.com/ozontech/oze-canopen (дата обр. 25.12.2025) ; GitHub repository.
9. *Ozon Tech*. oze□canopen□viewer. — 2025. — URL: https://github.com/ozontech/oze-canopen-viewer (дата обр. 25.12.2025) ; GitHub repository.
10. *Narain N.* ican. — 2025. — URL: https://github.com/nnarain/ican (дата обр. 25.12.2025) ; GitHub repository.
11. *Narain N.* Building CAN tools using Rust. — 08.2022. — URL: https://nnarain.github.io/2022/08/21/Building-CAN-tools-using-Rust.html (дата обр. 25.12.2025) ; Blog post.
12. *Narain N.* CANopen Device Simulator. — 05.2023. — URL: https://nnarain.github.io/2023/05/14/CANopen-Device-Simulator.html (дата обр. 25.12.2025) ; Blog post.