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 34 стр., 0 табл., 0 рис., 12 ист.

Ключевые слова: CAN, CANOPEN, RUST

Написать нормальный реферат в конце

СОДЕРЖАНИЕ

ВВЕДЕНИЕ **6**

1 Zencan [3] [4] **11**

1.1 Назначение и область применения 11

1.2 Поддерживаемые механизмы CANopen 12

1.3 Архитектура 12

1.4 Интеграция с транспортом 13

1.5 Зрелость и поддерживаемость 14

1.6 Вывод 14

2 OZE-CanOpen [8] [9] **15**

2.1 Назначение и область применения 15

2.2 Поддерживаемые механизмы CANopen 17

2.3 Архитектура 19

2.4 Интеграция с транспортом 20

2.5 Зрелость и поддерживаемость 21

2.6 Вывод 21

3 Ican [10] [11] **22**

3.1 Назначение и область применения 22

3.2 Поддерживаемые механизмы CANopen 25

3.3 Архитектура 27

3.4 Интеграция с транспортом 29

3.5 Зрелость и поддерживаемость 30

3.6 Вывод 30

ЗАКЛЮЧЕНИЕ **32**

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ **33**

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

Для введения в столько конкретную тему стоит рассказать что вообще что такое CAN.

Допустим, вы являетесь инженером-электронщиком и разрабатываете различные электронные механизмы. У этих механизмов вполне могут быть разнесены некоторые элементы, например какой-то датчик находится в одном месте, а блок обработки сигналов - в другом. В таком случае самым тривиальным решением будет взять и соединить их проводами! Однако такой подход не всегда является оптимальным и рано или поздно вы столкнётесь с проблемой вездесущности этих самых проводов и кабелей. Огромные траты материалов на проводку - не самая большая проблема, намного хуже, на мой взгляд - обслуживать потом такую систему, разобраться среди десятков и сотен различных проводов крайне сложно. Немного подумав, вы решаете объединить какие-то провода в жгуты, а следующим логическим шагом является переход от соединений ”точка-точка”к шинной архитектуре, где по общей линии передаются сигналы между различными устройствами.

Но теперь вы сталкиваетесь с другой проблемой - как научить устройство принимать только те сигналы которые назначались конкретно ему ? Можно ввести какое-нибудь мультиплексирование по времени, но как быть с системами реального времени в которых дорога каждая секунда или крайне высока цена ошибки ? Одним из способов заставить десятки электронных блоков в машине или промышленной установке обмениваться данными по одной общей шине так, чтобы это было надёжно, предсказуемо по времени и устойчиво к помехам - является введение шины CAN.

Шина CAN (Controller Area Network) - это система связи, используемая в транспортных средствах/машинах для позволяют электронным блокам управления (ЭБУ) обмениваться данными друг с другом без участия главного компьютера. Например, шина CAN обеспечивает быстрый и надежный 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 идентифи-

катора. В такой системе:

Каждый узел обрабатывает только то сообщение, которое назначалось конкретно ему.

Арбитраж происходит без разрушения кадра - при передаче сигналов от нескольких узлов одновременно победит то сообщение, у которого идентификатор приоритетнее.

CAN самостоятельно контролирует корректность данных на уровне канала.

Однако CAN - всего лишь шина, он даёт транспорт для коротких сообщений, но в сами сообщения он не лезет - для этого нужен какой-то надстроечный протокол на более абстрактном уровне. Здесь и возникает CANopen.

Стандарт CANopen полезен тем, что обеспечивает готовую к использованию совместимость между устройствами (узлами), например, промышленное оборудование. Кроме того, оно 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 живёт от включения питания до штатной работы. Этот протокол

Протокол имеет шесть ключевых особенностей [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, 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. Два наиболее заметных примера - SDO и PDO:

- SDO (Service Data Objects) - это “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.

В инженерных терминах: SDO - чтобы настроить и проверить, PDO - чтобы работать. Отдельно в этот же ряд обычно ставят NMT (Network Management) - механизм управления сетью и состояниями устройств: он отвечает за то, в каком режиме сейчас находится узел и можно ли ему “доверять” процессный обмен.

3. **CANopen-автомат.** CANopen - формализованная модель состояний узла и управление ими через NMT. Для CANopen устройство не просто “подключено к CAN”, оно находится в одном из определённых состояний (например, состояние для настройки, для нормальной работы, для остановки). Это важно потому, что многие действия допустимы не всегда: типично конфигурацию делают в состоянии, где 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 становится более предсказуемой: 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 конфигура-

ция устройства и как к ней обращаться. Практический смысл простой: вместо того чтобы каждый производитель придумывал свой способ настройки, CANopen говорит “все параметры лежат в словаре, к ним обращаются стандартным способом”. Доступ к OD чаще всего идёт через SDO: клиент читает/пишет конкретные элементы словаря. Поэтому OD и SDO концептуально связаны: OD - это 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 приобрел сервомотор ServoMotor3000 для интеграции в конвейерную ленту. При этом оператор редактирует EDS устройства, добавляя специфические для интеграции данные, например, указывая битрейт устройства и идентификатор узла. Модифицированный EDS можно экспортировать в виде файла конфигурации устройства (DCF).

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

Исторически и индустриально сложилось так, что основная масса библиотек и стеков для CAN и CANopen реализована на C (C++). Причина тривиальна: C уже долгое время доминирует в embedded-разработке, где CAN

наиболее распространён; он 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 обвязки вокруг конфигурации.

В этом смысле рассматриваемая область давно ”закрыта”с точки зрения рассматриваемых 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, сопровождаемости и устойчивости системного кода.

Сегодня Rust всё чаще рассматривается как естественный преемник C для задач низкоуровневого программирования: он сохраняет возможность работать близко к железу и контролировать ресурсы и при этом предлагает более строгие гарантии корректности. Однако, фреймворки и библиотеки для CAN/CANopen на Rust заметно малочисленнее, чем их аналоги на C, и в среднем находятся на более ранней стадии зрелости. Для многих проектов характерны ограничения по полноте реализации, менее стабильные API, а также меньшая проверенность в долгоживущих промышленных системах. Иными словами, Rust экосистема остаётся фрагментированной и в значительной мере незаполненной. Именно этот вакуум и определяет мотивацию дальнейшей работы.

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

Добавить коммуникационные объекты

Добавить методологию

Добавить ссылок на материалы

1 Zencan [3] [4]

1.1 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. 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.

Среди своей ниши он выглядит одним из самых перспективных, хотя открыто заявляется, что проект находится в стадии разработки [3].

Если разложить Zencan по прикладным сценариям [4]:

- **Диагностика:** И хотя главной задачей фреймворка не является диагностика - пара механизмов там всё-таки реализованно. Фреймворк позволяет проводить сканирование всех устройств на шине и чтение метаданных. Способен анализировать сигналы HEARTBEAT для первичной оценки в реальном времени. Также есть некоторые нереализованные возможности о планах по внедрению которых заявлено [4].

- **Конфигурация:** Является главной задачей Zencan, поскольку он и задумывался для чтения конфигурации из TOML и генерация кода для узлов.

- **Реалтайм:** Фреймворк способен работать с PDO и, соответственно, с задачами в realtime, однако он не гарантирует жёсткого соблюдения настоящего времени, впрочем, это сложно гарантировать, поскольку в большей степени это зависит от других факторов, таких как драйвер CAN и нагрузка шины.

- **Мониторинг:** Не реализован в полной мере. Zencan обладает своей CLI утилитой которая и позволяет использовать вышеописанный функционал [5], в том числе, что сейчас нам наиболее интересно: HEARTBEAT и считывание метаданных устройств на шине - этого в

целом можно считать достаточным для базового мониторинга, однако функционала например для логирования или построения графиков там нет.

1.2 Поддерживаемые механизмы CANopen

– **SDO** - реализованно, но вероятнее всего будут изменения. В целом с этим можно работать, но необходимо быть осторожным.

– **PDO** - реализованно.

– **NMT** - поддерживает на уровне, достаточным для управления жизненным циклом узла.

– **SYNC** - реализованно, но не полностью.

– **TIME** - не заявлено

– **EMCY** - не заявлено

– **HEARTBEAT** - реализованно в полной мере.

1.3 Архитектура

Одной из главных архитектурных особенностей Zencan является использование как источника истины не OD и не EDS, а конфигурационных файлов, создаваемой пользователем. Такой подход был выбран небезосновательно - автор пишет о том что, его не устраивал стандартный EDS по двум основным причинам:

1. EDS показался автору избыточным и трудным для ручного редактирования
2. Для Zencan необходимо было добавлять ”служебные”настройки и это было сделано сразу внутри конфигурационного файла

Из этого конфигурационного TOML строится OD и часть служебных структур.

Далее Zencan предоставляет пользователю конструктор Node - тип данных инкапсулирующий в себе всё поведение узла, подключенного к CAN шине.

Это ядро, которое реализует поведение CANopen-устройства на шине: оно знает, как принимать и отвечать на типовые CANopen сообщения, как читать/писать значения из OD, как отправлять PDO, как жить в NMT-состояниях, и т.д. При этом по философии оно не привязано к конкретному железу и не навязывает тебе отдельный runtime: ты сам решаешь, где и как крутить этот “движок” - в superloop, в RTOS-задаче, или в async-задаче. Главное, что оно сделано так, чтобы легко встраивалось: приложение само принимает CAN-кадры (в любом удобном контексте) и периодически даёт движку шанс обработать накопившееся и при необходимости отправить ответы. И хотя этот Node должен создавать пользователь, фреймворк генерирует все служебные данные, которые ему необходимы самостоятельно, а именно [6]:

– ”почтовый ящик”для принятия входящих сообщений

– служебное состояние узла

– OD

Единственное, что обязан задать пользователь самостоятельно при конкретно создании Node это ID узла. Однако далее пользователь должен задать какое-то поведение узла, а именно:

1. Приём CAN кадров: необходимо задать их преобразование в структуру NODE_MBOX
2. Периодические вызов обработки
3. Прикладную задачу

Чтобы устройством было удобно пользоваться, к стеку добавлен клиентский слой для Linux: он работает через SocketCAN и даёт утилиты для типовых задач - найти устройство, назначить ему Node-ID (через LSS), залить конфигурацию OD, запустить NMT, смотреть heartbeat/телеметрию.

1.4 Интеграция с транспортом

В Zencan можно выделить три способа подключения:

1. **Linux через SocketCAN/tokio.** При запуске клиента/утилиты на ПК (или на embedded Linux) шина доступна как can0/vcan0. В этом случае Zencan даёт готовый адаптер, который открывает SocketCAN интерфейс и возвращает sender/receiver для работы с CAN кадрами. Используется в первую очередь в zencan-client и zencan-cli.

2. **Embedded(MCU, без привязки к конкретному драйверу).** Используется в случае написании прошивки узла CANopen на MCU. При таком подходе приходится писать собственный node.

3. **Виртуальный транспорт.** Используется для прогона программ без реального железа. Из-за того что Zencan опирается на простые абстракции передачи/приёма CAN кадров (трейты для sync/async), пользователь способен написать адаптер, который реализует эти интерфейсы и прокидывает кадры куда угодно: в VecDeque, в mpsc каналы, в запись/воспроизведение логов, в два узла в памяти.

1.5 Зрелость и поддерживаемость

На момент подготовки текста репозиторий [3] имеет умеренную активность: 27 звёзд, 8 форков, около 180 коммитов, 12 открытых issues и 1 открытый PR. Это живой проект, но пока не крупная экосистема с широким внешним контрибьютингом.

Опубликованные пакеты находятся на версиях 0.0.x, что означает что API не стабилизировано, возможны частые несовместимые изменения. Например, zencan-client 0.0.1 датирован 29 октября 2025.

Автор заявляет, что проект находится на стадии прототипа и возможны изменения в API [3]

У проекта есть отличная документация, покрывающая большую часть его функционала.

1.6 Вывод

Zencan представляет собой ранний, но концептуально цельный CANopen-стек на Rust, ориентированный на сектор встроенных устройст: `no_std`, отсутствие heap-аллоцирования, статическая модель данных и явный контроль

транспорта и планировщика со стороны приложения [6]. Его сильная сторона - путь ”конфигурация (TOML) → генерация OD/типов → узел как вызываемый движок”: это снижает порог входа для создания собственных CANopen-устройств и делает интеграцию предсказуемой [4].

С практической точки зрения проект уже закрывает базовый контур управления и конфигурирования: LSS/NMT/SDO server/PDO mapping явно заявлены, а работоспособные сценарии (LSS fastscan, назначение Node-ID, загрузка конфигурации, запуск NMT, наблюдение статуса/last seen) показаны на примере CLI [7]. Проект сам маркируется как прототип с недостающими возможностями и ожидаемой нестабильностью API [3]. Кроме того, полнота покрытия ”всего CANopen”по публичной документации 0.0.1 подтверждена лишь частично (в частности, детали режимов SDO и наличие ряда сервисов уровня EMCY/TIME/SYNC не зафиксированы явно в описании возможностей) [6].

Следовательно, Zencan разумно использовать как объект исследования и как инженерную базу для прототипов и экспериментов в Rust-экосистеме (особенно там, где важны `no_std` и предсказуемость ресурсов), но в роли ”готового промышленного стека CANopen”он требует дополнительной верификации по нужным частям стандарта и анализа рисков, связанных с ранней стадией развития и возможным API churn [3].

2 OZE-CanOpen [8] [9]

2.1 Назначение и область применения

OZE-CanOpen - это открытый проект (компания Ozon Tech) на Rust, реализующий базовый стек протоколов CANopen, предназначенный в первую очередь для прослушивания и мониторинга шины, а также для работы в роли мастер-узла CANopen в PC-среде. В отличие от полноценных реализаций узлов, OZE-CanOpen не содержит собственной Object Dictionary (OD) и сосредоточен на обмене сообщениями и их разборе. Проект ориентирован пока 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 разложить возможности OZE-CanOpen по прикладным сценариям:

- **Диагностика:** Хотя основной задачей OZE-CanOpen не является автоматизированная диагностика, он предоставляет средства для просмотра и анализа трафика на CAN-шине. Библиотека умеет парсить все типы CANopen-пакетов 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, специфических механизмов активного сканирования устройств на шине (например, поиска всех Node-ID или опроса объекта Identity) в библиотеке не реализовано.

- **Конфигурация:** Работа с конфигурацией узлов является одной из предполагаемых областей применения OZE-CanOpen, хотя решается она не автоматически, а вручную через SDO. Библиотека предоставляет SDO-клиент для каждого узла, что позволяет читать и записывать параметры в OD удалённых устройств программно. По сути, пользователь, зная нужные индексы и подиндексы, может отправлять SDO-запросы на чтение/запись конфигурационных параметров. Однако отсутствуют средства высокого уровня для загрузки/выгрузки конфигураций из файлов EDS/DCF - все операции конфигурирования выполняются ”вручную”или с помощью написанного кода.

- **Реалтайм:** OZE-CanOpen способен обрабатывать PDO-трафик в реальном времени и поддерживает синхронизацию по SYNC. Благодаря асинхронной архитектуре на Tokio, при работе на PC 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 жестких гарантий временных задержек и детерминизма OZE-CanOpen не дает, точность соблюдения таймингов зависит от драйвера CAN, загрузки системы и т.п.

– **Мониторинг:** Функциональность мониторинга в OZE-CanOpen реализована преимущественно через внешние утилиты. В состав экосистемы входит графическая утилита oze-canopen-viewer, использующая библиотеку: она позволяет в реальном времени наблюдать за шиной CANopen, отображая типы пакетов, 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 может добавить это самостоятельно либо воспользоваться 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.

2.2 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 через асинхронный mutex). SDO-сервер (обслуживание 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 производит (так как не моделирует узел-источник), однако мастер-устройство может через нее рассылать PDO, если необходимо, формируя нужные фреймы.

– **NMT:** поддерживается на уровне мастера. OZE-CanOpen может рассылать команды NMT (Start, Stop, Reset) по шине - для этого достаточно отправить кадр с COB-ID 0x000 и соответствующими данными через интерфейс SocketCAN, что можно делать из кода библиотеки. Также библиотека отслеживает heartbeat от узлов (прямо парсит их как особый тип сообщения), что позволяет ей узнавать 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 слать SYNC-кадры с заданным интервалом. Прием и распознавание входящего SYNC (COB-ID 0x80) также поддерживается в парсере, хотя специальной обработки (например, уведомления приложению о наступлении цикла) библиотека не предоставляет - это остается на усмотрение пользователя. В целом, базовая функциональность SYNC (отправка/идентификация) присутствует, расширенные возможности (типа управления временем рассылки в привязке к задачам) не документированы.

– **TIME:** не заявлено в документации и, судя по исходному коду, не поддерживается (кадры с COB-ID 0x100 не обрабатываются явно).

– **EMCY:** не заявлено отдельно. Библиотека может распознать экстренные сообщения (COB-ID 0x80 + Node-ID) как неизвестный тип, так как явной обработки для EMCY в текущем коде нет. Таким образом, прямой поддержки сервиса аварийных сообщений нет - они пройдут

через OZE-CanOpen как обычные CAN-фреймы, которые пользователь может перехватить и обработать вручную.

– **HEARTBEAT:** поддерживается полностью. OZE-CanOpen распознаёт сообщения Heartbeat (COB-ID 0x700 + Node-ID) и определяет состояние узла по содержимому байта состояния. Эти данные могут быть использованы, например, для отслеживания ”живости”узлов и времени с последнего отклика. Также, благодаря этому, в утилите-Viewer реализовано отображение состояния узлов и времени последнего полученного heartbeat.

2.3 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 легковесна, 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 создается объект интерфейса, который запускает фоновые задачи: отдельные потоки/таски для приёма и передачи кадров (параллельный receiver и transmitter), а также монитор интерфейса. Это обеспечивает неблокирующую обработку - прием сообщений с шины не задерживается отправкой и наоборот. Пользователю предоставляется высокоуровневый API: например, функция `canopen::start(''vcan0'', Some(bitrate)` открывает канал, запускает необходимые таски и возвращает объект Interface и набор управляющих хендлов. Через объект Interface можно получить, например, SDO-клиент для конкретного узла (`interface.get_sdo_client(node_i` и затем выполнять с ним операции (чтение/запись) 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 PC). В перспективе (после внедрения `no_std`) планируется поддержка пользовательских драйверов CAN на микроконтроллерах [8], для чего архитектура заложена модульной. Уже сейчас доступны привязки к Python. Таким образом, архитектура OZE-CanOpen нацелена на гибкость: ядро - это движок 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 архитектурную правильность решения: 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 в прикладных задачах.

2.4 Интеграция с транспортом

В текущей версии OZE-CanOpen поддерживает два способа подключения к CAN-шине:

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 `socketcan`-драйвером Rust, что позволяет прослушивать и отправлять сообщения на указанном интерфейсе.

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

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

2.5 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 и имеется один Pull Request. Активность разработки не высокая - последний значимый коммит датируется мартом 2025 года Пакет опубликован на Crates.io в версии 0.1.0 (16 марта 2025), то есть API ещё не стабилизировано и может существенно меняться при последующих выпусках. Автор(ы) прямо помечают проект как ”прототип”и ожидают возможного изменения API.

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

Внешних контрибьюторов пока мало, основной вклад - от команды OzonTech.

2.6 Вывод

OZE-CanOpen представляет собой ранний, но перспективный стек CANopen для Rust, нацеленный скорее на задачи интеграции и отладки, чем на реализацию полноценных узлов устройств. Его сильной стороной является упрощенная 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-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, нельзя из коробки реали-

зовать устройство-ведомый узел), не реализованы некоторые сервисы стандарта (EMCY, TIME, LSS) и не гарантируется стабильность API. Полнота соответствия спецификации CANopen требует проверки - например, работу сегментированных SDO, правильность таймаутов, обработку ошибочных ситуаций библиотека покрывает 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 для изучения опыта реализации CANopen на современном языке (Rust). При планировании же использования в конечных продуктах следует учитывать риски ранней стадии: возможные изменения API, ограниченную функциональность и зависимость от дальнейшей поддержки со стороны разработчиков. В случае, если проект будет активно развиваться и выйдет на стабильную версию, он может занять свою нишу как легкий CANopen-стек для Rust, дополняющий существующие решения на C.

3 Ican [10] [11]

3.1 Назначение и область применения

Ican - это набор утилит и библиотек на Rust для работы с CAN-шиной, включающий, в том числе возможности по поддержке протокола CANopen. Проект позиционируется как ”современные инструменты CAN”для инженеров-разработчиков. Основная форма распространения - консольное приложение `ican`, предоставляющее функционал, аналогичный утилитам из пакета Linux CAN-utils (*candump*, *cansniffer*, *cansend*), но расширенный 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 легко применять на ПК для отладки и тестирования CAN/CANopen-сетей - например, подключив PC через CAN-адаптер к промышленной сети или используя виртуальную шину. В архитектуру также заложена возможность поддержки других источников CAN-данных (альтернативные драйверы, файлы ло-

гов 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 является достаточно уникальным инструментом. 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 своем блоге отмечает, что существующие средства его не удовлетворяли: например, *cansniffer* не умеет работать с расширенными кадрами, не отображает напрямую значения объектов CANopen и требует держать в голове расшифровку PDO/SDO [11]. Ican же призван устранить эти недостатки, предоставив ”всё в одном”для инженера: и sniffer, и отправитель кадров, и декодер CANopen. Проект начат в 2022 году и все еще развивается энтузиастом, поэтому также прямо не предназначен для промышленного применения без доработок.

Если распределить возможности Ican по прикладным сценариям:

– **Диагностика:** Ican изначально разработан для облегчения диагностики CAN/CANopen-сетей. 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 умеет распознавать 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 восьмибайтового HEX можно увидеть, что пришло PDO с определенными 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. Также Ican отслеживает HEARTBEAT-сообщения (распознавая состояние узлов) и может помогать выявлять, какие узлы активны и в каком состоянии. Автоматического сканирования узлов (например, LSS fastscan) в Ican нет, но пользователь, имея EDS устройства, 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 real-time управления, однако имеет возможности для работы с периодическими реальными 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 датчик, посылающий PDO с заданной частотой, 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 здесь 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 utility *cansniffer*. При наличии EDS, Ican может сразу показывать значения отдельных объектов, входящих в PDO, обновляя их в режиме реального времени. Также отображаются пришедшие SDO-ответы, что позволяет следить, как идут последовательные передачи данных (например, процесс загрузки конфигурации). Графического интерфейса или веб-интерфейса у Ican нет - мониторинг осуществляется в текстовом консольном окне. Логирование сообщений не выполняется автоматически, 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 возмож-

ностью интерпретации, но без излишеств (никаких встроенных графиков, GUI и т.п.).

3.2 Поддерживаемые механизмы CANopen

– **SDO:** поддерживается частично, в основном на уровне клиента и декодера. В Ican реализован разбор SDO-пакетов (как исходящих запросов, так и входящих ответов) - утилита распознает SDO Download/Upload по COB-ID (0x600/0x580) и умеет отображать направление и сырые данные. Более того, при использовании EDS она способна интерпретировать индекс и подиндекс запрашиваемого объекта, теоретически даже выводить название параметра. Однако полноценный SDO-стек (сегментирование блочных передач, хранение OD, обработка запросов) в Ican отсутствует - утилита не выступает SDO-сервером. Для пользователя основная возможность - посылать SDO-запросы вручную и видеть результаты. Это достаточный минимум: например, читать отдельные параметры или записывать их для настройки узла. Механизмы вроде подтверждения размера блоков, повторной попытки при таймауте - зависят от реализации SocketCAN и вручную не контролируются. В целом, SDO-составляющая в Ican есть и является одним из ключевых элементов, но она реализована лишь в объеме, нужном для мониторинга и простых операций.

– **PDO:** поддержка PDO в Ican выражается в 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. какие объекты и с каким размером входят в конкретный PDO [11] - и на основе этого расшифровать байты PDO в набор значений переменных. Например, TPDO1 устройства может содержать два 16-битных параметра - Ican, зная это из EDS, при получении PDO разделит 8 байт на две половины, преобразует их в целые и покажет отдельно каждый параметр. Это сильно облегчает анализ обмена процессными данными. Отправлять PDO (эмулировать устройство) Ican напрямую не умеет, хотя пользователь всегда может вос-

пользоваться командой `send` для посылки любого CAN ID. Но логики по формированию PDO-пакетов (на основе локальных данных) у утилиты нет, 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 (если настроены heartbeats). Что касается активного управления NMT: специализированной команды для отправки NMT не предусмотрено. Ican никак не интерпретирует команду NMT на своём уровне (не подтвердит её выполнение, кроме как последующим изменением heartbeat). В целом, минимальная необходимая поддержка NMT есть - увидеть состояния узлов и отправить команды - но она не обёрнута в удобный интерфейс, а требует знаний от пользователя.

– **SYNC:** реализована базовая поддержка. Ican распознает кадры SYNC (0x080) как отдельный тип CANopen-фрейма. При мониторинге, приход SYNC может отображаться, хотя особой информации (кроме факта синхронизации) он не несет. 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 (типа вызова callback по приходу SYNC) нет - опять же, Ican является внешним инструментом, а не частью прошивки. Но для проверки реакций устройств на SYNC-сигналы или для организации синхронного сбора данных во время тестирования эта возможность ценна.

– **TIME:** не поддерживается.

– **EMCY:** явной поддержки аварийных сообщений не видно. Ican распознает EMCY-кадры лишь как просто CAN-фреймы, так как их COB-ID (0x080+NodeID) не перечислен в парсере - сообщение с, например, 0x081 будет ошибочно трактовано как Sync с NodeID=1 (есть такой

нюанс в текущей реализации парсинга). Это можно считать недоработкой. Таким образом, расшифровки кода ошибки и регистра ошибок Ican не делает. Тем не менее само наличие EMCY-сообщений в дампе утилита не пропустит - пользователь увидит их (хотя бы как необработанные кадры) и сможет вручную интерпретировать при необходимости.

– **HEARTBEAT:** поддерживается полностью, 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 пропал (heartbeat перестал приходить), это также станет очевидно из вывода (прекратится обновление). Таким образом, утилита покрывает функцию базового менеджера живучести узлов.

3.3 Архитектура

Архитектурно Ican сочетает в себе черты системной утилиты и библиотечных компонентов. Сердцем проекта является движок обработки CAN/CANopen, написанный на Rust с опорой на асинхронные возможности. Используется Tokio и адаптированная библиотека `tokio-socketcan` для приёма/отправки CAN-фреймов без блокировки. Особенность реализации - автор переработал `socketcan` под совместимость с `embedded-hal` Traits для CAN. Это означает, что логика чтения/записи кадров в Ican написана абстрактно: она могла бы работать и с другим источником, реализующим стандартный Trait (например, с аппаратным CAN контроллером на MCU через HAL-драйвер). Фактически, в кодовой базе Ican CAN-интерфейс - это опция: драйвер выбирается строкой подключения (URI). Например, `socketcan://vcan0` подгрузит модуль для SocketCAN. Такая архитектура делает проект потенциально переносимым и расширяемым - можно реализовать поддержку, скажем, CANAL (CAN over serial) или PCAN, дописав соответствующий модуль, не меняя остальной код.

Важным компонентом архитектуры является парсер CANopen. В Ican введена структура (enum) `CanOpenFrame`, описывающая высокоуровневый вид CANopen-сообщения: Sync, Heartbeat(NmtState), Pdo(PdoType, Data), Sdo(SdoTyp Data) и т.д. Есть функция `parse(frame)`, которая из сырого CAN-фрейма

получает пару (NodeID, CanOpenFrame) либо ошибку, если ID не относится к CANopen. Этот парсер инкапсулирует знания о разметке CANopen-адресного пространства и позволяет остальному коду оперировать понятиями уровня CANopen, а не голыми ID.

Отдельно реализован модуль для парсинга EDS. Автор вынес его в отдельный пакет, который читает INI-файл EDS и формирует структуру данных, содержащую описание OD устройства - список всех объектов, их типы, допустимые диапазоны, карты PDO и пр. В архитектуре Ican EDS-структура 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. вывести не просто ”SDO response for index 0xABCD а указать название параметра. Эта часть описана не подробно.

Вышеописанные компоненты собраны в консольном приложении: в функции `main` реализован разбор аргументов (какой интерфейс, какая команда – dump/send/monitor), запуск асинхронных задач по приему кадров и, если нужно, периодической отправке, а также логика отображения данных. Для режима `monitor` используется, по всей видимости, обновление консоли. Хотя конкретная реализация UI в коде GitHub не освещена, можно предположить, что монитор строится как таблица: в строках - COB-ID (или имя объекта), в колонках - 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 как библиотека может быть включен в стороннюю программу, Ican предназначен как самостоятельное приложение. Однако, части его кода (парсер, EDS-читатель) потенциально могут быть использованы и отдельно, если выделить их в 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 в crates.io и API у него крайне нестабильное.

3.4 Интеграция с транспортом

Возможности интеграции Ican с различными транспортными средами формально заложены, но практически ограничены. Как уже отмечалось, основной транспорт - SocketCAN. При запуске утилиты достаточно указать интерфейс (например, `vcan0` или `can0`) и команду, и Ican подключится к соответствующему сокету ОС для обмена. Внутри используется `socketcan-hal` - модифицированная библиотека, которая адаптирует вызовы CAN к `embedded-hal` интерфейсу. Это сделано, 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 Trait `CANInterface` для нужного источника. Однако, из коробки подобных драйверов нет.

Для встраиваемых систем Ican напрямую не предназначен. Несмотря на использование `embedded-hal`, утилита зависима от Tokio и std. Тем не менее, ядро (парсинг CANopen) потенциально могло бы работать на микроконтроллере, если обеспечить ему поток CAN-кадров. Автором не заявлено планов портирования в no_std, и скорее всего, подход здесь другой: Ican применяется на уровне ноутбука/ПК, подключенного к отлаживаемой системе. Таким образом, интеграция с транспортом остаётся в рамках “подключи и работай” 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, и даже 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 (через дополнительный софт) и с помощью Ican осуществлять с ними обмен, проверяя логику.

3.5 Зрелость и поддерживаемость

Проект Ican, начатый в 2022 году, пока находится на стадии активной разработки одним основным разработчиком. На GitHub репозиторий имеет всего 8 звёзд и 0 форков, что говорит о небольшой известности. Коммит-история (80 коммитов на момент обзора) показывает развитие функционала 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 выпуска (release) ещё не публиковалось - пользователи устанавливают утилиту из исходников. Это означает, что автор не считает проект завершённым или стабильным для широкого распространения.

Документация как таковая отсутствует, роль мануала выполняет README с примерами использования и блоговые записи (которые достаточно информативны, но не формализованы как справочник).

3.6 Вывод

Ican представляет собой экспериментальный, но очень полезный инструмент для разработчиков, работающих с CANopen. В отличие от традиционных ”стеков”для встраиваемых устройств, Ican решает другую задачу - он помогает увидеть и управлять CANopen-сетью с уровня рабочего места разработчика. Можно сказать, что Ican - это аналог диагностического сканера, сочетающий функции анализатора протокола и генератора сообщений.

Однако Ican находится на ранней стадии развития, да и в целом, вероятно, мертв.

Практическое применение Ican на текущий момент - это 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 в них внутри (с помощью EDS, если доступен), или чтобы быстро отправить команду на смену состояния без написания отдель-

ной программы. Для постоянного мониторинга на объекте (в составе продукта) Ican вряд ли предназначен, но как вспомогательный инструмент в наборе инженера он весьма ценен. В сравнении с громоздкими коммерческими анализаторами (типа CANopen Magic) или комбинацией нескольких утилит, Ican предлагает легковесное и скриптуемое решение с открытым кодом.

Подводя итог, Ican заполняет определенную нишу: это CANopen-ориентирован отладочный инструмент, созданный разработчиком для разработчиков. В дальнейшей перспективе, если проект будет развиваться, он может стать основой для целого набора open-source инструментов (симуляторы, мастер-конфигураторы и т.д.) вокруг CANopen, что принесет пользу сообществу. Пока же его стоит использовать с осторожностью, тщательно проверяя результаты и учитывая, что ответственность за корректность некоторых аспектов лежит на пользователе.

ЗАКЛЮЧЕНИЕ

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.* zencan. — 2025. — URL: https://github.com/mcbridejc/zencan (дата обр. 25.12.2025) ; GitHub repository.
4. *McBride J.* Introducing Zencan. — 06.2025. — URL: https://jeffmcbride.net/blog/2025/06/05/introducing-zencan (дата обр. 25.12.2025).
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.