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 17 стр., 0 табл., 0 рис., 6 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

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, хотя и с ограничениями. Сейчас эта часть находится в разработке и вероятно будут улучшения. В целом с этим можно работать, но необходимо быть осторожным.

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

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

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

– **TIME** - не реализованно

– **EMCY** - не реализованно

– **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

1.5 Зрелость и поддерживаемость

1.6 Практическая демонстрация

1.7 Вывод

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.