INNOPOL

ООО "ИННОПОЛ-ТЕХНОЛОГИИ"
443070, САМАРА, УЛ. ПЕСЧАНАЯ Д.1 ОФ. 601
+7-993-466-23-04
OFFICE@INNOPOL.TECH

Документация к разработанному ПО для зарядной станции

Самара, 2023

https://innopol.tech/

СОДЕРЖАНИЕ

1. Описание используемой аппаратной части

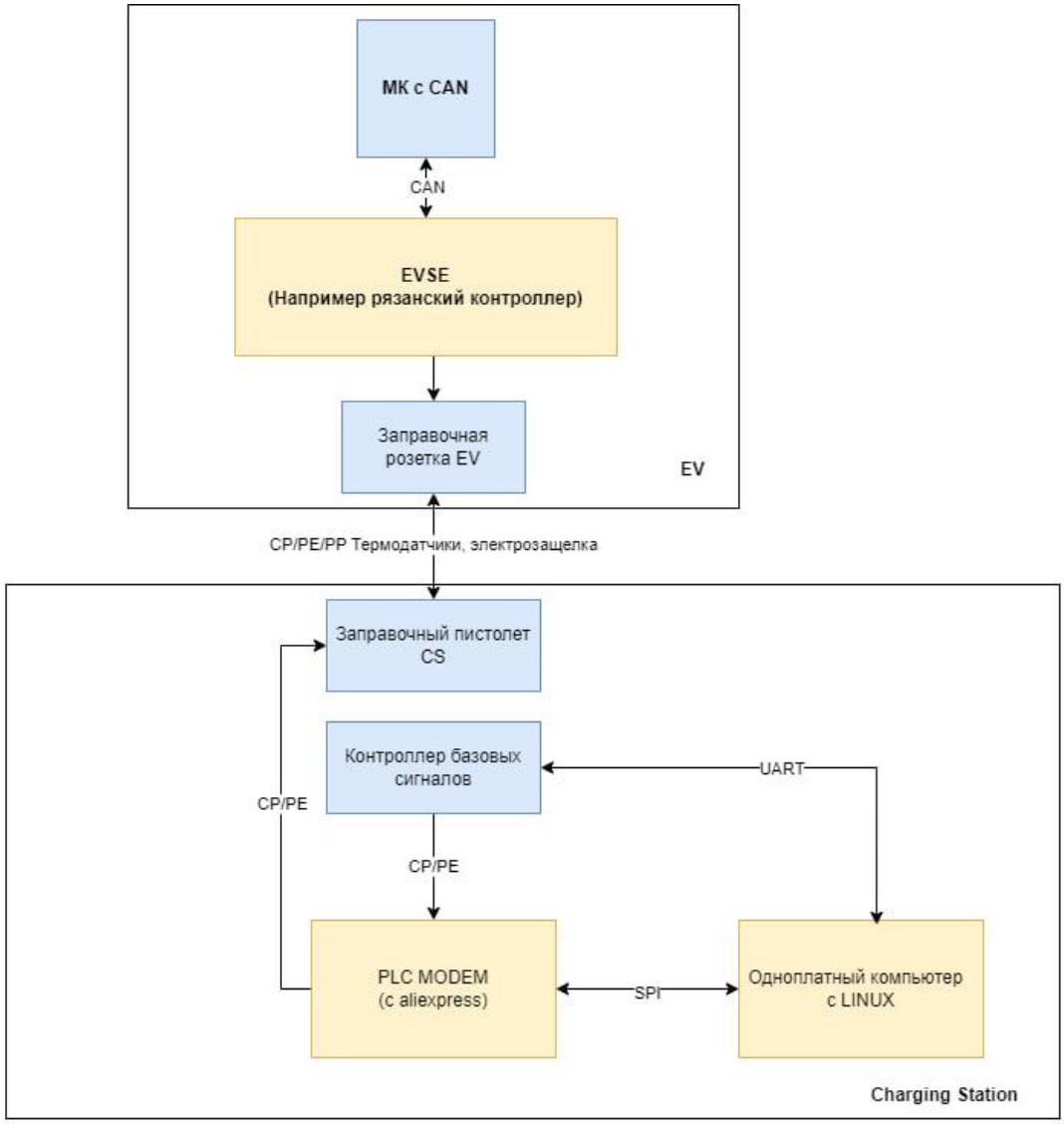
Данный документ описывает разработанное встраиваемое ПО для зарядной станции (EV) в соответствии со стандартом ISO 15118.

На рисунке 1 приведена структурная схема аппаратной реализации разработки в соответствии со стандартом ISO 15118. Ввиду отсутствия в наличии электромобиля и комплекта заправочного пистолета с заправочной розетки они были эмулированы доступными аппаратно-программными средствами. На рисунке 2 приведена схема аппаратной реализации, использования для разработки встраиваемого ПО и ее отладки.

Основные отличия:

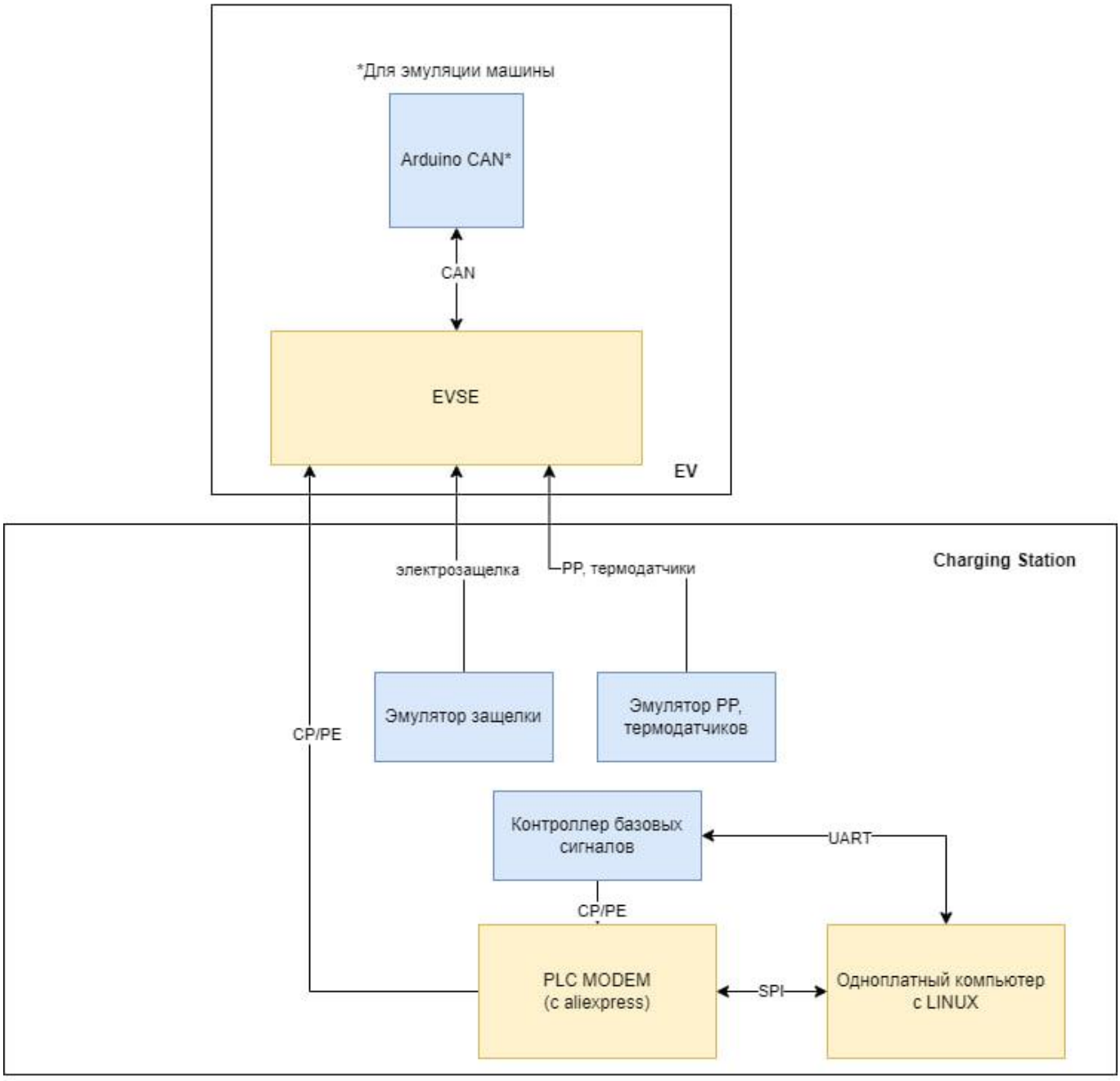
1. Бортовая сеть электромобиля с бортовым компьютером заменены на модуль Arduino CAN.

2. Комплект заправочного пистолета и розетки был заменен на эмулятор электрозащелки пистолета и эмуляторы термодатчиков и контакта PP.



Структурная схема аппаратной реализации ISO15118

Рисунок 1 – Структурная схема аппаратной реализации разработки в соответствии со стандартом ISO 15118



Структурная схема **тестовой** аппаратной реализации ISO15118 в рамках реализации задачи

Рисунок 2 – Структурная схема аппаратной реализации, использованная для разработки встраиваемого ПО

Таким образом список используемых устройств следующий:

1. EVSE - БОРТОВОЙ КОНТРОЛЛЕР БЫСТРОГО ЗАРЯДА ООО «ПРОМЭЛЕКТРОНИКА»
2. PLC MODEM - QCA7000 PEV.
3. Одноплатный компьютер с Linux – не хуже следующих характеристик (Cortex-A7, 800MHz, RAM 256MB DDR3, ROM 256MB NandFlash, 8GB eMMC, Linux4.1.15).
4. Контроллер базовых сигналов с поддержкой ШИМ и UART интерфейсом для соединения с Linu (STM32/AVR и др.)

5. В качестве эмулятора защелки, контакта PP и термодатчиков использовалась навесная электронная схема. Рекомендации по аппаратной части приведены в разделе 3.3 настоящего документа.
6. Arduino CAN – любой микроконтроллер с поддержкой CAN.

2. Описание разработанного встроенного ПО для зарядной станции

Основной репозиторий проекта, устанавливается на одноплатный компьютер с Linux:

_PLC_PYTHON - физический канальный уровень общения с PLC модем *(см. раздел 4 Процесс установки связи на физическом и канальном уровне через PLC)*.

_SDP_PYTHON - сервис поиска зарядки *(см. раздел 5 Процесс определения SECC через SDP)*.

_V2G_PYTHON - сервис V2G для высокоуровненного общения EVSE и EV *(см. раздел 6 Процесс общения в V2G цикле)*.

base.py - сервер работы с базовым сигналом (CP/PE).

Примечание:

1. *V2G сервер сконфигурирован с авторизацией без сертификата и методом зарядки постоянный ток DC.*
2. *В сервере base.py протокол общения с контроллером базовых сигналов открытый и пишется под конкретное устройство. В текущем коде протокол реализован под наше исполнение с _ARDUINO_LOCK_AND_BASE.*

Вспомогательный репозиторий проекта *(разрабатывался под конкретные условия, не задокументирован, предоставляется по запросу):*

_ARDUINO_CAN - инициализация бортового контролера быстрого зарядки по CAN. Используется Arduino Uno с Платой модуля CAN Bus MCP2515 с SPI интерфейсом с приемопередатчиком TJA1050 HW-184.

_ARDUINO_LOCK_AND_BASE - аппаратная и программная эмуляция защёлки пистолета и контроллер базовых сигналов (CP/PE).

Зависимости Python, необходимые для работы ПО на Linux:

netifaces==0.11.0

psutil==5.9.5

scapy==2.5.0

py4j==0.10.9.7

pydantic==2.4.2

typing_extensions==4.8.0

pyserial==3.5

3. Рекомендации по подключению используемых устройств

3.1 Подключение одноплатного компьютера с Linux и PLC модема.

Сборка Linux должна быть версии не менее 4.1.15 и иметь драйвер PLC. Рекомендуется обновить пакеты Linux командой *sudo apt-get update.*

Устанавливаются зависимости для Python:

netifaces==0.11.0

psutil==5.9.5

scapy==2.5.0

py4j==0.10.9.7

pydantic==2.4.2

typing_extensions==4.8.0

pyserial==3.5

Подключение PLC модема QCA7000 и устройства с Linux осуществляется через SPI. Предварительно в Linux необходимо включить периферию SPI.

После подключения PLC модема необходимо убедиться через команду *dmesg*, что драйвер Linux его определил:

pi@raspberrypi:~ $ dmesg | grep qca

[8.679849] qcaspi spi0.0: ver=0.2.7−i, clkspeed=12000000, burst len=5000, pluggable=0

[8.679951] qcaspi spi0.0: Using random MAC address: f6:45:27:00:4e:6b

pi@raspberrypi:~ $

Далее установите утилиту *plctool*:

pi@raspberrypi:~$curl−s https://packagecloud.io/install/repositories/mhei/open−plc−utils/script.deb.sh | sudo bash pi@raspberrypi:~ $ sudo apt install open−plc−utils

Проверьте версию прошивки PLC модема:

pi@raspberrypi:~ $ sudo plctool −r

eth1 00:B0:52:00:00:01 Request Version Information

eth1 00:01:87:FF:FF:2B QCA7000 MAC−QCA7000−1.1.3.1531−00−20150204−CS

pi@raspberrypi:~ $

Еще одна проверка связи с PLC модемом:

sudo plcstat -t -i "eth0"

Рекомендуется присвоить сетевому интерфейсу PLC модема статический адрес, который в дальнейшем использовать в конфигурациях серверов SDP и V2G. В противном случае при перезапуске питания Linux придется прописать новый адрес сетевого интерфейса PLC модема каждый раз заново.

3.2 Подключение одноплатного компьютера с Linux и контроллера базовых сигналов

Контроллер базовых сигналов CP/PE с подготовленной прошивкой для базовых сигналов подключить к PLC модему и устройство с Linux по интерфейсу UART.

Отредактировать в файле *base.py* часть кода, связанную с протоколом общения по UART между Linux и контроллером базовых сигналов. Опциально отредактировать прошивку контроллера базовых сигналов под приведённый протокол общения в файле *base.py*.

3.3 Подключение эмулятора защелки, контакта PP и термодатчиков и эмулятора бортовой сети CAN к бортовому контроллеру быстрого заряда.

Питание:

Для питания бортового контроллера быстрого заряда (далее EVCC) используется источник постоянного напряжение 12 Вольт, с допустимым током не менее 2 Ампер. Питание подводится к контактам 4M (+) и 2M (-) [0].

CAN:

Линия CAN подключается к контактам 2E (CAN H) и 2F (CAN L) [0]. Скорость приёма-передачи – 500 кбит/сек [1]. Терминатор 120 Ом по умолчанию может быть включен или не включен. Для ручного включения необходимо замкнуть контакты 1L и 1M [1].

Датчики температуры:

Для функционирования EVCC необходимо подключить 3 датчика температуры 2A-PTC0, 3A-PTC1, 4A-PTC2, 3B-aGND [0]. Для эмуляции можно использовать подстроечный резистор, который должен быть подключен следующим образом:

aGND < - > RES < - > (PTC0, PTC1, PTC2).

Линия aGND **не подключается** к общей земле.

Например: резистор ~380 Ом, дает температур ~17 градусов по всем датчикам.

PE:

Контакт 2L [0], подключается к общей земле (PLC модем и контроллер базовых сигналов).

PP:

Контакт 2K [0], замыкается на общую землю через резистор ~740 Ом. Это позволяет получить эквивалентное сопротивление PP в 1500 Ом ([1], пакет InletStatus). Другие сопротивления (680, 220, 100 Ом [1]) не позволяют запустить процесс зарядки с использованием HLC (DC EIM).

CP:

Контакт 2J [0], подключается к источнику ШИМ сигнала [-12В; +12В] (контроллер базовых сигналов) и контакту CP PLC модема. Для тестирования использовался однополярный сигнал [0В;

+12В], который также работал. Со стороны контроллер зарядной станции (SECC) необходим последовательный резистор 1 кОм [2]. Земля генератора подключается к общей земле.

Управление защелкой:

Биполярные сигналы управления DC мотором защелки (типа T2HBI24) исходят из контактов 3L и 3M [0]. Уровень напряжения сигнала [-12В; +12В] (в случае питание контролера заряда от 12В).

Линия запирающего механизма идет по контакту 4В [0]. Открытым состоянием считается float состояние контакта. Закрытым состоянием считается замыкание контакта на aGND.

Для **эмуляции закрытия защелки** необходимо с контактов 3L и 3M определить **положительный** импульс и до окончания действия импульса необходимо замкнуть контакт 4В с aGND. Это состояние **необходимо удерживать** вплоть до подачи сигнала на открытие защелки.

Для **эмуляции открытия защелки** необходимо с контактов 3L и 3M определить **отрицательный** импульс и до окончания действия импульса необходимо разомкнуть контакт 4В (контакт в воздухе). Это состояние **необходимо удерживать** вплоть до подачи сигнала на закрытие защелки.

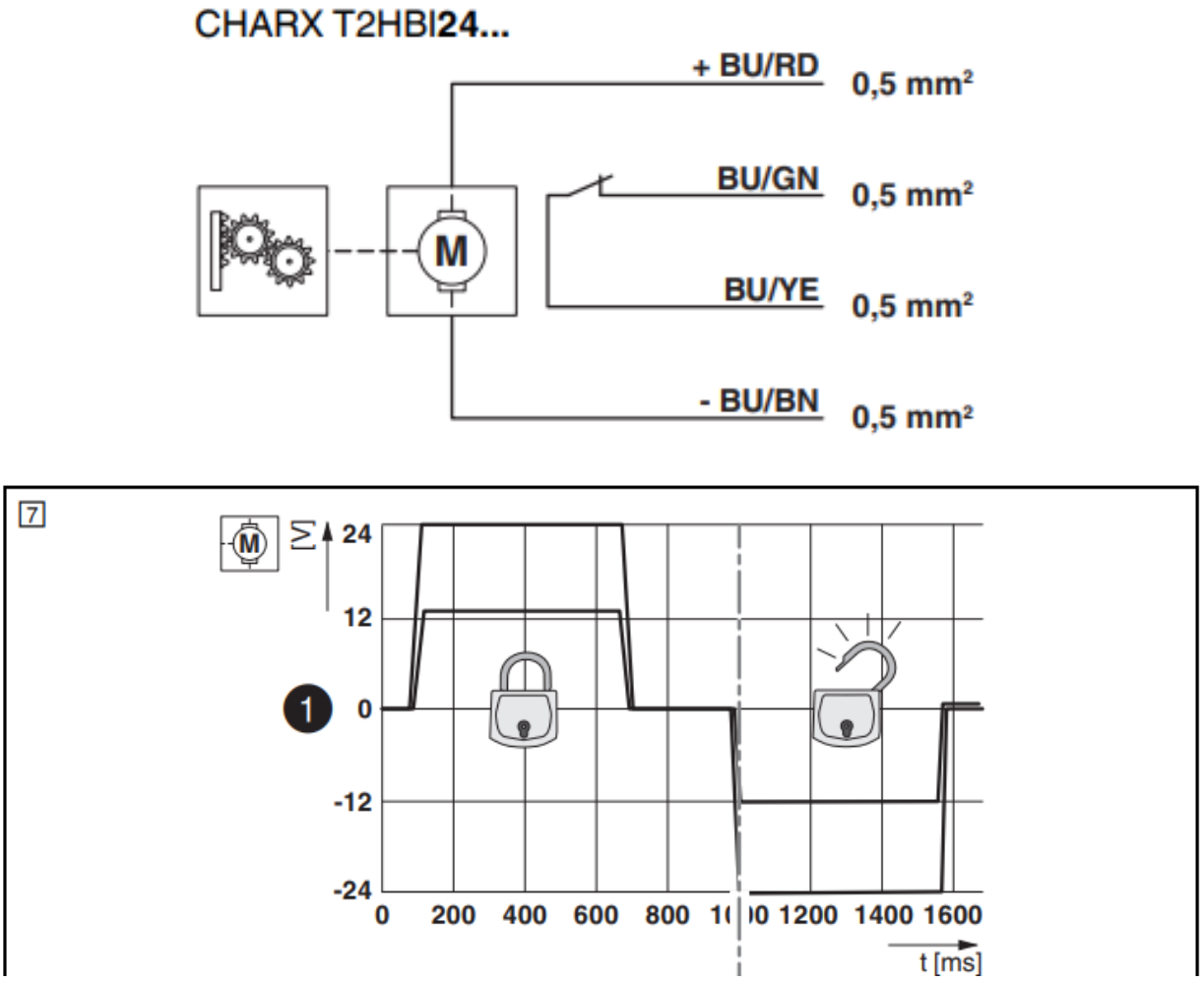


Рисунок 3 – Схема подключения мотора защелки и управляющий импульс на мотор с контроллера заряда [3]

Эмулятор бортовой сети CAN электромобиля должен постоянно поддерживать постоянную связь по CAN. Перед сопряжением нужно установить начальные параметры [1]. Далее все пакеты должны передаваться с интервалом 100 мс:

```
//V2G_DepartureTime
0x18FF4082 = {0xFF, 0xFF, 0xFF, 0xFF, 0x1};

//V2G_EVMaximumVoltageLimit
0x18FF3982 = {0x10, 0x01, 0x32, 0x00, 0x00};

//V2G_EVMaximumPowerLimit
0x18FF3882 = {0x10, 0x01, 0xFA, 0x00, 0x00};

//V2G_EVMaximumCurrentLimit
0x18FF3782 = {0x10, 0x01, 0x05, 0x00, 0x00};

//V2G_EVEnergyRequest
0x18FF3682 = {0x10, 0x01, 0x32, 0x00, 0x00};

//V2G_EVEnergyCapacity
0x18FF3582 = {0x10, 0x02, 0xC8, 0x00, 0x00};

//V2G_EVTargetVoltage
0x18FF3382 = {0xFF, 0x00, 0x0, 0x00, 0x00};

//V2G_EVTargetCurrent
0x18FF3482 = {0x06, 0x00, 0x0, 0x00, 0x00};

//V2G_RemainingTimeToBulkSOC
0x18FF3282 = {0xFF, 0xFF, 0x03, 0x00, 0x01};

//V2G_RemainingTimeToFullSOC
0x18FF3182 = {0xFF, 0xFF, 0x03, 0x00, 0x01};

//Requests
0x18FF2082 = {0x00, 0x00, 0x00, 0x00, 0x00,0x0, 0x00, 0x00};
```

//VehicleStatus

0x18FF3082 = {0x00, 0x00, 0x040, 0x0, 0x00, 0x00, 0x00};

//ChargeFromVehicle

0x18FF2182 = {0x00, 0x00, 0x00, 0x00, 0x55, 0x01};

Источники:

0 - Бортовой контроллер быстрого заряда (Контроллер быстрого заряда CCS) ПШМА.468362.002РЭ.). ООО «ПРОМЭЛЕКТРОНИКА».

1 - Бортовой контроллер быстрого заряда ЛПАС.468362.002 - протокол взаимодействия (универсальный). ООО «ПРОМЭЛЕКТРОНИКА».

2 - СИСТЕМА ТОКОПРОВОДЯЩЕЙ ЗАРЯДКИ ЭЛЕКТРОМОБИЛЕЙ часть 1, ГОСТ Р МЭК 61851-1-2013. (Electric vehicle conductive charging system — Part 1: General requirements, IEC 61851-1:2010). https://files.stroyinf.ru/Data2/1/4293774/4293774646.pdf

3 – RU Инструкция по монтажу для электромонтажника Зарядная розетка для электромобиля CCS типа 2 (T2HBI24). https://data2.manualslib.com/pdf7/306/30569/3056878-phoenix_contact/charx_t2hbi_1ac32dc125_series.pdf?027447ca34a663ef2b7ad59e1b3cd0c4

4. Процесс установки связи на физическом и канальном уровне через PLC

4.1 Сфера применения

Данный документ описывает код для физического и канального уровня SECC на языке Python, для Windows и Linux. Программа не охватывает опциональные случаи при установке соединения (A.9.3 Validation of matching decision, A.9.6 Amplitude map exchange). Программа рассчитана на подключение единственного EVCC и не поддерживает мульти-сессии.

Основные отличия

Программа является переведенной и дополненной версией проекта с открытым исходным кодом - https://github.com/SwitchEV/pyslac

Основные отличия, по сравнению с исходной версией:

- Перевод всех комментариев.
- Исключение работы с переменными средами.
- Изменение работы с сокетами для приема/передачи raw пакетов на сетевой интерфейс. Замена BPF на библиотеку scapy.
- Зацикливание работы программы, с обработкой исключений.
- Доработка всех FIXME и TODO.
- Добавление RPC клиента базовых сигналов и RPC сервера канального уровня.
- Модификации для возможности работы на платформах Linux и Windows.

4.2. Общий алгоритм

Общий алгоритм работы представлен ниже. Далее будет рассмотрен каждый блок.

4.2.1 Инициализация

Точка входа в приложение – файл Main.py, функция run() -> main().

К блоку инициализации относится часть функции main(), выделенная на рисунке.

```
async def main():
    logger.info(f"PLC SECC Start")

    root_dir  = os.path.dirname(os.path.abspath(__file__))
    json_file = open(os.path.join(root_dir, "cs_configuration.json"))
    cs_config = json.load(json_file)
    json_file.close()                                          # Загрузка EVSE_ID и связанных с ними интерфейсов

    base.OpenLINK(cs_config["client_base_addr"])               # Запуск RPC клиента базовых сигналов
    datalink.StartServer(cs_config["server_data_link_addr"],
                         cs_config["server_data_link_port"])   # Запуск RPC сервера канального уровня

    tasks = [prepare(cs_config)]
    await wait_for_tasks(tasks)

def run():
    asyncio.run(main())
```

В данном блоке происходит считывание параметров из файла конфигурации «cs_configuration.json». Блок не имеет обработку исключений, ошибка в блоке приведет к завершению программы.

Содержимое файла «cs_configuration.json»:

cs_configuration.json > server_data_link_port

```
{
  "number_of_evses": 1,
  "parameters": [
  {"evse_id": "DE*SWT*E123456789",
    "network_interface": "Ethernet"
  }
  ],
  "client_base_addr" : "http://192.168.0.201:8001/BASE",
  "server_data_link_addr" : "192.168.0.77",
  "server_data_link_port" : 8002
}
```

В файле необходимо задать следующие параметры: ид зарядной станции, название сетевого интерфейса (подключенного к PLC), адреса и порты: RPC сервера базовых сигналов, и RPC сервера канального уровня.

Server_data_ling_addr – указывается адрес сетевого интерфейса PLC модема.

Client_base_addr – указывается адрес сервера базовых сигналов, на котором запущен файл base.py.

4.2.2 Запуск клиента RPC

К блоку запуска клиента RPC относится часть функции main(), выделенная на рисунке.

```
async def main():
    logger.info(f"PLC SECC Start")

    root_dir  = os.path.dirname(os.path.abspath(__file__))
    json_file = open(os.path.join(root_dir, "cs_configuration.json"))
    cs_config = json.load(json_file)
    json_file.close()                                          # Загрузка EVSE_ID и связанных с ними интерфейсов

    base.OpenLINK(cs_config["client_base_addr"])               # Запуск RPC клиента базовых сигналов
    datalink.StartServer(cs_config["server_data_link_addr"],
                         cs_config["server_data_link_port"])   # Запуск RPC сервера канального уровня

    tasks = [prepare(cs_config)]
    await wait_for_tasks(tasks)

def run():
    asyncio.run(main())
```

Блок выполняет первое подключение к RPC серверу базовых сигналов. Блок не имеет обработку исключений, ошибка в блоке приведет к завершению программы.

Сам клиент расположен в файле Client_base.py, клиент представлен Singleton-ом.

4.2.3 Запуск сервера RPC

К блоку запуска сервера RPC относится часть функции main(), выделенная на рисунке.

```
async def main():
    logger.info(f"PLC SECC Start")

    root_dir  = os.path.dirname(os.path.abspath(__file__))
    json_file = open(os.path.join(root_dir, "cs_configuration.json"))
    cs_config = json.load(json_file)
    json_file.close()                                          # Загрузка EVSE_ID и связанных с ними интерфейсов

    base.OpenLINK(cs_config["client_base_addr"])               # Запуск RPC клиента базовых сигналов
    datalink.StartServer(cs_config["server_data_link_addr"],
                         cs_config["server_data_link_port"])   # Запуск RPC сервера канального уровня

    tasks = [prepare(cs_config)]
    await wait_for_tasks(tasks)

def run():
    asyncio.run(main())
```

Блок запускает RPC сервер канального уровня. Блок не имеет обработку исключений, ошибка в блоке приведет к завершению программы.

Сам сервер расположен в файле Server_data_link.py, сервер представлен Singleton-ом.

4.2.5 Настройка PLC

К блоку настройки PLC относится функция prepare(), выделенная на рисунке.

```
async def prepare(cs_config):
    logger.info("PLC SECC Prepare")
                                        # Допустима только одна сессия, с одним интерфейсом
    if cs_config["number_of_evses"] != 1 or \
        (len(cs_config["parameters"]) != cs_config["number_of_evses"]):
            raise AttributeError("Number of evses provided is invalid.")

    evse_params: dict = cs_config["parameters"][0]
    evse_id: str = evse_params["evse_id"]
    network_interface: str = evse_params["network_interface"]   # Извлечение ID и интерфейса

    try:
        slac_session = SlacEvseSession(evse_id, network_interface)      # Создание сессии
        await slac_session.evse_set_key()                               # Установка параметров приватной сети
    except Exception as e:
        logger.error(
            f"PLC chip initialization failed for "
            f"EVSE {evse_id}, interface "
            f"{network_interface}: {e}. \n"
            f"Please check your settings."
        )
        return

    await enable_hlc_and_trigger_slac(slac_session)                     # Запуск обработчика базовых сигналов
```

Блок создает сессию и выполняет установку параметров приватной сети в PLC. Блок имеет обработку исключений, однако ошибка в блоке приведет к завершению программы.

4.2.6 Рабочий цикл

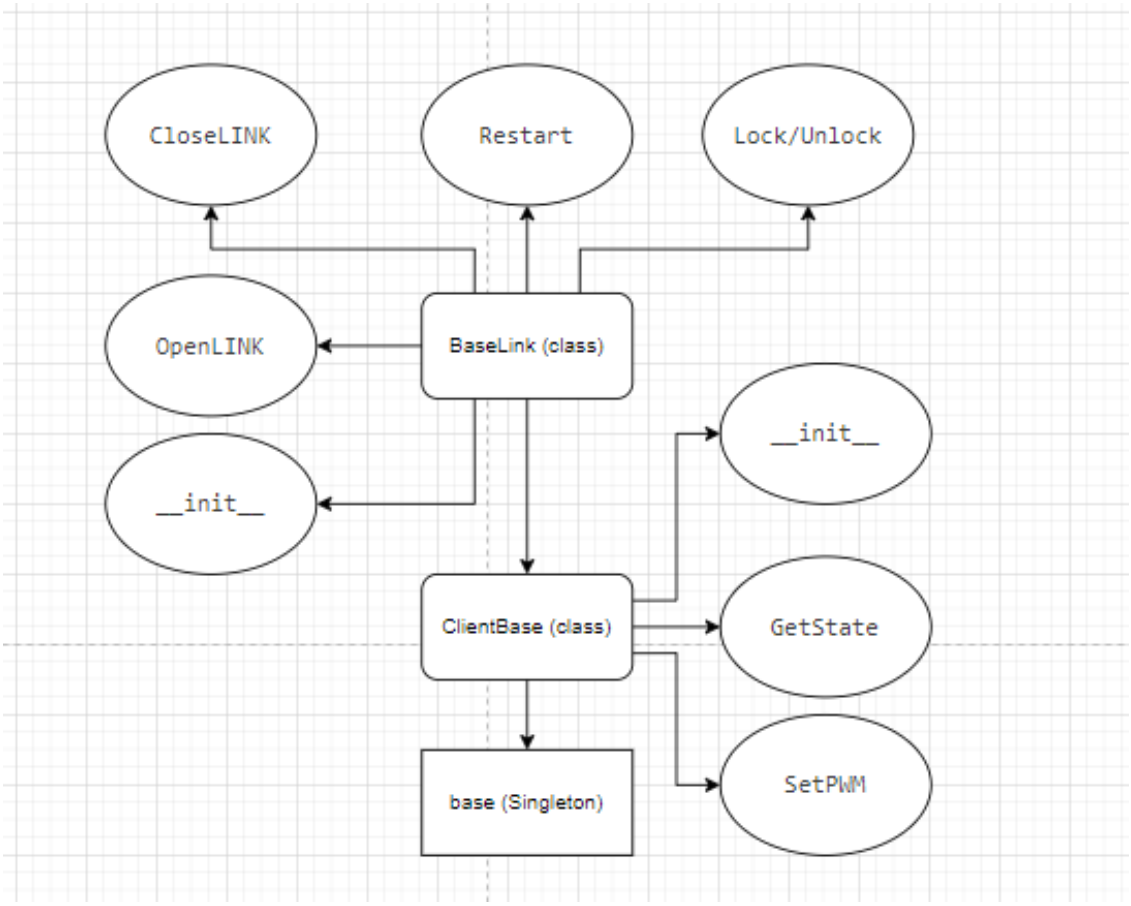
К блоку рабочего цикла относится функция enable_hlc_and_trigger_slac (), выделенная на рисунке.

```
async def enable_hlc_and_trigger_slac(session):
    """
    Обработка состояний базовых сигналов
    """

    Controller = SlacSessionController()    # Контроллер сессии
    while(True):
        try:
            await base.SetPWM(100)                          # Устанавливаем ШИМ=100%
            while(True):
                state = await base.GetState()
                await Controller.process_cp_state(session, state)
                await asyncio.sleep(0.1)
        except Exception as e:                              # Главная функция контроля зациклена
            logger.error(f"HLC_SLAC Exception!!! {e} ")
            await asyncio.sleep(1)
```

Блок запрашивает состояния ([1], таблица А.3) у сервера базовых сигналов и передает их в контроллер сессии. в PLC. Блок имеет обработку исключений, ошибки в блоке игнорируются.

4.3 Обзор клиента RPC

Рабочий объект – base является единственным экземпляром класса ClientBase. ClientBase наследуется от BaseLink.



Функции базового класса:

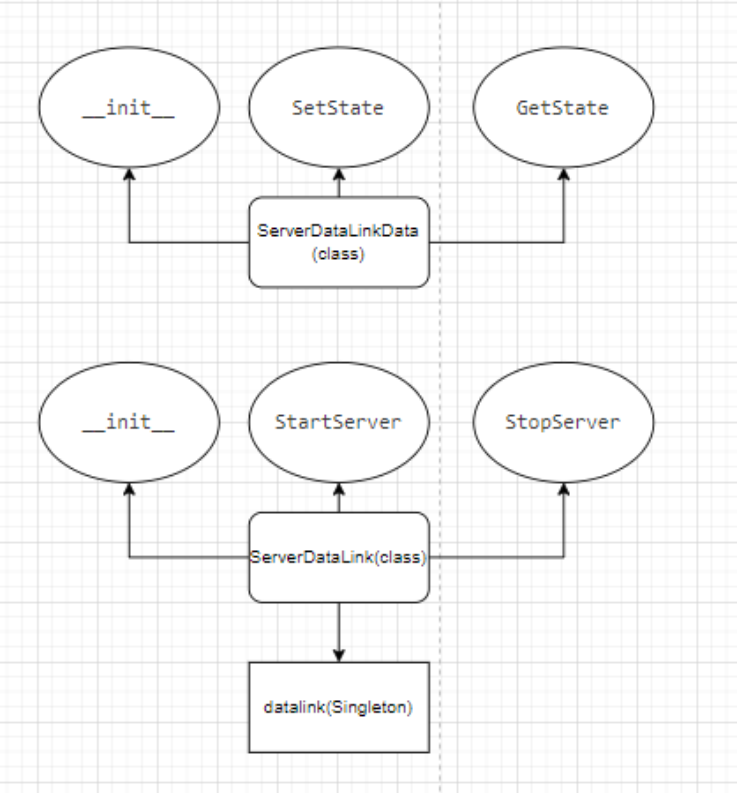
- __init__ - инициализация переменных
- OpenLINK – попытка подключения к RPC серверу, в случае неудачи происходит генерация исключения.
- CloseLINK – попытка отключения от сервера, в случае неудачи исключение игнорируется.
- Restart – несколько попыток переподключения к серверу, в случае неудачи происходит генерация исключения.
- Lock/Unlock – управление блокировкой запросов.

Функции клиента:

- __init__ - инициализация базового класса
- GetState – запрос текущего состояния ([1], таблица А.3), в случае неудачи происходит повторная попытка подключения, результат не обрабатывается.
- SetPWM – установка % заполнения ШИМ), в случае неудачи происходит повторная попытка подключения, если она успешна, происходит повторная попытка отправки, её результат не обрабатывается.

4.4 Обзор сервера RPC

Рабочие объекты – datalink является единственным экземпляром класса ServerDataLink, data_link_data - является единственным экземпляром класса ServerDataLinkData. ServerDataLinkData содержит функции с общим доступом. RequestHandler содержит путь к общим функциям на сервере.



Функции сервера:

- __init__ - инициализация переменных
- StartServer – создание и запуск потока сервера, в случае неудачи происходит генерация исключения.
- StopServer – остановка сервера, закрытие потока.

Общие функции сервера:

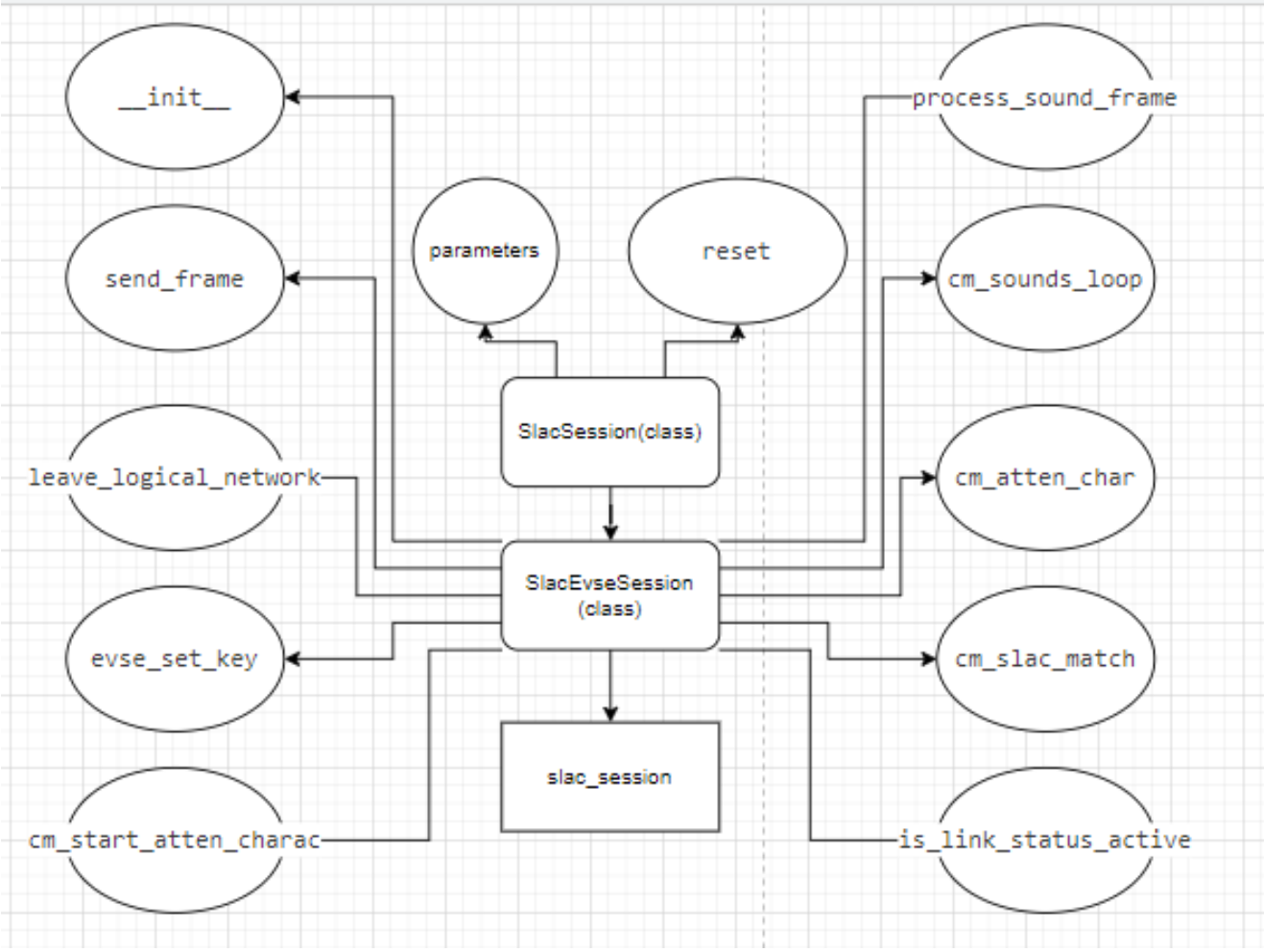
- __init__ - инициализация переменных
- SetState – запрос текущего состояния (“ongoing”, “ failed”, | completed”,).
- GetState – установка текущего состояния (“ongoing”, “ failed”, | completed”,).

4.5 Обзор класса SlacEvseSession

Рабочий объект – slac_session является экземпляром класса SlacEvseSession. SlacEvseSession наследуется от SlacSession.

SlacEvseSession – содержит макрофункции для выполнения алгоритма установления связи (A.9 Matching EV – EVSE process, [0]).

SlacSession – содержит рабочие и вспомогательные параметры.



Обзор функций класса SlacEvseSession:

1. __init__

- Инициализация параметров
- Получение MAK адреса сетевого интерфейса.
- Создание raw сокета
- Инициализация базового класса

2. send_frame

- Перенаправление пакета для отправки на сокет

3. rcv_frame

- Перенаправление ожидания приема пакета на сокет
- Задание таймаута ожидания

4. leave_logical_network

- Выполнение evse_set_key
- Сброс параметров базового класса

5. evse_set_key (Установка параметров PLC (EVSE))

- Генерация параметров приватной сети.

- Создание и отправка пакета CM_SET_KEY.REQ
- Ожидание пакета CM_SET_KEY.CNF
- Создание и отправка пакета CM_NW_INFO.REQ
- Ожидание пакета NW_INFO.CNF

6. evse_slac_parm(Обмен параметрами)

- Ожидание пакета CM_SLAC_PARM.REQ
- Сохранение параметров
- Создание и отправка пакета CM_SLAC_PARM.CNF

7. atten_charac_routine

- Выполнение последовательности макрофункций проверки уровня сигнала и передачи параметров приватной сети EV
- cm_start_atten_charac()
- cm_sounds_loop()
- cm_atten_char()
- cm_slac_match()

8. cm_start_atten_charac Измерение уровня сигнала)

- Ожидание пакета CM_START_ATTEN_CHAR.IND
- Проверка параметров

9. cm_sounds_loop (Измерение уровня сигнала)

- Ожидание очередного пакета CM_MNBC_SOUND.IND или CM_ATTEN_PROFILE.IND
- Обработка пакета в process_sound_frame()
- Выход по таймауту

10. process_sound_frame

- Обработка пакета CM_MNBC_SOUND.IND или CM_ATTEN_PROFILE.IND

11. cm_atten_char (Измерение уровня сигнала)

- Создание и отправка CM_ATTEN_CHAR.IND
- Ожидание пакета CM_ATTEN_CHAR.RSP

12. cm_slac_match (Обмен параметрами приватной сети)

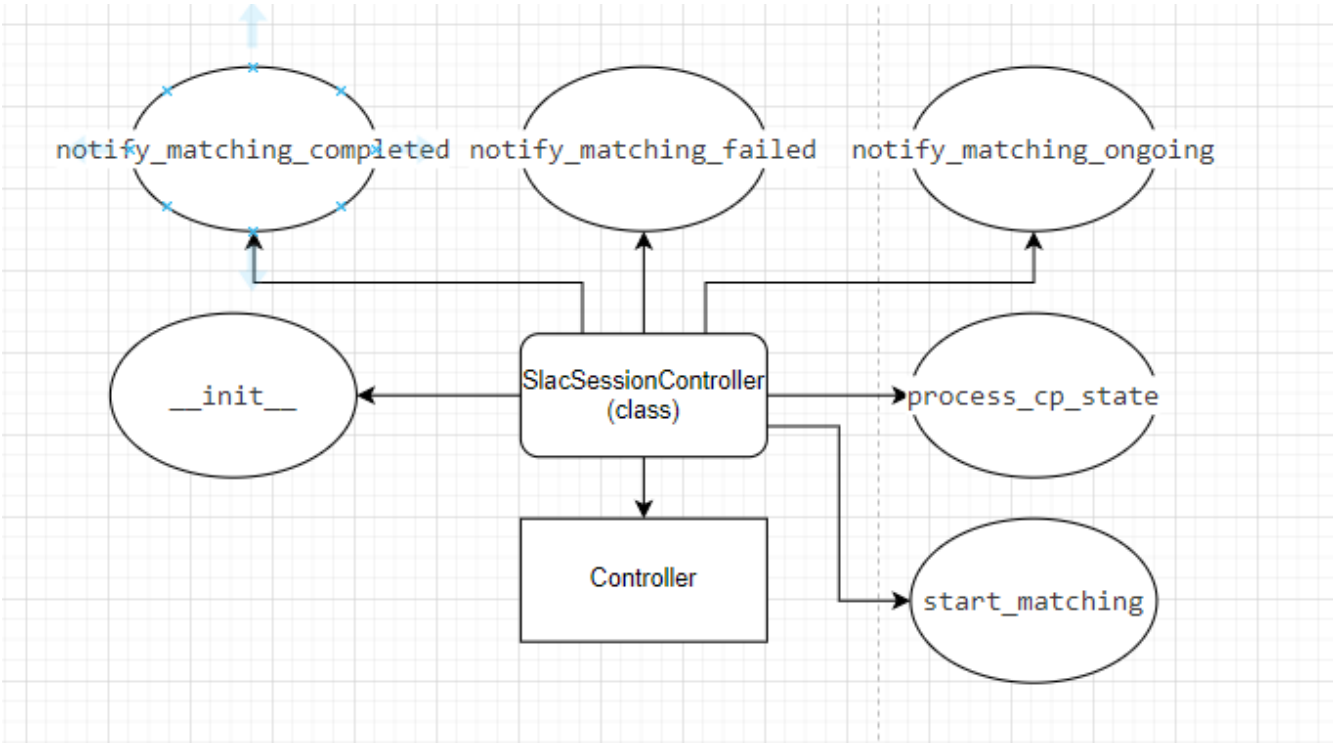
- Ожидание пакета CM_SLAC_MATCH.REQ
- Сохранение параметров
- Создание и отправка пакета CM_SLAC_MATCH.CNF

13. is_link_status_active

- Создание и отправка пакета LINK_STATUS.REQ
- Ожидание CM_LINK_STATUS.CNF

4.6 Обзор класса SlacSessionController

Рабочий объект – Controller является экземпляром класса SlacSessionController.



Функции класса:

1. __init__

- Инициализация переменных

2. notify_matching_ongoing

- Передача на RPC сервер канального уровня состояния «ongoing»

3. notify_matching_failed

- Передача на RPC сервер канального уровня состояния «failed»

4. notify_matching_completed

- Передача на RPC сервер канального уровня состояния «completed»

5. process_cp_state (Триггер начала сопряжения)

- Обработка только по «фронту» изменения состояния
- Для A,E,F – отмена задачи сопряжения, установка ШИМ=100%, переустановка параметров приватной сети.
- Для B,C,D – создание задачи сопряжения.

6. start_matching (Алгоритм установки соединения)

- Попытка выполнить evse_slac_parm (Обмен параметрами)
- Попытка выполнить atten_charac_routine (Измерение уровня сигнала и Обмен параметрами приватной сети)
- В случае успешного подключения, периодическая проверка подключения.
- В случае неудачи перезапуск подключения (ограниченное кол-во попыток).

- В случае, когда попытки закончились – перегенерация параметров приватной сети.

Источник:

(Road vehicles — Vehicle to grid communication interface - Part 3: Physical and data link layer requirements, ISO 15118-3:2015). http://forum.abok.ru/index.php?act=attach&type=post&id=147620

(требуется регистрация на форуме)

5. Процесс определения SECC через SDP

5.1 Сфера применения

Данный документ описывает код для прикладного уровня SDP на языке Python, для Windows и Linux. Программа осуществляет выдачу IPv6 адреса и порта (TCP) SECC для установки V2G loop. Программа не допускает использования TLS.

Основные отличия

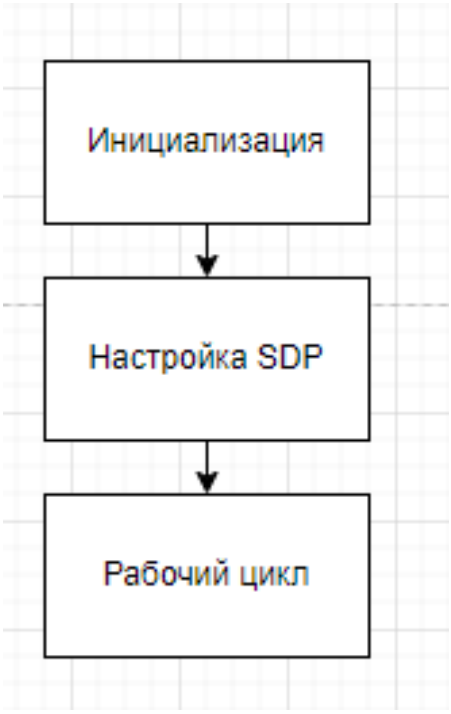
Программа основана на проекте с открытым исходным кодом - https://github.com/EVerest/ext-switchev-iso15118. Программа выделена в отдельный сервис для снижения сложности кода.

Основные отличия, по сравнению с исходной версией:

- Изоляция от основного кода проекта.
- Перевод всех комментариев.
- Исключение работы с переменными средами.
- Добавлена работа с сокетами под Windows.
- Зацикливание работы программы, с обработкой исключений.

5.2. Общий алгоритм

Общий алгоритм работы представлен ниже. Далее будет рассмотрен каждый блок.



5.2.1 Инициализация

Точка входа в приложение – файл Main.py, функция run() -> main().

К блоку инициализации относится часть функции main(), выделенная на рисунке.

```python
async def main():
    logger.info(f"SDP SECC Start")

    root_dir  = os.path.dirname(os.path.abspath(__file__))
    json_file = open(os.path.join(root_dir, "cs_configuration.json"))
    cs_config = json.load(json_file)
    json_file.close()                                          # Загрузка параметров

    tasks = [prepare(cs_config)]
    await wait_for_tasks(tasks)
```

В данном блоке происходит считывание параметров из файла конфигурации «cs_configuration.json». Блок не имеет обработку исключений, ошибка в блоке приведет к завершению программы.

Содержимое файла «cs_configuration.json»:

```
{} cs_configuration.json > [ ] parameters > {} 0
1   {
2     "number_of_evses": 1,
3     "parameters": [
4     {
5       "network_interface": "Ethernet",
6       "secc_addr" : "fe80::3ce3:4a6c:3c82:d516",
7       "secc_port" : 64473
8     }
9     ]
10  }
11
```

В файле необходимо задать следующие параметры: название сетевого интерфейса (подключенного к PLC), адрес и порт TCP сервера SECC V2G.

Команда ip addr show dev eth0 | sed -e's/^.*inet6 \([^]*\)\/.*$/\1/;t;d

5.2.2 Настройка SDP

К блоку настройки SDP относится часть функции prepare(), выделенная на рисунке.

```python
async def prepare(cs_config):
    logger.info("SDP SECC Prepare")
                                                    # Допустима только одна сессия, с одним интерфейсом
    if cs_config["number_of_evses"] != 1 or \
        (len(cs_config["parameters"]) != cs_config["number_of_evses"]):
            raise AttributeError("Number of evses provided is invalid.")

    evse_params: dict = cs_config["parameters"][0]
    network_interface: str = evse_params["network_interface"]   # Извлечение параметров
    secc_addr    : str = evse_params["secc_addr"]
    secc_port    : int = evse_params["secc_port"]

    while(True):
        try:
            receive_loop = ReceiveLoop(network_interface, secc_addr, secc_port)
            await receive_loop.run()
        except Exception as e:                                  # Бесконечный цикл для SDP сервера
            logger.error(f"SDP Exception!!! {e} ")
            await asyncio.sleep(1)
```

Блок извлекает параметры из прочитанного файла конфигурации. Блок не имеет обработку исключений, ошибка в блоке приведет к завершению программы.

5.2.3 Рабочий цикл

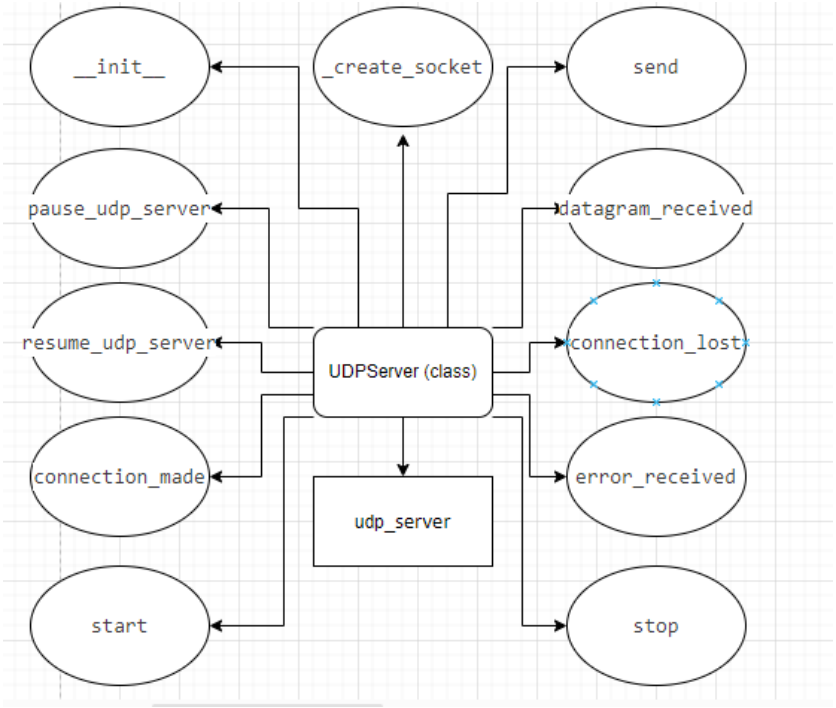
К блоку рабочего цикла относится часть функции prepare(), выделенная на рисунке. Относится к главе 7.10.1 «Протокол обнаружения SECC» и [0] «Протокол обнаружения SECC(SDP)»

```
async def prepare(cs_config):

    logger.info("SDP SECC Prepare")
                                        # Допустима только одна сессия, с одним интерфейсом
    if cs_config["number_of_evses"] != 1 or \
        (len(cs_config["parameters"]) != cs_config["number_of_evses"]):
            raise AttributeError("Number of evses provided is invalid.")

    evse_params: dict = cs_config["parameters"][0]
    network_interface: str = evse_params["network_interface"]   # Извлечение параметров
    secc_addr    : str = evse_params["secc_addr"]
    secc_port    : int = evse_params["secc_port"]

    while(True):
        try:
            receive_loop = ReceiveLoop(network_interface, secc_addr, secc_port)
            await receive_loop.run()
        except Exception as e:                              # Бесконечный цикл для SDP сервера
            logger.error(f"SDP Exception!!! {e} ")
            await asyncio.sleep(1)
```

Блок выполняет рутину по обработке SDP запросов. Блок имеет обработку исключений, ошибки в блоке игнорируются.

5.3. Обзор класса UDPServer

Рабочий объект – udp_server является экземпляром класса UDPServer.

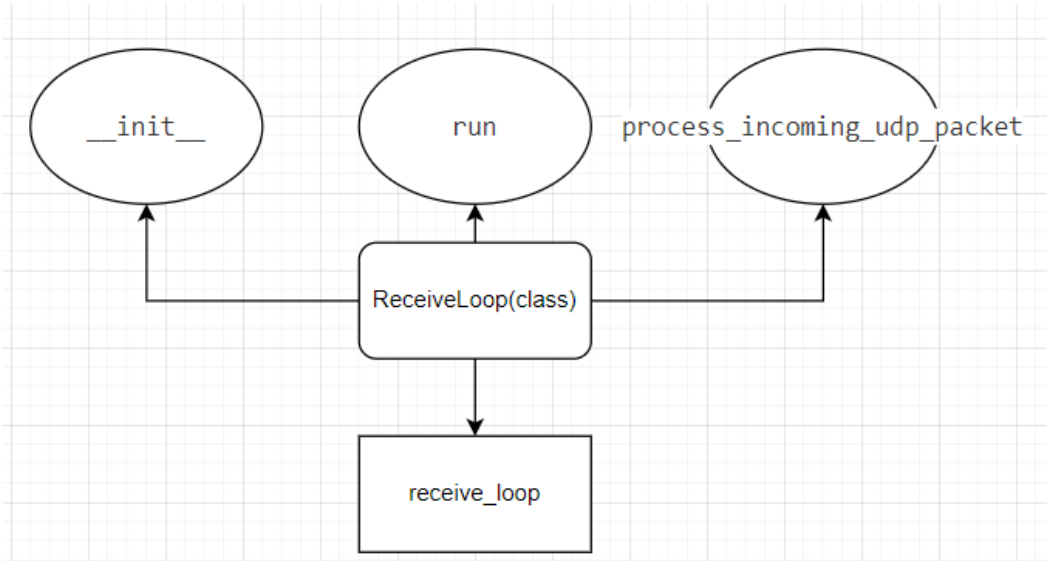


Функции класса:

- __init__ - инициализация переменных
- _create_socket – создание сокета (UDP, Multicast) для платформ Linux и Windows.
- start – создание UDP сервера в цикле событий asyncio.
- send – отправка сообщения сервером по адресу клиента.
- pause_udp_server – переключение SDP сервера в режим ожидания.
- resume_udp_server - переключение SDP сервера в активный режим.
- connection_made - callback от asyncio, сокет успешно запущен и подключен к порту.
- datagram_received - callback от asyncio, был принят пакет UDP, адрес отправителя сохраняется.
- error_received - callback от asyncio, в процессе получения пакета произошла ошибка.
- connection_lost - callback от asyncio, произошла внутренняя ошибка связанная сокетом.

5.4. Обзор класса ReceiveLoop

Рабочий объект – receive_loop является экземпляром класса ReceiveLoop.



Функции класса:

1. __init__:
   - Инициализация переменных
2. run:
   - Создание общедоступной очереди сообщений.
   - Создание и запуск UDP сервера.
   - Переход в цикл process_incoming_udp_packet().
   - Закрытие UDP сервера в случае ошибок.

3. process_incoming_udp_packet, цикл:
    - Проверка работоспособности UDP сервера.
    - Ожидание пакета UDP.
    - Попытка извлечения V2GTP пакета.
    - Попытка извлечения SDP пакета.
    - Составление ответа на SDP запрос (SDP и V2G пакеты).
    - Отправка ответа.

Источник:

Интерфейс связи автомобиль — электрическая сеть Часть 2 Требования к протоколу сетевого и прикладного уровней, ГОСТР 58123- 2018. (Road vehicles — Vehicle-to-Grid Communication Interface — Part 2: Network and application protocol requirements, ISO 15118-2).

https://files.stroyinf.ru/Data/698/69814.pdf

6. Процесс общения в V2G цикле

6.1 Сфера применения

Данный документ описывает код для осуществления процесса общения по V2G циклу связи, между EVCC и SECC. Программа поддерживает DC, и не поддерживает AC режимы. Программа не поддерживает работу с сертификатами и TLS.

Основные отличия

Программа является модификацией проекта с открытым исходным кодом - https://github.com/EVerest/ext-switchev-iso15118.

Основные отличия, по сравнению с исходной версией:

- Изоляция SDP от основного кода проекта.
- Перевод всех комментариев.
- Исключение работы с переменными средами.
- Модификации для работы на Linux и Windows.
- Замена вставок Everest и чистка по сторонним протоколам.
- Зацикливание работы программы, с обработкой исключений.

6.2 Общий алгоритм

Общий алгоритм работы представлен ниже. Далее будет рассмотрен каждый блок.

6.2.1 Инициализация

Точка входа в приложение – файл Main.py, функция run() -> main().

К блоку инициализации относится часть функции main(), выделенная на рисунке.

```
async def main():
    logger.info(f"SECC Start")

    root_dir  = os.path.dirname(os.path.abspath(__file__))
    json_file = open(os.path.join(root_dir, "cs_configuration.json"))
    cs_config = json.load(json_file)
    json_file.close()                                          # Загрузка параметров SECC из файла

    config = Config()                                          # Создание класса конфигурации SECC
    config.load_conf(cs_config)

    base.OpenLINK(cs_config["client_base_addr"])               # Запуск RPC клиента базовых сигналов
    datalink.OpenLINK(cs_config["datalink_base_addr"])         # Запуск RPC клиента канального уровня

    await prepare(config)
```

В данном блоке происходит считывание параметров из файла конфигурации «cs_configuration.json». Блок не имеет обработку исключений, ошибка в блоке приведет к завершению программы.

Содержимое файла «cs_configuration.json»:

```
{
  "number_of_evses": 1,
  "parameters": [
  {
    "evse_id": "DE*SWT*E123456789",
    "network_interface": "Ethernet",
    "secc_addr" : "fe80::3ce3:4a6c:3c82:d516",
    "secc_port" : 64473,
    "log_level": "INFO",
    "secc_enforce_tls": false,
    "free_charging_service":false,
    "free_cert_install_service":true,
    "use_cpo_backend":false,
    "allow_cert_install_service":false,
    "standby_allowed":false
  }
  ],
  "client_base_addr" : "http://192.168.0.201:8001/BASE",
  "datalink_base_addr" : "http://192.168.0.77:8002/DATA_LINK"
}
```

В файле необходимо задать следующие параметры: ID зарядной станции, название сетевого интерфейса (подключенного к PLC), адрес и порт для TCP сервера SECC V2G, уровень логирования, {некоторые параметры SECC}, адреса и порты: RPC сервера базовых сигналов, и RPC сервера канального уровня.

6.2.2 Настройка V2G

К блоку настройки V2G относится часть функции main(), выделенная на рисунке.

```
async def main():
    logger.info(f"SECC Start")

    root_dir  = os.path.dirname(os.path.abspath(__file__))
    json_file = open(os.path.join(root_dir, "cs_configuration.json"))
    cs_config = json.load(json_file)
    json_file.close()                                          # Загрузка параметров SECC из файла

    config = Config()                                          # Создание класса конфигурации SECC
    config.load_conf(cs_config)

    base.OpenLINK(cs_config["client_base_addr"])               # Запуск RPC клиента базовых сигналов
    datalink.OpenLINK(cs_config["datalink_base_addr"])         # Запуск RPC клиента канального уровня

    await prepare(config)
```

```
def load_conf(self, cs_config) -> None:
    """
    Извлечение параметров из файла и сохранение их как полей класса Config.
    """
    logger.info("SECC Extract config")
                                                    # Допустим только один набор параметров
    if cs_config["number_of_evses"] != 1 or \
        (len(cs_config["parameters"]) != cs_config["number_of_evses"]):
            raise AttributeError("Number of evses provided is invalid.")

    evse_params: dict = cs_config["parameters"][0]

    self.iface      = evse_params["network_interface"]
    self.addr       = evse_params["secc_addr"]
    self.port       = evse_params["secc_port"]

    self.log_level  = evse_params["log_level"]

    self.evse_id    = evse_params["evse_id"]

    # Указывает должен ли всегда запускаться TLS, не зависимо от контекста
    self.enforce_tls: bool = evse_params["secc_enforce_tls"]

    # Является ли сервис зарядки бесплатным (определяется через OCPP)
    self.free_charging_service: bool = evse_params["free_charging_service"]
```

Блок извлекает параметры из прочитанного файла конфигурации, и сохраняет их в виде полей класса. Блок не имеет обработку исключений, ошибка в блоке приведет к завершению программы.

6.2.3 Запуск RPC клиента базовых сигналов

К блоку запуска клиента RPC относится часть функции main(), выделенная на рисунке.

```
async def main():
    logger.info(f"SECC Start")

    root_dir  = os.path.dirname(os.path.abspath(__file__))
    json_file = open(os.path.join(root_dir, "cs_configuration.json"))
    cs_config = json.load(json_file)
    json_file.close()                                          # Загрузка параметров SECC из файла

    config = Config()                                          # Создание класса конфигурации SECC
    config.load_conf(cs_config)

    base.OpenLINK(cs_config["client_base_addr"])               # Запуск RPC клиента базовых сигналов
    datalink.OpenLINK(cs_config["datalink_base_addr"])         # Запуск RPC клиента канального уровня

    await prepare(config)
```

Блок выполняет первое подключение к RPC серверу базовых сигналов. Блок не имеет обработку исключений, ошибка в блоке приведет кзавершению программы.

Сам клиент расположен в файле Client_base.py, клиент представлен Singleton-ом.

6.2.4 Запуск RPC клиента канального уровня

К блоку запуска клиента RPC относится часть функции main(), выделенная на рисунке.

```
async def main():
    logger.info(f"SECC Start")

    root_dir  = os.path.dirname(os.path.abspath(__file__))
    json_file = open(os.path.join(root_dir, "cs_configuration.json"))
    cs_config = json.load(json_file)
    json_file.close()                                          # Загрузка параметров SECC из файла

    config = Config()                                          # Создание класса конфигурации SECC
    config.load_conf(cs_config)

    base.OpenLINK(cs_config["client_base_addr"])               # Запуск RPC клиента базовых сигналов
    datalink.OpenLINK(cs_config["datalink_base_addr"])         # Запуск RPC клиента канального уровня

    await prepare(config)
```

Блок выполняет первое подключение к RPC серверу базовых сигналов. Блок не имеет обработку исключений, ошибка в блоке приведет кзавершению программы.

Сам клиент расположен в файле Client_datalink.py, клиент представлен Singleton-ом.

6.2.5 Рабочий цикл

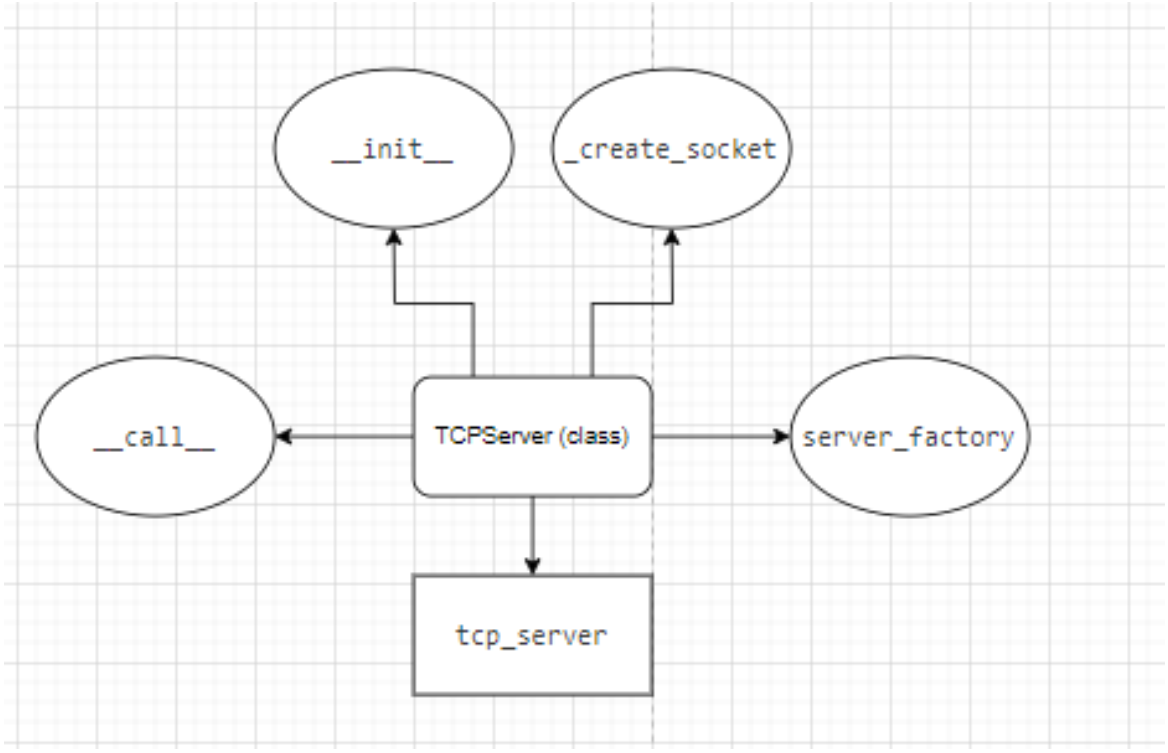
К блоку рабочего цикла относится часть функции prepare(), выделенная на рисунке. Относится к «Последовательность V2G loop – DC»

```
async def prepare(config : Config):
    logger.info(f"Starting 15118 version: {__version__}")

    while(True):
        try:
            v2g_evse_controller = await EVSEController.create(config=config) # Создание экземпляра контроллера
            await v2g_evse_controller.set_status(ServiceStatus.STARTING)
                                                    # Создание экземпляра сессии
            session = CommunicationSessionHandler(config, ExificientEXICodec(), v2g_evse_controller)
            await session.start_session_handler(config.iface)       # Запуск сессии
        except Exception as exc:
            logger.error(f"SECC terminated: {exc}")
            await asyncio.sleep(1.0)
```

Блок выполняет рутину по обработке V2G запросов, общению с RPC серверами и OCPP сервером (заглушка). Блок имеет обработку исключений, ошибки в блоке игнорируются.

6.3 Обзор класса TCPServer

Рабочий объект – tcp_server является экземпляром класса TCPServer.

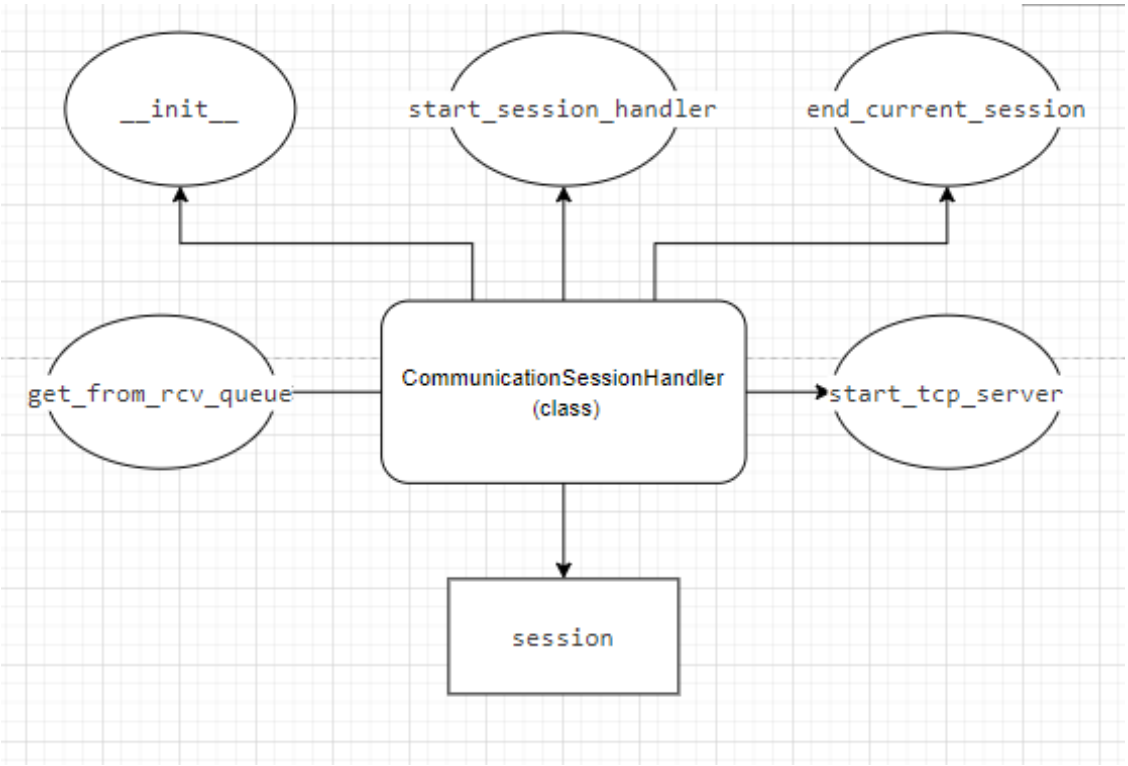


Функции класса:

- __init__ - инициализация переменных
- _create_socket – создание сокета (TCP) для платформ Linux и Windows.
- server_factory – создание TCP сервера в цикле событий asyncio.
- __call__ - callback от asyncio, TCP клиент успешно подключился к серверу, передает стримы на чтение/запись.

6.4 Обзор класса CommunicationSessionHandler

Рабочий объект – session является экземпляром класса CommunicationSessionHandler.



Функции класса:

1. __init__:

- Инициализация переменных
- Установка выбранного кодека EXI

2. start_session_handler:

Запуск TCP сервера.

Запуск обработки событий из общей очереди.

3. get_from_rcv_queue, цикл:

- Ожидание пакета TCP, или события (пауза, завершение сессии).
- Для TCP – возобновляем или создаем новый процесс сессии.

4. end_current_session:

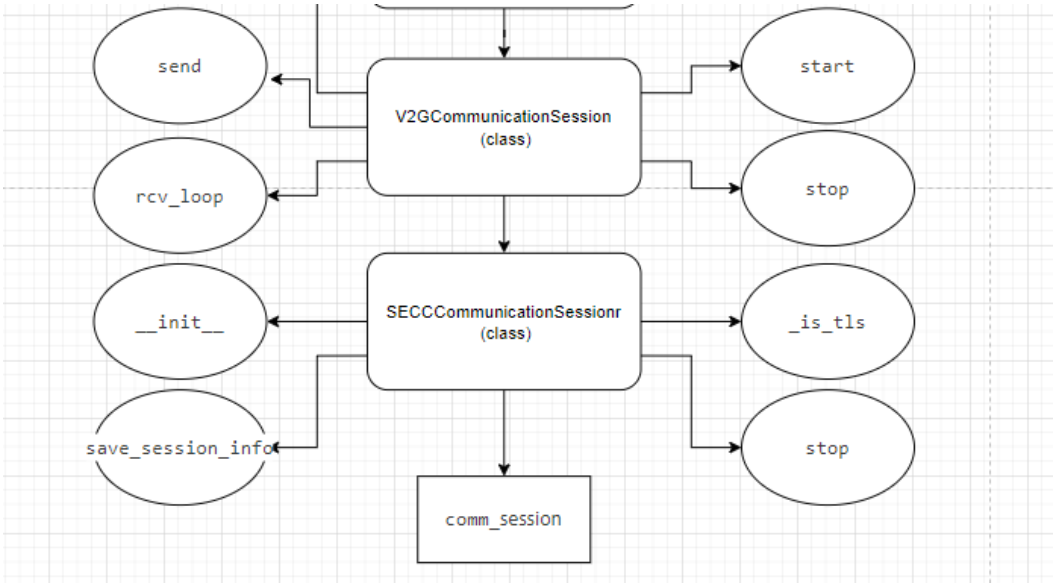
- Завершение процесса сессии
- Завершение процесса TCP сервера
- Перезапуск TCP сервера

6.5 Обзор класса SECCCommunicationSession

Рабочий объект – comm_session является экземпляром класса SECCCommunicationSession.

SECCCommunicationSession наследуется от V2GCommunicationSession.

V2GCommunicationSession наследуется от SessionStateMachine.



Функции класса:

1. __init__:

- Инициализация переменных
- Инициализация базового класса

2. save_session_info:

- Сохранение контекста сессии.

3. _is_tls:

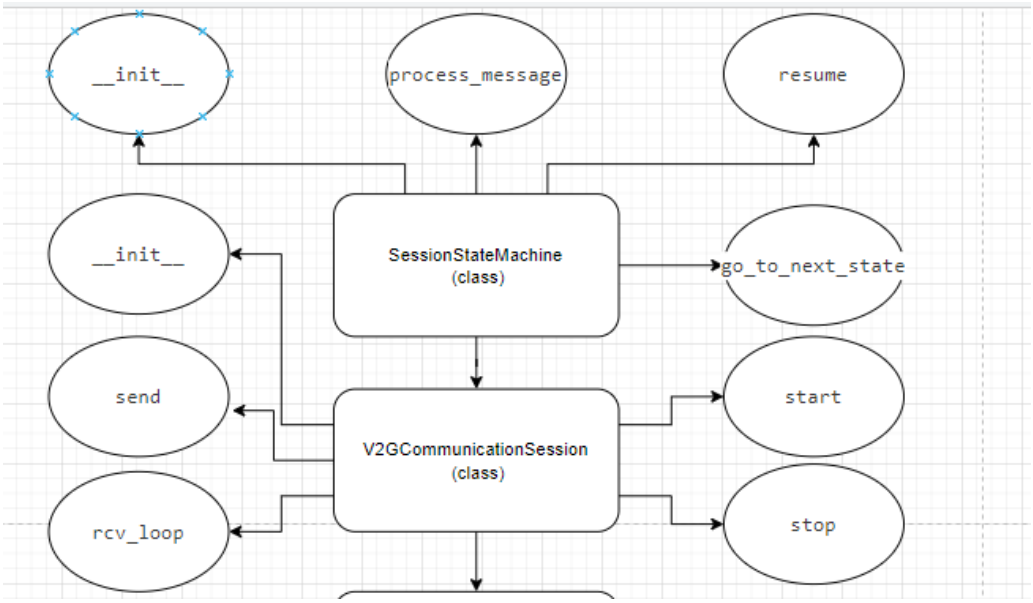
- Проверка, используется ли TLS

4. stop:

- Остановка зарядки
- Остановка TCP клиента и общения на канальном уровне

6.6 Обзор класса V2GCommunicationSession

V2GCommunicationSession наследуется от SessionStateMachine.



Функции класса:

1. __init__:

- Инициализация переменных
- Инициализация базового класса

2. start:

- Запуск цикла обработки сообщений из стрима TCP клиента

3. send:

- Отправка сообщения в стрим TCP клиента

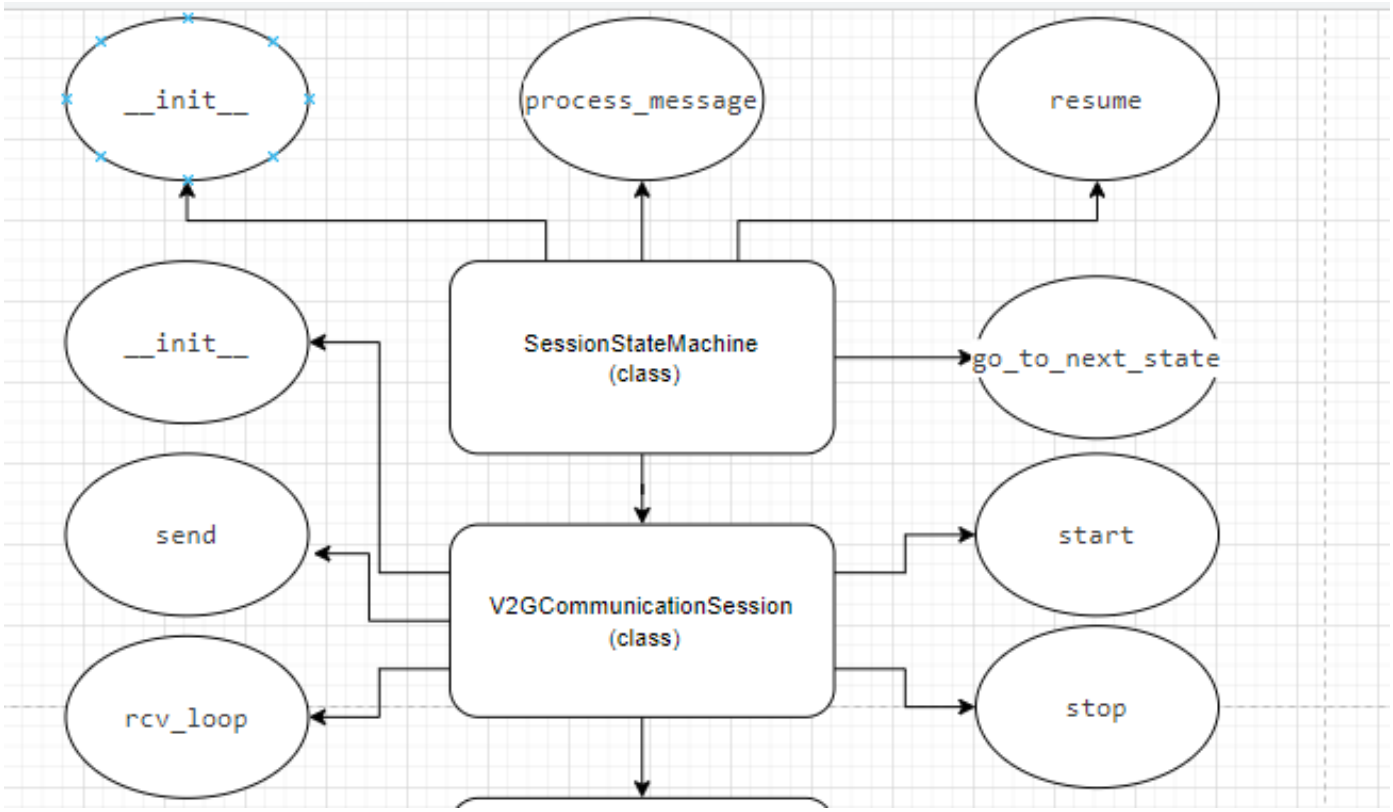
4. stop:

- Остановка TCP клиента и общения на канальном уровне

5. rcv_loop, цикл:

- Ожидание данных от TCP клиента (с таймаутом)
- Обработка сообщения в process_message(). Эта функция едина для каждого состояния машины обработки.
- Отправка ответа TCP клиенту.
- Переход в следующее состояние

6.7 Обзор класса SessionStateMachine



Функции класса:

1. __init__:

- Инициализация переменных

2. go_to_next_state:

- Перевод машины состояний в следующее состояние

3. resume:

- Возобновление сохранённой ранее сессии.

4. process_message:

- Извлечение V2GTP сообщения из массива байт
- Извлечение сообщения из V2GTP пакета, и декодирование из EXI.
- Обработка сообщения в process_message(). Эта функция уникальна для каждого состояния машины обработки

Источник:

Интерфейс связи автомобиль — электрическая сеть Часть 2 Требования к протоколу сетевого и прикладного уровней, ГОСТР 58123- 2018. (Road vehicles — Vehicle-to-Grid Communication Interface — Part 2: Network and application protocol requirements, ISO 15118-2).

https://files.stroyinf.ru/Data/698/69814.pdf

7. Алгоритм проверки работоспособности

Для проверки работоспособности разработанного ПО и установления связи между зарядной станцией и электромобилем в соответствии со стандартом ISO 15118 с использованием аппаратных средств, указанных в данном документе, требуется:

1. Включить эмулятор бортовой сети CAN автомобиля.
2. Включить эмуляторы защелки пистолета, контакта PP и термодатчиков.
3. Включить контроллер базовых сигналов.
4. Включить питание PLC модема.
5. Включить устройство с Linux.
6. Проверить командой на Linux *dmesg | grep qca u sudo plcstat -t -i "eth0"* что PLC модем подключился.
7. Узнать адреса сетевого интерфейса PLC модема и добавить указанные адреса в конфиги сервера PLC, SDP и V2G (см. разделы 3.2.1, 5.2.1 и 6.2.1) – если заранее сетевому интерфейсу PLC не установлен статический IP адрес.
8. Запустить сервер базовых сигналов base.py.
9. Запустить файл main.py директории _PLC_PYTHON и дождаться успешной синхронизации без ошибок.
10. Запустить файл main.py директории _SDP_PYTHON и дождаться успешной синхронизации без ошибок.
11. Запустить файл main.py директории _V2G_PYTHON и дождаться успешной синхронизации без ошибок.
12. Включить бортовой контроллер быстрого заряда.

Далее описывается, какой результат должен наблюдать пользователь

1. После инициализации, бортовой контроллер быстрого заряда размыкает защелку и проверяет, что она открыта.
2. Затем контроллер переходит в режим ожидания, его ожидаемые параметры:

- Все температурные датчики имеют состояние «подключен» и температуру от -39 до 59 градусов.
- Состояние защелки разъема (InletMotorStatus) – открыта.
- Состояние подключения разъема (ConnectionCPStatus) – не подключен.
- Сопротивление PP (PlugPresentResistance) – 1500 Ом,
- Состояние подключения разъема (PlugPresentStatus) – подключен.
- Коэффициент заполнения ШИМ линии CP (Duty Cycle) – 100%.
- Напряжение на линии CP (Voltage) – 9 В.
- Состояние сообщений V2G (MsgStatus) – 0.

- Ошибка машины состояний (StateMachineError) – 0.
- Статус машины состояний (StateMachineStatus) – 1.

3. Затем Контроллер зарядной станции подает ШИМ=5% по линии CP.
4. Бортового контроллер зарядного устройства закрывает защелку и проверяет, что она закрыта.
5. Начинается установление связи по физическому и канальному уровню.
6. После успешной установки данного соединения некоторые параметры изменятся:
   - Состояние защелки разъема (InletMotorStatus) – открыта.
   - Состояние подключения разъема (ConnectionCPStatus) –подключен.
   - Частота ШИМ линии CP (Frequency) – частота 1000 кГц.
   - Коэффициент заполнения ШИМ линии CP (Duty Cycle) – 5%.
   - Состояние сообщений V2G (MsgStatus) – 1.
   - Статус машины состояний (StateMachineStatus) – 4.