Приложение 4

Исходный код программного обеспечения контроллера управления бионическим протезом

{**Файл точки входа приложения**}

```
import os
import sys
import threading
import time
import logging

from arm_prosthesis.external_communication.core.communication import Communication
from arm_prosthesis.external_communication.services.telemetry_service import TelemetryService
from arm_prosthesis.hand_controller import HandController
from arm_prosthesis.config.configuration import load_config
from arm_prosthesis.services.adc_reader import AdcReader
from arm_prosthesis.services.gesture_repository import GestureRepository
from arm_prosthesis.services.motor_driver_communication import MotorDriverCommunication
from arm_prosthesis.services.settings_dao import SettingsDao


class App:
    def __init__(self):
        self._config = load_config('./config/config.ini')
        self.init_logger()
        self._logger = logging.getLogger('Main')
        self._logger.info('Logger init. Start app.')
        self._logger.info(f'App settings:\n{self._config}')

        self._settings_dao = SettingsDao(self._config.settings_path)
        self._logger.info(f'Prosthesis settings:\n{self._settings_dao.get()}')

        self._driver_communication = MotorDriverCommunication()
        self._hand = HandController(self._driver_communication)
        self._gesture_repository = GestureRepository(self._config.gestures_path)
        self._telemetry_service = TelemetryService(self._gesture_repository, self._driver_communication)
        self._communication = Communication(self._hand, self._config, self._gesture_repository, self._telemetry_service,
                                            self._settings_dao)
        self._adc_reader = AdcReader()

        self._driver_communication_thread = threading.Thread(target=self._driver_communication.run)
        self._communication_thread = threading.Thread(target=self._communication.run)
        self._hand_controller_thread = threading.Thread(target=self._hand.run)
        self._adc_reader_thread = threading.Thread(target=self._adc_reader.run)

    def run(self):
        self._logger.info('App start init workers.')

        self._driver_communication_thread.start()
        self._communication_thread.start()
        self._hand_controller_thread.start()
        self._adc_reader_thread.start()
```

```
        self._logger.info('App started.')
        self._hand_controller_thread.join()
        self._logger.info('App closed.')

    def init_logger(self):
        session_name = time.strftime("%Y_%m_%d_%H_%M_%S")
        stdout_handler = logging.StreamHandler(sys.stdout)
        handlers = [stdout_handler]

        if self._config.log_to_file:
            if os.path.isdir(self._config.path_to_log) is False:
                os.makedirs(self._config.path_to_log)

            log_file = self._config.path_to_log + '/' + session_name + '.log'
            print("Log file is: " + log_file)
            file_handler = logging.FileHandler(filename=log_file)
            handlers.append(file_handler)

        logging.basicConfig(
            level=logging.INFO,
            format='%(asctime)s %(levelname)-8s [%(threadName)s] [%(filename)s:%(lineno)d] %(message)s',
            handlers=handlers
        )

if __name__ == '__main__':
    app = App()
    app.run()
```

{**Файл приложения для исполнения жестов протеза**}

```
import logging
import time
from queue import Queue

from arm_prosthesis.models.gesture import Gesture
from arm_prosthesis.models.gesture_action import GestureAction
from arm_prosthesis.models.motor_positions import MotorPositions
from arm_prosthesis.models.positions import Positions
from arm_prosthesis.services.motor_driver_communication import MotorDriverCommunication


class HandController:
    # EC:56:23:F3:91:FC - Honor 10 Lite
    # 98:D3:71:F9:7A:02 - HCF97A02
    _logger = logging.getLogger('Main')

    # Id жестов по умолчанию для команд execute by raw и set_positions
    _uuid_set_positions = '39d4dab8-e2b5-4751-9b1c-d09ecff94f30'

    def __init__(self, driver_communication: MotorDriverCommunication):
        self._set_gesture_queue: 'Queue[Gesture]' = Queue()
        self._driver_communication = driver_communication
        self._logger.info('Hand controller initialized')

    def run(self):
        self._logger.info('Hand controller running')
        while 1:
```

```
        self._logger.info('Wait new gesture')
        # if there is a gesture in queue, we start execute it
        gesture: Gesture = self._set_gesture_queue.get()
        self._logger.info('New gesture receive from the queue. Start execute gesture')
        self._gesture_executor(gesture)

    def _gesture_executor(self, gesture: Gesture):
        if gesture is None:
            raise TypeError

        action_number = 0
        repeat_counter = 0

        self._logger.info(f'Start execute gesture with uuid - {gesture.uuid} ({gesture.name}).')

        if gesture.actions is None:
            self._logger.error('Gesture actions is none. Mock actions list to empty list.')
            actions_list = []
        else:
            actions_list = gesture.actions

        number_of_actions = len(actions_list)
        self._logger.info(
            f'Count of actions {number_of_actions}. Is iterable gesture - {gesture.iterable}. Number of repetitions -'
            f' {gesture.repetitions}')

        # a repeated gesture can be performed until a new gesture arrives
        while (self._set_gesture_queue.empty() and (gesture.iterable or
               (action_number < number_of_actions and
                repeat_counter < gesture.repetitions))):

            action = actions_list[action_number]
            self._logger.debug(f'Get action with index {action_number}')

            motor_positions = MotorPositions(action.little_finger_position, action.ring_finger_position,
                                             action.middle_finger_position, action.index_finger_position,
                                             action.thumb_finger_position, action.thumb_ejector_position)

            self._driver_communication.set_new_positions(motor_positions)

            # if all actions in list were completed
            if number_of_actions == action_number + 1:
                action_number = 0
                repeat_counter += 1
                self._logger.debug(f'All actions is done. Repeat counter is {repeat_counter}')
            else:
                action_number += 1

            self._logger.debug(f'Delay {action.delay} ms before next action')
            time.sleep(action.delay / 1000)

    def set_positions(self, positions: Positions):
        self._logger.info('Set positions execute')
        action = GestureAction(positions.little_finger_position, positions.ring_finger_position,
                               positions.middle_finger_position, positions.index_finger_position,
                               positions.thumb_finger_position, positions.thumb_ejector_position, 0)
```

```
        actions_list = [action]
        gesture = Gesture(self._uuid_set_positions, "SET_POSITIONS", 0, False, 1, actions_list)

        self._set_gesture_queue.put(gesture)

    def perform_gesture(self, gesture: Gesture):
        self._logger.info('Perform gesture start')
        self._set_gesture_queue.put(gesture)
```

{**Файл приложения для взаимодействия с контроллером линейных приводов по SPI**}

```
import logging
import time
from queue import Queue
from typing import List

import crc8
import spidev

import enums_pb2 as enums
from arm_prosthesis.models.actuator_controller_queue import ActuatorControllerQueue, ActuatorControllerCommand
from arm_prosthesis.models.driver_telemetry import DriverTelemetry
from arm_prosthesis.models.motor_positions import MotorPositions
from arm_prosthesis.utils.stoppable_thread import StoppableThread


class ActuatorControllerService:
    _logger = logging.getLogger('Main')
    _default_interval = 0.5
    _telemetry_thread: StoppableThread = None

    def __init__(self):
        self._set_positions_queue: 'Queue[ActuatorControllerQueue]' = Queue()
        self._empty_payload = [0xFF] * 9
        self._telemetry_request = [0x00] * 9
        self._telemetry_request[8] = int.from_bytes(self._get_crc8_for_request(self._telemetry_request), "big")
        self._telemetry = DriverTelemetry(0, 0, 0, 0, 0, 0, 0)

    @property
    def telemetry(self) -> DriverTelemetry:
        return self._telemetry

    def enable_telemetry(self, interval_in_ms=None):
        if self._telemetry_thread is not None:
            raise Exception('Telemetry already started')

        if interval_in_ms is None:
            interval_seconds = self._default_interval
        else:
            interval_seconds = interval_in_ms / 1000

        if interval_seconds < 0:
            raise Exception('Incorrect interval')

        self._telemetry_thread = StoppableThread(self._telemetry_runner, interval_seconds)
        self._telemetry_thread.start()
        self._logger.info('Telemetry started')
```

```
def disable_telemetry(self):
    if self._telemetry_thread is None:
        raise Exception('Telemetry not started')

    self._telemetry_thread.stop()
    self._telemetry_thread = None
    self._logger.info('Telemetry stopped')

def _telemetry_runner(self):
    queue_command = ActuatorControllerQueue(ActuatorControllerCommand.TELEMETRY)
    self._set_positions_queue.put(queue_command)

def run(self):
    spi = spidev.SpiDev()
    spi.open(0, 0)

    spi.bits_per_word = 8
    spi.max_speed_hz = 500000

    self._logger.info('Motor driver communication running')
    while 1:
        new_command: ActuatorControllerQueue
        new_command = self._set_positions_queue.get()

        requests: List[int]
        if new_command.command_type == ActuatorControllerCommand.SET_POSITIONS:
            self._logger.info('New positions receive from the queue. Send to driver')
            request = self._create_set_positions_request(new_command.motor_positions)
            logging.info(f"Send to driver: {request}")
        else:
            if new_command.command_type == ActuatorControllerCommand.TELEMETRY:
                request = self._telemetry_request
            else:
                raise Exception('Not supported')

        spi.xfer(request)
        time.sleep(0.02)

        # receive telemetry
        result = spi.xfer(self._empty_payload)

        self._set_telemetry(result)
        if new_command.command_type != ActuatorControllerCommand.TELEMETRY:
            logging.info(f"Receive from driver: {result}")

def set_new_positions(self, positions: MotorPositions):
    queue_command = ActuatorControllerQueue(ActuatorControllerCommand.SET_POSITIONS, positions)
    self._set_positions_queue.put(queue_command)

def _set_telemetry(self, response_driver: bytes):
    if response_driver[0] == 0:
        type_work = enums.DRIVER_STATUS_CONNECTION_ERROR

        crc_calculator = crc8.crc8()
        for i in range(0, len(response_driver)):
            crc_calculator.update(response_driver[i].to_bytes(1, 'little'))
```

```
        if response_driver[-1] == crc_calculator.digest():
            type_work = self._type_work_convert(response_driver[1])

        self._telemetry = DriverTelemetry(type_work, response_driver[2],
                                          response_driver[3], response_driver[4],
                                          response_driver[5], response_driver[6],
                                          response_driver[7])

    @staticmethod
    def _type_work_convert(state_code: int):
        if state_code == 0:
            return enums.DRIVER_STATUS_INITIALIZATION
        if state_code == 1:
            return enums.DRIVER_STATUS_ERROR
        if state_code == 2:
            return enums.DRIVER_STATUS_ERROR
        if state_code == 3:
            return enums.DRIVER_STATUS_SLEEP
        if state_code == 4:
            return enums.DRIVER_STATUS_SETTING_POSITION
        if state_code == 5:
            return enums.DRIVER_STATUS_ERROR

    @staticmethod
    def _get_crc8_for_request(request):
        crc_calculator = crc8.crc8()
        for i in range(0, len(request) - 1):
            crc_calculator.update(request[i].to_bytes(1, 'little'))

        return crc_calculator.digest()

    def _create_set_positions_request(self, positions: MotorPositions):
        protocol_driver_package = [0x00] * 9
        protocol_driver_package[0] = 1
        protocol_driver_package[1] = 0
        protocol_driver_package[2] = positions.little_finger_angle_position
        protocol_driver_package[3] = positions.ring_finger_angle_position
        protocol_driver_package[4] = positions.middle_finger_angle_position
        protocol_driver_package[5] = positions.index_finger_angle_position
        protocol_driver_package[6] = positions.thumb_finger_angle_position
        protocol_driver_package[7] = positions.thumb_ejector_angle_position
        protocol_driver_package[8] = int.from_bytes(self._get_crc8_for_request(protocol_driver_package),
                                    byteorder="big")
        return protocol_driver_package
```

{**Файл приложения для хранения жестов протеза в файловой системе**}

```
import json
import logging
import os
from typing import List, Dict

from arm_prosthesis.external_communication.models.dto.gesture_dto import GestureDto


class GestureRepository:
```

```
_logger = logging.getLogger('Main')

_gestures_dictionary: Dict[str, GestureDto]
_path_to_gesture_folder: str
_common_info_file_name = 'info.json'
_gestures_directory_name = 'gestures'
_gestures_file_extension = '.gesture'

_common_info = {
   "last_time_sync": 0
}

def __init__(self, path_to_gesture_folder: str):

   if type(path_to_gesture_folder) is not str:
      self._logger.critical(f'Path to gestures incorrect. Is {path_to_gesture_folder}')
      raise Exception(f'Path to gestures incorrect. Is {path_to_gesture_folder}')

   self._path_to_gesture_folder = path_to_gesture_folder
   self._path_to_gestures = os.path.join(self._path_to_gesture_folder, self._gestures_directory_name)
   self._path_info_file = os.path.join(self._path_to_gesture_folder, self._common_info_file_name)

   self._gestures_dictionary = {}
   self._load_dictionary()

def add_gesture(self, current_time: int, new_gesture: GestureDto):
   if new_gesture.id in self._gestures_dictionary:
      self._logger.info(f'Update gesture {new_gesture.id}')
   else:
      self._logger.info(f'Adding new gesture {new_gesture.id}')

   with open(os.path.join(self._path_to_gestures, new_gesture.id + self._gestures_file_extension),
         'wb') as gesture_file:
      gesture_file.write(new_gesture.serialize())

   self._gestures_dictionary[new_gesture.id] = new_gesture
   self.update_time_sync(current_time)

def remove_gesture(self, current_time: int, gesture_id: str):
   self._logger.info(f'Remove gesture {gesture_id}')
   os.remove(os.path.join(self._path_to_gestures, gesture_id + self._gestures_file_extension))

   del self._gestures_dictionary[gesture_id]
   self.update_time_sync(current_time)

def get_gesture_by_id(self, gesture_id: str) -> GestureDto:
   self._logger.info(f'Get gesture {gesture_id}')
   return self._gestures_dictionary[gesture_id]

def get_all_gestures(self) -> List[GestureDto]:
   self._logger.info(f'Get all gestures')
   return list(self._gestures_dictionary.values())

@property
def last_time_sync(self) -> int:
   return self._common_info['last_time_sync']
```

```
    def update_time_sync(self, new_time_sync):
        self._logger.info(f'Update last time sync to new value: {new_time_sync}')

        self._common_info['last_time_sync'] = new_time_sync

        with open(self._path_info_file, 'w') as info:
            json.dump(self._common_info, info)

    def _load_dictionary(self):
        self._logger.info(f'Load gestures')

        if os.path.isdir(self._path_to_gestures) is False:
            self._create_default()

        with open(self._path_info_file, 'r') as info:
            self._common_info = json.load(info)

        self._logger.info(f'Loaded time sync: {self._common_info["last_time_sync"]}')

        for file in os.listdir(self._path_to_gestures):
            if file.endswith(self._gestures_file_extension):
                with open(os.path.join(self._path_to_gestures, file), 'rb') as gesture_file:
                    gesture_dto = GestureDto()
                    gesture_dto.deserialize(gesture_file.read())
                    self._gestures_dictionary[gesture_dto.id] = gesture_dto

        self._logger.info(f'Loaded gestures count: {len(self._gestures_dictionary)}')

    def _create_default(self):
        self._logger.info('Start create new gestures directory')
        os.makedirs(self._path_to_gestures)

        self.update_time_sync(0)
```

{**Файл приложения для хранения настроек протеза**}

```
import json
import logging
import os

from arm_prosthesis.external_communication.models.dto.get_settings_dto import GetSettingsDto
from arm_prosthesis.external_communication.models.dto.set_settings_dto import SetSettingsDto
from arm_prosthesis.models.mode_type import ModeType


class SettingsDao:
    _logger = logging.getLogger('Main')

    _settings: GetSettingsDto

    def __init__(self, path_to_settings_file: str):
        if type(path_to_settings_file) is not str:
            self._logger.critical(f'Path to settings incorrect. Is {path_to_settings_file}')
            raise Exception(f'Path to settings incorrect. Is {path_to_settings_file}')

        self._path_to_settings_file = path_to_settings_file
        self._settings = self._load_settings()
```

```
def get(self) -> GetSettingsDto:
    return self._settings

def save(self, settings: SetSettingsDto):
    self._settings.type_work = settings.type_work
    self._settings.enable_emg = settings.enable_emg
    self._settings.enable_display = settings.enable_display
    self._settings.enable_driver = settings.enable_driver
    self._settings.enable_gyro = settings.enable_gyro

    self._save_settings_to_file(self._settings)

def _load_settings(self) -> GetSettingsDto:
    self._logger.info(f'Load settings')

    if os.path.isfile(self._path_to_settings_file) is False:
        self._create_default()

    return self._load_settings_from_file()

def _create_default(self):
    self._logger.info('Start create new settings')

    settings = GetSettingsDto()
    settings.type_work = ModeType.Auto
    settings.enable_emg = False
    settings.enable_driver = False
    settings.enable_display = False
    settings.enable_gyro = False

    self._save_settings_to_file(settings)

def _save_settings_to_file(self, settings: GetSettingsDto):
    settings_content_default = {
        "type_work": settings.type_work,
        "enable_emg": settings.enable_emg,
        "enable_display": settings.enable_display,
        "enable_gyro": settings.enable_gyro,
        "enable_driver": settings.enable_driver
    }

    with open(self._path_to_settings_file, 'w') as settings_file:
        json.dump(settings_content_default, settings_file)

def _load_settings_from_file(self) -> GetSettingsDto:
    settings = GetSettingsDto()

    with open(self._path_to_settings_file, 'r') as settings_file:
        settings_content = json.load(settings_file)
        settings.type_work = ModeType(settings_content['type_work'])
        settings.enable_emg = settings_content['enable_emg']
        settings.enable_display = settings_content['enable_display']
        settings.enable_gyro = settings_content['enable_gyro']
        settings.enable_driver = settings_content['enable_driver']

    return settings
```

{**Файл приложения для исполнения протокольный команд протеза**}

```
import logging
import os
import time
import traceback
from queue import Queue

from arm_prosthesis.config.configuration import Config
from arm_prosthesis.external_communication.core.connectors.mqtt_connector import MqttConnector
from arm_prosthesis.external_communication.core.connectors.rfcc_connector import RFCCConnector
from arm_prosthesis.external_communication.models.command_type import CommandType
from arm_prosthesis.external_communication.models.dto.delete_gesture_dto import DeleteGestureDto
from arm_prosthesis.external_communication.models.dto.get_gestures_dto import GetGesturesDto
from arm_prosthesis.external_communication.models.dto.get_mio_patterns_dto import GetMioPatternsDto
from arm_prosthesis.external_communication.models.dto.get_settings_dto import GetSettingsDto
from arm_prosthesis.external_communication.models.dto.get_telemetry_dto import GetTelemetryDto
from arm_prosthesis.external_communication.models.dto.perform_gesture_by_id_dto import PerformGestureByIdDto
from arm_prosthesis.external_communication.models.dto.perform_gesture_by_raw_dto import PerformGestureRawDto
from arm_prosthesis.external_communication.models.dto.save_gesture_dto import SaveGestureDto
from arm_prosthesis.external_communication.models.dto.set_mio_patterns_dto import SetMioPatternsDto
from arm_prosthesis.external_communication.models.dto.set_positions_dto import SetPositionsDto
from arm_prosthesis.external_communication.models.dto.set_settings_dto import SetSettingsDto
from arm_prosthesis.external_communication.models.dto.start_telemetry_dto import StartTelemetryDto
from arm_prosthesis.external_communication.models.dto.update_last_time_sync_dto import UpdateLastTimeSyncDto
from arm_prosthesis.external_communication.models.request import Request
from arm_prosthesis.external_communication.models.response import Response
from arm_prosthesis.external_communication.services.dto_to_entity_converter import DtoToEntityConverter
from arm_prosthesis.services.mio_patterns_service import MioPatternsService
from arm_prosthesis.services.motor_driver_communication import ActuatorControllerService
from arm_prosthesis.services.myoelectronics_service import MyoelectronicsService
from arm_prosthesis.utils.stoppable_thread import StoppableThread
from arm_prosthesis.external_communication.services.telemetry_service import TelemetryService
from arm_prosthesis.hand_controller import HandController
from arm_prosthesis.models.positions import Positions
from arm_prosthesis.services.gesture_repository import GestureRepository
from arm_prosthesis.services.settings_dao import SettingsDao
from errors_pb2 import Error


class Communication:
    _logger = logging.getLogger('Main')
    _settings: GetSettingsDto
    _telemetry_thread: StoppableThread = None
    _mqtt_connector: MqttConnector = None
    _rfcc_connector: RFCCConnector = None
    _interval_100_hz_in_ms = 10
    _interval_2_days_in_ms = 172800000

    def __init__(self, hand_controller: HandController,
                 config: Config,
                 gesture_repository: GestureRepository,
                 telemetry_service: TelemetryService,
                 settings_dao: SettingsDao,
                 myoelectronics_service: MyoelectronicsService,
                 driver_communication: ActuatorControllerService,
```

```
            mio_patterns_service: MioPatternsService):
    self._gesture_repository = gesture_repository
    self._settings_dao = settings_dao
    self._hand_controller = hand_controller
    self._config = config
    self._telemetry_service = telemetry_service
    self._myoelectronics_service = myoelectronics_service
    self._driver_communication = driver_communication
    self._settings = self._settings_dao.get()
    self._mio_patterns_service = mio_patterns_service

    self._request_queue: 'Queue[Request]' = Queue()

    if self._config.mqtt_enabled:
        self._mqtt_connector = MqttConnector(self._config, self.request_queue)

    if self._config.rfcomm_enabled:
        self._rfcc_connector = RFCCConnector(self._config, self.request_queue)

    self._myoelectronics_service.pattern_observable.subscribe(
        lambda pattern: self._handle_recognized_pattern(pattern)
    )

    self._logger.info('Communication initialized')

def _handle_recognized_pattern(self, pattern):
    self._logger.info(f'Handle pattern: {pattern}')

    if pattern is None:
        return

    gesture_id = self._mio_patterns_service.get_gesture_id_by_pattern(pattern)
    self._logger.info(f'Gesture id for pattern: {gesture_id}')

    if gesture_id is not None:
        gesture = self._gesture_repository.get_gesture_by_id(gesture_id)

        if gesture is not None:
            self._hand_controller.perform_gesture(DtoToEntityConverter.convert_gesture_dto_to_gesture(gesture))
        else:
            self._logger.info(f'Gesture not found: {gesture_id}')

@property
def request_queue(self) -> 'Queue[Request]':
    return self._request_queue

def _send_telemetry(self):
    if (self._mqtt_connector and self._mqtt_connector.connected) \
            or (self._rfcc_connector and self._rfcc_connector.connected):

        telemetry = self._telemetry_service.get_telemetry()
        telemetry_response = Response(CommandType.Telemetry, telemetry.serialize())

        if self._mqtt_connector and self._mqtt_connector.connected:
            self._mqtt_connector.write_response(telemetry_response)

        if self._rfcc_connector and self._rfcc_connector.connected:
```

```
            self._rfcc_connector.write_response(telemetry_response)

    def run(self):
        self._logger.info('Communication running')

        if self._mqtt_connector:
            self._mqtt_connector.start()

        if self._rfcc_connector:
            self._rfcc_connector.start()

        while 1:
            self._logger.info('Wait new request')
            request = self._request_queue.get()
            self._logger.info('New request receive from the queue. Start handle request')
            self.handle_request(request)

    def handle_request(self, request: Request):
        try:
            logging.info(f'Request {request.command_type}')
            if request.command_type == CommandType.SetPositions:
                self.handle_set_positions_request(request.payload)
                request.response_writer.write_response(Response(CommandType.Ok, None))
                return

            if request.command_type == CommandType.SaveGesture:
                self.handle_save_gesture(request.payload)
                request.response_writer.write_response(Response(CommandType.Ok, None))
                return

            if request.command_type == CommandType.DeleteGesture:
                self.handle_delete_gesture(request.payload)
                request.response_writer.write_response(Response(CommandType.Ok, None))
                return

            if request.command_type == CommandType.PerformGestureId:
                self.handle_perform_gesture_by_id(request.payload)
                request.response_writer.write_response(Response(CommandType.Ok, None))
                return

            if request.command_type == CommandType.PerformGestureRaw:
                self.handle_perform_gesture_raw(request.payload)
                request.response_writer.write_response(Response(CommandType.Ok, None))
                return

            if request.command_type == CommandType.GetGestures:
                gestures_dto = self.handle_get_gesture()
                request.response_writer.write_response(Response(CommandType.GetGestures, gestures_dto.serialize()))
                return

            if request.command_type == CommandType.GetSettings:
                settings_dto = self.handle_get_settings()
                request.response_writer.write_response(Response(CommandType.GetSettings, settings_dto.serialize()))
                return

            if request.command_type == CommandType.SetSettings:
                self.handle_set_settings(request.payload)
```

```
            request.response_writer.write_response(Response(CommandType.Ok, None))
            return

        if request.command_type == CommandType.UpdateLastTimeSync:
            self.handle_update_last_time_sync(request.payload)
            request.response_writer.write_response(Response(CommandType.Ok, None))
            return

        if request.command_type == CommandType.GetTelemetry:
            get_telemetry_dto = self.handle_get_telemetry()
            request.response_writer.write_response(
                Response(CommandType.GetTelemetry, get_telemetry_dto.serialize()))
            return

        if request.command_type == CommandType.StartTelemetry:
            self.handle_start_telemetry(request.payload)
            request.response_writer.write_response(Response(CommandType.Ok, None))
            return

        if request.command_type == CommandType.StopTelemetry:
            self.handle_stop_telemetry()
            request.response_writer.write_response(Response(CommandType.Ok, None))
            return

        if request.command_type == CommandType.GetMioPatterns:
            get_mio_patterns_dto = self.handle_get_mio_patterns()
            request.response_writer.write_response(
                Response(CommandType.GetTelemetry, get_mio_patterns_dto.serialize()))
            return

        if request.command_type == CommandType.SetMioPatterns:
            self.handle_set_mio_patterns(request.payload)
            request.response_writer.write_response(Response(CommandType.Ok, None))
            return

        raise Exception(f'Command {request.command_type} not supporting')
    except:
        e = traceback.format_exc()
        logging.error(f'Error request handling: {e}')
        error = Error()
        error.message = e
        error_response = Response(CommandType.Error, error.SerializeToString())
        request.response_writer.write_response(error_response)

def handle_set_positions_request(self, payload: bytes):
    logging.info(f'Start handling set positions')
    set_position = SetPositionsDto()
    set_position.deserialize(payload)
    positions = Positions(set_position.little_finger_position, set_position.ring_finger_position,
                          set_position.middle_finger_position, set_position.index_finger_position,
                          set_position.thumb_finger_position)
    self._hand_controller.set_positions(positions)

def handle_save_gesture(self, payload: bytes):
    logging.info(f'Start handling save gesture')
    save_gesture_dto = SaveGestureDto()
    save_gesture_dto.deserialize(payload)
```

```
    self._gesture_repository.add_gesture(save_gesture_dto.time_sync, save_gesture_dto.gesture_dto)

def handle_delete_gesture(self, payload: bytes):
    logging.info(f'Start handling delete gesture')

    delete_gesture_dto = DeleteGestureDto()
    delete_gesture_dto.deserialize(payload)

    self._gesture_repository.remove_gesture(delete_gesture_dto.time_sync, delete_gesture_dto.id)

def handle_update_last_time_sync(self, payload: bytes):
    logging.info(f'Start handling update last time sync')

    update_last_time_sync_dto = UpdateLastTimeSyncDto()
    update_last_time_sync_dto.deserialize(payload)
    self._gesture_repository.update_time_sync(update_last_time_sync_dto.last_time_sync)

def handle_perform_gesture_by_id(self, payload: bytes):
    logging.info(f'Start handling perform gesture by id')

    perform_gesture_by_id_dto = PerformGestureByIdDto()
    perform_gesture_by_id_dto.deserialize(payload)

    gesture = self._gesture_repository.get_gesture_by_id(perform_gesture_by_id_dto.id)

    self._hand_controller.perform_gesture(DtoToEntityConverter.convert_gesture_dto_to_gesture(gesture))

def handle_perform_gesture_raw(self, payload: bytes):
    logging.info(f'Start handling perform gesture raw')

    perform_gesture_raw_dto = PerformGestureRawDto()
    perform_gesture_raw_dto.deserialize(payload)

    self._hand_controller.perform_gesture(
        DtoToEntityConverter.convert_gesture_dto_to_gesture(perform_gesture_raw_dto.gesture_dto))

def handle_get_gesture(self) -> GetGesturesDto:
    logging.info(f'Start handling get gesture')

    get_gestures_dto = GetGesturesDto()
    get_gestures_dto.last_time_sync = self._gesture_repository.last_time_sync
    get_gestures_dto.gestures_dto = self._gesture_repository.get_all_gestures()
    return get_gestures_dto

def handle_get_settings(self) -> GetSettingsDto:
    logging.info(f'Start handling get settings')

    current_settings = self._settings_dao.get()
    return current_settings

def handle_set_settings(self, payload: bytes):
    logging.info(f'Start handling set settings')

    settings_dto = SetSettingsDto()
    settings_dto.deserialize(payload)
```

```
    old_emg = self._settings.enable_emg

    self._settings_dao.save(settings_dto)

    if settings_dto.power_off:
        logging.info(f'Power off')
        os.system("sudo shutdown now -h")
        exit(0)

    self._settings = self._settings_dao.get()

    if self._settings.enable_emg != old_emg:
        if self._settings.enable_emg:
            self._myoelectronics_service.start()
        else:
            self._myoelectronics_service.stop()

def handle_start_telemetry(self, payload):
    logging.info(f'Start handling start telemetry')

    if self._telemetry_thread is not None:
        raise Exception('Telemetry already started')

    start_telemetry_dto = StartTelemetryDto()
    start_telemetry_dto.deserialize(payload)

    if start_telemetry_dto.interval_ms < self._interval_100_hz_in_ms \
            or start_telemetry_dto.interval_ms > self._interval_2_days_in_ms:
        raise Exception('Incorrect interval')

    interval_in_seconds = start_telemetry_dto.interval_ms / 1000

    self._driver_communication.enable_telemetry()
    self._telemetry_thread = StoppableThread(target=self._send_telemetry, timeout=interval_in_seconds)
    self._telemetry_thread.start()

def handle_get_telemetry(self) -> GetTelemetryDto:
    logging.info(f'Start handling get telemetry')

    get_telemetry_dto = GetTelemetryDto()
    telemetry = self._telemetry_service.get_telemetry()
    get_telemetry_dto.telemetry = telemetry
    return get_telemetry_dto

def handle_stop_telemetry(self):
    logging.info(f'Start handling stop telemetry')

    if self._telemetry_thread is None:
        raise Exception('Telemetry not started')

    self._driver_communication.disable_telemetry()
    self._telemetry_thread.stop()
    self._telemetry_thread = None

def handle_get_mio_patterns(self) -> GetMioPatternsDto:
    logging.info(f'Start handling get mio patterns')
```

```
        get_mio_patterns_dto = GetMioPatternsDto()
        mio_patterns_dto = self._mio_patterns_service.get_mio_patterns()
        for mio_pattern_dto in mio_patterns_dto:
            get_mio_patterns_dto.patterns_dto.append(mio_pattern_dto)

        return get_mio_patterns_dto

    def handle_set_mio_patterns(self, payload):
        logging.info(f'Start handling set mio patterns')

        set_mio_patterns_dto = SetMioPatternsDto()
        set_mio_patterns_dto.deserialize(payload)

        self._mio_patterns_service.update_mio_patterns(set_mio_patterns_dto.patterns_dto)
```

{**Файл приложения для парсинга формата протокола**}

```
from enum import Enum

from datetime import datetime
import crc8

from arm_prosthesis.external_communication.core.connectors.irequest_writer import IPackageReceiver
from arm_prosthesis.external_communication.core.connectors.package_dto import PackageDto
from arm_prosthesis.external_communication.models.command_type import CommandType


class ProtocolState(Enum):
    SFD = 1
    TYPE = 2
    SIZE = 3
    PAYLOAD = 4
    CRC8 = 5


class ProtocolParser:
    _sfd = b'\xfd\xba\xdc\x01\x50\xb4\x11\xff'

    _state: ProtocolState
    _current_package: PackageDto
    _buffer: bytearray
    _last_receive_timestamp: float

    def __init__(self, package_receiver: IPackageReceiver):
        self._state = ProtocolState.SFD
        self._payload_size = 0
        self._buffer = bytearray()
        self._crc_calculator = crc8.crc8()
        self.package_receiver = package_receiver

    @property
    def current_request(self) -> PackageDto:
        return self._current_package

    @property
    def state(self) -> ProtocolState:
        return self._state
```

```
def update(self, data: bytes):
    # Если таймаут приема истек, то сбрасываем буфер и начинаем прием с нуля
    receiver_time = datetime.now().timestamp()
    if self._state != ProtocolState.SFD and receiver_time - self._last_receive_timestamp > 5:
        self._buffer.clear()
        self._state = ProtocolState.SFD
        self._crc_calculator = crc8.crc8()

    for byte in data:
        self._buffer.append(byte)

        if self._state is not ProtocolState.SFD and self._state is not ProtocolState.CRC8:
            self._crc_calculator.update(byte.to_bytes(1, 'little'))

        self._update_states()

    self._last_receive_timestamp = receiver_time

def _update_states(self):
    if self.state == ProtocolState.SFD:
        if len(self._buffer) == 8:
            if self._buffer == self._sfd:
                self._current_package = PackageDto()
                self._state = ProtocolState.TYPE
            else:
                self._buffer.pop(0)
    else:
        if self.state == ProtocolState.TYPE:
            self._current_package.command_type = CommandType(self._buffer[-1])
            self._state = ProtocolState.SIZE
        else:
            if self.state == ProtocolState.SIZE:
                if len(self._buffer) == 11:
                    self._current_package.payload_size = (self._buffer[-1] << 8) | self._buffer[-2]

                    if self._current_package.payload_size == 0:
                        self._state = ProtocolState.CRC8
                    else:
                        self._state = ProtocolState.PAYLOAD
            else:
                if self.state == ProtocolState.PAYLOAD:
                    if self._current_package.payload_size + 11 == len(self._buffer):
                        self._current_package.payload = bytes(self._buffer[11:])
                        self._state = ProtocolState.CRC8
                else:
                    if self.state == ProtocolState.CRC8:
                        self._current_package.received_crc8 = self._buffer[-1].to_bytes(1, 'little')
                        self._current_package.real_crc8 = self._crc_calculator.digest()
                        self._crc_calculator = crc8.crc8()

                        self.package_receiver.receive_package(self._current_package)

                        self._buffer.clear()
                        self._state = ProtocolState.SFD
                    else:
                        raise Exception('Invalid protocol state')
```

```
    @staticmethod
    def create_package(command_type: CommandType, payload: bytes) -> PackageDto:
        package = PackageDto()

        package.command_type = command_type

        if payload is None:
            package.payload_size = 0
        else:
            package.payload_size = len(payload)
            package.payload = payload

        return package

    def serialize_package(self, package: PackageDto):
        ser_package = bytearray()
        package_crc_calculator = crc8.crc8()

        ser_package.extend(self._sfd)
        ser_package.append(package.command_type.value)
        ser_package.extend(package.payload_size.to_bytes(2, 'little'))

        if package.payload_size != 0:
            ser_package.extend(package.payload)

        package_crc_calculator.update(ser_package[8:])
        crc = package_crc_calculator.digest()
        ser_package.extend(crc)

        return ser_package
```

{**Файл приложения для обеспечения соединения по RFCC протоколу (Bluetooth master)**}

```
import logging
import threading
import time
from queue import Queue

from bluedot.btcomm import BluetoothServer

from arm_prosthesis.config.configuration import Config
from arm_prosthesis.external_communication.core.connectors.irequest_writer import IPackageReceiver
from arm_prosthesis.external_communication.core.connectors.iresponse_writer import IResponseWriter
from arm_prosthesis.external_communication.core.connectors.package_dto import PackageDto
from arm_prosthesis.external_communication.core.protocol_parser import ProtocolParser
from arm_prosthesis.external_communication.models.command_type import CommandType
from arm_prosthesis.external_communication.models.request import Request
from arm_prosthesis.external_communication.models.response import Response


class RFCCConnector(threading.Thread, IResponseWriter, IPackageReceiver):
    _logger = logging.getLogger('Main')
    _bluetooth_server: BluetoothServer = None
    _response_mutex = threading.Lock()
    _count_connected: int
```

```
def __init__(self, config: Config, request_transmitter: 'Queue[Request]'):
    threading.Thread.__init__(self)
    self._config = config

    if not config.rfcomm_enabled:
        self._logger.fatal('RFCC is disabled but is trying to create')
        raise Exception('RFCC is disabled but is trying to create')

    self._count_connected = 0
    self._request_transmitter = request_transmitter
    self._protocol_parser = ProtocolParser(self)

@property
def connected(self) -> bool:
    if self._bluetooth_server is not None:
        return self._count_connected > 0
    else:
        return False

def run(self):
    self._logger.info('RFCC running start')
    self._bluetooth_server = BluetoothServer(data_received_callback=self._data_received_handler,
                                             auto_start=False, power_up_device=True, encoding=None,
                                             when_client_connects=self._client_connect_handler,
                                             when_client_disconnects=self._client_disconnect_handler)
    self._logger.info('RFCC server created')

    while True:
        self._logger.info('RFCC server try to start')
        try:
            self._bluetooth_server.start()
        except OSError as e:
            self._logger.info(f'RFCC server start error: {e}')
            time.sleep(30)
            continue
        except Exception as e:
            self._logger.exception(e)
            raise e

        self._logger.info('RFCC started')
        break

def _client_connect_handler(self):
    self._logger.info('New device connected')
    self._count_connected = self._count_connected + 1

def _client_disconnect_handler(self):
    self._logger.info('Device disconnected')
    self._count_connected = self._count_connected - 1

    if self._count_connected < 0:
        self._count_connected = 0

def _data_received_handler(self, data):
    self._logger.debug(f'RFCC receive {len(data)} bytes')
    self._protocol_parser.update(data)
```

```
    def write_response(self, response: Response):
        self._response_mutex.acquire()

        payload_length = 0
        if response.payload is not None:
            payload_length = {len(response.payload)}

        if response.command_type is not CommandType.Telemetry:
            self._logger.info(
                f'RFCC try to send response with type {response.command_type} and payload length {payload_length}')
        package = self._protocol_parser.create_package(response.command_type, response.payload)
        self.send(self._protocol_parser.serialize_package(package))

        self._response_mutex.release()

    def receive_package(self, package: PackageDto):
        self._logger.info(f'RFCC receive new package {package.command_type} with size {package.payload_size} bytes')
        new_request = Request(package.command_type, package.payload, self)
        self._request_transmitter.put(new_request)

    def send(self, payload: bytes):
        if self._bluetooth_server is None:
            self._logger.critical('RFCC not running, but send invoke')
            raise ConnectionError('RFCC not running, but send invoke')

        self._bluetooth_server.send(payload)
```

{**Файл protocol buffer структур приложения**}

```
syntax="proto3";
package handcontrol;

enum ModuleStatusType {
  MODULE_STATUS_INITIALIZATION = 0;
  MODULE_STATUS_WORK = 1;
  MODULE_STATUS_ERROR = 2;
  MODULE_STATUS_CONNECTION_ERROR = 3;
  MODULE_STATUS_DISABLED = 4;
}

enum DriverStatusType {
  DRIVER_STATUS_INITIALIZATION = 0;
  DRIVER_STATUS_ERROR = 1;
  DRIVER_STATUS_CONNECTION_ERROR = 2;
  DRIVER_STATUS_DISABLED = 3;
  DRIVER_STATUS_SLEEP = 4;
  DRIVER_STATUS_SETTING_POSITION = 5;
}

message Error {
  string message = 1;
}

message GetGestures {
  int64 last_time_sync = 1;
  repeated Gesture gestures = 2;
}
```

```
message SaveGesture {
  int64 time_sync = 1;
  Gesture gesture = 2;
}

message DeleteGesture {
  int64 time_sync = 1;
  uuid.UUID id = 2;
}

message PerformGestureById {
  uuid.UUID id = 1;
}

message PerformGestureRaw {
  Gesture gesture = 1;
}

message SetPositions {
  int32 pointer_finger_position = 1;
  int32 middle_finger_position = 2;
  int32 ring_finger_position = 3;
  int32 little_finger_position = 4;
  int32 thumb_finger_position = 5;
}

message Gesture {
  uuid.UUID id = 1;
  string name = 2;
  int64 last_time_sync = 3;
  bool iterable = 4;
  int32 repetitions = 5;
  repeated GestureAction actions = 6;
}

message GestureAction {
  int32 pointer_finger_position = 1;
  int32 middle_finger_position = 2;
  int32 ring_finger_position = 3;
  int32 little_finger_position = 4;
  int32 thumb_finger_position = 5;
  int32 delay = 6;
}

message UpdateLastTimeSync {
  int64 last_time_sync = 1;
}

message MioPattern {
  int64 pattern = 1;
  uuid.UUID gesture_id = 2;
}

message GetMioPatterns {
  repeated MioPattern patterns = 1;
}
```

```
message SetMioPatterns {
  repeated MioPattern patterns = 1;
}

message GetSettings {
  bool enable_emg = 1;
  bool enable_display = 2;
  bool enable_gyro = 3;
  bool enable_driver = 4;
}

message SetSettings {
  bool enable_emg = 1;
  bool enable_display = 2;
  bool enable_gyro = 3;
  bool enable_driver = 4;
  bool power_off = 5;
}

message Telemetry {
  enums.ModuleStatusType emg_status = 1;
  enums.ModuleStatusType display_status = 2;
  enums.ModuleStatusType gyro_status = 3;
  enums.DriverStatusType driver_status = 4;
  int64 last_time_sync = 5;
  int32 emg = 6;
  uuid.UUID executable_gesture = 7;
  int32 power = 8;
  int32 pointer_finger_position = 9;
  int32 middle_finger_position = 10;
  int32 ring_finger_position = 11;
  int32 little_finger_position = 12;
  int32 thumb_finger_position = 13;
}

message GetTelemetry {
  Telemetry telemetry = 1;
}

message StartTelemetry {
  int32 interval_ms = 1;
}

message UUID {
  string value = 1;
}
```