**TODO: Не забудь изменить стиль «названия рисунка», изменить сейчас нельзя, потому что Word – инвалид**

**Введение**

Развлечения всегда были неотъемлемой частью жизни человека. По этой причине в данной работе я хотел бы разобрать одно из самых передовых и быстро развивающихся технологий в области развлечений – «лазертаг», а именно программное обеспечение для клиентской части.

**1 Анализ предметной области**

В конце 70-х годов была представлена одна из первых версий взаимодействия устройств по ИК сигналу. Для обмена данными был придуман устоявшийся протокол передачи – «Miles tag», который дошел до современного вида лазертага. По этому протоколу построены все правила передачи информации между устройствами. Устройствами являются муляжи оружий, повязки на голову и дополнительные устройства для разнообразия сценариев игры. Муляж оружия – «тагер», умеет излучать ИК сигнал для имитации выстрела, является slave устройством для повязки. Повязка на голову или жилет имеют несколько модулей для приёма ИК сигнала, которые оснащены световыми индикаторами, является Master устройством для тагера. Радиобаза – устройство для передачи информацией между игроками и компьютером со специальным программным обеспечением. Программное обеспечение является способом взаимодействия администратора лазертаг клуба с оборудованием для настройки игровых параметров и ведения статистики игры.

Целью данной работы является разработка вышеупомянутого программного обеспечения, для настройки всех типов устройств специального лазертаг оборудования, учёта статистики, автоматизации игровых событий и взаимодействия с сервером для сохранения характеристик игроков. Существует множество способов подключения радиобазы для дальнейшей передачи информации, но нужно реализовать 2 – соединение с радиобазой по COMPORT и Bluetooth, для компьютеров и телефонов соответственно. Протокол команд для соединения и передачи данных идентичные в обоих случаях, отличаются лишь физические способы отправки.

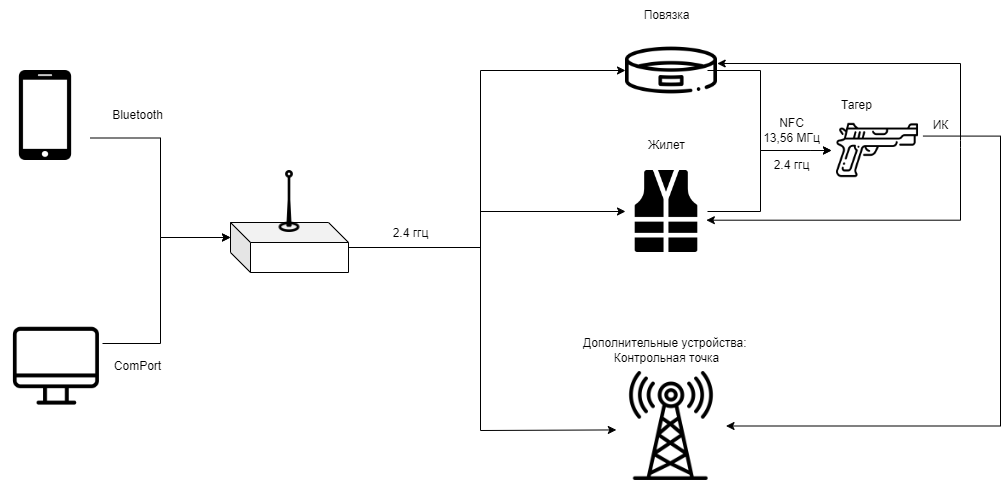


Рисунок 1 - Абстрактное представление работы оборудования

На рисунке 1, представлена абстрактная схема взаимодействия всех устройств. Радиобаза пересылает данные на частоте 2.4 ГГц для Master устройств. Повязки и жилеты связываются с тагерами на частоте 13,56 МГц, после чего начинают общаться между собой на частоте 2.4 ГГц. Тагер излучает инфракрасный луч, передавая по нему данные о выстреле для других Master устройств.

В данной работе нужно было реализовать обмен данными только на стороне программного обеспечения.

**1.1 Протокол обмена данными между устройствами и программным обеспечением**

Для полноценной работы оборудования должно быть реализовано 3 протокола обмена данными:

1. Для обмена между радиобазой и программным обеспечением, по COMPORT и Bluetooth
2. Для обмена между радиобазой и устройствами на частоте 2.4 ГГц
3. Для обмена между устройствами и тагерами на частоте 2.4 ГГц и 13.56 МГц
4. Для передачи данных по ИК-порту от тагеров на устройства

В случае с оборудованием, представленным для данной работы, радиобаза является ретранслятором, поэтому 1 и 2 протоколы обмена были объединены в один.

Протокол обмена данными между ПО и устройствами был разделен на группы команд:

1. Команды для базы:

|  |  |  |
| --- | --- | --- |
| Поиск базы | 0xFF | - |
| Группа команд для базы | 0xFE | … |
| Данные о базе (UID, ревизия, версия) | 0xFE | 0x01 |
| Выключить поиск новых игроков | 0xFE | 0x10 |
| Включить поиск игроков | 0xFE | 0x11 |
| Выключить обратную связь | 0xFE | 0x12 |
| Включить обратную связь | 0xFE | 0x13 |

Таблица 1 - Команды для радиобазы

1. Команды на базу для общей рассылки на устройства:

|  |  |
| --- | --- |
| 0х02 | Пауза (не воспринимаем выстрелы, не могут подвязать тагеры, и перевожу свои тагеры в состояние не активен, с сохранением жизни и здоровья) |
| 0х03 | Стоп раунда (не воспринимаем выстрелы, не могу подвязать тагеры, перевожу свои тагеры в состояние не активен, запись стоп игра, стоп сбора статистики) |
| 0х04 | Конец игры (не воспринимаем выстрелы, не могу подвязать тагеры, перевожу свои тагеры в состояние не активен, жизнь и здоровье в ноль, запись конец игры, стоп сбора статистики) Переход через 30с в настройки |
| 0х05 | Разблокировать игрока (принимаю выстрел и активен тагер) |
| 0х06 | Старт раунда (принимаем выстрелы, активируем тагеры, начать сбор статистики, передавать статистику, оживить и восстановить патроны) |

Таблица 2 - Команды для общей рассылки

1. Команды на базу для индивидуальной посылки на устройство. Такие команды обновляются, добавляются и чистятся с выходом новых версий устройств. Также они делятся еще на несколько групп
   1. Команды на главные устройства (повязки, жилеты)
   2. Команды на дополнительные устройства (контрольные точки)
   3. Команды на привязываемые устройства (тагеры)

Пример таких команд соответственно каждой группе:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Номер | UID главного устройства | UID привязываемого устройства | Описание | Размер значения | Диапазон значений |
| 0x35 | UID повязки | - | Здоровье | 2 байта | 1-999 |
| 0xB0 | UID повязки | UID тагера | Урон | 1 байт | 0-15 |
| 0x41 | UID контрольной точки | - | Сценарий | 1 байт | 0-3 |

Таблица 3 - Пример индивидуальных команд

1. Команды с базы – возможные ответы, которые база может прислать на разные команды:

|  |  |  |
| --- | --- | --- |
| Номер | Размер | Описание ответа |
| 0x80 | 1 байт | OK. Ответ на посылки, не требующие ответа |
| 0x81 | 4 байта | UID не подключен к базе |
| 0x82 | 21 байт | Конфиг повязки |
| 0x83 | #### | UID повязки. (0xFF Конец) Ответ на команду 0х19 |
| 0x84 | #### | UID тагеров. (0xFF Конец) Одним пакетом. Ответ на команду 0х29 |
| 0x90 | 1 байт | Таймаут. Не получил ответ на индивидуальную посылку от повязки. |
| 0x91 | 1 байт | Значение в команде не верное |
| 0x92 | 1 байт | Команда не распознана |

Таблица 4 - Ответы с радиобазы

Важно учитывать, что, по мере роста функционала оборудования, будут добавляться новые команды, поэтому следовало реализовать архитектуру обмена данными в программном обеспечении так, чтобы при изменении протокола взаимодействия с радиобазой не требовалось добавлять новые методы для каждой нововведённой команды. Отправка и приём команд должны быть гибкими и универсальными.

**1.2 Протокол обмена данными между программным обеспечением и сервером**

Сервер является неотъемлемой частью для продвинутых лазертаг клубов. С помощью сервера, игроки могут отслеживать свои достижения из любой точки, анализировать свою игру и игру соперников. Клубы могут соревноваться между собой в интенсивности проводимых игр и привлекать новых игроков. Всё это возможно благодаря возможности взаимодействия программного обеспечения с сервером. Протокол обмена данными между сервером и ПО строится на:

1. HTTP метод POST, самый распространённый способ отправки и приёма данных. Он может отправлять любые типы данных: текст, файлы, изображения. Вся нужная информация отправляется в теле HTTP-запроса, что делает POST надежным и безопасным методом передачи данных.
2. JSON формат – текстовый формат обмена данными, состоящий из объектов, которые представляют собой пары ключ-значение, где ключ – строка, а значением может быть любой тип данных, даже другой объект

Любой POST запрос на сервер лазертага включает в себя объект JSON. На любой запрос возвращается ответ в JSON с обязательным полем status, которое может принимать значения ok и error.

Все запросы на сервер:

1. Получение игроков и команд клуба:

|  |  |
| --- | --- |
| act | 10 |
| login | логин клуба |
| hash | hash пароля |
| v | версия api |

1. Добавление нового клуба:

|  |  |
| --- | --- |
| act | 11 |
| login | уникальный логин клуба |
| hash | хеш пароля |
| name | название клуба |
| email | e-mail |
| contacts | телефон, другая контактная информация |
| city | название города |
| v | версия api |

1. Добавление команды

|  |  |
| --- | --- |
| act | 12 |
| idclub | идентификатор клуба |
| login | логин клуба |
| hash | хеш пароля клуба |
| name | название команды |
| dopinfo | комментарий |
| v | версия api |

1. Добавление игрока

|  |  |
| --- | --- |
| act | 13 |
| idclub | идентификатор клуба |
| login | логин клуба |
| hash | хеш пароля клуба |
| name | имя игрока |
| dopinfo | комментарий |
| v | версия api |

1. Редактирование игрока:

|  |  |
| --- | --- |
| act | 15 |
| idclub | идентификатор клуба |
| login | логин клуба |
| hash | хеш пароля клуба |
| idplayer | идентификатор редактируемого игрока |
| name | имя игрока |
| fio | ФИО игрока |
| city | название города игрока |
| email | задать email (не влияет на email-логин пользователя) |
| phone | задать телефон (не влияет на контакты, указанные пользователем) - строка |
| bdate | дата рождения пользователя (строка в формате *04.04.1990*) |
| sex | пол (0 - мужской, 1 - женский) |
| dopinfo | комментарий |
| currteam | текущая команда |
| status | статус |
| v | версия api |

1. Редактирование команды:

|  |  |
| --- | --- |
| act | 16 |
| idclub | идентификатор клуба |
| login | логин клуба |
| hash | хеш пароля клуба |
| idteam | идентификатор редактируемой команды |
| name | название команды |
| dopinfo | комментарий |
| v | версия api |

1. Удаление игрока:

|  |  |
| --- | --- |
| act | 17 |
| idclub | идентификатор клуба |
| login | логин клуба |
| hash | хеш пароля клуба |
| idplayer | идентификатор удаляемого игрока |
| v | версия api |

1. Удаление команды:

|  |  |
| --- | --- |
| act | 18 |
| idclub | идентификатор клуба |
| login | логин клуба |
| hash | хеш пароля клуба |
| idteam | идентификатор удаляемой команды |
| v | версия api |

1. Отправка игры на сервер:

Структура json v1

**version** - версия протокола отправки игры (int);

**login** - логин клуба (string)

**hash** - хэш пароля клуба (string)

**hash\_game** - md5 для защиты от повторной загрузки одной и той же игры (string)

**is\_test** - если 1, то запись в БД не происходит, но выводится тестовая информация

**game\_info** - информация об игре

**name** - название игры (string)

**poligon** - название места проведения игры, если есть (string)

**date\_start\_game** - дата и время начала игры в unixtime по utc (int)

**duration\_of\_the\_game** - продолжительность игры в секундах (int)

**winning\_team\_ID** - игровой идентификатор команды-победителя (int)

**winning\_player\_ID** - игровой идентификатор игрока-победителя (int)

**teams** - список команд (array)

**players** - список игроков (array)

**events** - игровые события (попадания, убийства и т.п.) (array)

**1.4 Создание и анализ технического задания**

Исходяиз анализ предметной области и вышеописанных протоколов можно составить техническое задание. Оно звучало следующим образом:

Необходимо разработать программное обеспечение, которое должно работать на компьютере и смартфоне. Включает в себя конфигураторы всех устройств, отправка данных на которые, зависит от платформы. Должно быть реализовано взаимодействие с клубом на сервере и отправка игры.

**1.4 Сравнительный анализ аналогичных разработок**

Исходя из того, что нужно реализовать программное обеспечение для нескольких платформ, нужно было разработать 2 похожих по функционалу и дизайну приложения.

**3 Разработка**

В данной главе рассмотрена разработка всех классов и методов для построения основного функционала приложения.

**3.1 Создание проекта и сцены под Desktop**

Меню конфигуратора включает в себя множество параметров, которые были утверждены и расставлены в дизайне приложения. Под все основные меню выделены объекты, включающие в себя компонент «CanvasGroup». Данное решение было принято за счёт удобства взаимодействия с этим компонентом. Встроенный класс был расширен двумя методами, отвечающими за прозрачность, с помощью которых реализовано открытие и закрытие основных меню.

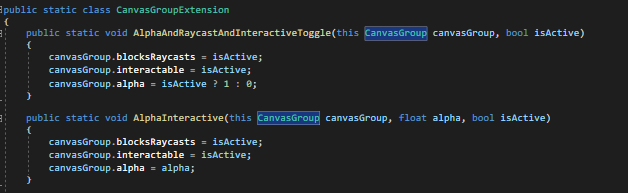


Рисунок 2 - Методы расширения CanvasGroup

Все элементы конфигуратора были и разделены на «настройки повязки», «настройки тагера», а они, разбиты на «основные настройки» и «дополнительные настройки», согласно дизайну.



Рисунок 3 - Пример расположения элементов конфигуратора на сцене

Каждый параметр – объект с наследником класса «MultyTypeIntParameter», который содержит в себе встроенный InputField, нижний и верхний порог значения и событие изменения параметра. Каждый наследник переопределяет метод разбора внесенного пользователем значения. Например, в поле ввода «здоровье», пользователь может занести только целое число не меньше 1 и не больше 999.

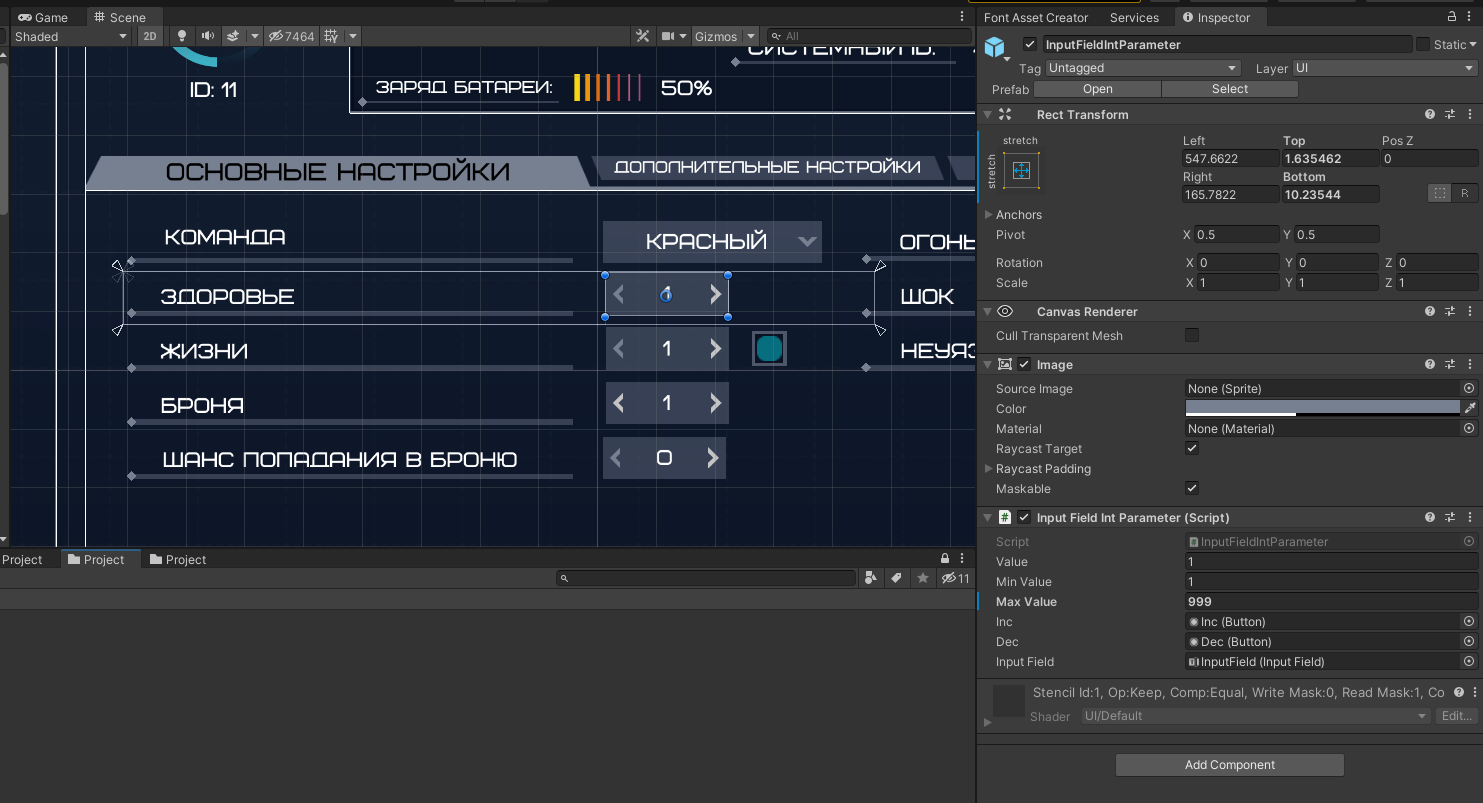


Рисунок 4 - Параметр конфигуратора

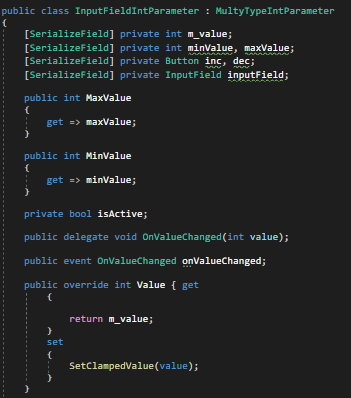


Рисунок 5 – класс наследник MultyTypeIntParameter

Классы конфигуратора, как и UI, разделены на «конфигуратор повязки» и «конфигуратор тагера» - «HeadBandConfiguration», «TaggerConfiguration» соответственно. Каждый содержит в себе соответствующие сериализованные объекты параметров, методы инициализации девайса и добавление слушателей на события изменения параметров. **Всё это можно расписать побольше если нужно будет увеличить объем**

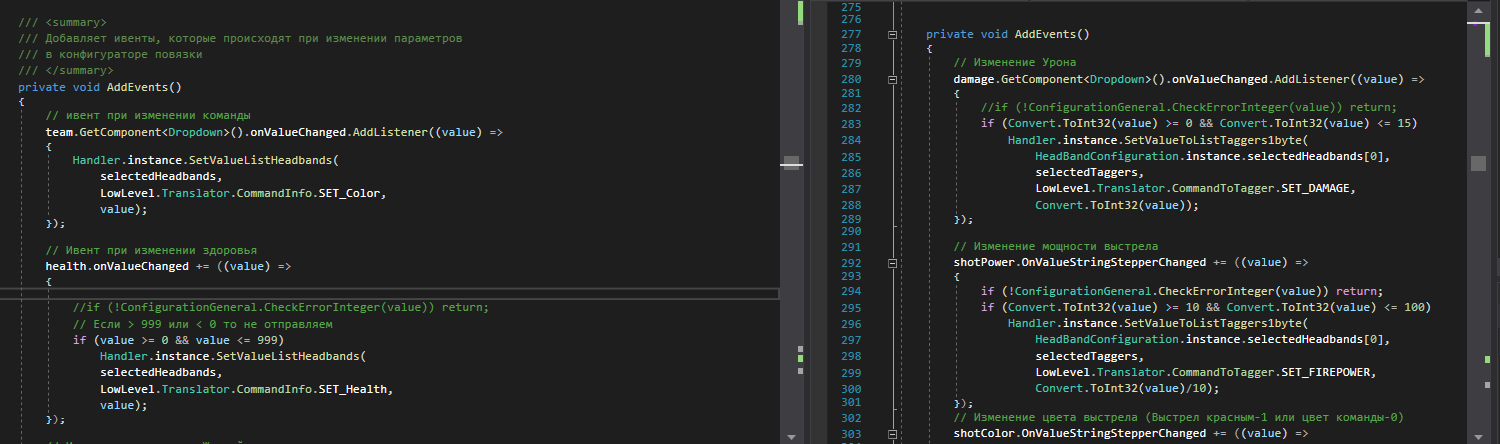


Рисунок 6 - Обработка событий изменения параметров

Каждое изменение параметра обращается к классу одиночке Handler, через который передаются все команды на COMPORT. Он содержит в себе объект реализации интерфейса ICommander, через который идёт основной обмен данными.

**3.3 Разработка методов обмена данными**

Первое, что требовалось реализовать – многопоточность. Для разработки многопоточных приложений на Unity следует использовать сторонние библиотеки или усложнённый функционал, потому что движок запрещает вызывать почти все основные вызовы не из основного потока. Для выполнения поставленной задачи был взят готовый скрипт «UnityThread», который отслеживает возвращение из стороннего потока и синхронизируется с основным во встроенных методах – Update/FixedUpdate. При старте приложения требуется инициализировать UnityThread, после чего создаётся DontDestroyOnLoad объект, который будет производить синхронизацию ответов из других потоков. В Handler инициализируется объект типа «Intaracton», который также, как и Commader реализует все методы интерфейса ICommander. Всё это сделано для того, чтобы объект типа Interaction был мостом для вызова методов Commander, которые должны выполняться уже не из главного потока. Пример работы метода отправки данных на тагер через новый поток:

1. TaggerConfigurator обращается к Handler, чтобы отправить введенные пользователем данные:

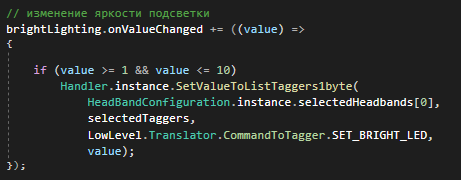


Рисунок 7 - обращение к Handler

1. Handler вызывает метод «SetToTagger» у объекта commander, который имеет тип Interaction:

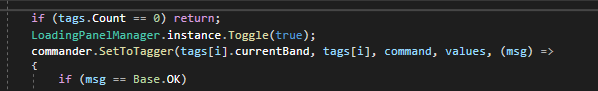


Рисунок 8 - вызов метода в Commander

1. Класс Interaction реализовывает метод интерфейса ICommander так, чтобы создался новый поток и объект типа Commander исполнялся уже не из основного потока, а Action callback вернулся обратно в Interaction для дальнейшей синхронизации потоков в Update через UnityThread:

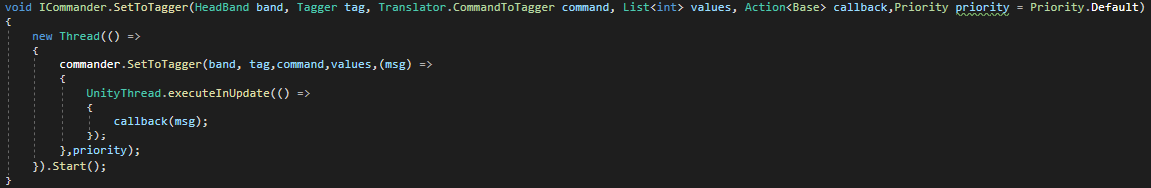


Рисунок 9 - Создание нового потока

1. После исполнения операции через обратный вызов обрабатывается результат выполнения команды:



Рисунок 10 - Приём ответа из метода в Commander

Автоматизация фонового взаимодействия программного обеспечения с радиобазой создала довольно весомое количество отправки и получения не важных данных, из-за чего могла возникнуть проблема переполнения буфера. Для избежания возможных ошибок, было принято разделить методы обмена на приоритеты. Например, запрос на обнаружение новых устройств, который вызывается раз в несколько секунд менее важен, чем запрос на изменения какой-либо конфигурации, которую применил пользователь. Было принято решение о добавление 3 приоритетов:

|  |  |
| --- | --- |
| Приоритет | Описание |
| Low | Низкий приоритет – выполняется только в том случае, если очередь и команд на выполнение пустая. Если на момент вызова метода с низким приоритетом очередь будет занята хотя-бы одним элементом на ожидание, то данный метод не будет выполняться и занимать очередь. |
| Default | Обычный приоритет – выполнится, как только очередь методов с обычным приоритетом дойдёт до него, а очередь с методами высокого приоритета будет пуста. |
| Hight | Высокий приоритет – метод займет очередь для высоких приоритетов и выполнится либо сразу, либо после вызванных ранее методов с таким-же приоритетом. |

Таблица 5 - Приоритеты исполнения команд

Для управления очередями были объявлены переменные, которые хранят в себе индексы методов на исполнение в очереди.



Рисунок 11 - Переменные управления очередями

HereWeGoAgain – Флаг, который следит за тем, не занят ли COMPORT передачей данных.

lastQuery – Индекс последнего в очереди на выполнение метода

lastQueryHightPriority – Индекс последнего в высокоприоритетной очереди на выполнение метода

currentQuery – Индекс метода в очереди, который на данный момент выполняется.

currentQueryHightPriority – Индекс метода в высокоприоритетной очереди, который на данный момент выполняется.

COUNT\_ERROR\_WAIT\_QUEUE – Переменная хранящая в себе время до ошибки. Если на протяжении этого времени, умноженного на 10, индексы очереди не изменят значения (очередь не сдвинется), то, возможно, какой-либо метод не смог закончить своё выполнение и не вернул значение. Переменная была создана для исключения необходимости перезапуска программного обеспечения в случае ошибки в коде.

Метод занимания очереди принимает в себя 1 аргумент – приоритет, от которого алгоритм изменяет определенные индексы. По умолчанию приоритет «Default»



Рисунок 12 - Аргумент метода ожидания очереди

Также WaitQueue возвращает ответ типа bool.

True – Можно начинать выполнение команды.

False – Следует отменить выполнение команды.

False вернется только в том случае, если будет низкий приоритет на выполнение.

Если приоритет «Default» - то WaitQueue вернёт значение на True, только тогда, когда очередь высокоприоритетных команд будет пуста и индекс очереди обычных команд дойдёт до нужной.

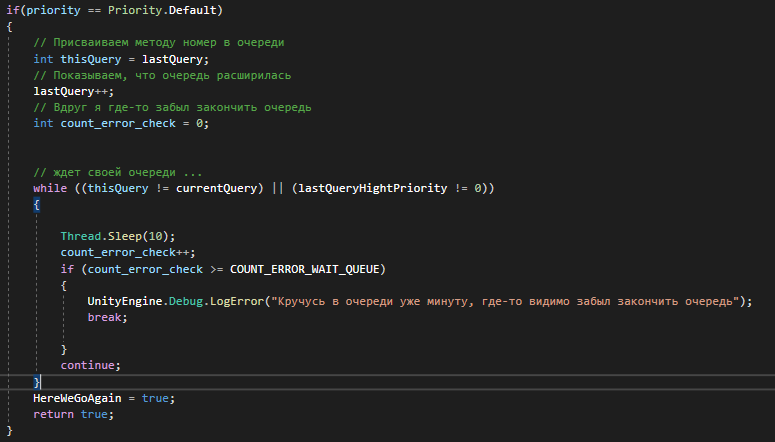


Рисунок 13 - Алгоритм ожидания очереди для команд с обычным приоритетом

Если приоритет «Low» - то WaitQueue вернёт значение True, только если ни одна из очередей не будет занята более приоритетными задачами на выполнение, иначе вернётся False и метод вызывавший WaitQueue должен будет прекратить выполнение.

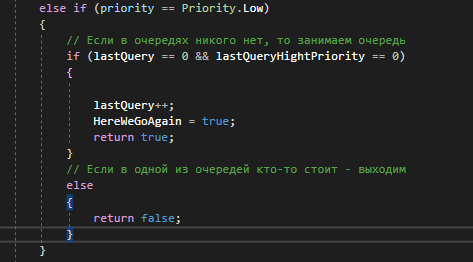


Рисунок 14 - Алгоритм ожидания очереди для команд с низких приоритетом

Если приоритет «Hight» - то WaitQueue вернёт значение на True, только тогда, когда индекс очереди высокоприоритетных команд дойдёт до нужной.

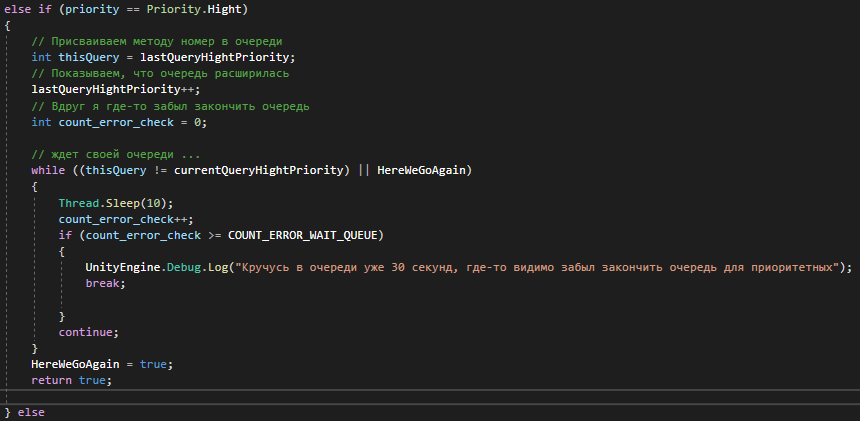


Рисунок 15 - Алгоритм ожидания очереди для команд с высоким приоритетом

Передача и приём данных был реализован в классе Translator. Данный класс содержит в себе методы, которые взаимодействуют с классом из метаданных «SerialPort», через который и реализуется отправка и приём данных по физическому COMPORT’у. В Commander инициализирован объект типа Translator. Несколько основных методов, реализованных в классе Translator:

«Opros\_to» конвертирует все данные на оправку, переданные в аргументах, в тип byte, собирает их в массив и добавляет в буфер на отправку, через встроенный метод Write у класса «SerialPort», заранее очистив буфер. Метод «Opros\_to» переопределён под разные задачи:



Рисунок 16 - Методы для отправки данных

«Opros\_From» принимает данные из COMPORT.

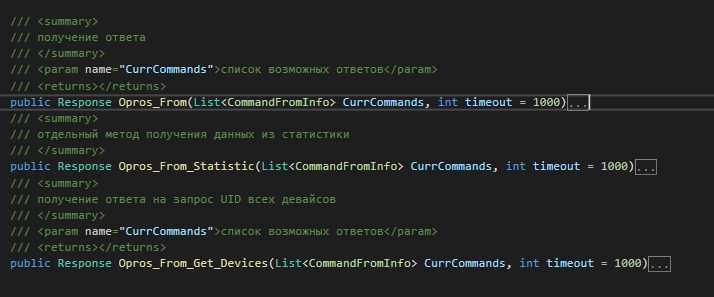


Рисунок 17 - Методы для приёма данных

Данные с COMPORT приходят через разный интервал времени после отправки, поэтому был реализован цикл, в котором ведется опрос на наличие байт в буфере приёма.

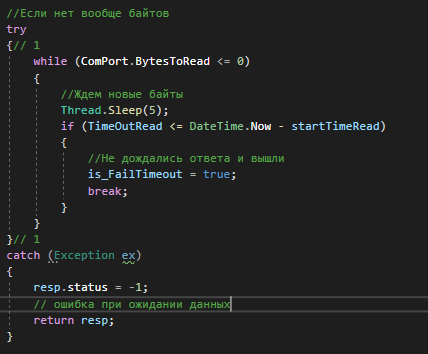


Рисунок 18 - цикл проверки присутствия данных в буфере приёма

Ответ на переданные данные возвращается через Callback, где синхронизируется с основным потоком программы, после чего результат обрабатывается