

Глава 7. GIT UART

Ежелев Г. И.

17 декабря 2025 г.

Структура проекта

В нашем проекте уже появилось несколько файлов с совершенно разным функционалом - это и управление тактированием и временем, и управление **GPIO** - General Purpose Input Output, и ADC, а скоро еще и **UART** - это цифровой протокол передачи данных.

Настала пора навести порядок в проекте. Начнем с файловой структуры проекта, откройте проводник, чтобы убедиться что у вас так же:

- ▶ **Корень** - в нем лежит файл *.uvprojx - файл-проект.
- ▶ Папка **src**(source - источник/исходник) - в ней лежит весь разработанный вами код и библиотеки.
- ▶ Папка **Objects** - все скомпилированные программные компоненты. Каждый файл - отдельный объект, созданный компилятором. Далее их будет собирать воедино **Linker** от слова Link - связь/ссылка.

Возникает довольно логичный вопрос - как именно **распределять файлы и папки**. Вспомните любой свой проект - в какой-то момент он **становится таким громоздким**, что в нем очень трудно разобраться вам самим. В таком случае **жесткая формальная структура** приходит на помощь:

- ▶ **Layer-first** - подход, в котором все файлы распределяются по папкам в зависимости от их **“слоя в проекте”**. Нам важно не то, какие функции используют этот файл, а то чем он является сам - все **интерфейсы в папке interfaces**, вся **обработка датчиков в sensors** и т.д..
- ▶ **Feature-first** - здесь все файлы, ответственные за одну **“фичу”** группируются в одну папку:
`camera/{uart.cpp, camera.cpp, gpio.cpp}`
- ▶ В нашем случае лучше использовать именно первый вариант, т.к. мы делаем универсальный проект.

- ▶ Очевидно, что **каждая пара *.cpp и *.h лежат в отдельной папке** с таким же названием.
- ▶ В каждом файле должен быть `#include "project_config.h"`
- ▶ Каждый файл должен быть **добавлен в группу** (папки в левой части экрана)
- ▶ Каждая папка должна быть указана в options for target(значок волш палочки) → C/C++ → include paths
И все подпапки!!!

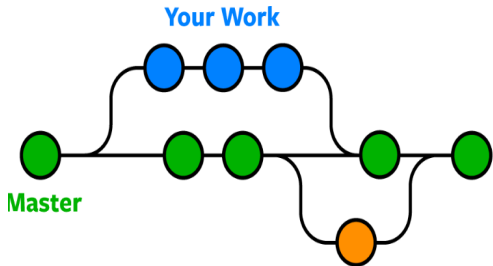
Хороший пример: https://github.com/haaroner/ITMO_Clown/tree/main/Java/lab3_4

Плохой: https://github.com/haaroner/ITMO_Clown/tree/main/Java/Lab2

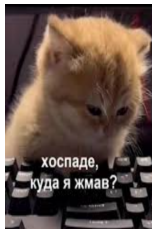
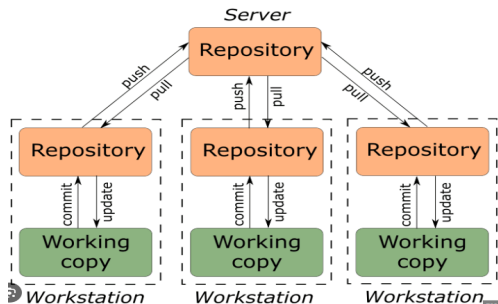
Следующая проблема ой, а куда я жмав, все сломалось. Чтобы не тратить много времени имеет смысл сохранять **стабильные версии** проекта. Можно сделать zip и отправить себе в телеге способ невероятно удобный.

Но если брать более адекватный вариант то это GIT распределенная система управления версиями.

- ▶ В ней ваш проект состоит из **веток(branches)**
- ▶ Каждая ветка **таймлайн версий проекта**.
- ▶ Несколько веток позволяют разрабатывать проект параллельно и из каждой брать по кусочку
- ▶ В каждой ветке **сохранены все версии** вы можете откатиться к любой
- ▶ GIT поддерживает работу с локальными репозиториями и удаленными (remote). Обновление репозитория через **push**, скачивание конкретной версии - **pull**



Someone Else's Work



Очень советую самостоятельно углубиться в эту тему.

Сейчас только самое важное. Подробности:

[https://www.perplexity.ai/search/
kak-sdelat-pervyi-push-na-gith-80L1XRYNSeKcqL5k0RofHw](https://www.perplexity.ai/search/kak-sdelat-pervyi-push-na-gith-80L1XRYNSeKcqL5k0RofHw).

UART (Universal Asynchronous Receiver/Transmitter) — универсальный асинхронный приёмопередатчик — это интерфейс для последовательной передачи данных между устройствами без общего тактового сигнала. Он преобразует параллельные данные из шины микроконтроллера в последовательный поток битов и обратно, используя две линии: TX (передача) и RX (приём). Каждый байт передается в формате кадра:

- ▶ Изначально линия находится в состоянии покоя - HIGH
- ▶ Стартовый бит всегда - LOW. По спадающему фронту происходит синхронизация устройств.
- ▶ Момент чтения каждого следующего бита определяется этим фронтом и baud rate - скорость передачи, которая на обоих устройствах должна быть одинакова.

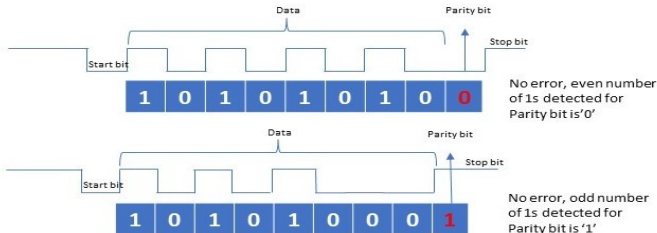
- ▶ Далее идет (обычно) 8 бит данных начиная с **LSB** (Low significant bit), опциональный **PARITY BIT** (бит четности - простая проверка корректности приема байта) и 1 или 2 **STOP BITS** - возврат линии в покой.
- ▶ Одна логическая последовательность байтов называется **пакет**. У пакета, обычно есть явное начало и конец, чтобы было понятно, какой байт что значит, например:
255, X, Y, 254
- ▶ UART имеет стандартные скорости - загляните какие
- ▶ устройства подключаются **крест-накрест** tx1 в rx2 и tx2 в rx1

UART - особенности

UART не самый крутой протокол и имеет множество недостатков:

- ▶ Проблема **рассинхронизации**
- ▶ Много **служебных битов**
- ▶ Невозможно подключить более двух устройств для **full duplex** (двухсторонней) коммуникации.
- ▶ Не самая высокая скорость по сравнению с другими протоколами

Все это с лихвой компенсируется его простотой в использовании



UART - код

Пожалуйста, не копируйте. Пишите сами, ориентируясь на пример. Все места с xxx - вам нужно подумать что вставить по аналогии с уже сделанным кодом.

https://github.com/haaroner/ezh239/tree/main/STM32_25_26/7_UART_GIT

UART - checksum

В процессе передачи данных могут возникать **ошибки передачи**. Причины могут быть совершенно разными и сейчас нас это не интересует.

Есть различные подходы к решению проблемы ошибок в пакетах, все они обобщенно называются **checksum** - "чек-суммы" или проверочные суммы. Отправитель посылает данные, и считате эту сумму по специальному алгоритму. Приемник по полученным данным тоже считает эту сумму и сравнивает с присланной

Расстояние Хэмминга - максимальное количество обнаруживаемых ошибок, т.е. максимальное расстояние между двумя последовательностями, которые будут распознаны алгоритмом, как корректные.

UART - checksum. Виды

- ▶ **Parity Bit** - бит четности. Один бит, расположенный (обычно) в конце байта. Если в байте четное количество единиц - то бит четности 1, иначе 0.
- ▶ **Среднее значение** по байтам
- ▶ Код **Хэмминга**
- ▶ **crc8**, **crc16**, ...
- ▶ На самом деле подойдет **любая** операция, для которой можно явно найти обратное значение.

Код Хэмминга

Каждый бит r_i - бит четности для определенной группы информационных битов. Бит r_{i+1} - половина каждого сектора, покрываемого r_i . Каждый синдром S вычисляется как $S_i = r_i \oplus i_1 \oplus i_2 \oplus i_3$, где i_1, i_2, i_3 - биты конкретного r . Если S_i равен 1 - в этой части есть ошибка. Все S_i задают адрес бита с ошибкой



N ->	1	2	3	4	5	6	7	
2^x	r_1	r_2	i_1	r_3	i_2	i_3	i_4	S
1	X		X		X		X	s_1
2		X	X			X	X	s_2
4				X	X	X	X	s_3

$$r_1 = i_1 \oplus i_2 \oplus i_4$$

$$r_2 = i_1 \oplus i_3 \oplus i_4$$

$$r_3 = i_2 \oplus i_3 \oplus i_4$$

$$s_1 = r_1 \oplus i_1 \oplus i_2 \oplus i_4$$

$$s_2 = r_2 \oplus i_1 \oplus i_3 \oplus i_4$$

$$s_3 = r_3 \oplus i_2 \oplus i_3 \oplus i_4$$

► Какое расстояние Хэмминга в этом коде?

CRC8

CRC8 - алгоритм с намного **большим кодовым расстоянием**.

В следующих слайдах будет приведено строгое обоснование работы алгоритма, которое покажет вам в насколько простых и прикладных задачах математическое решение круче и эффективнее обычного алгоритмического.

Если вы что-то не поймете, возможно, это нормально. Следующая секция скорее для ознакомления.

- ▶ Сами по себе числа это просто знаки, как и знаками являются операции.
- ▶ В каждом конкретном случае мы можем задавать совершенно разные операции и у чисел будет разное поведение
- ▶ Строго говоря нам даже не важно, числа это или другие объекты. Например, мы умеем складывать и умножать многочлены, значит для них тоже справедливы многие мат. утверждения, что и для чисел.
- ▶ Обычно, удобно, чтобы ваши числа и операции удовлетворяли следующим правилам:
 $ab = ba, a(bc) = (ab)c, 1 * a = a, a * a^{-1} = 1$ и те же свойства по сложению (существование обратного элемента по умножению для a , вообще-то не гарантировано)
- ▶ Если у всех элементов есть обратный - то это множество - поле, иначе кольцо

В школе это утверждается как аксиомы, хотя вообще существуют числа, где это не так. Главное, что нам нужно - с такими правилами нам гарантированно, что никакая операция между любыми такими числами не даст результат, выходящий за это множество.

Например \mathbb{Z} (целые числа) - кольцо, а \mathbb{R} (все действительные) - поле. Проверьте!

Оказывается если взять любое кольцо и умножить каждый элемент на число из кольца, то получится новое кольцо \mathbb{I} , обладающее интересным свойством: для любых x из кольца и i из \mathbb{I} $xi \in I$. Такое множество называется **Идеал**. пример $2 * \mathbb{Z}$ - все четные числа. При умножении любого числа на четное, очевидно, будет четное

К чему это всё?

- ▶ Если любые 2 числа a и b такие, что: $a - b \in I$ то $a \equiv b \pmod{n}$ Проверьте на примере $2 * \mathbb{Z}$
- ▶ Как упоминалась раньше многочлены тоже можно складывать и умножать. Значит из них тоже можно составить кольцо и идеал по какому-то многочлену.
- ▶ Если построить множество всех многочленов, с коэффициентами из $0, 1$ то они будут соответствовать всем возможным последовательностям битов

Итак преамбулу мы завершили, возвращаемся к CRC8

CRC8 через многочлены над полем GF(2)

Пусть первый байт соответствует многочлену $M(x)$, crc8 - $R(x)$, а порождающий многочлен (по которому сделан идеал) - это $G(x)$ с коэффициентами либо 0, либо 1. Его степень должна быть равна 8 для максимального расстояние Хэмминга, почему так мы выясним чуть позже.

Тогда, аналогично тому, как любое число можно поделить с остатком, можно и многочлен $M(x) * x^8$ на $G(x)$:

$$M(x) * x^8 = G(x) * Q(x) + R(x)$$

Тогда сумма самого байта ($M(x)$) и чек суммы ($R(x)$) $C(x)$ попадает в идеал:

$$C(x) = M(x) * x^8 - R(x) = G(x) * Q(x) \equiv G(x)(\text{mod } G(x)) \equiv 0$$