

Глава 7. GIT UART

Ежелев Г. И.

30 января 2026 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

GIT

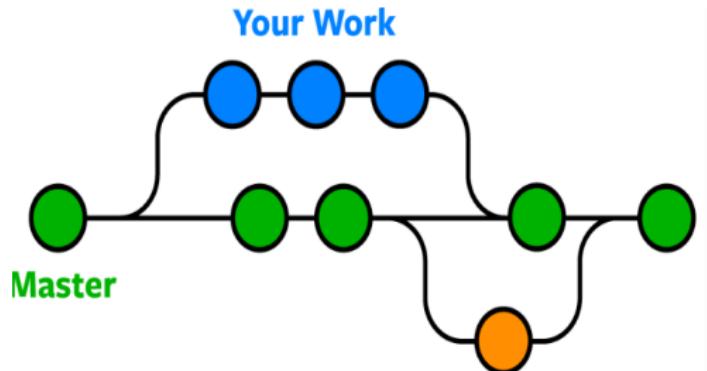
Следующая проблема ой, а куда я жмав, все сломалось. Чтобы не тратить много времени имеет смысл сохранять стабильные версии проекта. Можно сделать zip и отправить себе в телеге способ невероятно удобный.

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

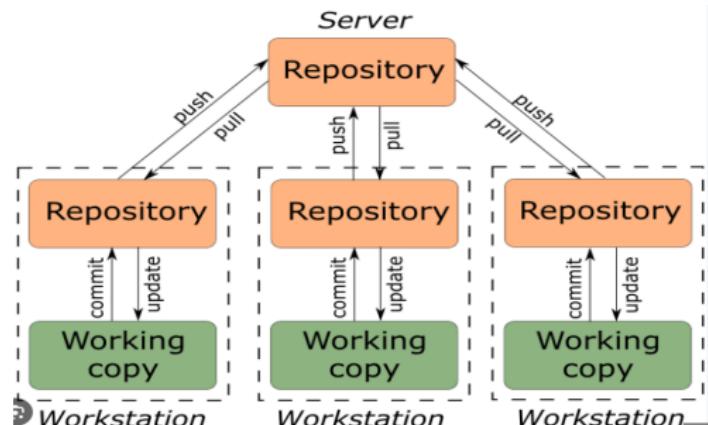
- ▶ В ней ваш проект состоит из веток(branches)
- ▶ Каждая ветка таймлайн версий проекта.
- ▶ Несколько веток позволяют разрабатывать проект параллельно и из каждой брать по кусочку
- ▶ В каждой ветке сохранены все версии вы можете откатиться к любой
- ▶ GIT поддерживает работу с локальными репозиториями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

GIT

STM32
Глава 7.
GIT UART
Ежелев Г. И.



Your 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.

UART

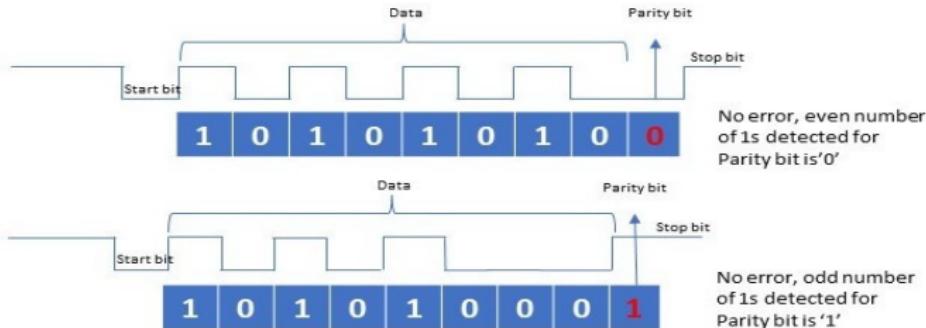
- ▶ Далее идет (обычно) 8 бит данных начиная с **LSB** (Low significant bit), опциональный **PARITY BIT** (бит четности - простая проверка корректности приема байта) и 1 или 2 **STOP BITS** - возврат линии в покой.
- ▶ Одна логическая последовательность байтов называется **пакет**. У пакета, обычно есть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



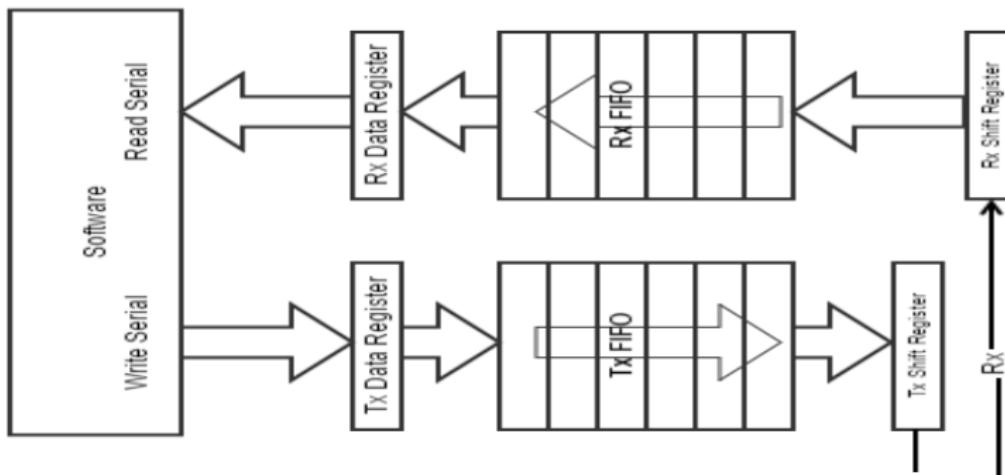
UART - код

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

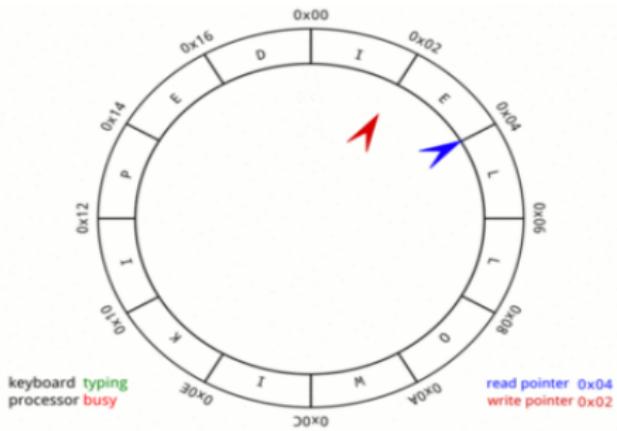
UART - FIFO buffer

FIFO - first input - first output. Часто называют сдвиговый регистр. Т.е. каждый бит, который вы хотите отправить отправляется в сдвиговый регистр трансмиттера. Трансмиттер отправляет последовательно байт за байтом. Ресивер принимает эти пакеты и так же последовательно передает устройству. Посредством шины эти два блока памяти на разных устройствах объединяются в один сдвиговый регистр, работающий как конвейер.



UART - буфер на массиве

- ▶ Чтобы заставить работать массив как буфер, т.е. часть сдвигового регистра (отправка и прием байтов не мгновенная). Нужно его закольцевать.
- ▶ Создадим два массива - `uint8_t tx[16]` и `rx[16]`
- ▶ Для каждого массива по 2 указателя - (`byte_to_send`, `last_byte`. Первый указывает, какой следующий байт нужно отправить, второй, какой последний байт лежит в массиве (после него лежат старые значения)



Код

Сначала напишем код без буфера. У нас имеется 3 флага в регистрах UART:

- ▶ TXE - TX Empty - устройство готово к очередной передаче = 1
- ▶ TC - Transmission Complete - Передача завершена
- ▶ RXNE - RX Not Empty - Приемник не пуст = 1

```
1 void write(uint8_t byte)
2 {
3     while (USART_GetFlagStatus(USART6, USART_FLAG_TXE) == RESET);
4     USART_SendData(USART6, byte);
5     while (USART_GetFlagStatus(USART6, USART_FLAG_TC) == RESET);
6 }
7
8 uint8_t read()
9 {
10    while (USART_GetFlagStatus(USART6, USART_FLAG_RXNE) == RESET);
11    return USART_ReceiveData(USART6);
12 }
```

Настройка UART

```
1 USART_InitTypeDef usart;
2 RCC_APB2PeriphClockCmd(RCC_APB2Periph_USART6, ENABLE);
3 usartUSART_BaudRate = ???;
4 usartUSART_WordLength = ???;
5 usartUSART_StopBits = ???;
6 usartUSART_Parity = USART_Parity_No;
7 usartUSART_HardwareFlowControl = USART_HardwareFlowControl_None;
8 usartUSART_Mode = USART_Mode_Tx | USART_Mode_Rx ;
9 USART_Init(USART6, &usart); //Настройка USART
10 USART_Cmd(USART6, ENABLE); // Включение/выключение UART
```

- ▶ **BaudRate** - скорость
- ▶ **WordLength** - количество бит в пакете
- ▶ **StopBits** - Длительность остановочных битов
(т.к. они всегда равны 1 это эквивалентно
минимальной задержке между пакетами)
- ▶ **Parity** - наличие бита четности, **FlowControl** -
использование сигнала для синхронизации
- ▶ **Mode** - Какие шины использовать.

UART. Буфер

Теперь сделаем продвинутый UART.

- ▶ Работа с буфером будет **внутри прерывания** -
`extern "C"void USART6_IRQHandler(void)`
- ▶ Две части - Обработка **отправки и приема** данных
- ▶ Код лежит в репозитории в папке 07

UART. Буфер

- ▶ Отметим, что на каждый буфер TX и RX **по 2** **указателя**, по одному данные записываются в буфер, по другому извлекаются, если один обгонит другой вы потеряете логичность данных.
- ▶ Отдельно отметим функции связанные с **CRITICAL_ZONE** - они **откладывают** **прерывания** для обеспечения корректности доступа к данным (если вы работаете с буфером, а в этот момент срабатывает прерывание, которое меняет этот буфер может возникнуть ошибка).
Если внутри критической зоны вы поставите задержку - программа зависнет, т.к. задержки работают по прерываниям на таймере.

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 посыпает данные, и считает 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 байти. 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.

Кольцо

- ▶ 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 а, 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. Например кватернионы, матрицы тоже не выполняют всех условий, хотя их можно складывать и умножать.
- ▶ Главное, что нам нужно - с такими правилами нам гарантированно, что никакая операция между любыми такими объектами не даст результат, выходящий за это множество.
- ▶ Например \mathbb{Z} (целые числа) - кольцо, а \mathbb{R} (все действительные) - поле. Проверьте!
- ▶ Главное отличительное свойство поля от кольца, в поле вы можете прыгнуть из любого числа в другое за одну операцию, в кольце не всегда. Попробуйте взять два числа из \mathbb{Z} : 3 и 2 и умножить 3 на какое-то целое число(из кольца целых), чтобы получилось 2. Такого нет. А у действительных оно есть: $\frac{2}{3} * 3 = 2$.

Оказывается если взять любое кольцо и умножить каждый элемент на число из кольца, то получится новое кольцо \mathbb{Z} , обладающее интересным свойством: для любых x из кольца и i из \mathbb{Z} $xi \in I$. 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 - **кольцо многочленов** - и идеал по какому-то многочлену.
- ▶ Если построить множество всех многочленов, с коэффициентами из 0, 1 то они будут соответствовать всем возможным последовательностям битов (и это даже окажется полем, но мы это уже не сможем так легко доказать)

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, а сам он должен быть простым, почему так мы выясним чуть позже.

CRC8 через многочлены над полем GF(2)

- ▶ Итак, $G(x)$ - порождающий ($\mathbb{I} = G(x) * GF(2)$),
 $M(x)$ - байт, $R(x)$ - чек сумма, $Q(x)$ - целая часть
от деления.
- ▶ Тогда, аналогично тому, как любое число можно
поделить с остатком, можно и многочлен $M(x) * x^8$
на $G(x)$:
$$M(x) * x^8 = G(x) * Q(x) + R(x)$$
- ▶ Умножение на x^8 сделано, чтобы степени
совпадали у $M(x)$ и $G(x)$
- ▶ Тогда сумма самого байта ($M(x)$) и чек суммы
($R(x)$) - $C(x)$ попадает в идеал:
$$C(x) = M(x) * x^8 - R(x) = G(x) * Q(x) \equiv G(x) \equiv 0$$

(mod $G(x)$) почему я сказал про сумму а написал
про разность?

- ▶ Разность и сумма эквивалентны в GF(2): $1-1 = 1+1 \equiv 0$ (так же работают и побитовые операции)
- ▶ Если окажется, что произошла **ошибка передачи**, то $C(x) \neq 0(modG(x))$ Степень G(x) равна 8, значит расстояние Хэмминга - расстояние между двумя разрешенными (т.е. алгоритм сочетет их правильными) комбинациями - **равно 8**, что довольно солидно.
- ▶ Если порождающий многочлен (по которому сделан идеал, и на который **должна разделиться сумма** CRC8 и байта, чтобы алгоритм счел байт переданным без ошибок) будет иметь степень меньше 8, то и максимальное расстояние будет меньше (многочленов, которые разделяются на x^6 явно больше чем на x^8).
- ▶ Если порождающий многочлен **не простой** (приводимый), то $G(x) = H(x) * F(x)$ Степень каждого из них меньше, а по предыдущему пункту это **уменьшит расстояние Хэмминга**.

Код

Итак, идея, каждый пакет **можно представить как многочлен**.

Пример: $0x31 = 100110001 \sim x^8 + x^5 + x^4 + 1$

Отправитель **суммирует** все многочлены, соответствующие всем пакетам и **находит остаток от деления** этой суммы на произвольный многочлен.

Для **проверки передачи** по такой чек сумме просто сложим все пакеты на приемнике и в конце сложим с самой чек суммой(остатком). Поскольку сложение бинарное (коэффициенты только 1 и 0) полученная сумма **будет иметь нулевой остаток**.

Тогда, если приемник получил сумму с ненулевым остатком => произошла ошибка передачи.

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

где $M(x)$ - сумма байтов данных, $G(x)$ - многочлен, на который делим, $R(x)$ - остаток - чек сумма.

Код. Сумма многочленов

- ▶ Первый шаг - **сложение**. Пусть изначальное значение crc будет: $\text{crc}=0xFF=11111111 \sim x^8 + x^7 + x^6 + x^5 + x^4 + x^3 + x^2 + x + 1$
- ▶ **Сложим** его с многочленом $x^5 + x^2 + 1$.
Получим: $x^8 + x^7 + x^6 + 2*x^5 + 2*x^4 + x^3 + x^2 + x + 2 = x^8 + x^7 + x^6 + 0 * x^5 + 0 * x^4 + x^3 + x^2 + x + 0$
Т.к. в двоичной арифметике **$1+1 = 2 = 0$** (остаток от деления на 2 одинаковый). Можете проверить в калькуляторе. Отметим, что **переноса в другие разряды нет**, т.к. мы складываем многочлены, и сложение только между одинаковыми коэффициентами.
- ▶ То есть мы получили: две единицы или два нуля - получили 0. Одна единица - получили 1. Какая **логическая операция** ведет себя так же?

Это **операция XOR** - исключающее ИЛИ. В C++ обозначается ^

Будем проводить алгоритм с каждым байтом отдельно.

data - массив байтов сообщения, len - их количество.

```
1  uint8_t camera::crc8(uint8_t* data, int len)
2  {
3      uint8_t crc = 0xFF, i, j;
4      for (i = 0; i < len; i++) {
5          crc ^= data[i];
```

Код. Деление многочленов

- ▶ Поделим промежуточную сумму на многочлен $0x31 = 100110001$. В нем 9 бит.
- ▶ Будем сдвигать побитово сумму на 1 влево, чтобы наша 8 битная сумма стала 9-и битной. Если самый левый (MSB) бит будет равен 0 - скипаем шаг, наше число меньше чем $0x31$, ничего не разделится. В соответствующем разряде результата запишем 0.

Код. Деление многочленов

- ▶ Пусть самый левый бит равен 1 и мы можем поделить промежуточную сумму на 0x31.
Поскольку у нас коэффициенты либо 0 либо 1, значит в результате шага деления коэффициент больше 1 появиться не может.
- ▶ коэффициент 0 появиться тоже не может, это значит, что старший бит равен 0 (пред. слайд), а он 1.
- ▶ Значит коэффициент на этом шаге равен 1. А это значит, что для деления нужно просто вычесть 0x31 из текущего числа.
- ▶ Так и сделаем, сдвинем на 1 бит влево и вычтем 0x31 (XOR это и сложение и разность, т.к. -1 сравним с 1 по модулю 2).

Код. Деление многочленов

Так нужно проделать с каждым разрядом. Тогда в конце деления мы получим остаток от деления.

Заметьте что мы только что просто описали деления многочленов в столбик с остатком, с единственным отличием - коэффициенты только 1 и 0.

```
1  uint8_t camera::crc8(uint8_t* data, int len)
2  {
3      uint8_t crc = 0xFF, i, j;
4      for (i = 0; i < len; i++) {
5          crc ^= data[i];
6          for (j = 0; j < 8; j++) {
7              if (crc & 0x80) crc = (char)((crc << 1) ^ 0x31);
8              else crc <<= 1;
9          }
10     }
11     return crc;
12 }
```

Теперь для отправки данных нужно заполнить массив байтами данных и полученную чек сумму отправить в самом конце сообщения.

А приемник должен наполнить массив всеми принятыми данными(в том числе и чек суммой!) и посчитать crc8 для этого набора данных. Если результат равен 0 - сообщение корректно передано.

Другой способ посчитать crc8 по всем принятым байтам данных и сравнить ее с принятой чек суммой.

Код. Прием данных

```
1 if(usart2::available() >= 8)
2 {
3     camera_data = usart2::read();
4     if(camera_data == 255)
5     {
6         for(int i = 0; i < 9; i++)
7             data[i] = usart2::read();
8
9         if(data[8] == crc8(data, 8))
10        {
11            \\Обработка данных
```

пример приема пакета из 8 байт: стартовый 255
(остальные не равны 255), 6 байт информации и 8-й
байт - чек сумма.