+7-913-939-20-48.

Денис Олегович Бирюков

Оглавление

[Device 1](#_Toc463861996)

[MK programming 2](#_Toc463861997)

[Интерфейс ввода-вывода 3](#_Toc463861998)

[Регистры конфигурации 3](#_Toc463861999)

[Регистры данных 4](#_Toc463862000)

[Регистр установки/сброса 4](#_Toc463862001)

[Регистр блокировки 4](#_Toc463862002)

[Unknown 4](#_Toc463862003)

[C - language 5](#_Toc463862004)

[Const volatile static 5](#_Toc463862005)

[Квалификатор типа const. 6](#_Toc463862006)

[Квалификатор типа volatile. 6](#_Toc463862007)

[Спецификатор класса памяти extern. 6](#_Toc463862008)

[Structures 7](#_Toc463862009)

[Побитовые операторы 8](#_Toc463862010)

[Полезные функции: 8](#_Toc463862011)

[itoa 8](#_Toc463862012)

[HEX 9](#_Toc463862013)

[16-09-2016 9](#_Toc463862014)

[23-09-2016 9](#_Toc463862015)

[ADC – analog to digital converter 10](#_Toc463862016)

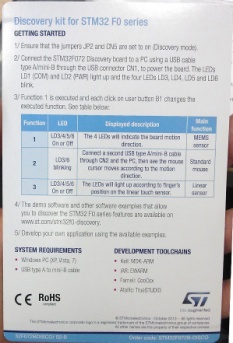
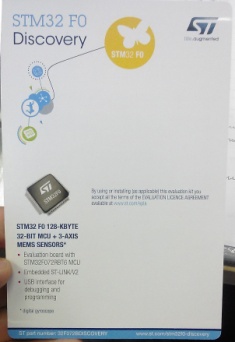
# Device

Наш микроконтроллер stm32f072rb

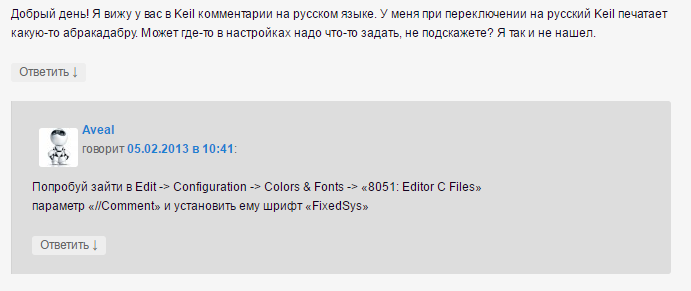
<http://www.st.com/content/st_com/en/products/microcontrollers/stm32-32-bit-arm-cortex-mcus/stm32f0-series/stm32f0x2/stm32f072rb.html>

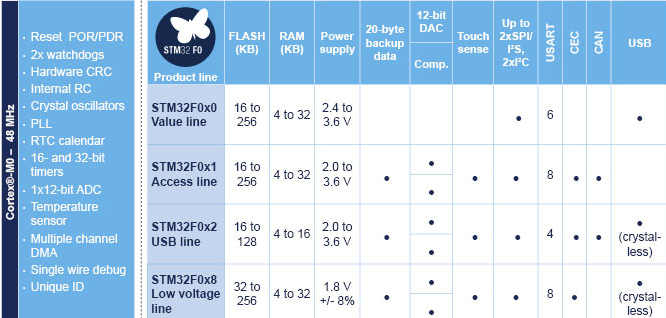
Отладочная плата: Discovery kit with STM32F072RB MCU

<http://www.st.com/content/st_com/en/products/evaluation-tools/product-evaluation-tools/mcu-eval-tools/stm32-mcu-eval-tools/stm32-mcu-discovery-kits/32f072bdiscovery.html>









# MK programming

**GPIO** - Интерфейс ввода/вывода общего назначения (англ. general-purpose input/output, GPIO. Через них, порты, мк связан с внешним миром.

**AHB** - advanced high-performance bus. Отходит от ядра.

**APB** - advanced peripheral bus

**SRAM** - Статическая память с произвольным доступом (static random access memory) — полупроводниковая оперативная память, в которой каждый двоичный или троичный разряд хранится в схеме с положительной обратной связью, позволяющей поддерживать состояние без регенерации, необходимой в динамической памяти (DRAM). Тем не менее, сохранять данные без перезаписи SRAM может, только пока есть питание, то есть SRAM остается энергозависимым типом памяти.

**Регистр процессора** — сверхбыстрая память внутри процессора, предназначенная для хранения адресов и промежуточных результатов вычислений (регистр общего назначения/регистр данных) или данных, необходимых для работы самого процессора

**RCC** – reset and clock control

* system reset
* power reset
* RTC domain reset (сброс домена)

**ODR** - output data registr (GPIOx\_ODR)

Интерфейс ввода-вывода

Каждый интерфейс ввода-вывода имеет 4ре 32х битных регистра конфигурации:   
GPIOx\_MODER, GPIOx\_OTYPER, GPIOx\_OSPEEDR and GPIOx\_PUPDR  
два 32х битных регистра данных (data register)  
(GPIOx\_IDR and GPIOx\_ODR)  
и один set/reset регистр  
(GPIOx\_BSRR)  
Ports A and B also have a 32-bit locking register (GPIOx\_LCKR) and two 32-bit alternate function selection registers (GPIOx\_AFRH and GPIOx\_AFRL).

### Регистры конфигурации

Как можно догадаться из названия, эти регистры позволяют задавать режим работы каждой ножки отдельно. Всего их 4 для каждого порта:

* GPIOx\_MODER. Этот регистр задает направление ввода-вывода каждой ножки. Направление может быть: вход, выход, альтернативная ф-я, аналоговый. По-умолчанию(после сброса) направление всех ножек устанавливается на вход, кроме портов A и B(начальное состояние можно посмотреть в пункте 8.4.1. СР)
* GPIOx\_OTYPER. Этот регистр задает тип выхода: двухтактный или открытый сток. По умолчанию — двухтактный для каждой ножки
* GPIOx\_OSPEEDR. С помощью этого регистра задается частота тактирования каждого вывода. СР описывает частоты как: низкая, средняя, повышенная, высокая. После сброса скорость низкая(кроме В.3, тут высокая). Где же сами частоты? Их нужно смотреть в Даташите, об этом ниже.
* GPIOx\_PUPDR. Этот регистр управляет подтяжкой каждой ножки. Нам предлагают на выбор: без подтяжки, подтяжка к питанию, подтяжка к земле. По умолчанию подтяжки нет(кроме портов А и В, их состояние уточняйте в пункте 8.4.4 СР)

### Регистры данных

Тут все намного проще: два регистра, один на вывод, другой на ввод. Давайте рассмотрим их поподробнее:

* GPIOx\_IDR. Регистр ввода. Если вывод МК настроен на вход, то любой, поданный на ножку сигнал пройдет через триггер Шмитта и отразится в этом регистре в виде соответствующего 0 или 1.
* GPIOx\_ODR. Регистр вывода. То же самое, только задом-на-перед: то, что мы запишем в этот регистр(0 или 1) попадет на ножку МК в виде логического уровня цифрового сигнала.

### Регистр установки/сброса

* GPIOx\_BSRR. При помощи этого регистра осуществляется запись 0 или 1 в регистр вывода GPIOx\_ODR. Вы спросите, зачем же нам нужен этот дополнительный регистр, если можно прочитать регистр вывода, установить и сбросить в нем нужные нам биты и снова записать результат регистр? Тут дело в атомарности операций. Без использования регистра установки/сброса такой подход занимает сразу несколько операций(тактов): сначала необходимо записать данные из регистра вывода в один из общих регистров, затем применить к данным в общем регистре логические операции(для переключения бита/битов используют XOR, для установки в 0 используют NOT затем AND, для установки в 1 используют OR), и наконец снова записать полученное значение из общего регистра обратно в регистр вывода. Как видим одна такая операция на самом деле состоит из 3-4 подопераций и не является единичной(атомарной). Такова особенность большинства архитектур. Неатомарность операций плоха тем, что если в момент выполнения одной из подопераций возникнет запрос прерывания, то МК переключится на выполнение кода прерывания и только после этого вернется к нашей подоперации. Следовательно если в коде обработчика прерывания используется GPIO — можно получить на выводе МК не то состояние(помните про жизни людей?=) ). Есть много подходов к решению этой проблемы, словом некоторые(маскируемые) прерывания можно запретить на время выполнения операции, можно контролировать атомарность программно. Но STM предоставляет готовое решение — регистр установки/сброса, обращения через который атомарны и выполняются быстрее. О прерываниях мы поговорим в следующих уроках, а пока можно заглянуть сюда и вот сюда. При работе с МК знание булевой алгебры на начальном уровне обязательно, этой статьи нам будет вполне достаточно(только не забывайте, что она из цикла по MSP430, а мы изучаем STM32).

### Регистр блокировки

* GPIOx\_LCKR. Этот регистр осуществляет управление механизмом блокировки конфигурации.Что бы не забивать голову с самого начала, останавливаться на этом регистре, впрочем как и на самом механизме, мы не будем. При желании можно заглянуть в пункт 8.3.6 GPIO locking mechanism СР. Упомяну лишь, что даже в заблокированном состоянии доступна работа с регистром установки/сброса(GPIOx\_BSRR) и регистром вывода(GPIOx\_ODR).

## Unknown

**CMSIS -** библиотека. Единый стандарт для всех cortex. В нем стандартизирован доступ к разной периферии разных микроконтроллеров STM. Также библиотека SPL – standard peripheral library. В виде структур реализована настройка всевозможной периферии.

**Keil uVision** - среда разработки программного обеспечения для микроконтроллеров

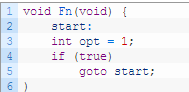
**ARM Cortex-A5** — процессорное ядро, разработанное ARM Holdings и реализующее архитектуру набора команд ARM v7. Появилось в конце 2009 года. Предназначен для приложений, которые требуют управления виртуальной памятью на высоком уровне в маломощных операционных системах. Процессор распространён среди широкого спектра устройств: от смартфонов и мобильных телефонов до промышленных устройств.

# C - language

**uint32\_t** for bit manipulations, especially on 32-bit registers

**static** это ключевое слово как и const и определяет static переменную или функцию что значит:  
1. Если static переменная была объявлена внутри функции то она сохраняет свое значение между вызовами этой функции.  
2. Если static переменная или функция была объявлена как глобальная то она видна только в том файле где была объявлена.

**volatile uint32\_t -** Модификатор volatile запрещает компилятору оптимизировать обращения к аргументу[[1]](#footnote-2). Значение переменной может меняться извне. Под действием управления ОС, аппаратных средств и другого потока. Поскольку значение может меняться, компилятор каждый раз разгружает его из памяти.   
Пример: зацикленная программа. Всегда единица. Но извне может податься «0», тогда произойдет выход из цикла. Только opt должна быть *volatile int opt = 1;*



**Задержка**  
Нет стандартной функции программной задержки. Написал функцию, которая отнимает 1. Обычно программные задержки не используются – не точны и считают мусор. Лучше использовать таймеры.

void Soft\_Delay(volatile uint32\_t number)  
{  
        while(number--);  
}

## Const volatile static

### Квалификатор типа const.

Все переменные, определенные с использованием этого ключевого слова не могут изменить своего значения во время выполнения программы. На то они и константы 😉 Чаще всего спецификатор const используется так: когда в программе много раз используется одно и то же числовое значение, создается переменная, имеющая определенное имя, значение которой равно этому числовому значению. Так как переменная не меняется при выполнении программы, используется ключевое слово const. И теперь вместо использования числового значения можно использовать имя константы, что заметно облегчает понимание написанного кода и последующую работу с ним. Поэтому настоятельно рекомендуется в качестве имен переменных использовать что-либо осмысленное, а не просто первый пришедший на ум набор букв. Например, нужно объявить константу, которая будет хранить значение максимального количества принятых байт. Тогда:

// Удачное имя переменной

const int maxNumOfBytes = 64;

// Неудачное имя переменной

const int var123 = 64;

В принципе это основное применение спецификатора const, поэтому, думаю, нет смысла углубляться, так что идем дальше.

### Квалификатор типа volatile.

Этот замечательный спецификатор позволяет нам использовать переменную, значение которой может изменяться неявно в процессе выполнения программы. Что же это значит? А вот пример:

bool test = FALSE;

if (test)

{

....

}

Так как наша переменная test нигде явно не изменяется (то есть не стоит слева от оператора присваивания) то заботливый компилятор в результате оптимизации прочитает ее значение только один раз и больше не станет, переменная то на его взгляд нигде не изменяется! А эта переменная на самом деле может принять другое значение в результате выполнения другого потока, либо изменить свое значение неявно в прерывании. А это уже катастрофа! И для того, чтобы ее предотвратить как раз и используется спецификатор volatile. То есть объявить нашу переменную мы должны так:

*volatile bool test = FALSE;*

### Спецификатор класса памяти extern.

Если программа состоит из нескольких файлов, как почти всегда и бывает, то без ключевого слова extern не обойтись ) Пусть в разных файлах используется одна и та же переменная x. Значит каждый файл должен знать о существовании этой переменной, но просто объявить переменную во всех файлах не прокатит — компилятор этого не допустит. Тут то и придет на помощь спецификатор extern. Вот пример: пусть программа состоит из двух файлов, в каждом из которых есть функции, использующий переменную x:

Файл номер 1

unsigned char x;

void main()

{

x = 0x55;

...

...

}

Файл номер 2

extern unsigned char x;

void testFunction()

{

x += 0x10;

}

При таком объявлении переменной ошибки не будет и все скомпилируется успешно.

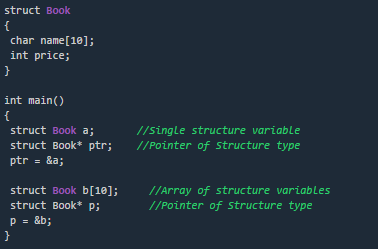
Тут есть важный момент — а именно разница между понятиями определить и объявить переменную. При объявлении переменной ей присваивается определенный тип и значение, а при определении — для нее выделяется память. Таким образом, если перед именем переменной стоит спецификатор extern, то мы объявляем переменную, не определяя ее.

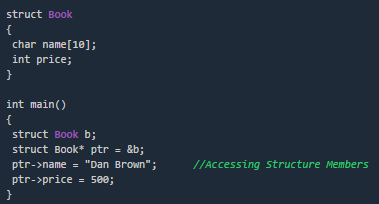
## Structures

**Объявление указателя на структуру**struct addr \*addr\_pointer;

**Адрес переменной-структуры**p = &person;

**Чтобы с помощью указателя на структуру получить доступ к ее членам, необходимо использовать оператор стрелка ->**p->balance





Таким образом, строка GPIOC->MODER:

GPIOC – указатель на структуру  
MODER, OTYPER, OSPEEDR, PUPDR – члены структуры (переменные структуры)

RCC – указатель на структуру  
AHBENR – член структуры

## Побитовые операторы



## Полезные функции:

### itoa

void itoa(int n, char s[])

{

int i, sign;

if ((sign = n) < 0) /\* записываем знак \*/

n = -n; /\* делаем n положительным числом \*/

i = 0;

do { /\* генерируем цифры в обратном порядке \*/

s[i++] = n % 10 + '0'; /\* берем следующую цифру \*/

} while ((n /= 10) > 0); /\* удаляем \*/

if (sign < 0)

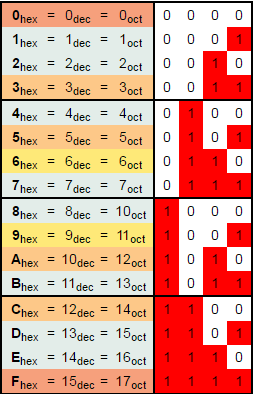
s[i++] = '-';

s[i] = '\0';

reverse(s);

}

## HEX



# 16-09-2016

**Тактирование:**  
 прочесть содержимое регистра  
 обнулить  
 пишем новое значение  
 сохраняем

Т.е. прочесть, записать, обнулить.

# 23-09-2016

Научились читать состояние кнопки.

Макросы:  
Атомарная запись в регистре. Можем выставлять и сбрасывать регистры.

Было: чтение, накладывание по маске новое значение, складывание на место – не атомарная операция.

**ODR**У ODR есть регистр BSRR. Младшая за выставление 1, остальные не трогаются. Просто пишем. Если хотим сбросить, то пишем в 16 единицу. Логику делает сам. Можем сразу несколько битов писать.

GPIOC->BSRR = GPIO\_BSRR\_BR\_6;

**Прерывания.**

У каждого кортекса (да и вообще) есть набор вещей, которые одинаковы (gpio и регистры разные для разных). Одинаковы, например, системный таймер. Просто считает до какого-то значения. Потом прерывается и заново. Может генерировать прерывания или нет.

Пишем в мэйне

SysTick\_Config(количество тиков)

Переменная SyStem COre clock /1000. SystemCoreClock - открыть вкладку device. Вектор - адрес по которому она прерывает. Вызывается функция - SysTick\_Handler. Нужно самим написать эту фунцию. Заводим переменную на кр светодиод

# ADC – analog to digital converter

**Разрешение АЦП** — минимальное изменение величины аналогового сигнала, которое может быть преобразовано данным АЦП — связано с его разрядностью. В случае единичного измерения без учёта шумов разрешение напрямую определяется разрядностью АЦП.

**Разрядность АЦП** характеризует количество дискретных значений, которые преобразователь может выдать на выходе

Например, двоичный 8-разрядный АЦП способен выдать 256 дискретных значений (0…255), поскольку {\displaystyle 2^{8}=256}

**Разрешение по напряжению** равно разности напряжений, соответствующих максимальному и минимальному выходному коду, делённой на количество выходных дискретных значений.

Разрешение: (3.6-2.4)/2^{12} = 1.2/4096 = 0.293mkV

На практике разрешение АЦП ограничено отношением сигнал/шум входного сигнала.При этом реально достижимое разрешение описывается **эффективной разрядностью** (англ. effective number of bits, ENOB), которая меньше, чем реальная разрядность АЦП

То есть любой высокоскоростной АЦП крайне чувствителен к качеству оцифровывающей тактовой частоты, подаваемой пользователем

Частота, с которой производятся цифровые значения, получила название **частота дискретизации АЦП.**

Поскольку реальные АЦП не могут произвести аналого-цифровое преобразование мгновенно, входное аналоговое значение должно удерживаться постоянным, по крайней мере, от начала до конца процесса преобразования (этот интервал времени называют время преобразования). Эта задача решается путём использования специальной схемы на входе АЦП — устройства выборки-хранения (УВХ). УВХ, как правило, хранит входное напряжение на конденсаторе, который соединён со входом через аналоговый ключ: при замыкании ключа происходит выборка входного сигнала (конденсатор заряжается до входного напряжения), при размыкании — хранение. Многие АЦП, выполненные в виде интегральных микросхем, содержат встроенное УВХ.

Если последовательность цифровых значений, выдаваемая АЦП, где-либо преобразуется обратно в аналоговую форму цифро-аналоговым преобразователем, желательно, чтобы полученный аналоговый сигнал был максимально точной копией исходного сигнала. Если входной сигнал меняется быстрее, чем делаются его отсчёты, то точное восстановление сигнала невозможно, и на выходе ЦАП будет присутствовать ложный сигнал. Ложные частотные компоненты сигнала (отсутствующие в спектре исходного сигнала) получили название alias (ложная частота, побочная низкочастотная составляющая). Частота ложных компонент зависит от разницы между частотой сигнала и частотой дискретизации. Например, синусоидальный сигнал с частотой 2 кГц, дискретизованный с частотой 1.5 кГц, был бы воспроизведён как синусоида с частотой 500 Гц. Эта проблема получила название **наложение частот (aliasing).**

Для предотвращения наложения спектров сигнал, подаваемый на вход АЦП, должен быть пропущен через фильтр нижних частот для подавления спектральных компонент, частота которых превышает половину частоты дискретизации. Этот фильтр получил название anti-aliasing (антиалиасинговый) фильтр, его применение чрезвычайно важно при построении реальных АЦП.

1. https://en.wikipedia.org/wiki/Volatile\_(computer\_programming) [↑](#footnote-ref-2)