

# Курс

---

## 1. Предисловие

Специальность “Информационно-измерительная техника” направлена на создание и применение устройств и систем, составляющих основу информационных технологий в различных отраслях промышленности. Особое внимание должно уделяться компьютерной или микропроцессорной технике как со стороны аппаратного, так и программного обеспечения.

В современном мире неотъемлемой частью практически любого измерительного устройства является микроконтроллер. Важной особенностью применения микроконтроллеров в измерительных устройствах является тот факт, что для надежной работы такого устройства необходимо не только надежная аппаратура, но и качественное и надежное программное обеспечение управляющее микроконтроллером. В настоящее время существует очень много методических пособий и книг по разработке устройств с использованием микроконтроллеров, однако вопросы разработки программного обеспечения сводятся к простым примерам на языке ассемблера и Си. Кроме того, существующие пособия значительно отстают от быстременяющихся изменений в микропроцессорной технике и тем более языках программирования. Если еще недавно прорывом в программирование был выход стандарта С11, то уже сегодня существует стандарт С20 и уже активна работа по стандарту С++23. Следует также заменить, что автором не найдено ни одной книги или пособия, которые бы затрагивали, например, такие области разработки ПО для микроконтроллеров, как архитектура программного обеспечения, использования UML и средств моделирования архитектуры.

Предыдущие методические пособия для курса ПОИП, например, [2] были ориентированы на широкие области применения информационных технологий, начиная от микроконтроллеров и заканчивая базами данных. Однако по мнению автора, невозможно хорошо разобраться и усвоить столь большой объем разногласовой информации. В итоге курс и лабораторные работы дают лишь поверхностное представление о разработке программного обеспечения, а будущие инженеры не до конца усваивают материал и не могут детально разобраться в принципах разработки программного обеспечения для измерительных устройств. Основываясь на данном предубеждении, автором выбран иной путь, а именно более узкоспециализированное и детальное рассмотрение принципов разработки программного обеспечения измерительных устройств на базе современных микроконтроллеров.

Большую помощь в разработке методического материала оказал обучающийся на кафедре ИНИТ Загоскин Я. Современные быстроизменяющиеся и эволюционирующие условия диктуют и новый подход к образованию, а именно все больший упор делается на самообразование, самоусовершенствование и самостоятельный поиск нужной информации с технической документации, системах поиска, книгах.

Поэтому довольно большая часть разделов предлагается студентам для самостоятельного

изучения и выполнения в качестве домашней практической работы. Большое влияние на составление данного методического пособия оказал труд [1] Недяка С.П., Шаропина Ю.Б. откуда были заимствованы некоторые подходы и организационная структура методического пособия.

## 2. Введение

Данное методическое пособие предназначено для выполнения лабораторных работ с использованием отладочных плат XNUCLEO F411RE для измерения физических величин на основе микроконтроллера STM32F411 с архитектурой Cortex-M4, операционной системы реального времени FreeRtos и языка программирования C++ 14 интегрированной средой разработки IAR Embedded Workbench for ARM ver. 8.20

Методическое пособие написано в рамках курса “Программное обеспечение измерительных процессов”, но оно будет полезным всем желающим освоить принципы разработки измерительных устройств на современных микроконтроллерах.

Предполагается, что для изучения данного курса у студентов есть хорошие знания микропроцессорной техники и навыки разработки ПО на языке C++ полученные на ранних курсах.

Методическое пособие состоит из 3 разделов. Первая часть ориентирована на создание проекта в IAR Embedded Workbench, работу с периферией микроконтроллера и возможности отладки системы IAR Embedded Workbench, работе с детальной разработкой простых классов на языке UML в пакете starUML. Вторая часть посвящена использованию прерываний, работе с операционной системой FreeRtos взаимодействию между задачами. Заключительная часть ориентирована на принцип разработки архитектуры программного обеспечения, шаблонам проектирования, разработки детальной архитектуры.

Модули связаны с курсовым проектированием в котором буду задействованы все части данного методического пособия.

В 2017 году компания “Метран” безвозмездно предоставила ЮУрГУ на кафедру информационно-измерительная техника 10 отладочных комплектов на базе микропроцессора Stm32F411RE на ядре Cortex M4 с различными модулями расширения включающие в себя модули Bluetooth, WiFi, графическим индикатором, различными сенсорами, включающие в себя датчики Холла, датчики влажности, температуры, звука, освещенности, дыма, положения и многое другое. По этой причине курс ПОИП и лабораторный практикум выполняется на отладочных платах XNUCLEO 411RE.

## Лекция 1

### 1. Основные термины и определения

## *Встраиваемые вычислительные системы (Embedded systems)*

Встраиваемая система (встроенная система, англ. *embedded system*) – специализированная микропроцессорная система управления, концепция разработки которой заключается в том, что такая система будет работать, будучи встроенной непосредственно в устройство, которым она управляет.

## *Микроконтроллер (англ. Micro Controller Unit, MCU)*

Микросхема, предназначенная для управления электронными устройствами. Типичный микроконтроллер сочетает на одном кристалле функции процессора и периферийных устройств, содержит ОЗУ и (или) ПЗУ. По сути, это однокристальный компьютер, способный выполнять простые задачи.

## *Контроллер*

1. Изделие для автоматизации и управления.
2. Микросхема или часть микросхемы реализующая отдельную функцию или задачу управления.

## *Отладочная, оценочная или демонстрационная плата*

Электронный модуль, как правило, в бескорпусном изготовлении, содержащий минимально необходимый набор микросхем для разработки ПО для МК.

## *Интегрированная среда разработки. IDE(англ. IDE, Integrated development environment)*

Система программных средств, используемая программистами для разработки программного обеспечения.

Обычно, среда разработки включает в себя:

- текстовый редактор,
- компилятор и/или интерпретатор,
- средства автоматизации сборки,
- отладчик.

## *SWD - Serial Wire Debug.*

Двухпроводной отладочный порт

## *Компилятор*

Программа выполняющая трансляцию исходного кода из предметно-ориентированного языка на машинно-ориентированный язык.

## *Компоновщик(Линковщик)*

Программа собирающая исходный код на машино-ориентированном языке и производящую сборку в исполняемый модуль

## *Стек*

Абстрактный тип данных, представляющий собой список элементов, организованных по принципу LIFO  
(англ. *last in – first out*, «последним пришёл – первым вышел»).

Возможны три операции со стеком: добавление элемента (иначе проталкивание, *push*), удаление элемента (*pop*) и чтение головного элемента (*peek*).

Мы будем использовать определение Стека, в значении Аппаратный стек

## *Аппаратный стек*

В микроконтроллере стек - это непрерывная область памяти, адресуемая специальными регистрами SP  
(указатель стека)

## *Регистр*

Сверхбыстрая память внутри процессора, предназначенная для хранения адресов и промежуточных результатов вычислений (регистр общего назначения/регистр данных) или данных, необходимых для работы самого процессора.

# **2. Среда разработки программ для микроконтроллера**

В учебных целях мы будем использовать интегрированную среду разработки IAR Workbench for ARM. Компания IAR бесплатно предлагает для ознакомления две версии своего продукта: версию evolution с полным функционалом и ограничением времени использования 30 дней и версию kickstart (в имени дистрибутива есть буквы KS) с ограничением на размер генерируемого исполняемого кода -32 кбайт), но без ограничения

времени использования.

Еще каких-то 10-15 лет назад для создания простейших программ, минимально необходимым набором инструментального ПО являлись: текстовый редактор, транслятор ассемблерного кода и симуляторы для отладки. С развитием микропроцессоров, с ростом объема оперативной памяти и памяти программ и широчайшим распространением МК в различных областях техники, а также требований к надежности и качеству разрабатываемого программного обеспечения минимального набора стало не хватать.

Для создания качественных программ и повторного использования уже отлаженного кода, в виде библиотек, появились редакторы связей (линковщики, компоновщики), появились отладчики, и более совершенные трансляторы и, наконец, стало возможным и обоснованным применение компиляторов (примерно с середины 90-х прошлого века), появился диалект Embedded C\С++. И все эти средства для удобства использования стали объединять в один программный продукт - так появились интегрированные среды разработки (IDE) и целая отрасль разработки ПО. Одними из лидеров в этой области являются фирмы IAR Systems."

Для студенческих нужд размера кода в 32КБ более чем достаточно. В курсе мы будем использовать IAR Embedded Workbench for ARM ver 8.40. Состав этого инструмента показан на Рисунке [\[IAR Workbench\]](#).

## **2.1. Состав интеграционной среды разработки IAR Workbench**

Процесс разработки программного обеспечения в общем случае ничем не отличается от процесса разработки приложения для обычных компьютеров, который включает в себя проектирование (Design), разработка кода(Develop), отладка(Debug)

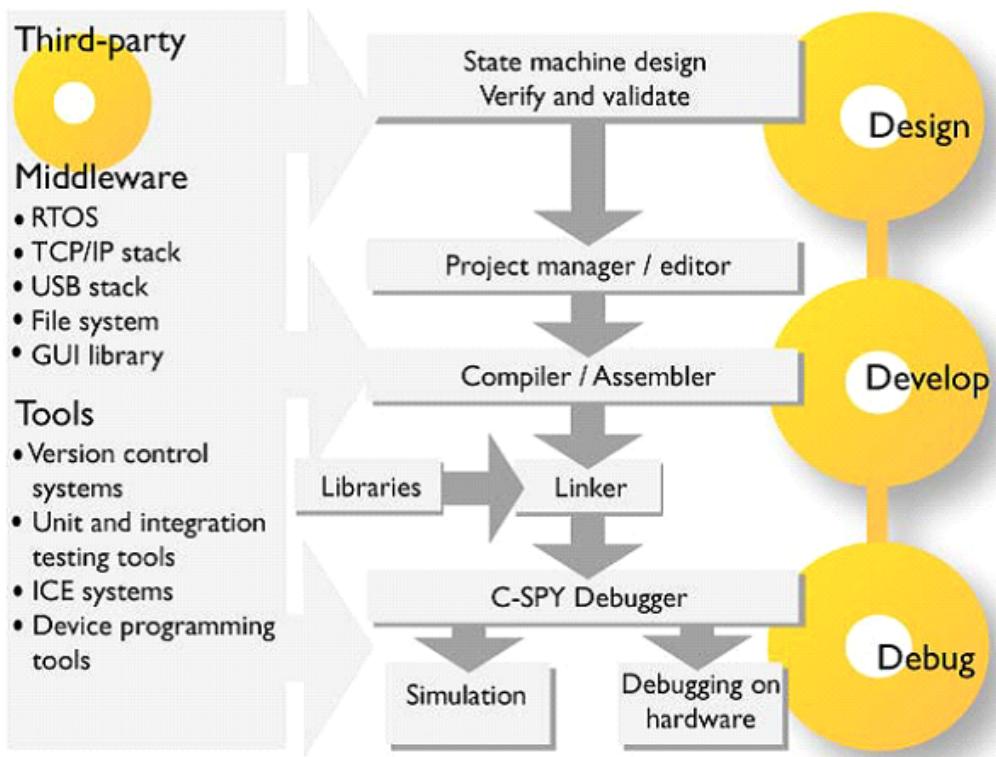


Рисунок 1. Процесс разработки с точки зрения IAR Workbench

## 2.2. Процесс создания исполняемого образа

Процесс преобразования кода на языке программирования высокого уровня C++ в файл, содержащий образ исполняемой программы, готовый для прошивки в микроконтроллер можно разделить на два этапа:

- Трансляция кода в объектный файл
- Компоновка кода в исполнительный файл

## 2.3. Трансляция кода

Трансляцию кода выполняет компилятор. Структурно процесс трансляции с помощью компилятора показан на рисунке [Схема Трансляции]. После трансляции вы можете получить на выходе либо файлы библиотеки, которые впоследствии можно будет использовать в других проектах, либо объектные файлы.

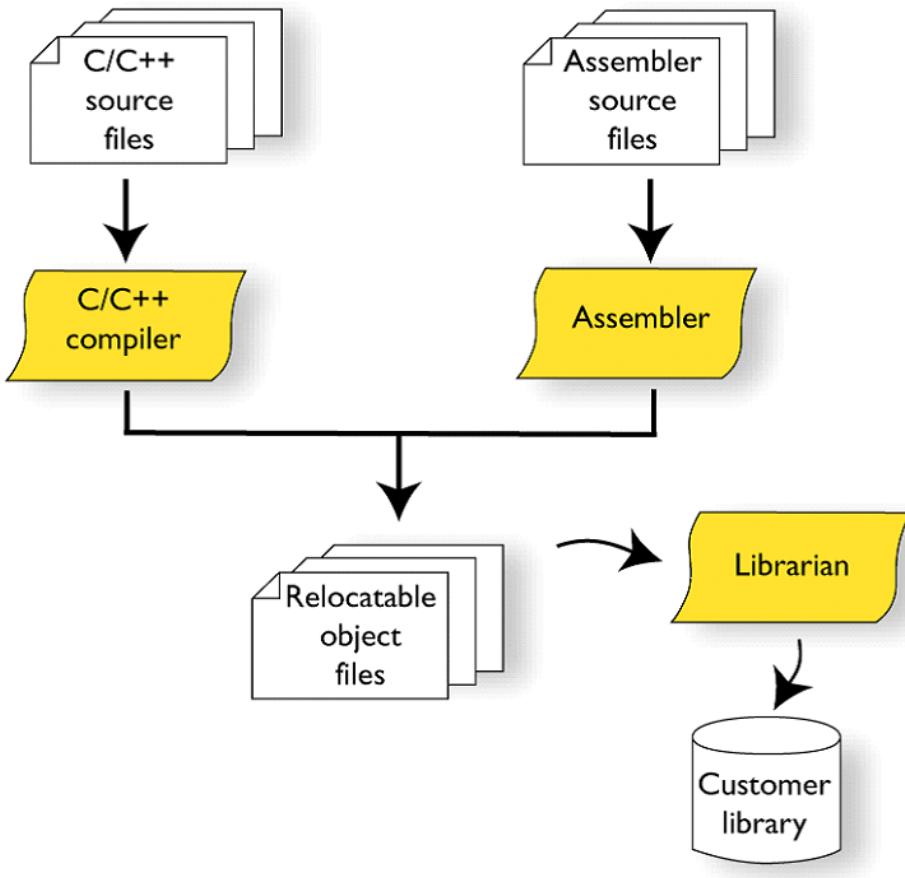


Рисунок 2. Схема процесса трансляции

## 2.4. Компоновка кода

Компоновку кода выполняет линковщик. Структурно процесс компоновки с помощью линковщика показан на [\[Схема компоновки\]](#).

На входе линковщика могут быть, внешние библиотеки, полученные на этапе трансляции в других проектах и программах, объектные файлы полученные на предыдущем этапе, стандартные(встроенные) библиотеки C++, и конфигурационный файл, описывающий настройки по размещению кода и данных в адресном пространстве микроконтроллера. Компоновщик создает исполняемый файл, который можно запустить на микроконтроллере

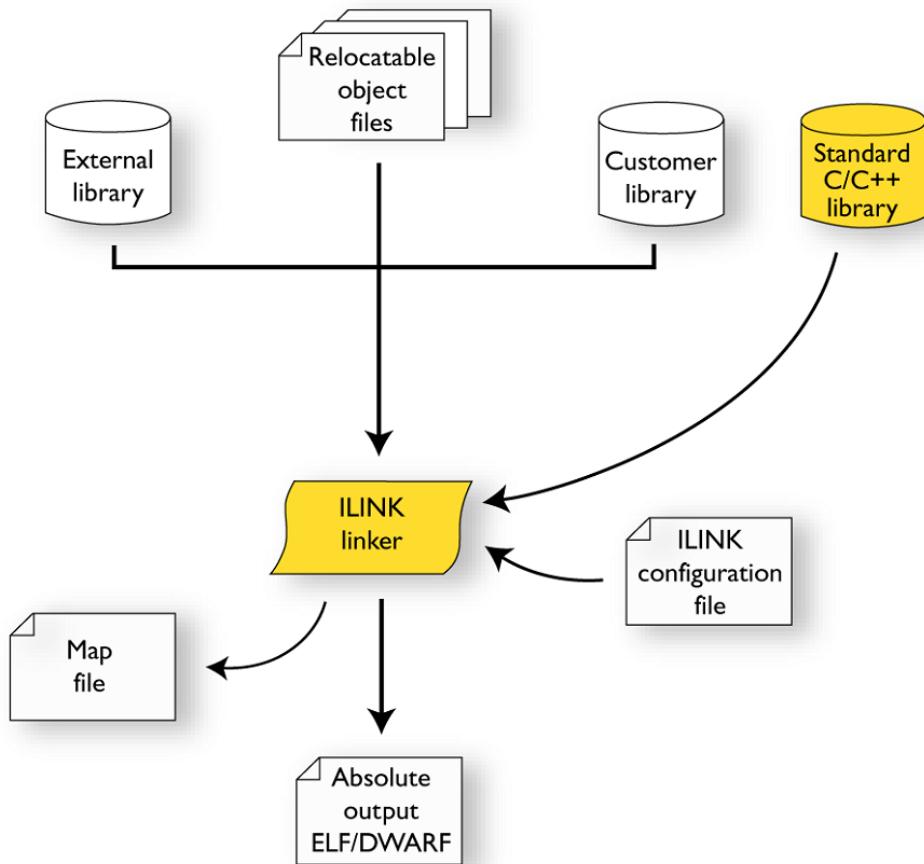


Рисунок 3. Схема процесса компоновки

## 2.5. Запуск и отладка

Последний этап, показанный на рисунке [IAR Workbench](#) - отладка. Компоновщик IAR создает файл в формате ELF, который содержит исполняемый образ программы. Этот файл может быть использован для:

- Загрузки в систему отладки IAR-CSPY или в любой другой отладчик, например GDB, способный читать ELF формат
- Загрузки образа в ПЗУ микроконтроллера используя программатор.

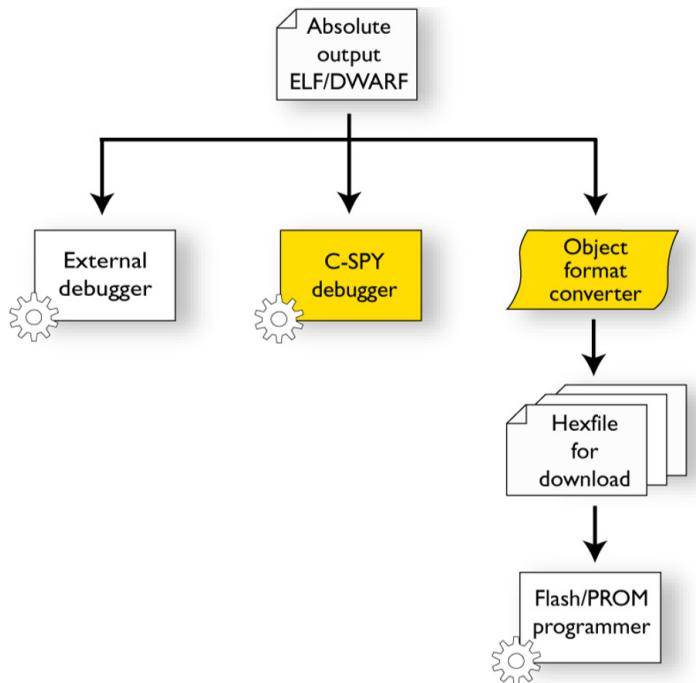


Рисунок 4. Возможные варианты использования выходного файла

### 3. Запуск программного обеспечения

Функция `int main()` является точкой входа программы, для пользователя программа начинается с вызова этой функции и выполнения тела этой функции. Однако на самом деле, еще до функции `main()` микроконтроллер выполняет множество различных действий, например, инициализацию стека, глобальных переменных, констант.

#### 3.1. Инициализация стека

Сразу после подачи питания происходит инициализации указателя стека на конечный адрес стека.

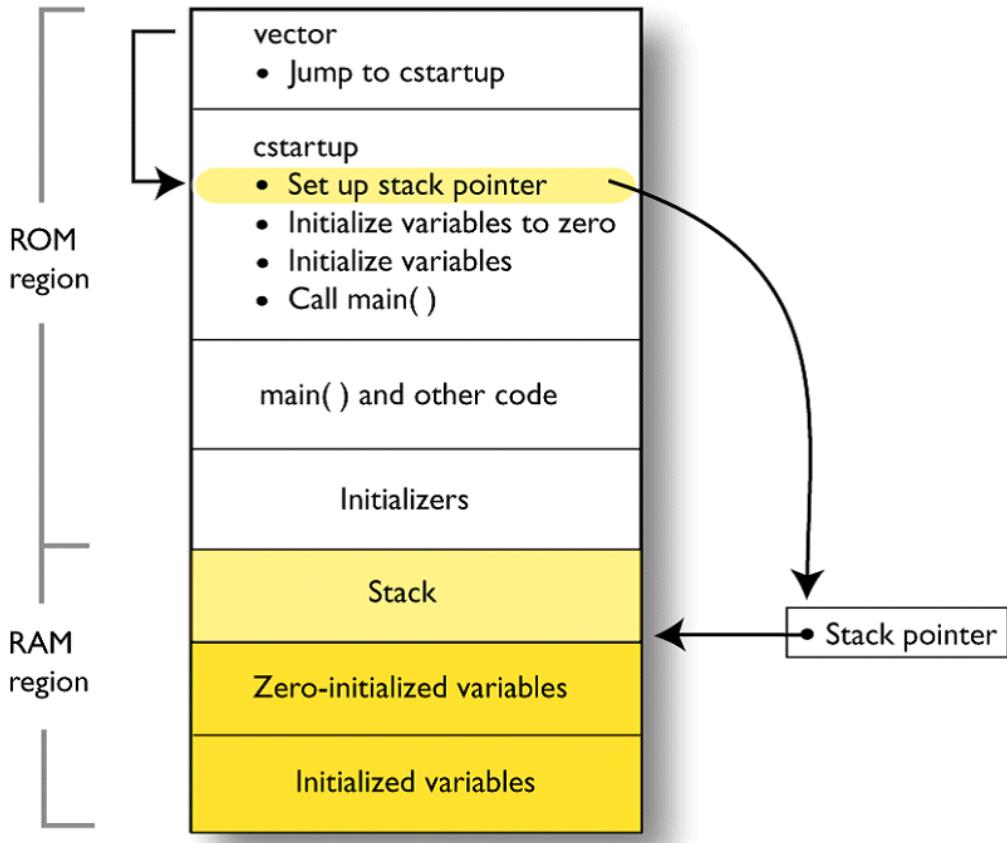


Рисунок 5. Стадия инициализации стека

### 3.2. Инициализация переменных в нулевые значения

После подачи питания на микроконтроллер, регистр адреса команды указывает на 0 адрес, микроконтроллер начинает работу с адреса 0. По адресу 0, находится таблица векторов перываний, по начальному вектору находится команда инициализации указателя стека на конечный адрес стека и далее перехода на функцию инициализации.

После подачи питания и инициализации стека, выполняется функция инициализации памяти нулями (данные указанные как zero-initialized data, непроинциализированные глобальные переменные, такие как int i;)

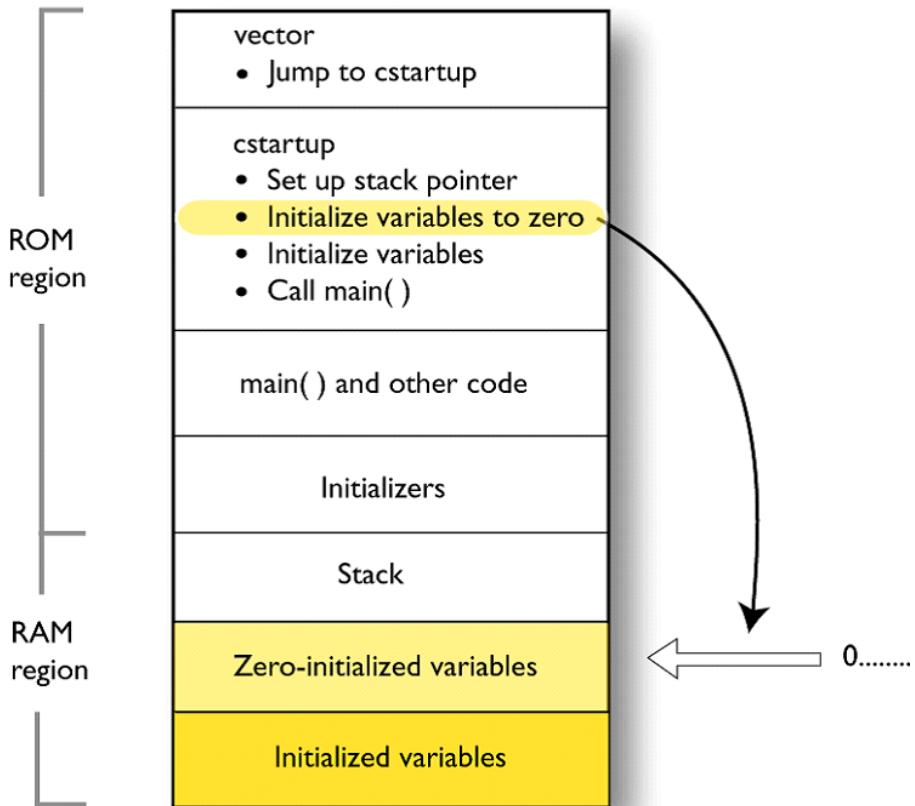


Рисунок 6. Стадия инициализации непроинициализированных переменных

### 3.3. Инициализация переменных

Далее должна произойти инициализация данных определенных как initialized data, например `int i = 6`. Значения инициализации для каждой переменной будут скопированы из ПЗУ в ОЗУ.

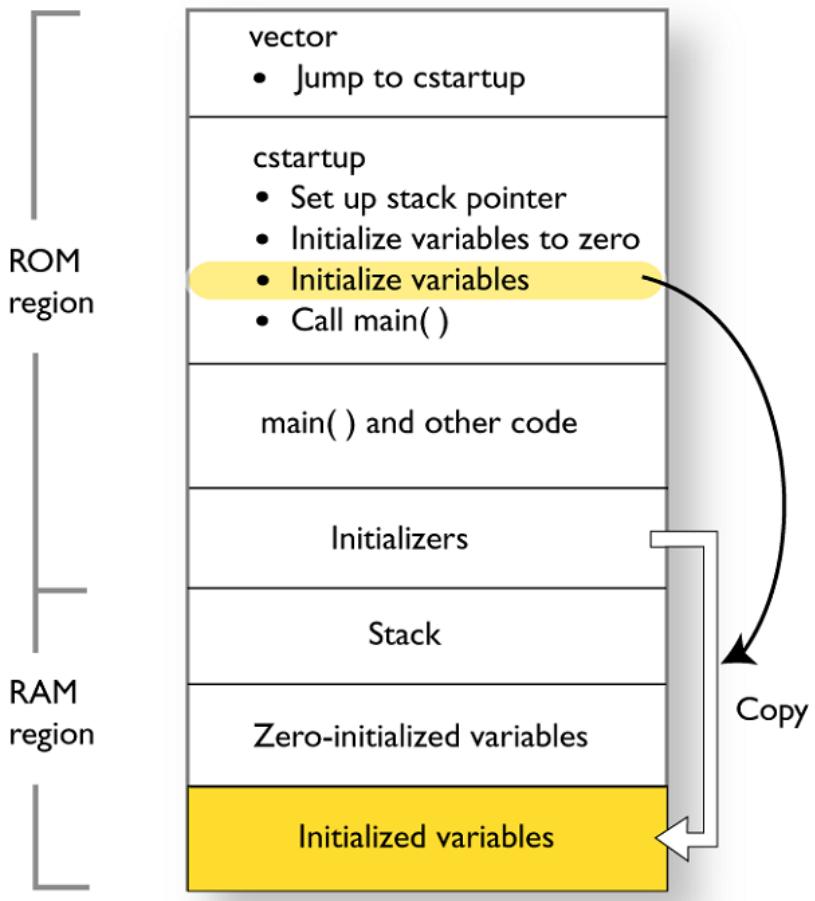


Рисунок 7. Стадия инициализации проинициализированных переменных

### 3.4. Запуск функции main()

Завершающий этап – это вызов функции main().

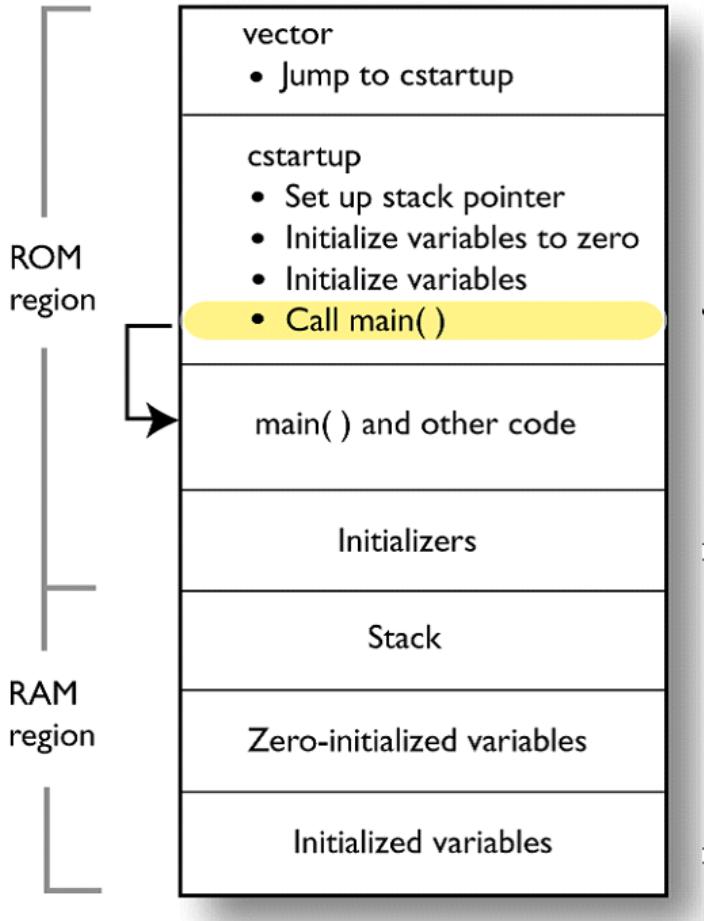


Рисунок 8. Стадия запуска функции `main()`

Как видно, перед тем как запуститься функция `main` необходимо выполнить инициализацию стека и переменных, кроме того, если в вашем проекте будут использоваться прерывания, то в таблицу векторов прерываний необходимо добавить переходы на адреса обработчиков ваших прерываний.

### 3.5. Преимущества IAR Embedded Workbench

За последние время в среде разработки IAR Embedded был сделан огромный скачок с точки зрения удобства использования, так и с точки зрения поддержки современных стандартов. Так версия 8.X получила поддержку стандарта C14, а начиная с версии 8.40 и поддержку стандарт C17 и это является огромным плюсом для разработки надежного, понятного и качественно ПО. Свои мысли по этому поводу я озвучил в статье [\[Можно ли использовать C++ вместо Си для небольших проектов в микроконтроллерах:\]](#)

Некоторые характеристики среды вы можете получить из Таблицы - [\[Характеристики IAR Embedded Workbench\]](#), данные взяты из [\[IAR C/C++ Development Guide\]](#)

Таблица 1. Характеристики IAR Embedded Workbench

Характеристика	IAR Embedded Workbench
Языки	C/C++
Стандарты языка	C++ 17 начиная с версии 8.40

Характеристика	IAR Embedded Workbench
Оптимизация кода	Да, кроме condition_variable, future, mutex, shared_mutex, thread, поддержка atomic урезана и реализована только для типов для которых есть аппаратная поддержка atomic специальными командами в микроконтроллерах
Контроль размера стека	Да
Поддержка RTOS	Да
Статический анализатор кода с набором правил	Да - MISRAC++2008, SECURITY,CERT, STDCHECKS
Динамический анализ кода	C-RUN
Сертификация и проверка соответсвию стандартам безопасности	Сертификация на безопасность по стандартам IEC 61508 и ISO 26262 экспертной организацией TUV SUD – SIL3 сертификат
Поддержка микроконтроллера STM32F411 RE	Полная

### 3.6. Вопросы по разделу

1. Дайте определение понятию “Интегрированной среде разработки”

Ответ:

2. Что такое компилятор и чем он отличается от транслятора?

Ответ:

3. Что такое компоновщик и какие функции он выполняет?

Ответ:

4. Почему важен процесс проектирования ПО какие задачи входят в этот процесс?

Ответ:

5. Дорисуйте процесс разработки ПО [\[IAR Workbench\]](#) с учетом итеративности связей в этом процессе

Ответ:

6. Зачем нужна отладка и в каких случаях она применяется? Для чего применяются точки остановки?

Ответ:

7. Какие еще важные IAR workbench можно добавить в таблицу [\[Характеристики IAR\]](#)

Ответ:

### 3.7. Запуск программного обеспечения

### 3.7.1. Файл cstartup.cpp

Действия по инициализации прописываются в файле cstartup. Этот файл может быть написан как на ассемблере, на Си, так и на С+. Поскольку мы будем использовать С+, то и файл будем использовать cstartup.cpp, который будет выглядеть примерно так

```
extern "C" void __iar_program_start(void) ;

class InterruptHandler {
public:
    static void DummyHandler() { for(;;) {} }
};

using tIntFunct = void(*)();
using tIntVectItem = union {tIntFunct __fun; void * __ptr;};
#pragma segment = "CSTACK"
#pragma location = ".intvec"
const tIntVectItem __vector_table[] = {
{ __ptr = __sfe("CSTACK") }, //инициализация стека
__iar_program_start, //переход на адрес функции __iar_program_start

InterruptHandler::DummyHandler,
...
InterruptHandler::DummyHandler,      ////TIM4
};

extern "C" void __cmain(void) ;
extern "C" __weak void __iar_init_core(void) ;
extern "C" __weak void __iar_init_vfp(void) ;

#pragma required = __vector_table
void __iar_program_start(void) {
    __iar_init_core();
    __iar_init_vfp();
    __cmain();
}
```

Немного проясним, что здесь написано, строка:

```
extern "C" void __iar_program_start( void );
```

Описывает прототип функции \_\_iar\_program\_start, которая будет отвечать за инициализацию переменных и запуск функции main(). Реализацию этой функции вы можете увидеть в самом конце файла cstarup.cpp

```
void __iar_program_start( void ) {
    __iar_init_core();
    __iar_init_vfp();
    __cmain();
}
```

Далее идет определение класса с описание одного единственного метода `handler()`. Это метод и будет тем самым обработчиком прерывания который вызовется при срабатывании соответствующего прерывания. Реализация метода проста – бесконечный цикл, т.е. попав в прерывание программа “навсегда” останется в нем:

```
__weak void DummyModule::handler() { for(;;) {} };
```

Это сделано для того, что пока не планируем использовать никаких прерываний, и если все таки каким то образом прерывание сработало, значит, что-то было сделано не так. В дальнейшем в разделе [Прерывания] будет показано, как сделать нужный нам обработчик прерывания, но сейчас мы не будем на этом заострять внимание. Следующий две строки определяют новый тип, который будет использоваться для задания элементов таблицы векторов прерываний:

```
typedef void( *intfunc )( void );
typedef union { intfunc __fun; void * __ptr; } intvec_elem;
```

Как видно этот тип есть объединение двух типов, указателя на функцию типа `void` и указателя на `void`. Это необходимо для того, чтобы правильно интерпретировать элементы таблицы. Ведь начальный вектор прерывания не содержит никакого обработчика, а просто содержит конечный адрес стека, а последующие вектора содержат адреса обработчиков, именно поэтому первый элемент таблицы векторов должен иметь тип указателя на `void`, а последующие указателей на функцию типа `void`. Собственно далее идет и сама таблица лежащая в выделенном для неё сегменте `.intvec`, который задается в настройках линковщика `#pragma location = ".intvec"` Таблица начинается с адреса стека, который также задается сегментом `CSTACK` в настройке линковщика, а следующий элемент таблицы есть адрес функции инициализации переменных, а затем адреса обработчиков для конкретных прерываний.

## 4. Использование C++

- Так же как когда-то Си пробивал себе дорогу в качестве стандарта для встроенного ПО, так и язык C++ уже вполне может заменить Си в этой области.

Язык программирования стандарта C++ и современные компиляторы имеют достаточно средств для того чтобы создавать компактный код и не уступать по эффективности коду, созданному на Си, а благодаря нововведениям быть понятнее и надежнее.. Начиная с версии IAR Workbench 8.40 компилятор поддерживает полезные нововведения стандарта C++17, такие, как например “структурные привязки”, “инициализация в ветвлениях”,

“встроенные переменные”.

- C++ является строго типизированным языком, а значит программы написанные на нем более безопасны, чем программы написанные на Си и меньше вероятность того, что программист допустит ошибку.
- C++ является языком программирования полностью поддерживающий парадигму программирования ООП, которая отлично подходит для разработки программного обеспечения измерительных устройств.

Ведь нужно понимать, что для измерительного устройства нам нужно описать логику работы, интерфейс взаимодействия с пользователем, реализовать расчеты, а не помнить, что для того чтобы считать данные с АЦП, нужно вначале его выбрать с помощью сигнала CS, находящегося на порту GPIOA.3 и установить его в единицу. Этим должен заниматься разработчик драйверов.

Большинство драйверов для работы с аппаратурой уже реализованы производителями микроконтроллеров, например, в библиотеках CMSIS и CMSIS\_HAL, ими можно воспользоваться для обращения к функциям доступа к аппаратуре, упростить и ускорить разработку.

Замечаниями по этому поводу может служить то, что эти библиотеки довольно громоздкие и для использования в небольших приложениях вряд ли подойдут, кроме того, не всегда они имеют необходимые сертификаты надежности, а потому при разработке реальных измерительных устройств, применение которых планируется в местах с повышенной безопасностью промышленных предприятий, вряд ли стоит пользоваться этими библиотеками.

Именно поэтому, мы будем использовать C++ от написания драйверов и уровня аппаратуры и до реализации логики работы с пользователем. Начнем же изучение с создания проекта, системы тактирования и небольшой программы мигания светодиодом.

## 4.1. Программа на C++

Как было сказано в разделе [Состав интеграционной среды разработки IAR Workbench](#) первоначально мы должны создать исходные файлы на языке программирования С. В С разделяют два типа файлов:

- Исходный файл (файл с расширением \*.cpp)
- Заголовочный файл (файл с расширением \*.h, \*.hpp)

Заголовочные файлы подключаются с помощью директивы #include и при трансляции просто вставляются в текст \*.cpp файла. Используются они для того, чтобы вынести общие определения, используемые в нескольких \*.cpp файлах в одно место.

Вот так может выглядеть ваша программа:

```
#include "gpioaregisters.hpp" //for Gpioa
#include "rccregisters.hpp"   //for RCC

int main()
{
    RCC::AHB1ENR::GPIOAEN::Enable::Set() ;
    GPIOA::MODER::MODER15::Output::Set() ;
    GPIOA::ODR::ODR15::Enable::Set() ;
    return 0 ;
}
```

## 5. Создание проекта и работа в IAR Workbench

- Создать новый проект Project⇒Create New Project.

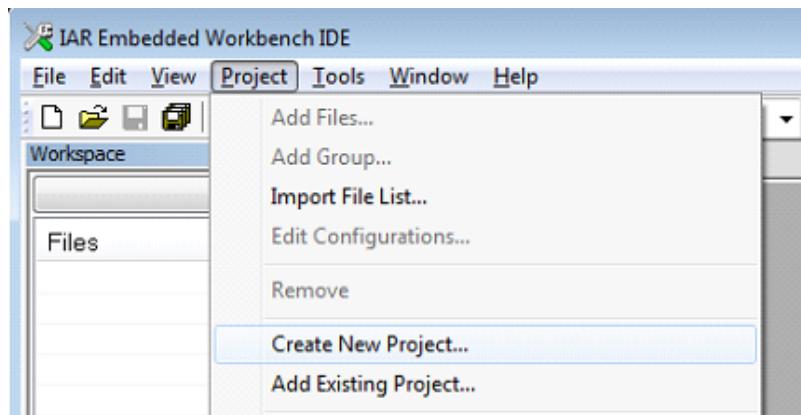


Рисунок 9. Создание нового проекта

### 5.1. Выбор шаблона проекта

- Выбирать шаблон проекта( ProjectTemplates): C++ - main

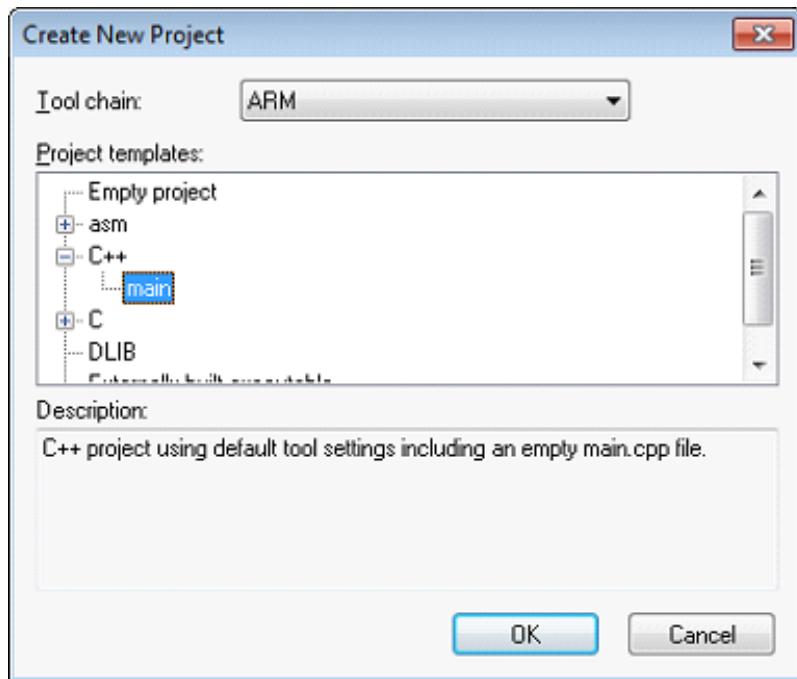


Рисунок 10. Выбор шаблона проекта

## 5.2. Выбор микроконтроллера

- Сохранить проект под каким-либо именем
- В свойствах проекта выбрать модель микроконтроллера ST  $\Rightarrow$  STM32F4  $\Rightarrow$  STM32F411  $\Rightarrow$  ST STM32F411RE см. [Выбор микроконтроллера](#). Для этого правой кнопкой мыши щелкнуть по проекту, выбирать Options и далее в категории General Option выбрать закладку Target.

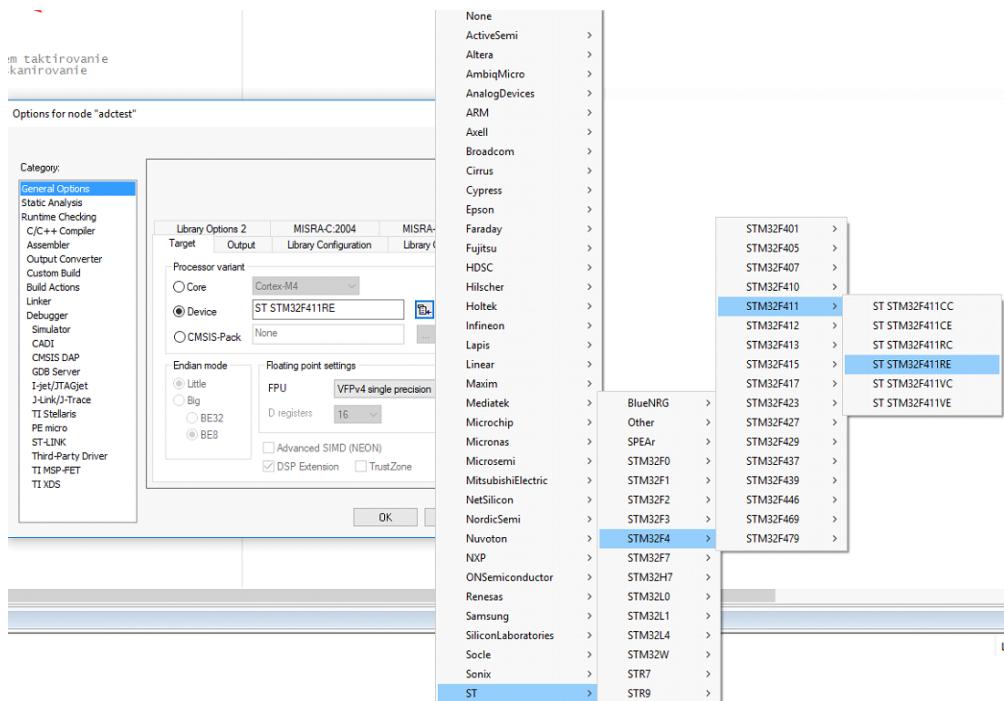


Рисунок 11. Выбор микроконтроллера

## 5.3. Запуск в режиме отладки

После создания проекта необходимо сохранить так называемое рабочее пространство или (workspace).

В рабочее пространство можно загружать несколько проектов (например, проекты всех лабораторных работ) и переключаться между проектами по мере необходимости.

После того, как проект сделан, и имеет вид показанный на [\[Вид созданного проекта\]](#), можно попробовать собрать проект, нажав кнопку Ctrl-F7, а затем загрузить полученный бинарный файл в микропроцессор и запустить на отладку с помощью кнопки Ctrl-D.

Все тоже самое можно сделать и с помощью кнопок быстрого доступа на панели инструментов, через меню среды или контекстное меню проекта (загрузить которое можно нажав на правую клавишу мыши на проекте).

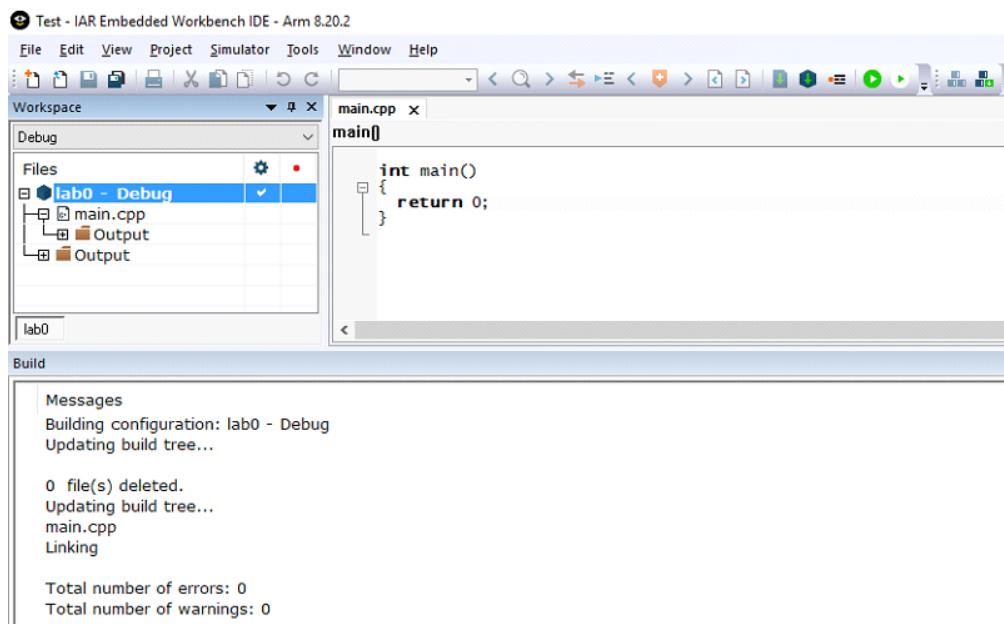


Рисунок 12. Вид созданного проекта

## 5.4. Запуск проекта в режиме симуляции

По умолчанию загрузка и отладка бинарного файла осуществляется в симулятор выбранного микроконтроллера. Поэтому, если вы выполнили все верно, то должно получиться нечто похожее, показанное на [\[Проект в режиме отладки\]](#).

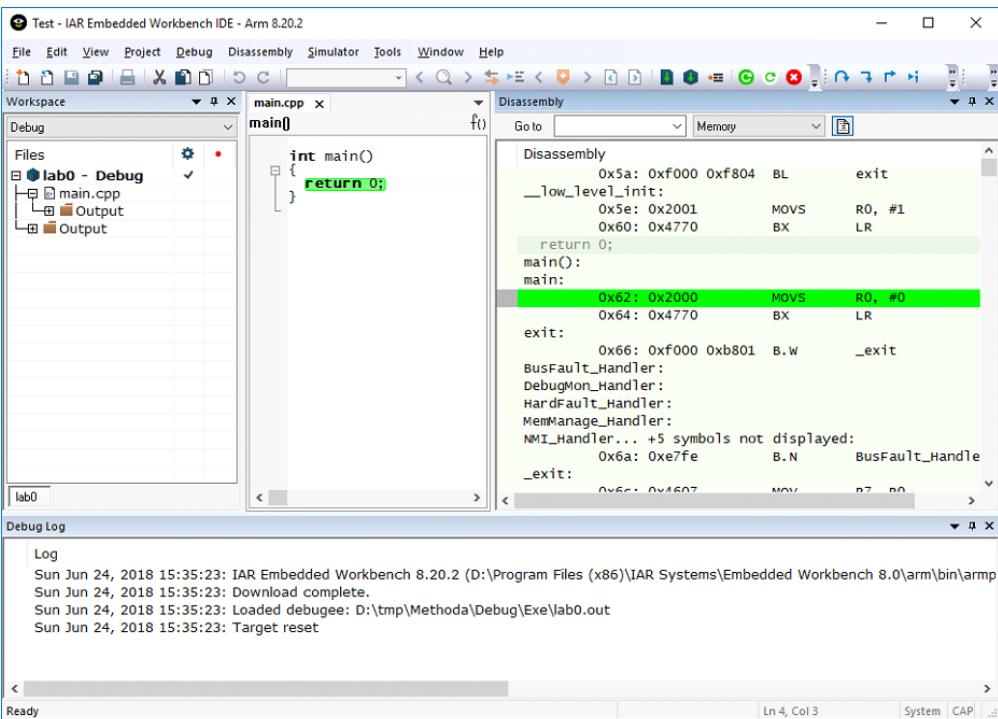


Рисунок 13. Проект в режиме отладки

На этом рисунке вы можете видеть, как сам ваш код написанный на C++, так и окно дизассемблера, показывающее как компилятор преобразовал ваш код в команды ассемблера. Зеленая строчка показывает текущую исполняющую строчку вашего кода и команду ассемблера.

Для того чтобы остановить отладку и выйти в режим разработки необходимо нажать кнопки Ctrl-Shift-D.

## 5.5. Выбор внутрисхемного отладчика

Чтобы загрузить программу в микроконтроллер необходимо вместо симулятора выбрать внутрисхемный отладчик, которым вы пользуетесь. Это можно сделать, встав на проект и нажать на правую кнопку мыши, далее выбрать пункт меню Options⇒Debugger⇒Driver и выбрать в нем нужный вам внутрисхемный отладчик, см [Выбор внутрисхемного отладчика](#). Мы будем использовать отладчик ST-Link.

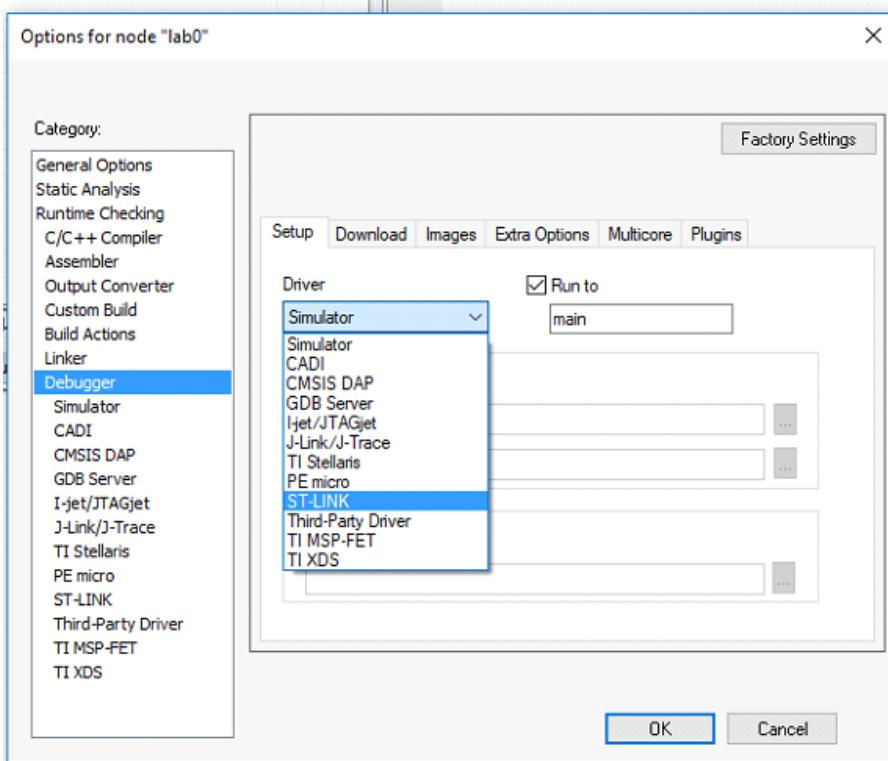


Рисунок 14. Выбор внутрисхемного отладчика

Теперь, если вы нажмете Ctrl-D, ваша программа загрузиться в микроконтроллер и отладка будет осуществляться непосредственно на ядре микроконтроллера. И так вы смогли сделать проект, откомпилировать пустую программу и загрузить её в симулятор и микроконтроллер, но всех этих действий недостаточно, для того, чтобы начать разрабатывать программное обеспечение. Рассмотрим, что же еще необходимо сделать для того, чтобы наш проект был полностью готов.

## 6. Структура проекта

Для того, чтобы разработка была быстрой и качественной, необходимо структурировать паку проекта.

- Не нужно писать весь код в одном файле. Лучше каждый класс описывать в отдельном файле
- Файлы с классами, ответственные за один компонент, лучше держать в папках с именем этого компонента
- Не превращаем проект в мусорку

### 6.1. Добавление файла (cstartup.cpp) в проект

В папку где вы сохранили проекта, необходимо скопировать файл cstartup.cpp. и добавить его к проекту: Для этого нужно нажать правую кнопку мыши на проекте и выбрав пункт Add⇒Add Files... как показано на [Добавление нового файла в проект](#), а затем выбрать файл startup\_stm32F411.cpp.

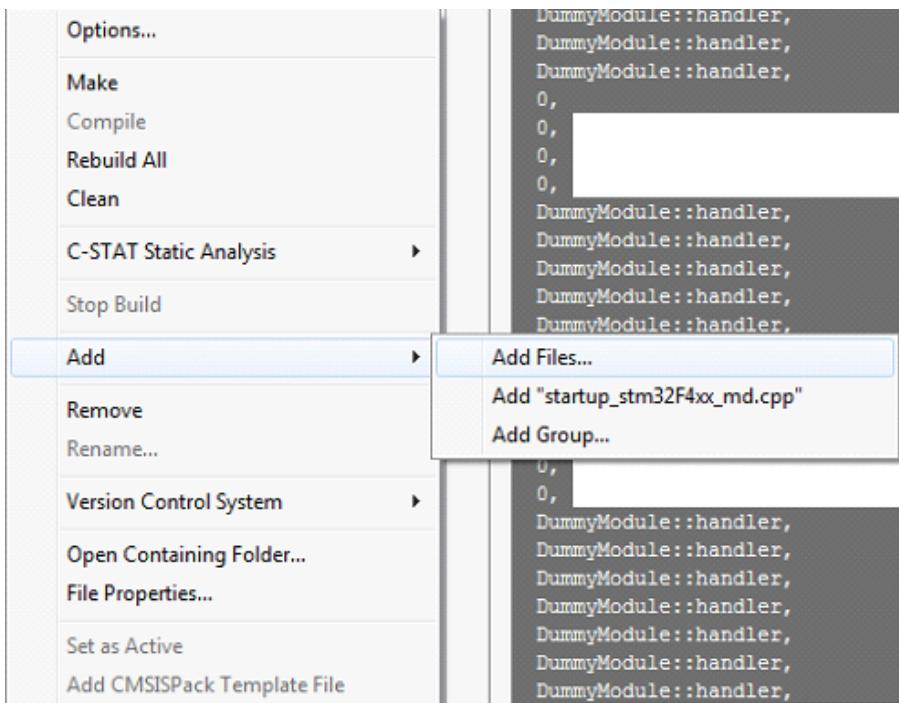


Рисунок 15. Добавление нового файла в проект

Как было сказано выше, в файле cstartup.cpp описывается таблица векторов прерываний и начальная инициализация. Поэтому первым делом нужно подключить файл cstartuo.cpp в проект. Тут следует иметь ввиду, что таблица векторов прерываний для разных микроконтроллеров разная, и соответственно файлы cstartup должен быть различных для разных микроконтроллеров. Чтобы не перепутать свой файлы для разных микроконтроллеров, назовем его startup\_stm32F411.cpp и подключим к проекту, нажав правую кнопку мыши на проекте и выбрав пункт Add⇒Add Files... (см. Рисунок 21 ), а затем выбрав файл startup\_stm32F411.cpp.

## 6.2. Начальная структура проекта

Добавив файл в проект у вас должно получиться следующая структура в среде IAR Workbench:

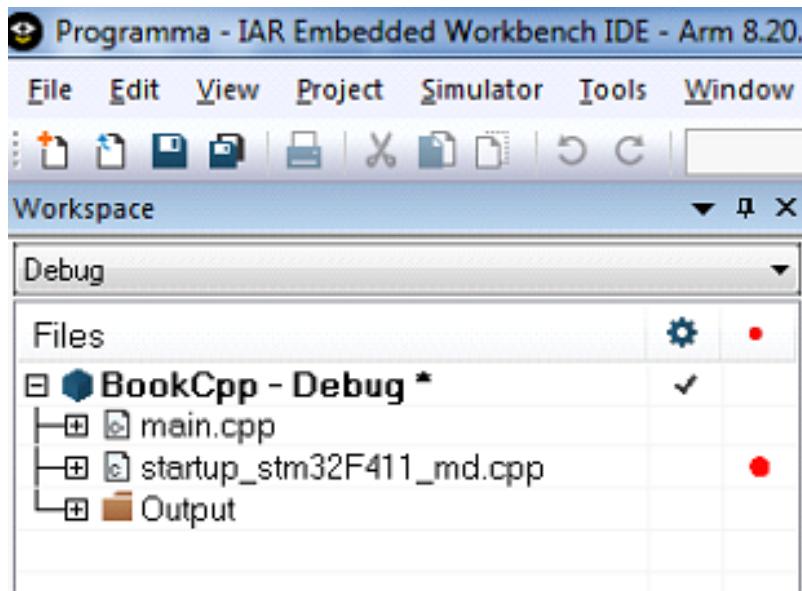


Рисунок 16. Начальная структура проекта

### 6.3. Доступ к папке проекта

Теперь нужно разобраться с тем как будет организован наш проект на диске и в системе контроля версий. Если мы нажмем правой мышкой на проекте и выберем пункт Open Containing Folder см. [Открытие папки проекта], то мы попадем в папку нашего проекта.

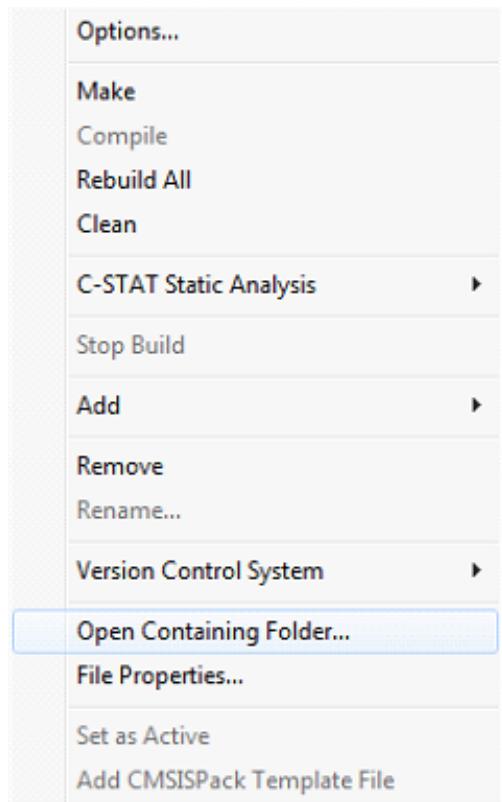


Рисунок 17. Открытие папки проекта

Мы увидим что у нас есть папки Debug и Settings, а также созданные нами файлы проекта, файлы рабочей области, main.cpp и startup\_stm32F411.cpp. В паке Debug хранятся объектные файлы, двоичные файлы для прошивки, листинг программы, созданные в режиме Debug

(т.е. в режиме, когда в программу добавляется некая служебная информация и функциональность для того, чтобы можно было поддерживать внутрисхемную отладку. Существует также режим Release, когда двоичный файл содержит только код программы).

В папке Settings хранятся настройки проекта и рабочей области.

## 6.4. Структура папки проекта

□ Нам нужна будет папка AbstractHardware/Registers. В которой находятся файлы с описанием полей регистров. Можно скопировать ее путем клонирования папки проекта преподавателя, набрав в командной строке:

```
git clone https://github.com/lamer0k/CortexLib.git
```

Вы можете скопировать папку преподавателя через Git, используя PowerShell. Для этого, нужно нажав на вашу папку правой кнопкой мыши, удерживая Shift, выбрать меню "Открыть окно PowerShell здесь".

□ В папке AbsstractHardware будут содержаться файлы для работы с регистрами, аппаратурой и периферией.

Папка AbstractHardware содержит зависимую от микроконтроллера часть.

□ Дополнительно создадим еще папку Application, в которой в дальнейшем будут содержаться файлы классов для работы с логикой программы.

Папка Application будет содержать полностью независимую часть, которую можно будет перенести на любую другую платформу и микроконтроллер. А папка AbstractHardware будет содержать модули зависящие от конкретного микроконтроллера.

□ В завершение добавим папку FreeRtos – она пригодиться нам при работе с ОСРВ.

AbstractHardware	20.09.2019 15:34	Папка с файлами
Application	20.09.2019 15:44	Папка с файлами
Common	20.09.2019 15:34	Папка с файлами
FreeRtos	20.09.2019 15:45	Папка с файлами
Tools	20.09.2019 15:34	Папка с файлами
CMakeLists.txt	20.09.2019 15:34	Текстовый докум...
iartproject.dep	20.09.2019 15:34	Файл "DEP"
iartproject.ewd	20.09.2019 15:34	Файл "EWD"
iartproject.ewp	20.09.2019 15:34	Файл "EWP"
iartproject.ewt	20.09.2019 15:34	Файл "EWT"
main.cpp	20.09.2019 15:34	JetBrains CLion
startup.cpp	20.09.2019 15:34	JetBrains CLion
stm32f411xE.icf	20.09.2019 15:34	Файл "ICF"

Рисунок 18. Финальное содержимое папки проекта

## 6.5. Изменение структуры проекта

Теперь необходимо создать точно такую же структуру в проекте IAR Workbench, как и

структура папок. Для этого необходимо нажать правой мышкой на проект, и выбрать меню Add⇒Ggroup и создать группы Abstract\_Hardware, Application, Common, FreeRtos.

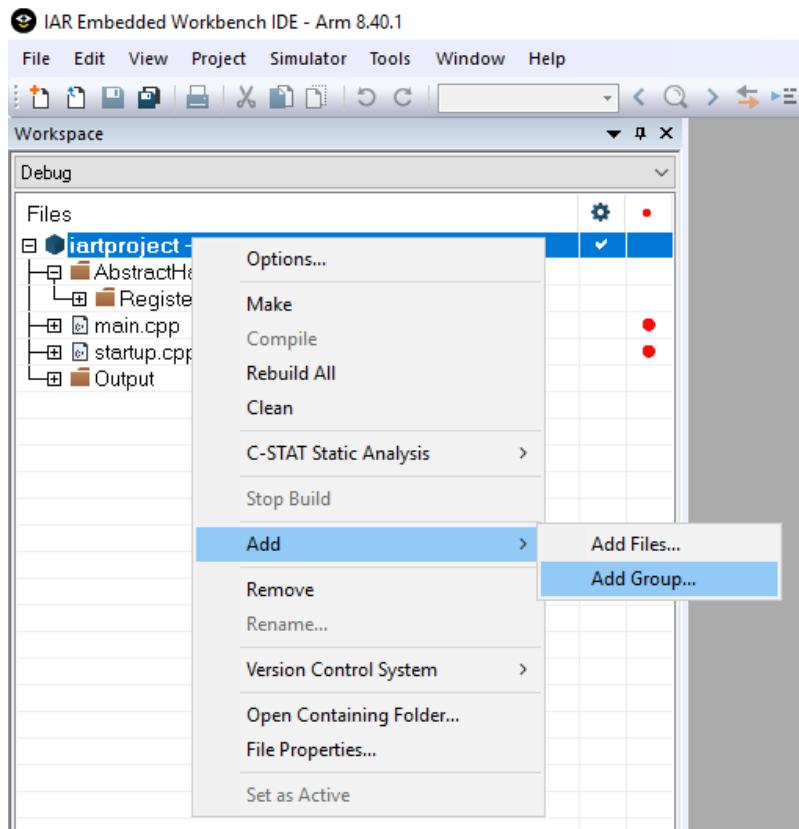


Рисунок 19. Изменение структуры проекта

## 6.6. Финальная структура проекта

В конечном итоге у вас должна появиться вот такая структура:

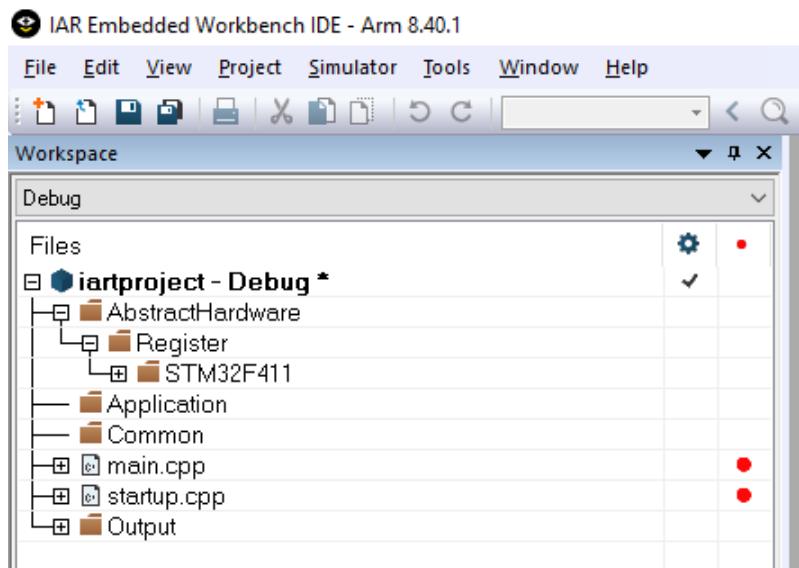


Рисунок 20. Финальная структура проекта

# 7. Окончательная настройка проекта

Для окончательной настройки проекта, нам понадобится настроить компоновщик и установить размер сегментов памяти, стека и кучи.

Перед тем как производить их настройку разберемся, что такие сегменты памяти, стек и куча и для чего они нужны.

## 7.1. Организация памяти

Существует несколько признанных архитектур микропроцессоров \* Архитектура ФонНеймана \* Гарвардская архитектура В традиционных микропроцессорах используется архитектура Фон Неймана (названную так в честь американского математика Джона Фон Неймана), см. [\[Архитектуры микропроцессоров\]](#) А.

Эта архитектура состоит из единого блока памяти, в котором хранятся и команды, и данные, и общей шины для передачи данных и команд в ЦПУ и от него. При такой архитектуре перемножение двух чисел требует по меньшей мере трех циклов: двух циклов для передачи двух чисел в ЦПУ, и одного – для передачи команды. Данная архитектура приемлема в том случае, когда все действия могут выполняться последовательно. По сути говоря, в большинстве компьютеров общего назначения используется сегодня такая архитектура.

Однако для быстрой обработки сигналов больше подходит гарвардская архитектура, см [\[Архитектуры микропроцессоров\]](#) В. Данная архитектура получила свое название в связи с работами, проведенными в Гарвардском университете под руководством Ховарда Айкена. Данные и код программы хранятся в различных блоках памяти и доступ к ним осуществляется через разные шины, как показано на схеме. Т.к. шины работают независимо, выбор команд программы и данных может осуществляться одновременно, повышая таким образом скорость по сравнению со случаем и спользования одной шины в архитектуре Фон Неймана.

На [\[Архитектуры микропроцессоров\]](#) С, представлена модифицированная гарвардская архитектура, где и команды, и данные могут храниться в памяти программ.

ARM является модифицированной гарвардской архитектурой.

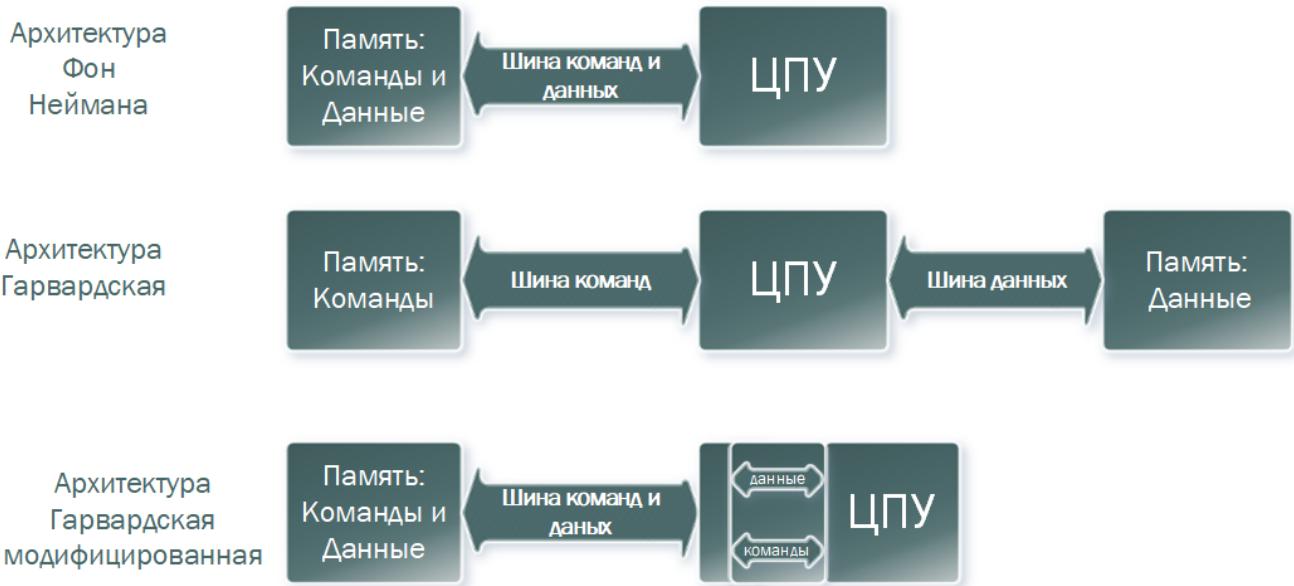


Рисунок 21. Архитектуры микропроцессоров

Доступ к памяти осуществляется по одной шине, а уже устройство управления памятью обеспечивает разделение шин при помощи управляющих сигналов: чтения, записи или выбора области памяти.

Данные и код могут находиться в одной и той же области памяти. В этом едином адресном пространстве может находиться и ПЗУ и ОЗУ и периферия. А это означает, что собственно и код и данные могут попасть хоть куда(в ОЗУ или в ПЗУ) и это зависит только от компилятора и линкера.

## 7.2. Настройка области памяти в компоновщике

Поэтому чтобы различить области памяти для ПЗУ(ROM) и ОЗУ их обычно указывают в настройках линкера.

В настройках линкера IAR 8.40.1 это выглядит вот так:

```
define symbol __ICFEDIT_region_ROM_start__ = 0x08000000;
define symbol __ICFEDIT_region_ROM_end__   = 0x0807FFFF;
define symbol __ICFEDIT_region_RAM_start__ = 0x20000000;
define symbol __ICFEDIT_region_RAM_end__   = 0x2001FFFF;
define region ROM_region      = mem:[from __ICFEDIT_region_ROM_start__ to
__ICFEDIT_region_ROM_end__];
define region RAM_region      = mem:[from __ICFEDIT_region_RAM_start__ to
__ICFEDIT_region_RAM_end__];
```

В данном микроконтроллере диапазон адресом для памяти следующий:

- ОЗУ(RAM) 0x20000000...0x2001FFF,
- ПЗУ(ROM) с 0x008000000...0x0807FFFF.

Вы легко можете поменять начальный адрес ROM\_start на адрес ОЗУ, скажем RAM\_start и

конечный адрес ROM\_end на адрес RAM\_end и ваша программа будет полностью расположена в ОЗУ.

Вы даже можете сделать наоборот и указать ОЗУ в области памяти ROM, и ваша программа успешно сберется и прошьется, правда работать не будет :)

Некоторые микроконтроллеры, такие как, AVR изначально имеют раздельное адресное пространство для памяти программ, памяти данных и периферии и потому там такие фокусы не пройдут, а программа по умолчанию записывается в ROM память.

#### Важно

Все адресное пространство в CortexM единое, и код и данные могут размещаться где угодно. С помощью настроек линкера можно задать регион для адресов ПЗУ(ROM) и ОЗУ(RAM) памяти. IAR располагает сегмент кода .text в регионе ROM памяти.

## 7.3. Объектный файл и сегменты

Выше я упомянул про сегмент кода, давайте разберемся, что это такое.

На каждый компилируемый модуль создается отдельный объектный файл, который содержит следующую информацию:

- Сегменты кода и данных
- Отладочную информацию в формате DWARF
- Таблицу символов

Нас интересуют сегменты кода и данных.

Сегмент это такой элемент, содержащий часть кода или данных, который должен быть помещен по физическому адресу в памяти. Сегмент может содержать несколько фрагментов, обычно один фрагмент на каждую переменную или функцию. Сегмент может быть помещен как в ПЗУ(ROM) так и ОЗУ(RAM).

В общем и целом, сегмент это наименьший линкуемый блок.

## 7.4. Атрибуты сегментов

Каждый сегмент имеет имя и атрибут, который определяет его содержимое. Атрибут используется для определения сегмента в конфигурации для линкера. Например, атрибуты могут быть: \* code — исполняемый код \* readonly — константные переменные \* readwrite — инициализируемые переменные \* zeroinit — инициализируемые нулем переменные

Конечно есть и другие типы сегментов, например сегменты, содержащие отладочную информацию, но нас будут интересовать только те, которые содержат код или данные нашего приложения.

Повторюсь, сегмент это наименьший линкуемый блок. Однако при необходимости линкеру можно указать и еще более мелкие блоки(фрагменты). Этот вариант рассматривать не будем, остановимся на сегментах.

## 7.5. Предопределенные имена сегментов в IAR Workbench

Во время компиляции данные и функции размещаются в различные сегменты. А во время линковки, линкер назначает им реальные физические адреса. В компиляторе IAR есть предопределенные имена сегментов, некоторые из них приведены ниже:

- .bss — Содержит статические и глобальные переменные инициализируемые 0
- .CSTACK — Содержит стек используемый программой
- .data — Содержит статические и глобальные инициализируемые переменные
- .data\_init — Содержит начальные значения для данных в .data секции, если используется директива инициализации для линкера
- .HEAP — Содержит кучу, используемую для размещения динамических данных
- .intvec — Содержит таблицу векторов прерываний
- .rodata — Содержит константные данные
- .text — Содержит код программы

На практике это означает, что если вы определили переменную `int val = 3;`, то сама переменная будет расположена компилятором в сегмент .data и помечена атрибутом `readwrite`, а число 3 может быть помещено либо в сегмент .text, либо в сегмент .rodata или, если применена специальная директива для линкера в .data\_init и также помечается им как `readonly`.

Сегмент .rodata содержит константные данные и включает в себя константные переменные, строки, агрегатные литералы и так далее. И этот сегмент может быть размещена где угодно в памяти.

## 7.6. Файл настройки компоновщика

Файл линкера имеет расширение \*.icf. В нашем проекте этот файл называется `stm32f411xE.icf`. Давайте теперь поймем, что же прописано в настройках линкера и почему.

```

define symbol __ICFEDIT_region_ROM_start__ = 0x08000000;
define symbol __ICFEDIT_region_ROM_end__ = 0x0807FFFF;
define symbol __ICFEDIT_region_RAM_start__ = 0x20000000;
define symbol __ICFEDIT_region_RAM_end__ = 0x2001FFFF;
define region ROM_region = mem:[from __ICFEDIT_region_ROM_start__ to
__ICFEDIT_region_ROM_end__];
define region RAM_region = mem:[from __ICFEDIT_region_RAM_start__ to
__ICFEDIT_region_RAM_end__];

// Разместить сегменты .rodata и .data_init (константы и инициализаторы) в
// (ПЗУ)ROM:
place in ROM_region { readonly };

// Разместить сегменты .data, .bss, .noinit, STACK и HEAP в (ОЗУ)RAM
place in RAM_region { readwrite, block STACK , block HEAP };

```

## 8. Настройка стека

### 8.1. Стек

Для начала определение из Википедии:

Стек (англ. Stack - стопка; читается стэк) - абстрактный тип данных, представляющий собой список элементов, организованных по принципу LIFO (англ. last in — first out, «последним пришёл — первым вышел»).

В стек можно положить данные, и можно данные забрать, причем те данные которые были положены в стек последним, забираем из стека первым

Стек – это организация памяти, выполненная компоновщиком. На уровне микроконтроллера для работы со стеком есть специальные ассемблерные команды (например PUSH – положить регистры в стек, и POP – взять из стека). Так же для сохранения и считывания данных из стека могут использоваться инструкции STR и LDR

Обычно в стеке сохраняются регистры когда вы вызываете подпрограмму, или проваливаетесь в прерывание, для того, чтобы когда вернуться обратно в вашу программу восстановить весь контекст и все переменные. Кроме того, если в вашей функции передается слишком много переменных и под все не хватит регистров, то компилятор расположит их также на стеке. Локальные переменные функции также создаются на стеке.

В традиционной реализации память для всех локальных переменных функции выделяется сразу, одним "кадром стека" в начале работы функции. Внутри этого кадра стека компилятор еще на стадии компиляции разработает некую фиксированную карту расположения локальных переменных. При этом он может (и будет) располагать локальные переменные в этой карте совершенно произвольным образом, руководствуясь

оптимизационными соображениями выравнивания, экономии памяти и т.д. и т.п.

## 8.2. Правила задания размера стека

В большинстве "традиционных" платформ стек растет сверху-вниз: от старших адресов к младшим. Поэтому прежде всего нужно верно указать размер или вершину стека. Для того, чтобы сделать это есть пара правил:

1. Всегда считаем, что все локальные переменные создаются на стеке (Хотя часть из них могут быть созданы и на регистрах)
2. У нас 16 регистров + регистры блока с плавающей точкой. Которые должны быть сохранены на стеке
3. Каждая вложенная подпрограмма должна сохранить на стеке все данные из пункта 1 и 2. Т.е. если вложенность будет 2, то и сохранять придется примерно в два раза больше данных
4. Каждое прерывание должно сохранить данные из пункта 1 и 2.

## 8.3. Установка размера стека

Обычно размер стека вычисляется эмпирически и задается с небольшим запасом.

Чтобы задать размер стека, нужно нажав на правую кнопку мыши на проекте, выбрать Option⇒Linker и нажать кнопку Edit, далее выбрать закладку Stack/Heap Size, см. [Установка размера стека и кучи]

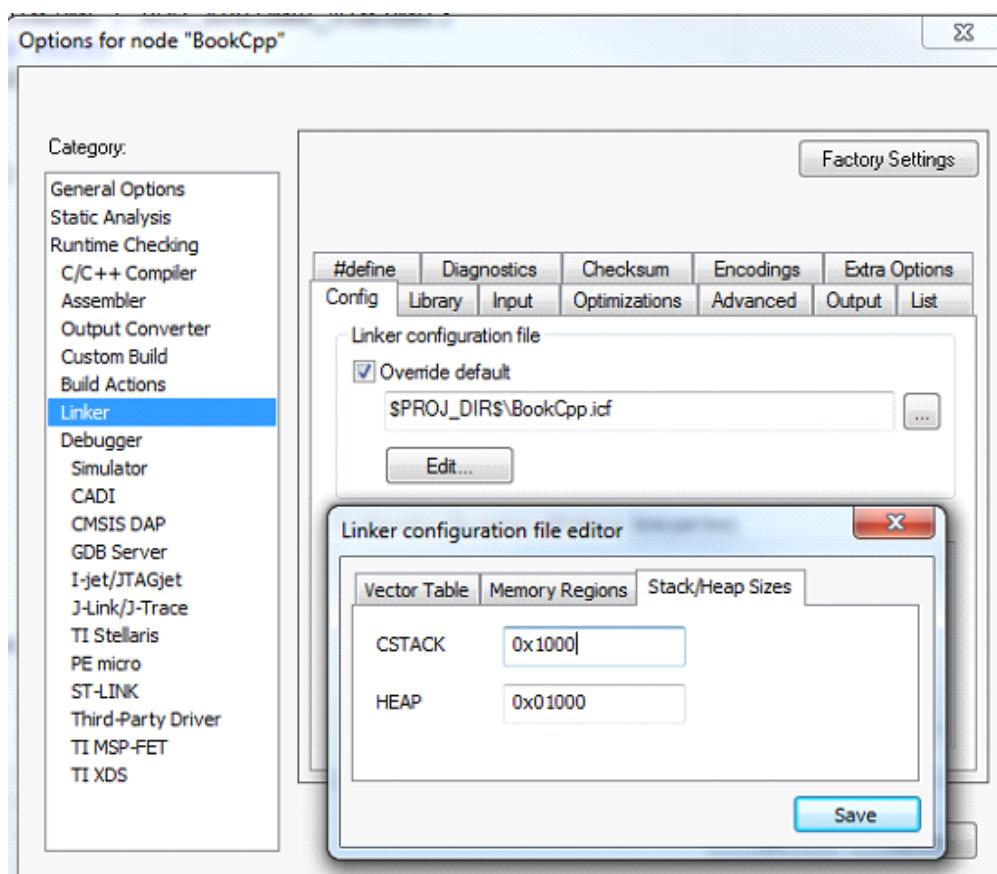


Рисунок 22. Установка размера стека и кучи

Тоже самое можно сделать руками в файле `stm32f411xE.icf`, поменяв значение символа `_ICFEDIT_size_cstack`

## 8.4. Контроль за размером стеком

IAR Workbench имеет встроенные средства для контроля стека на этапе сборки он может указать максимально возможный размер стека для вашего приложения для самой глубокой цепочки вызова функций.

Это значение можно использовать как ориентир при установке максимального значения стека. Однако следует помнить, что во-первых, в вашей программе возможно никогда не будет самой глубокой цепочки вложенности, а во вторых не всегда компоновщик сможет определить верно размер, например, при использовании ОСРВ, указатель стека постоянно изменяется и стек выделяется под каждую задачу отдельно, в итоге вся программа может работать вообще без единого стека и его размер можно минимальным. Зато придется указывать размер стека для каждой задачи при её создании. В любом случае, очень полезно знать об этой особенности и как её задействовать.

Для включения достаточно поставить галочку в меню `Option⇒Linker⇒Advanced⇒Enable stack usage analysis` см. [Опция анализа глубины стека]

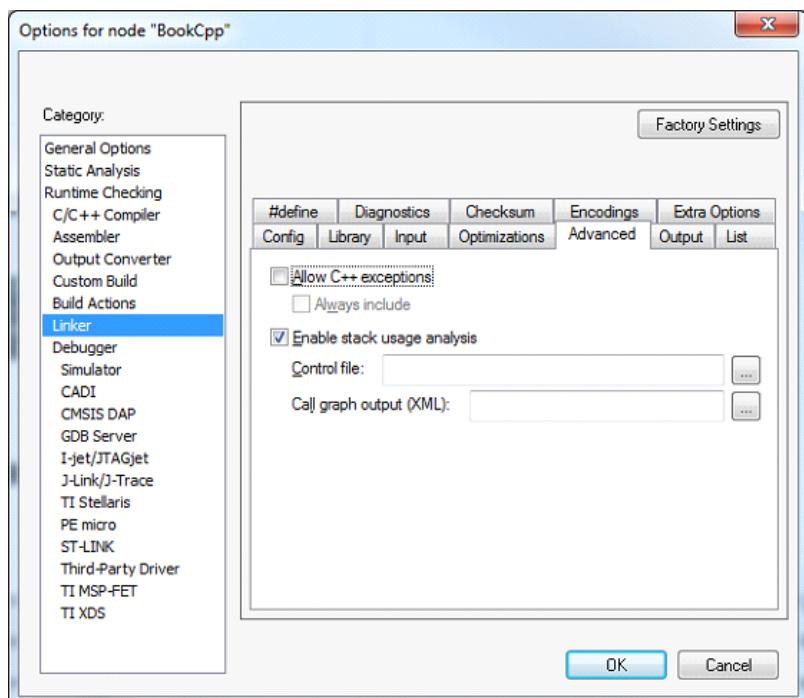


Рисунок 23. Опция анализа глубины стека

## 8.5. Доступ к данным по анализу размеру стека

После установки этой опции на выходе компоновщика в файле с расширением \*.map можно будет увидеть результат анализа, например, такой:

Call Graph Root Category	Max Use	Total Use
Program entry	896	896
Uncalled function	0	0
<b>Program entry</b>		
"__iar_program_start": 0x08005291		
<b>Maximum call chain</b>	896	bytes
"__iar_program_start"	8	
"__cmain"	0	
"main"	88	
"std::ostream::operator <<(float)"	80	
"std::numpunct<char>::grouping() const"	8	
"std::numpunct<char>::do_grouping() const"	8	
"std::string::basic_string(char const *)"	16	
"std::string::assign(char const *)"	16	
"std::string::assign(char const *, unsigned int)"	16	
"std::string::assign(const std::string&, unsigned int, unsigned int)"	32	
"std::string::_Grow(unsigned int, bool)"	16	
"std::string::_Copy(unsigned int, unsigned int)"	32	

В данном случае анализ стека показывает, что размер стека при максимальной цепочке вложенности может быть 896 байт.

## 8.6. Куча

Куча (англ. heap) - структура данных с помощью которой организуется динамически распределение памяти приложения. Размер кучи — размер памяти, выделенной операционной системой (ОС) для хранения кучи (под кучу).

Компоновщик выделяет раздел памяти под кучу в соответствии с заданным размером кучи, а при запуске программы происходит инициализация кучи, в ходе которой память, выделенная под кучу, отмечается как свободная.

Куча используется только при динамически выделяемой памяти, для нас это означает, что все объекты созданные с помощью оператора new будут расположены в куче.

Механизм выделения памяти описывать не будем, просто нужно запомнить, что если объект создан с помощью оператора new, то все его содержимое хранится в куче.

Я не советую использовать динамическое создание объектов. Так как динамическое выделение памяти не рекомендуется для использования в надежном ПО. Лучше делать все объекты статическими.

## 8.7. Определение размера кучи

Как определить размер кучи, необходимой под кучу. Можно вооружиться некоторыми правилами:

- Чтобы узнать размер объекта в куче, можно воспользоваться оператором `sizeof`, который может вернуть вам размер в байтах типа объекта (собственно, он будет равен размеру объекта расположенному в куче). Таким образом узнав размер всех объектов, можно приблизительно вычислить необходимый размер кучи
- Поскольку на кучи объекты могут как создаваться так и удаляться из неё, то куча может получаться неаргументированной, т.е. между объектами может быть пустая, незаполненная память. Поэтому если вы постоянно создаете и удаляете объекты, нужно учитывать этот факт и брать размер кучи с запасом.
- Размер кучи зависит от алгоритма работы вашей программы, если вы будете создавать и удалять последовательно объекты 100 раз, то нет никакого резона создавать кучу на 100 объектов, вполне разумно, создать кучу под 1-2 объекта с запасом на дефрагментацию – скажем 20% и все.

Как вы поняли использование кучу несет ряд трудностей с расчетом её размера, помимо этого использование кучи может тормозить выполнение программы., см, например, [\[Обзор одной российской RTOS\]](#). Поэтому во встроенном ПО использование кучи не приветствуется, по возможности её надо избегать, однако некоторые архитектурные приемы невозможны без использования динамических объектов (например для позднего связывания, или факта того, что мы не хотим использовать глобальные объекты), поэтому использовать в курсовых вы можете, но с одним условием, в нашем программном обеспечении созданные динамические объекты никогда не должны удаляться. Таким образом мы избежим дефрагментации кучи, а также слежением за памятью.

Для задачи размера кучи, нужно сделать те же действия что для задания размера стека, см. [Установка размера стека](#)

## 9. Задания

3 Задания, кто не успеет в лабораторной, завершить дома.

### 9.1. Задание 1 #Лекция 1 Задание 1

- Создать проект C++ с `main.cpp`
- Подключить к проекту файл `cstartup.cpp`
- Создать папки `AbstractHardware/Registers/FiledValues`, `Common`, `Application`, `FreeRtos`
- Создать структуру проекта в соответствии со структурой папок
- Настроить `STACK`, `HEAP`
- Скопировать содержимое папки `Registers` и `Common` с проекта преподавателя в свою папку

## 7. Написать программу в main.cpp

```
#include "gpiocregisters.hpp" //for GPIOC
#include "rccregisters.hpp"   //for RCC

int main()
{
    RCC::AHB1ENR::GPIOCEN::Enable::Set() ;
    GPIOC::MODER::MODER5::Output::Set() ;
    GPIOC::ODR::ODR5::Enable::Set() ;
    GPIOC::ODR::ODR5::Disable::Set() ;
    return 0 ;
}
```

1. Посмотреть видео: <https://youtu.be/uC0jJGfDxtM>

## 9.2. Задание 2

1. Откомпилировать и отлинковать программу
2. Загрузить программу в симуляторе
3. Сделать пошаговую отладку
4. Настроить Debugger на отладку через StLink
5. Подключить плату к компьютеру
6. Загрузить программу в плату
7. Выполнить пошаговую отладку
8. Описать полученный результат
9. Посмотреть видео: <https://youtu.be/c7CasTJKw7o>

## 9.3. Задание 3

1. Запустить анализатор стека. Узнать рекомендуемый размер стека.
2. Изменить в проекте размер стека на рекомендуемый
3. Создать тар файл
4. Описать что написано в тар файле
5. Поставить размер кучи НЕАР в 0. Объяснить почему так можно сделать. И почему STACK нельзя
6. Добавить проект в Git и сделать синхронизацию с GitHub
7. Сделать отчет по каждому пункту каждого задания в файле .adoc. Выложить файл в GitHub
8. Прислать ссылку на GitHub преподавателю для проверки
9. Посмотреть видео: <https://youtu.be/TajLTcjBgIg>

# Лекция 2

## 1. Портируемость проекта

Для того, чтобы ваш проект мог хорошо портироваться на другие типы микроконтроллеров мы должны принять некоторые меры.

- Применять одни и те же типы данных, имеющие один и тот же размер
- Разделять часть, которая отвечает за аппаратуру и аппаратные модули, зависящую от микроконтроллера и бизнес логику, которая не зависит от аппаратуры
- Использовать разделение реализации и интерфейсов

Сейчас нам важны типы данных.

### 1.1. Типы данных

Одно из главных правил портируемости состоит в том, что для разных ядер микроконтроллеров один и тот же тип переменной имел одинаковый размер. Для этого давайте разберемся, что такое тип и почему он может иметь разную длину? Для нашего микроконтроллера компилятор поддерживает следующие типы, см [Встроенные типы C++].

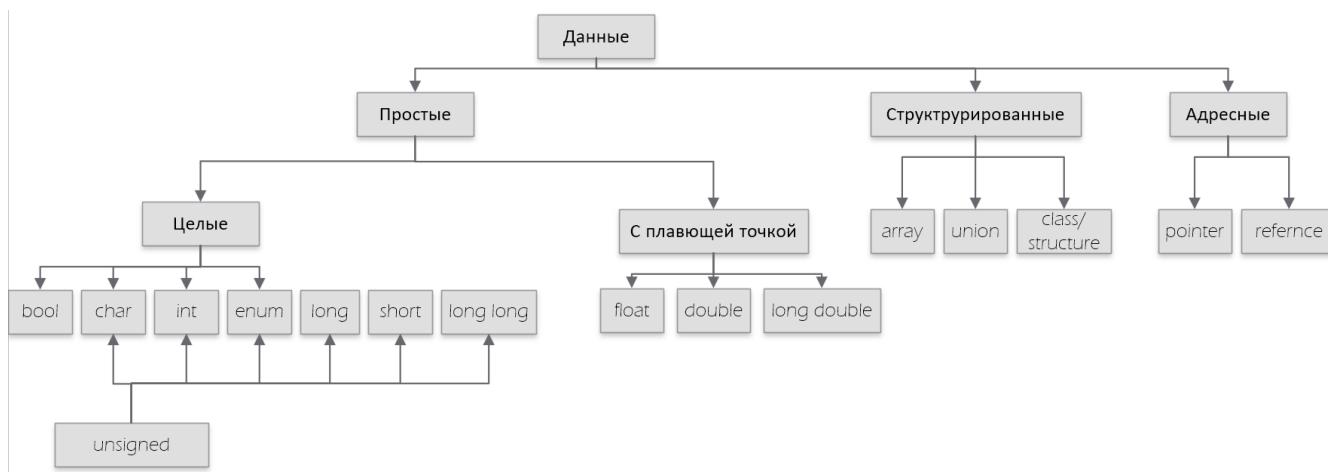


Рисунок 24. Типы данных в C++

### 1.2. Встроенные типы

Таблица 2. Встроенные типы C++

Тип	Длина	Комментарий
<b>bool</b>	1	Представляет значения, которые могут быть или <b>true</b> , или <b>false</b> .
<b>char</b>	1	Используется для символов ASCII в старых строках в стиле C или в объектах <code>std::string</code> , которые никогда не будут преобразовываться в Юникод.

Тип	Длина	Комментарий
<b>unsigned char</b>	1	Аналог байта. В C++17 стандарте появился тип std::byte
<b>int</b>	4	Целочисленное значение. Выбор по умолчанию для целых чисел
<b>unsigned int</b>	4	Беззнаковое целое число
<b>float</b>	4	Число с плавающей точкой, поддерживается аппаратно некоторыми микроконтроллерами
<b>double</b>	8	Число с плавающей запятой двойной точности. Выбор по умолчанию для значений с плавающей точкой

## 1.3. Модификаторы типов данных

Таблица 3. Встроенные типы C++ модификаторы

Тип	Длина	Комментарий
<b>short int</b>	2	Целочисленное знаковое значение укороченной длины
<b>unsigned short int</b>	2	Целочисленное беззнаковое значение укороченной длины
<b>long int</b>	8	Выбор по умолчанию для целочисленных значений. На платформах на которых int равен по длине unsigned short int может быть длиннее int
<b>unsigned long int</b>	8	Целое число двойной длины. На платформах на которых int равен по длине unsigned short int может быть длиннее int
<b>long double</b>	8	Число с плавающей точкой двойной точности с двойной точностью

## 1.4. Размеры типов данных

Размеры типов не четко определены и могут отличаться для различных микроконтроллеров. Для размеров типов существует правило:

```

1           <= sizeof(char)      <= sizeof() <= sizeof(short) <= sizeof(int) <=
sizeof(long)
1           <= sizeof(bool)     <= sizeof(long)
sizeof(char) <= sizeof(long)
sizeof(float) <= sizeof(double) <= sizeof(long double)
sizeof(T)    == sizeof(signed T) == sizeof(unsigned T)

```

Поэтому вместо прямых типов типа int, используйте псевдонимы, например:

`std::uint32_t` целое беззнаковое длиной 32 бита

`std::int64_t`      целое знаковое длиной 64 бита

`std::uint8_t`      целое знаковое длиной 8 бит

## 1.5. Пользовательские типы

Вы можете определить свой тип сами, либо сделать псевдоним типа. Любой класс или структура, определенная вами, будет являться вашим типом. Например:

```
template<typename T>
struct Complex
{
    Complex(T r, T im): real{r}, imaginary{im} {} ;
    operator T { return sqrt(real*real + imaginary* imaginary); }
    Complex operator +(Complex value)
    {
        return Complex(real+ value.real, imaginary + value.imaginary);
    }
    private:
    T real; // вещественная часть
    T imaginary // мнимая часть
};

int main()
{
    Complex<float> value1(3.0f, 4.0f);
    Complex<float> value2(1.0f, 2.0f);
    value1 += value2;
    return 0;
}
```

## 1.6. Псевдонимы типов

Для того, чтобы было понятнее работать с типом можно вводить их псевдонимы (alias). С помощью ключевого слова `using` ;

```

auto t = std::make_tuple(10, "Test", 3.14, 2U); ①
using tMyType = decltype(t); ②
using tShortType = std::tuple<int, string, double, tU32>; ③

void(tMyType & value) { ④
    ...
}

int main() {
    using tU32 = unsigned int; ⑤
    tU32 i = 10U; ⑥

    myfunction(t); ⑦
}

```

- ① Определяем кортеж из 4 элементов разного типа
- ② Объявляем псевдоним типа, который имеет кортеж (тип выводится компилятором)
- ③ Тоже самое что и <2> за исключением того, что указываем тип напрямую
- ④ Объявляем функцию, принимающую аргумент типа, который имеет кортеж
- ⑤ Объявляем псевдоним типа unsigned int
- ⑥ Определяем переменную типа unsigned int

## 1.7. Неявное преобразование типов

Базовые/простые типы неявно можно привести друг к другу. Т.е

```

int a = 0; ①

char a = 512; ②

int a = 3.14; ③

bool a = -4; ④

bool a = 0; ⑤

```

- ① Присваиваем знаковое целое(int) число переменной целого типа
- ② Присваиваем знаковое целое(int) число переменной типа char. Результат в a 0 ;
- ③ Присваиваем число с плавающей точкой(double) к переменной типа int. Результат в a 3
- ④ Присваиваем знаковое целое(int) к переменной типа bool. Результат в a true
- ⑤ Присваиваем знаковое целое(int) к переменной типа bool. Результат в a false

## 1.8. Явное преобразование типов

Так как компилятор может сделать за вас, то, что вы вообще не ожидаете, не нужно использовать неявное преобразование типа.

Вместо этого, лучше указать компилятору явное преобразование из одного типа в другой. В этом случае, вы говорите компилятору, что я понимаю, что я делаю, это именно так и задумано

Для преобразований из одного типа используют 4 вариантов преобразования:

- static\_cast
- const\_cast
- reinterpret\_cast
- dynamic\_cast

## 1.9. static\_cast

**static\_cast** позволяет сделать приведение близких типов (целые, пользовательских типов которые могут создаваться из типов который приводится, и указатель на void\* к указателю на любой тип).

Проверка производится на уровне компиляции, так что в случае ошибки сообщение будет получено в момент сборки приложения или библиотеки.

```
int a = static_cast<int>(0); ①  
int a = static_cast<int>(3.14); ②  
bool a = static_cast<bool>(-4); ③  
bool a = static_cast<bool>(0); ④  
float f = 3.14f ;      ⑤  
float f = static_cast<float>(3.14) ; ⑥  
Complex f = static_cast<3.14> ⑦
```

① Явно говорим, что 0 должен восприниматься как тип (int), хотя он и так является литералом типа int. Но все ли помнят об этом?

② Явно говорим, что 3.14 воспринимать как int, т.е взять только целую часть.

③ Явно говорим, -4 нужно воспринять как bool тип, в данном случае true.

④ Явно говорим, 0 нужно воспринять как bool тип, в данном случае false.

⑤ Явно говорим, что 3.14 это float

- ⑥ Явно говорим, что 3.14 это float
- ⑦ Комплексное число может создаться из double, поэтому тут будет работать static\_cast.

## 1.10. reinterpret\_cast

`reinterpret_cast` преобразует типы, несовместимыми друг с другом, и используется для:

- В свой собственный тип
- Указателя в интегральный тип
- Интегрального типа в указатель
- Указателя одного типа в указатель другого типа
- Указателя на функцию одного типа в указатель на функцию другого типа

```
auto ptr = reinterpret_cast<volatile uint32_t *>(0x40010000) ; ①  
auto value = *ptr ; ②
```

- ① Преобразует адрес 0x40010000 в указатель типа volatile uint32\_t
- ② Записывает в переменную value (типа) значение лежащее по указателю ptr, указывающего на адрес 0x40010000

## 2. Память

Как говорилось в первой лекции, ARM имеет общее адресное пространство для данных и команд.

Ядро ARM имеет 4 Гбайт последовательной памяти с адресов 0x00000000 до 0xFFFFFFFF.

Различные типы памяти могут быть расположены по эти адресам. Обычно микроконтроллер имеет постоянную память, из которой можно только читать (ПЗУ) и оперативную память, из которой можно читать и в которую можно писать (ОЗУ).

Также часть адресов этой памяти отведены под регистры управления и регистры периферии.

### 2.1. Память микроконтроллера CortexM4

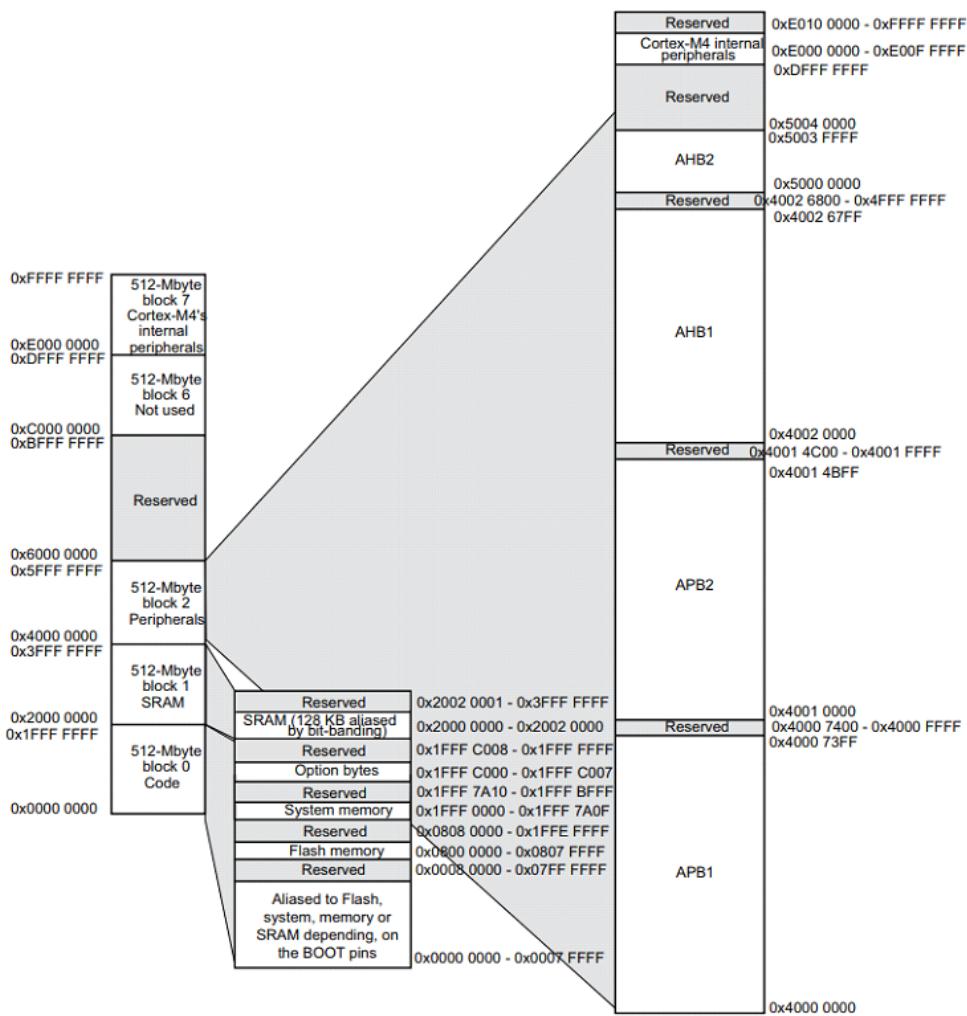


Рисунок 25. Карта памяти микропроцессора

Микроконтроллер на ядре Cortex M4 выполнен по Гарвардской архитектуре, память здесь разделена на три типа:

- ПЗУ (FLASH память в которой храниться программа)
- ОЗУ память для хранения временных данных (туда же можно по необходимости переместить программу и выполнить её из ОЗУ), память в которой находятся регистры отвечающие за настройку и работу с периферией и
- Память для хранения постоянных данных EEPROM.

Адресное пространство памяти программы (ПЗУ) находится по адресам **0x00000000** по **0x1FFFFFFF**

Адресное пространство ОЗУ находится по адресам **0x20000000** по **0x3FFFFFFF**

Адресное пространство для регистров периферии находится по адресам с **0x40000000** по **0x5FFFFFFF**

Памяти EEPROM микропроцессора STM32F411RE не содержит, см [\[Карта памяти микропроцессора\]](#). Более подробно вы можете изучить адресное пространство микропроцессора в спецификации на микропроцессор [\[12\]](#).

## 2.2. Память для расположения данных

Данные в памяти могут быть расположены 3 различными способами:

- Авто(локальные) переменные, которые являются локальными в функции располагаются в регистрах или в стеке.

Такие переменные "существуют" только внутри функции, как только функция закончится и вернется к вызывающему объекту, эти переменные становятся не валидными.

- Глобальные переменные или статические переменные. В этом случае они инициализируются единожды.

Static означает, что та память, которая была выделена под эту переменную не будет изменяться и закрепляется за этой переменной до конца работы приложения.

- Динамически размещаемые данные. Данные создаваемые на Куче(Heap)

Если заранее не известно, сколько объектов нужно создать, и сколько памяти они будут отнимать, то придется создавать их динамически, например с помощью оператора new, в таком случае, объекты будут создаваться в куче.

### 2.2.1. Память под функции(команды)

Для расположения функций используется та же самая память с границами от **0x00000000** - **0xFFFFFFFF**.

По умолчанию весь код будет лежать в сегменте .text, который расположен в readonly памяти (обычно в ROM), но можно разместить функции и в ОЗУ.

## 2.3. Указатели

Как мы уже поняли, данные могут находиться в ОЗУ или ПЗУ. Каждой переменной содержащей данные соответствует некий адрес памяти. К переменной можно обратиться непосредственно обращаясь к самой переменной, тогда мы можем напрямую писать или читать значение с адреса переменной, либо можно обратиться косвенно, через указатель

или ссылку.

Указатель это переменная, которая хранит адрес какой-то другой переменной:

```
int main() {  
    int c = 463 ; ①  
    int* ptr = &c ; ②  
    return 0;  
}
```

- ① Объявляем переменную **c** типа **int**
- ② объявляем указатель **ptr** на переменную **c** типа **int**

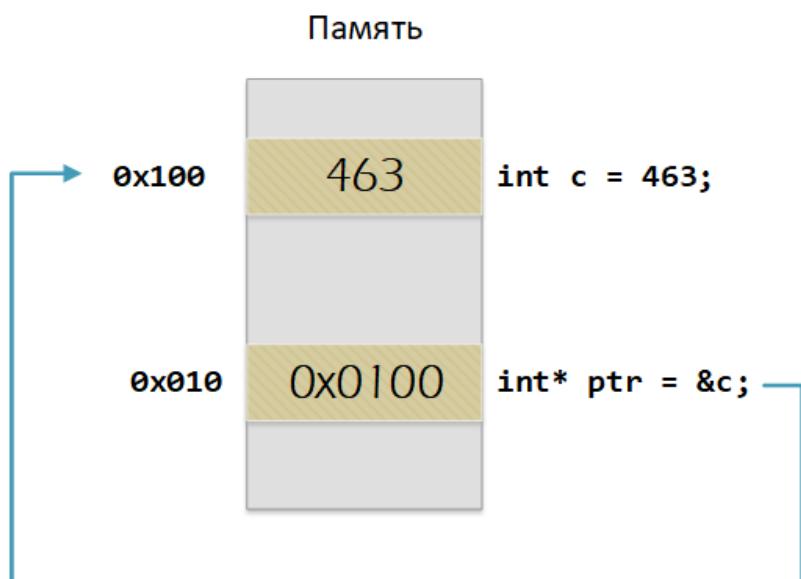


Рисунок 26. Указатель

Размер указателя для нашего микроконтроллера 4 байта (32 бита).

## 2.4. Взятие адреса и разыменование указателя.

```
int main() {  
    int c = 463 ; ①  
    int* ptr = &c ; ②  
    cout << &c ; ③  
    cout << c ; ④  
  
    *ptr = 5; ⑤  
    cout << c << ":" << *ptr; ⑥  
}
```

- ① Объявление переменной
- ② Оператор & - оператор взятия адреса.

- ③ Выведется адрес переменной **c** (0x100)
- ④ Выведется значение переменной **c** (463)
- ⑤ Операция разыменование указателя, записываем в переменную по адресу, который лежит в **ptr**, число 5
- ⑥ Вывод значения переменной **c** и значения лежащего по адресу, на который указывает указатель (5: 5) По сути **c** и **\*ptr** это одно и то же.

## 2.5. Операции над указателями

Указатели можно складывать, вычитать, сравнивать. Но указатели должны быть одного типа. Т.е. не нужно например складывать указатель типа **char \*** и **int \***

```
int main() {
    int arr[] = {1,2,3,4,5} ;      ①
    int* ptr = arr ;            ②

    ptr ++ ;                  ③
    int a = *(ptr + 4) ;        ④
    if(ptr != nullptr)         ⑤
        cout << a << ":" << *ptr; ⑥
}
```

- ① Объявление массива **arr** из 5 элементов. В целом можно считать, что массив **arr** это указатель на первый элемент массива.
- ② Обявления указателя на массив типа **int** ;
- ③ Увеличиваем указатель на 1. На самом деле мы смещаемся по адресам на размер равный **size\_of(int)**, т.е. на 4 байта. Т.е в данном случае указатель **ptr** стал указывать на элемент массива **arr[1]**.
- ④ Объявляем переменную **a** типа **int** и присваиваем ей значение **arr[4]**.
- ⑤ Сравнение указателя с **nullptr** указателем.
- ⑥ Вывод значения **a** и значения по адресу в указателе **ptr**. Вывод (5: 2)

## 2.6. Сложение указателей

```
int main() {
    int arr[] = {1,2,3,4,5} ;      ①
    int* ptr = arr ;            ②

    ptr ++ ;                  ③
    int a = *(ptr + 3) ;        ④
}
```

## Память

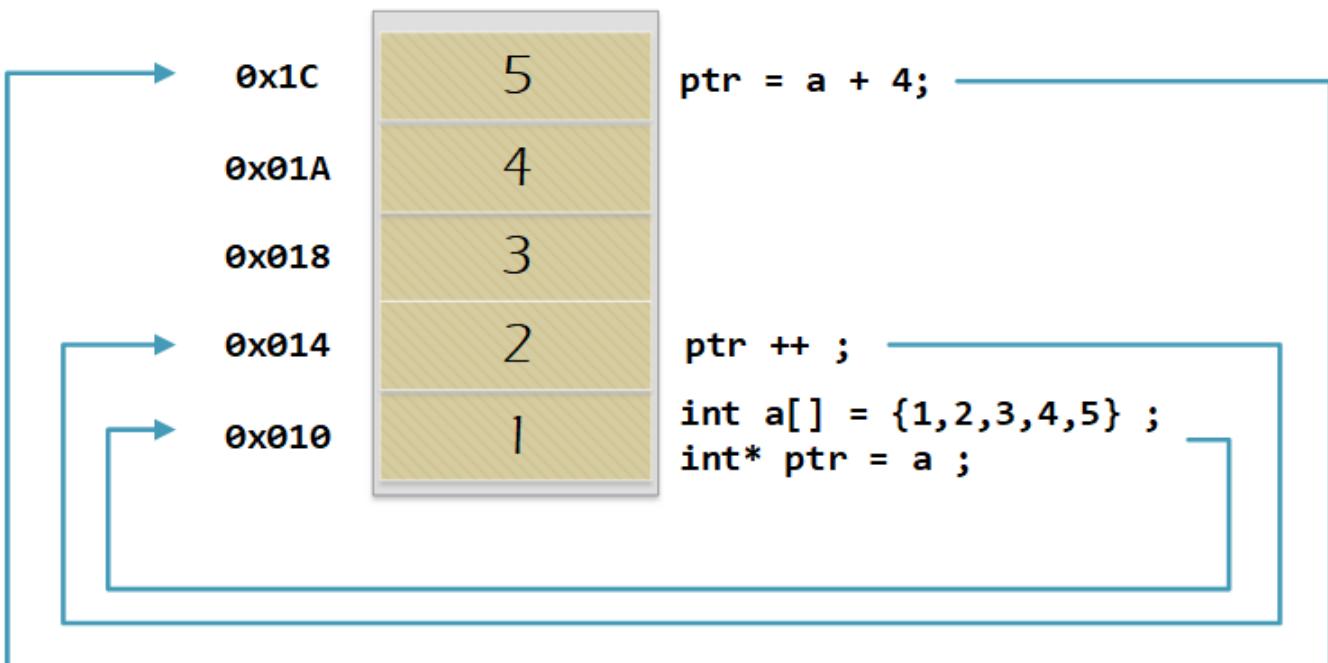


Рисунок 27. Сложение указателей

- ① Объявление массива `arr` из 5 элементов. В целом можно считать, что массив `arr` это указатель на первый элемент массива.
- ② Обявления указателя на массив типа `int` ;
- ③ Увеличиваем указатель на 1. На самом деле мы смещаемся по адресам на размер равный `size_of(int)`, т.е. на 4 байта. Т.е в данном случае указатель `ptr` стал указывать на элемент массива `arr[1]`.
- ④ Записываем в переменную `a` типа `int` данные, находящиеся по адресу, хранящиеся в указателе `ptr`, смещенному на 3.

## 2.7. Константный указатель и указатель на константу

```
int main() {
    const auto pi[] = {3.14, 3.14159} ;
    const double *ptr = pi ;
    *ptr = 3.14159 ;                                ①
    ptr++ ;                                         ②
    count << *ptr ;                                ③
    const double * const ptr1 = pi ;                ④
    ptr1++ ;                                         ⑤
    return 0 ;
}
```

- ① Пытаемся поменять значение по указателю `ptr` (`pi[0]`). Ошибка, указатель на константу, нельзя поменять значение константы

- ② Увеличиваем указатель на 1 (теперь указатель указывает на p[1]).
- ③ Вывод значения по указателю (3.14159)
- ④ Объявляем константный указатель на константу
- ⑤ Нельзя изменить указатель, он константный

## 2.8. Ссылка

```
int main(){
    int a = 0;
    int &ref = a ;           ①
    ref = 10;               ②
    cout << &ref << ":" << ref ; ③
    return 0 ;
}
```

- ① Объявляем ссылку на переменную **a**
- ② Записываем в переменную **a** число 10
- ③ Выводим адрес переменной **a** и значение переменной **a**

Ссылка это псевдоним переменной.

- У ссылки нельзя взять адрес. Если применить оператор взятия адреса к ней, то будет выведен адрес переменной, на которую она ссылается
- Ссылка ведет себя почти также как константный указатель. Её нельзя изменять, складывать, вычитать
- Ссылки нельзя сравнивать
- Ссылка не может быть не проинициализирована.

## 3. Регистр

- Существуют регистры общего назначения и специальные регистры. Регистры общего назначения расположены внутри ядра микроконтроллера(сверхбыстрая память).
- Регистры общего назначения - это сверхбыстрая память внутри процессора, предназначенная для хранения адресов и промежуточных результатов вычислений (регистр общего назначения/регистр данных) или данных, необходимых для работы самого процессора.
- Регистры специального назначения расположены в ОЗУ микроконтроллера и используются для управления процессором и периферийными устройствами.
- Каждый регистр в архитектуре ARM представляет собой ресурс памяти и имеет длину в 32 бита, где каждый бит можно представить в виде выключателя с помощью которого осуществляется управление тем или иным параметром микроконтроллера [10].

## 3.1. Регистры общего назначения

С точки зрения прикладного программиста, процессор располагает 16-ю 32-разрядными регистрами общего назначения (РОН, GPR), из которых три на деле имеют специальные функции:

- Оперативные регистры
- Вспомогательные регистры
- Специальные регистры

## 3.2. Оперативные регистры

Регистры R0-R3, R12 являются оперативными(sratch) регистрами. Любая функция может использовать эти регистры по своему усмотрению и уничтожать содержимое этих регистров.

Если функции нужны значения этих регистров после вызова другой функции, она должна сохранить их на стеке, а после вызова восстановить.

## 3.3. Вспомогательные регистры

Регистры от R4-R11 являются вспомогательными. Любая функция должна сохранить их на входе, а при выходе восстановить их значение.

## 3.4. Специальные регистры

- Регистр указателя на стек R13/SP, должен всегда указывать на последний элемент стека или ниже него.
- Регистр R15/PC есть программный счетчик.
- Регистр R14/LR, содержит адрес возврата функции.

## 3.5. Регистр специального назначения

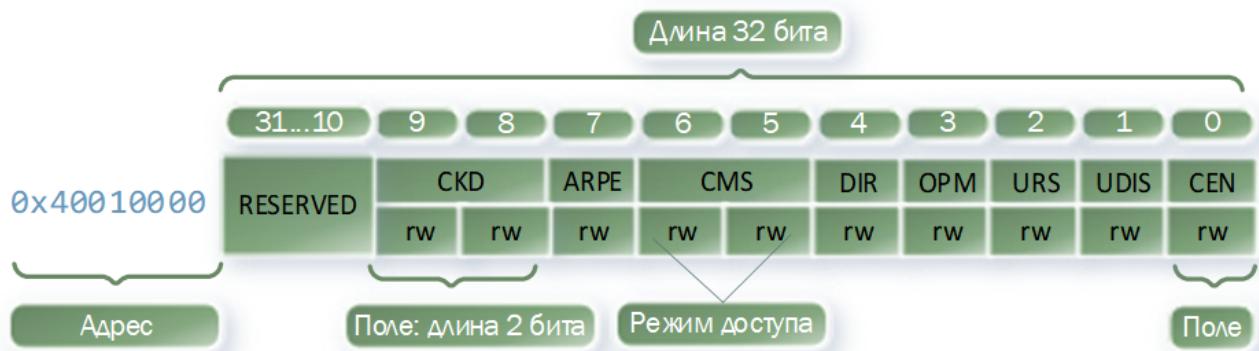


Рисунок 28. Схематичное изображение регистра

- Название регистра
- Адрес регистра обозначается 32-битным шестнадцатеричным числом.
- Тип доступа к ячейкам регистра.
- Длина - количество ячеек в одном регистре. Мы будем работать с 32-битными регистрами.
- Поле - набор ячеек регистра, отвечающих за работу одной из функций микроконтроллера
- Значение поля - есть пространство всех возможных величин, которые может принимать поле

Значение поля зависит от длины поля. Т.е. если поле имеет длину 2, то существует 4 возможные значения поля (0,1,2,3). Так же как у регистра, у полей и значений полей есть режим доступа (чтение, запись, чтение и запись)

## 3.6. Пример регистра специального назначения

Как было сказано выше регистры используются для управления микроконтроллером и его периферией. Например, чтобы запустить таймер 1 на счет, необходимо в Таймере1, в регистре **CR1(Control Register1)** в поле **CEN(Counter Enable)** установить значение 1 (Enable).

	31...10	9	8	7	6	5	4	3	2	1	0
0x40010000	RESERVED	CKD	ARPE	CMS	DIR	OPM	URS	UDIS	CEN		
		rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Рисунок 29. Регистр CR1 Таймера 1

Бит 0 CEN: Включить счетчик  
 0: Счетчик включен: Disable  
 1: Счетчик выключен: Enable

Здесь, например, CEN — это поле размером 1 бит имеющее смещение 0 относительно начала регистра. А Enable(1) и Disable(0) это его возможные значения.

## 3.7. Доступ к регистру специального назначения

Так как регистр специального назначения - это просто адресуемая ячейка памяти, то в коде это может мы можем обратиться к данным по этому адресу, разыменовывая указатель, указывающий на этот адрес:

```

int main()
{
    *reinterpret_cast<uint32_t *>(0x40010000) |= 1 << 0 ; ①
    TIM1::CR1::CEN::Enable::Set() ; ②
}

```

① Записываем 1 в нулевой бит ячейки памяти (регистра) по адресу 0x40010000

② Тоже самое, но с использование специального класса на C++

## 3.8. Работа с регистрами периферии через обертку на C++

Для того, чтобы настроить определенное периферийное устройство процессора, необходимо изменить значение поля соответствующем регистре.

Для более удобной работы с регистрами можно использовать C++ обертку. Эта обертка позволяет обращаться к регистрам в форме очень похоже с тем, как эти регистры описаны в документации.

Так, например, для запуска внешнего источника частоты, необходимо обратиться к регистру “CR” периферии “RCC”, полю “HSEON” и установить в нем значение Enable. Операция обращения к регистру выглядит следующим образом:

```

---
int main()
{
    RCC::CR::HSEON::Enable::Set() ;
}
---

```

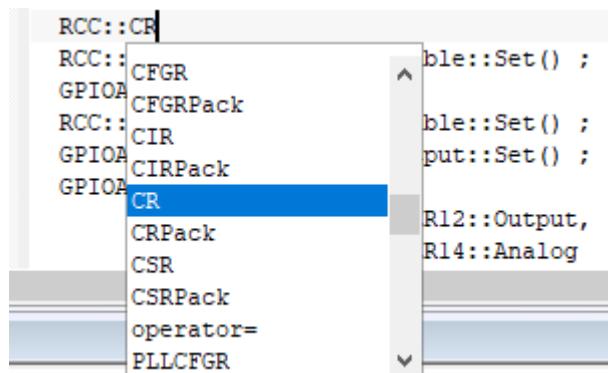


Рисунок 30. Подсказка для регистра CR модуля периферии RCC

## 3.9. Некоторые моменты при работе с оберткой C++ для регистров

Код для регистров был сгенерирован автоматически, см [13]. Поэтому по умолчанию все

значения полей называются в формате ValueX, где X-само значение. Поэтому тот момент когда нужно их использовать, нужно заглянуть в документацию и поменять слова Value, на что-то более внятное.

Для того, чтобы найти место где объявляется значение поля, необходимо правой мышкой нажать на значении и найти все его объявления.

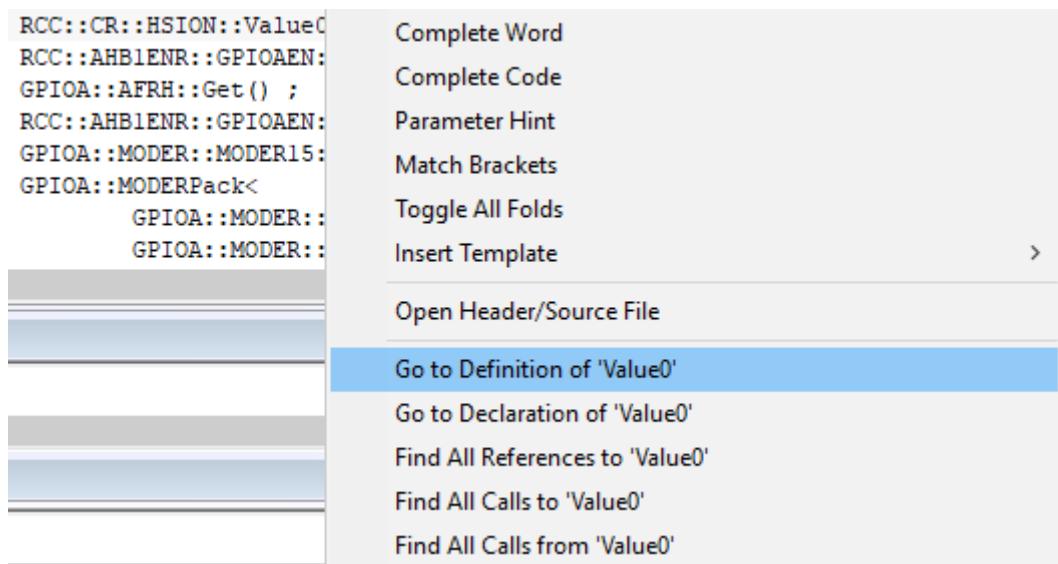


Рисунок 31. Поиск места объявления значения

На самом деле, все значения полей определены в файлах, которые лежат в папке: AbstractHardware\Registers\STM32F411\FieldValues

Можно открыть файл с именем [имя периферии]filevalues.hpp и найти там структуру названием ИМЯ ПЕРИФЕРИ\_ИМЯ РЕГИСТРА\_ИМЯ ПОЛЯ\_Values.

Например, для значений поля HSEON модуля периферии RCC, регистра CR, необходимо:

1. открыть файл AbstractHardware\Registers\STM32F411\FieldValues\rccfieldvalues.hpp,
2. найти структуру struct RCC\_CR\_HSEON\_Values
3. поменять в этой структуре Value0 на Disable, а Value1 на Enable.

## 4. Соглашение об вызовах

Соглашение об вызовах включает в себя:  
\* Объявление функции \* Компоновка С и С++ кода \*  
Последовательность использования оперативных регистров и вспомогательные регистров \*  
Вход в функцию \* Выход из функции \* Обработка адреса возврата

### 4.1. Объявление функции

Функция должна быть объявлена в таком порядке, чтобы компилятор мог узнать как её вызвать. Объявление функции может выглядеть следующим образом:

```
int MyFunction(int first, char * second);
```

Все что знает об этой функции компилятор, это то, что она принимает два параметра: целое и указатель на символ. И функция должна вернуть целое значение. Этого достаточно для компилятора, чтобы понять как вызывать эту функцию.

## 4.2. Компоновка С и С++ кода

В С+ , функция может компоноваться либо как С +, либо как С функция. Пример объявления функции с Си компоновкой:

```
extern "C" {
    int F(int);
}
```

Если вы хотите вызвать функции ассемблера из С++, то лучше объявить эту функцию, как имеющую тип компоновки Си

## 4.3. Вход в функцию

Параметры передающие в функцию могут использовать два метода:

- Через регистры
- Через стек

Для большей эффективности параметры передаются через регистры, но их число ограничено, поэтому если регистров не хватает, то используется стек. Для передачи параметров используются оперативные регистры R0:R3

## 4.4. Выход из функции

Функция может вернуть значение. Для возврата значения используются регистры R0:R1. Если значение больше 64 бит, то в регистр R0 записывается адрес где лежат данные.

Вызывающая функция обязана очистить стек, после того, как вызываемая функция вернула значение.

## 4.5. Операторы

- Арифметические операторы
- Операторы сравнения
- Логические операторы
- Побитовые операторы
- Составное присваивание
- Операторы работы с указателями и членами класса
- Функторы, тернарные операции, sizeof(), запятая, приведение типа, new

Все операторы можно переопределить

## 4.6. Арифметические операторы

Арифметические операторы предоставляют базовые арифметические действия над типами, такие как сложение, вычитание, деление, умножение, остаток от деления, присваивание. Любой оператор может быть определен для множества пользовательского типа. Т.е. вы можете создать свой тип и определить арифметические операторы для вашего типа. Например, можно определить арифметические операторы для множества комплексных чисел, которые могут быть представлены в виде вашего собственного пользовательского типа.

Таблица 4. Арифметические операторы

Операция	Оператор	Комментарий
Присваивание	=	$a = b$
Сложение	+	$a + b$
Вычитание	-	$a - b$
Унарный плюс	+	+ $a$
Унарный минус	-	- $a$
Умножение	*	$a * b$
Деление	/	$a / b$
Остаток от деления	%	$a \% b$
Инкремент (пост и предфиксный)	++	++ $a$ и $a++$
Декремент (пост и предфиксный)	--	-- $a$ и $a--$

## 4.7. Логические операторы

Логические операторы предоставляют действия над булевым типов. Результат действия этих операторов может быть только **true** или **false**

Таблица 5. Логические операторы

Операция	Оператор	Комментарий	Пример
Логическое отрицание, НЕ	!	! $a$	$!true \Rightarrow false$
Логическое умножение, И	&&	$a \&\& b$	$true \&\& false \Rightarrow false$
Логическое сложение, ИЛИ		$a     b$	$true     false \Rightarrow true$

## 4.8. Побитовые операторы

Побитовые операторы предоставляют действия с битами.

Таблица 6. Побитовые операторы

Операция	Оператор	Комментарий	Пример
Побитовая инверсия	<code>~</code>	<code>~a</code>	<code>unsigned char a = 0; ~a ⇒ 0xFF</code>
Побитовое И	<code>&amp;</code>	<code>a &amp; b</code>	<code>unsigned char a = 1, b = 3; a &amp; b ⇒ 1</code>
Побитовое ИЛИ	<code> </code>	<code>a   b</code>	<code>unsigned char a = 1, b = 3; a   b ⇒ 3</code>
Побитовое исключающее ИЛИ	<code>^</code>	<code>a ^ b</code>	<code>unsigned char a = 1, b = 3; a ^ b ⇒ 2</code>
Побитовый сдвиг влево	<code>&lt;&lt;</code>	<code>a &lt;&lt; b</code>	<code>unsigned char a = 1, b = 3; a &lt;&lt; b ⇒ 8</code>
Побитовый сдвиг вправо	<code>&gt;&gt;</code>	<code>a &gt;&gt; b</code>	<code>unsigned char a = 8, b = 3; a &gt;&gt; b ⇒ 1</code>

## 5. Отладочная плата

• STM32F411RET6 ядро: ARM® 32-bit Cortex™-M4	• CP2102: USB - UART преобразователь
• Arduino разъем: для подключения Arduino шилдов	• ICSP interface: Arduino ICSP
• USB разъем: USB коммуникационный интерфейс	• SWD interface: для программирования и отладки
• ST Morpho разъемы: для упрощения расширения	• 6-12 V DC вход питания
• Пользовательская кнопка	*Кнопка Сброса
• Индикатор питания	• Пользовательские светодиоды
• Индикаторы последовательного порта Rx/Tx	8 MHz кварцевый резонатор
• 32.768 KHz кварцевый резонатор	<a href="http://www.waveshare.com/xnucleo-F411RE.htm">http://www.waveshare.com/xnucleo-F411RE.htm</a>

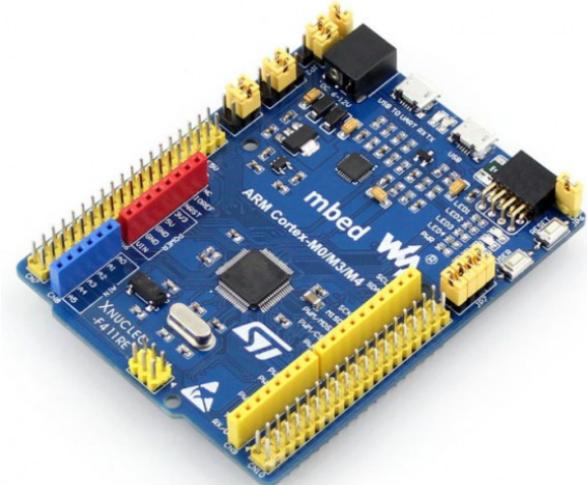


Рисунок 32. Отладочная плата

## 6. Микроконтроллер ST32F411RE

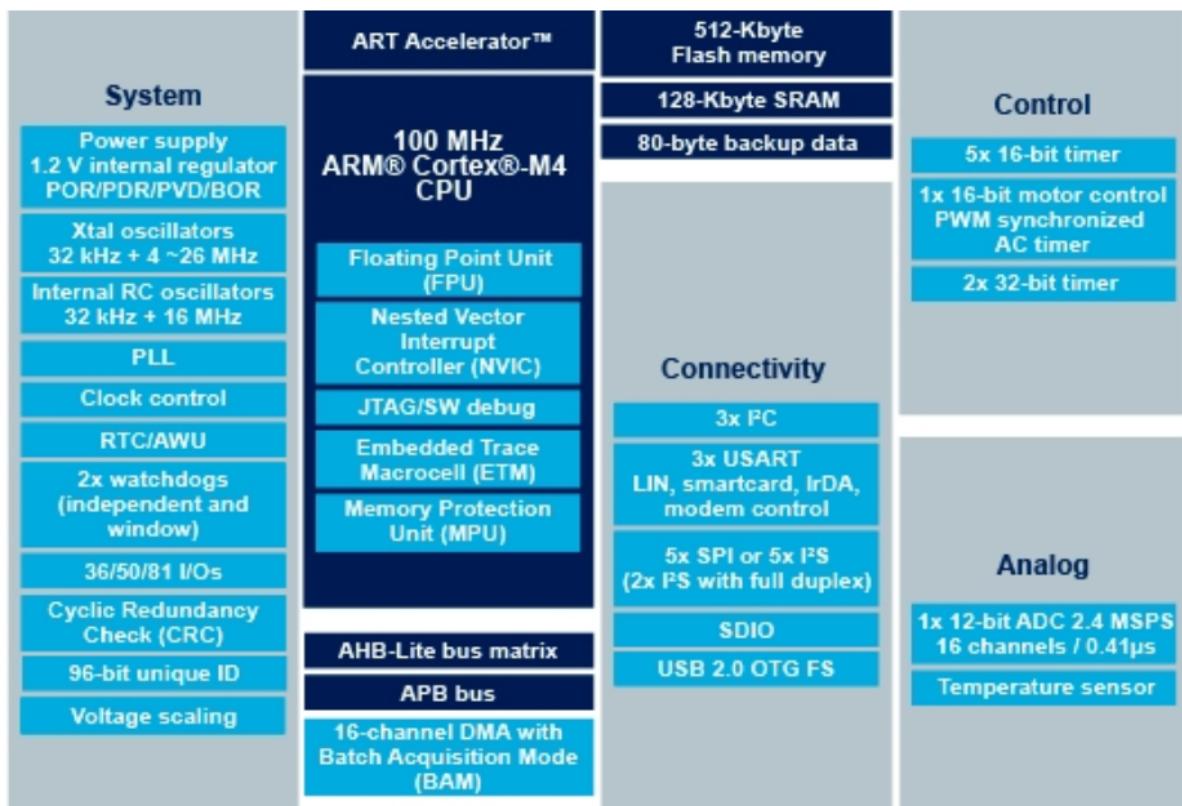


Рисунок 33. Функциональные блоки микроконтроллера STM32F411

### 6.1. Ядро CortexM4

OPERATION	INSTRUCTIONS	CM3	CM4
$16 \times 16 = 32$	SMULBB, SMULBT, SMULTB, SMULTT	n/a	1
$16 \times 16 + 32 = 32$	SMLABB, SMLABT, SMLATB, SMLATT	n/a	1
$16 \times 16 + 64 = 64$	SMLALBB, SMLALBT, SMLALTB, SMLALTT	n/a	1
$16 \times 32 = 32$	SMULWB, SMULWT	n/a	1
$(16 \times 32) + 32 = 32$	SMLAWB, SMLAWT	n/a	1
$(16 \times 16) \pm (16 \times 16) = 32$	SMUAD, SMUADX, SMUSD, SMUSDX	n/a	1
$(16 \times 16) \pm (16 \times 16) + 32 = 32$	SMLAD, SMLADX, SMLSD, SMLSX	n/a	1
$(16 \times 16) \pm (16 \times 16) + 64 = 64$	SMLALD, SMLALDX, SMLSQD, SMLSQX	n/a	1
$32 \times 32 = 32$	MUL	1	1
$32 \pm (32 \times 32) = 32$	MLA, MLS	2	1
$32 \times 32 = 64$	SMULL, UMULL	5-7	1
$(32 \times 32) + 64 = 64$	SMLAL, UMLAL	5-7	1
$(32 \times 32) + 32 + 32 = 64$	UMAAL	n/a	1
$32 \pm (32 \times 32) = 32$ (upper)	SMMLA, SMMLAR, SMMLS, SMMLSR	n/a	1
$(32 \times 32) = 32$ (upper)	SMMUL, SMMULR	n/a	1

Рисунок 34. Ядро CortexM4

- Ядро Cortex построено по гарвардской архитектуре с разделением шины данных и кода.
- Ядро Cortex-M4 поддерживает 8/16/32-разрядные операции умножения, которые выполняются за 1 цикл (деление со знаком (SDIV) или без (UDIV) занимает от 2 до 12 тактов в зависимости от размера операндов)
- Ядро Cortex-M4 поддерживает 8/16/32-разрядные операции умножения со сложением

## 6.2. Характеристики ядра CortexM4

Параметр	ARM7TDMI	ARM Cortex-M3	ARM Cortex-M4
Архитектура	ARMv4T (Фон Неймана)	ARMv7 (Гарвардская)	ARMv7 (Гарвардская)
Набор инструкций	Thumb/ARM	Thumb/Thumb-2	Thumb/Thumb-2, DSP, SIMD, FP
Конвейер	3 уровня	3 уровня + предсказание ветвлений	3 уровня + предсказание ветвлений
Прерывания	FIQ/IRQ	NMI (немаскируемые) + от 1 до 240 физических источников прерываний	NMI (немаскируемые) + от 1 до 240 физических источников прерываний
Длительность входа в обработчик прерывания	24-42 цикла	12 циклов	12 циклов
Длительность переключения между обработчиками прерываний	24 цикла	6 циклов	6 циклов

Режимы пониженного энергопотребления	Нет	Встроены	Встроены
Защита памяти	Нет	Блок защиты памяти с 8 областями	Блок защиты памяти с 8 областями
Производительность по тесту Dhrystone	0,95 DMIPS/МГц	1,25 DMIPS/МГц	1,25 DMIPS/МГц
Энергопотребление ядра	0,28 мВт/МГц	0,19 мВт/МГц	0,19 мВт/МГц
Аппаратный модуль работы с плавающей точкой	нет	нет	есть

## 6.3. Характеристики микроконтроллера

Микроконтроллер имеет следующие характеристики:

• 32 разрядное ядро ARM Cortex-M4	• Блок работы с числами с плавающей точкой FPU
• 512 кБайт памяти программ	• 128 кБайт ОЗУ
• Встроенный 12 битный 16 каналный АЦП	• DMA контроллер на 16 каналов
• USB 2.0	• 3x USART
• 5 x SPI/I2S	• 3x I2C
• SDIO интерфейс для карт SD/MMC/eMMC	• Аппаратный подсчет контрольной суммы памяти программ CRC
• 6 - 16 разрядных и 2 - 32 разрядных Таймера	• 1 - 16 битный для управления двигателями
• 2 сторожевых таймера	• 1 системный таймер
• Работа на частотах до 100Мгц	• 81 портов ввода вывода
• Питание от 1.7 до 3.6 Вольт	• Потребление 100 мкА/МГц

## 6.4. Блок диаграмма микроконтроллера

Блок схема микроконтроллера схематично изображена на рисунке [Блок диаграмма микроконтроллера](#).

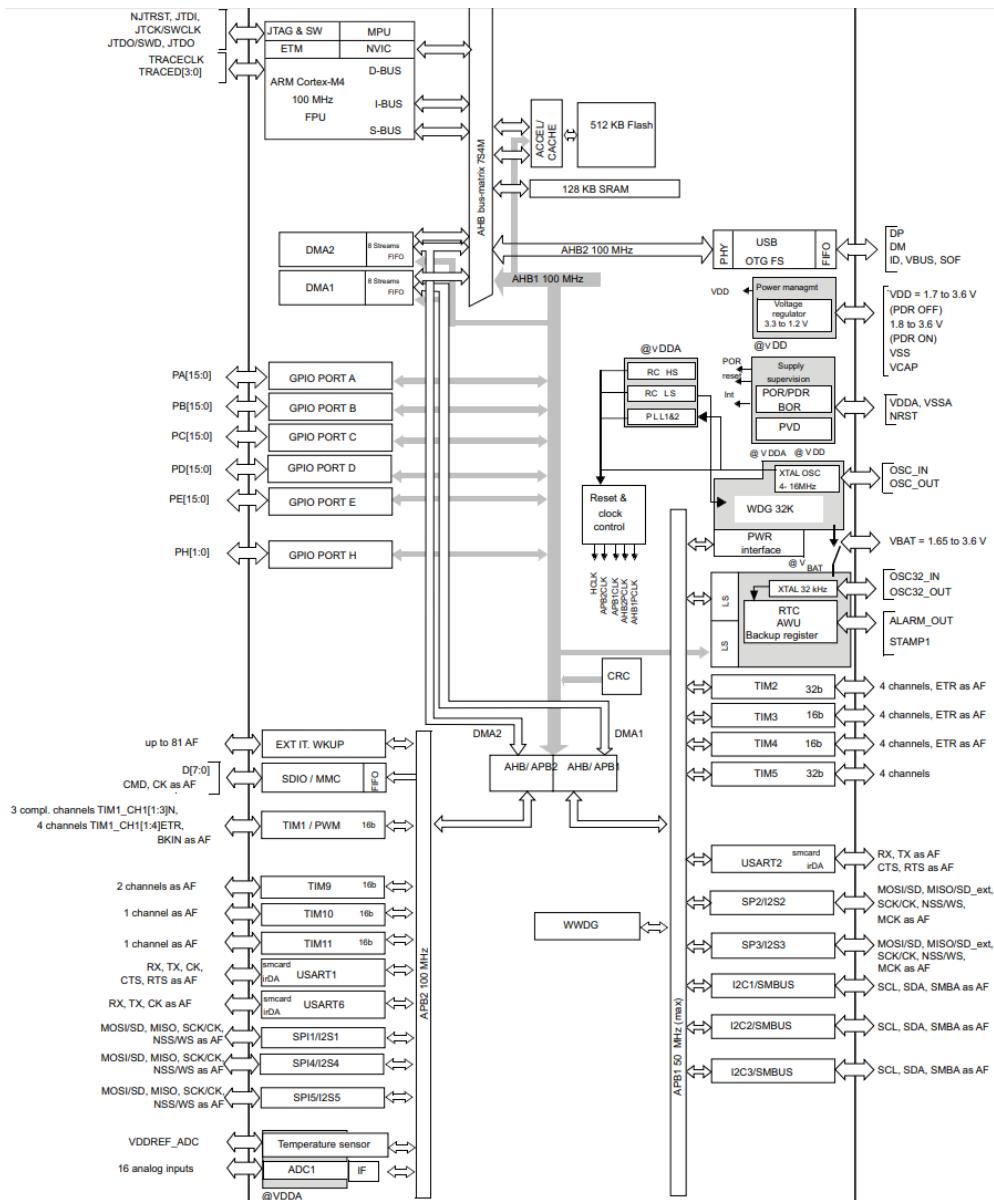


Рисунок 35. Блок диаграмма микроконтроллера

## 6.5. Дополнительные особенности микроконтроллера

Из дополнительных особенностей, которые понадобятся для лабораторных работ следует выделить:

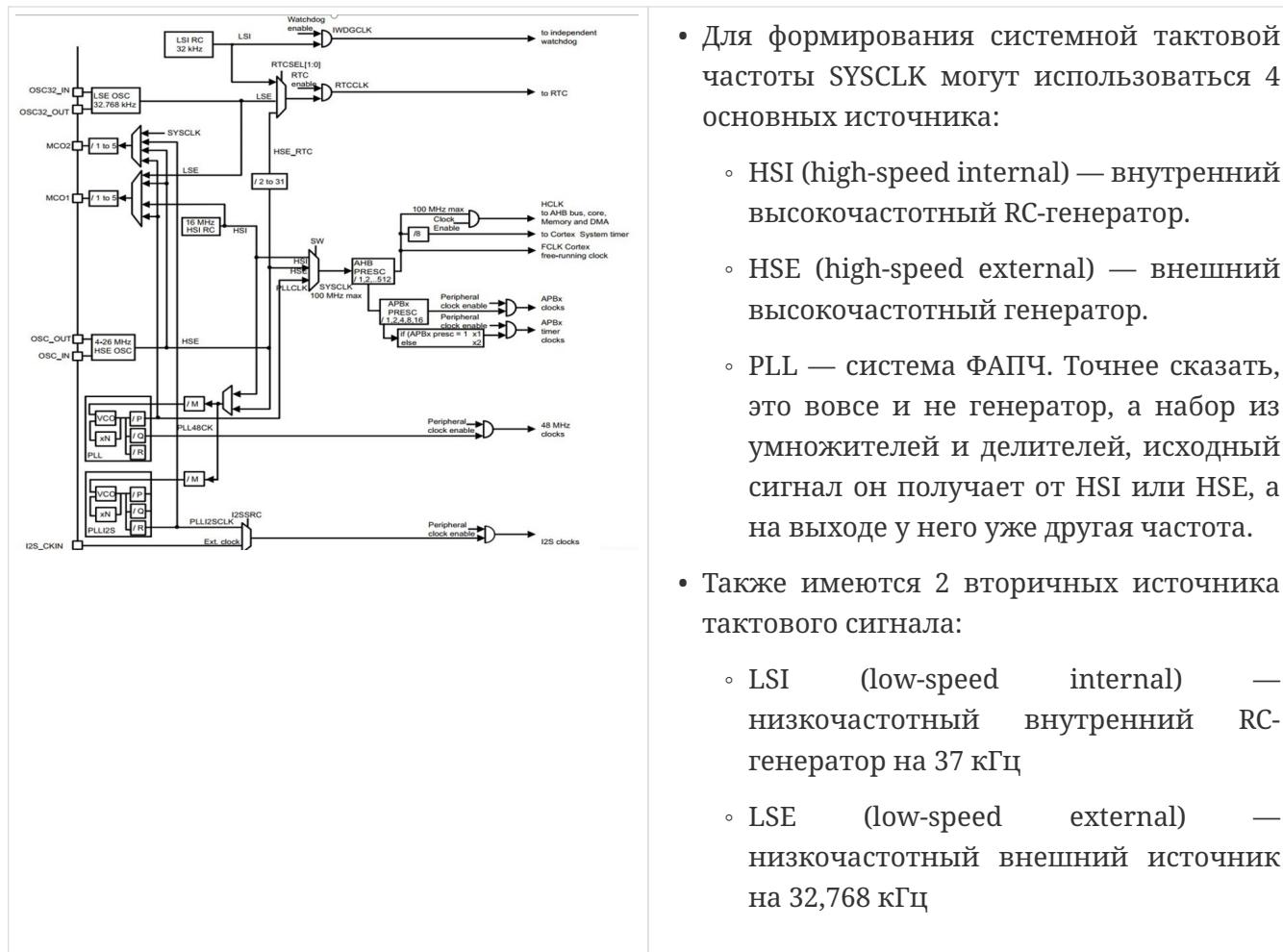
- Настраиваемые источники тактовой частоты
- Настраиваемые на различные функции порты
- Внутренний температурный сенсор
- Таймеры с настраиваемым модулем ШИМ

- DMA для работы с модулями (SPI, UART, ADC... )
- 12 разрядный ADC последовательного приближения
- Часы реального времени
- Системный таймер и спец. прерывания для облегчения и ускорения работы ОСРВ

## 7. Система тактирования

## 8. Блок диаграмма системы тактирования

Таблица 7. Система тактирования микроконтроллера STM32F411



### 8.1. Модуль тактирования.

Модуль тактирования (Reset and Clock Control) RCC

- Для формирования системной тактовой частоты SYSCLK могут использоваться 4 основных источника:
  - HSI (high-speed internal) — внутренний высокочастотный RC-генератор.
  - HSE (high-speed external) — внешний высокочастотный генератор.
  - PLL — система ФАПЧ. Точнее сказать, это вовсе и не генератор, а набор из

умножителей и делителей, исходный сигнал он получает от HSI или HSE, а на выходе у него уже другая частота.

- Также имеются 2 вторичных источника тактового сигнала:
  - LSI (low-speed internal) — низкочастотный внутренний RC-генератор на 37 кГц
  - LSE (low-speed external) — низкочастотный внешний источник на 32,768 кГц

## 8.2. Фазовая подстройка частоты PLL

- PLL Внутренний источник PLL тактируется от внешнего или внутреннего высокочастотных генераторов (HSE либо HSI).
  - С помощью регистров PLLM, PLLN, PLLP можно подобрать любую частоту до 100 МГц включительно по формуле:

$$f = f(\text{PLL clock input}) \times (\text{PLLN} / \text{PLLM}) / \text{PLLP}$$

- Кроме системной тактовой частоты SYSCLK, PLL также выдает частоту 48 МГц для интерфейса USB. При использовании USB входная частота для PLL должна быть в диапазоне от 2 МГц до 24 МГц.

$$f(\text{USB}) = f(\text{PLL clock input}) \times (\text{PLLN} / \text{PLLM}) / \text{PLLQ}$$

## 8.3. Дополнительные генераторы тактовой частоты

- LSE. Низкочастотный внешний генератор частоты.
  - Применение внешнего кварцевого/керамического резонатора на 32,768 кГц на входах OSC32\_IN, OSC32\_OUT. Высокостабильный источник, формирует тактовые сигналы для часов реального времени RTC, модуля ЖКИ, а также для таймеров TIM9/TIM10/TIM11.
  - Использование внешнего источника тактовой частоты (режим LSE bypass). Формируются тактовые сигналы для часов реального времени и ЖКИ. В этом режиме исходный сигнал поступает с генератора HSE. Входная частота может быть до 1 МГц, затем сигнал проходит через делитель с коэффициентом деления 2, 4, 8 или 16. Входной сигнал может быть прямоугольной, треугольной формы или синусоидой с 50% скважностью.
- LSI. Внутренний RC-генератор частотой около 37 кГц.
  - Как и LSE, позволяет тактировать часы реального времени и модуль ЖКИ. Кроме этого, поддерживает работоспособность независимого сторожевого таймера IWDG в режимах Stop и Standby.

## 8.4. Регистр управления частотой.

Clock Control register (CR) Как уже упоминалось, системная тактовая частота для серии "STM32F411" может быть до 100 МГц. Для ее формирования используются 3 основных источника — HSI, HSE, PLL. Включение и выключение основных генераторов производится через регистр RCC\_CR — Clock Control register.

Значение по умолчанию: 0x0000 XX81:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved				PLL2S RDY	PLL2S ON	PLLRDY	PLLON	Reserved				CSS ON	HSE BYP	HSE RDY	HSE ON
				r	rw	r	rw					rw	rw	r	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
HSICAL[7:0]								HSITRIM[4:0]				Res.	HSI RDY	HSION	
													r	rw	

**Bit 24 PLLON** Включить PLL. Этот бит устанавливается и сбрасывается программно, чтобы включить PLL. Бит не может быть сброшен, если PLL уже используется как системная частота.

- 0: ОТКЛЮЧИТЬ PLL 1: ВКЛЮЧИТЬ PLL

**Bit 16: HSEON** Включить HSE. Этот бит устанавливается и сбрасывается программно. Бит не может быть сброшен, если HSE уже используется как системная частота.

- 0: ОТЛЮЧИТЬ HSE 1: ВКЛЮЧИТЬ HSE

**Bit 0: HSION** Включить HSI. Этот бит устанавливается и сбрасывается программно. Очищается аппаратно при входе в режим Stop или Standby. Бит не может быть сброшен, если HSI уже используется как системная частота.

- 0: ВЫКЛЮЧИТЬ HSI 1: ВКЛЮЧИТЬ HSI

## 8.5. Регистр управления частотой. Контроль

Сразу после установки частоты, нужно проверить, что частота с нового источника стабилизировалась. Для этого используются те же поля того же регистра CR, оканчивающиеся на RDY (Ready)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved				PLL2S RDY	PLL2S ON	PLLRDY	PLLON	Reserved				CSS ON	HSE BYP	HSE RDY	HSE ON
				r	rw	r	rw					rw	rw	r	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
HSICAL[7:0]								HSITRIM[4:0]				Res.	HSI RDY	HSION	
													r	rw	

**Bit 25 PLLRDY** Флаг готовности частоты PLL. Этот бит устанавливается аппаратно

- 0: PLL НЕ ЗАПУЩЕН И НЕ ИСПОЛЬЗУЕТСЯ 1: PLL ИСПОЛЬЗУЕТСЯ

**Bit 17: HSERDY** Флаг готовности частоты HSE. Этот бит устанавливается аппаратно.

- 0: HSE НЕ ГОТОВ 1: HSE ГОТОВ

**Bit 1: HSIRDY** Флаг готовности частоты HSI. Этот бит устанавливается аппаратно

- 0: HSI НЕ ГОТОВ 1: HSI ГОТОВ

## 8.6. Регистр конфигурации частоты. Выбор источника

После включения генераторов частоты, необходимо выбрать один из них в качестве источника для системной частоты SYSCLK. Выбор осуществляется через регистр RCC\_CGCR — Clock Configuration Register. Значение по умолчанию: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved				PLL12S RDY	PLL12S ON	PLLRDY	PLLON	Reserved			CSS ON	HSE BYP	HSE RDY	HSE ON	
				r	rw	r	rw				rw	rw	r	rw	
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
HSICAL[7:0]								HSITRIM[4:0]				Res.	HSI RDY	HSION	
r	r	r	r	r	r	r	r	rw	rw	rw	rw		r	rw	

**Bits 3:2 SWS[1:0]** Статус частоты SYSCLK.

- 00: ИСТОЧНИК ЧАСТОТЫ HSI 01: ИСТОЧНИК ЧАСТОТЫ HSE
- 10: ИСТОЧНИК ЧАСТОТЫ PLL 11: РЕЗЕРВ

**Bits 1:0 SW[1:0]** Выбор источника частоты SYSCLK.

- 00: HSI 01: HSE
- 10: PLL 11: НЕ ИСПОЛЬЗУЕТСЯ

## 8.7. Регистр конфигурации частоты. Делители

Следующие секции регистра HPRE (AHB prescaler), PPRE1 (APB1 prescaler), PPRE2 (APB2 prescaler) — задают коэффициенты деления системной частоты SYSCLK, которая после предделителей поступает на матрицы шин.

## AHB (Advanced High Speed Busses)

матрица высокоскоростных шин. Она "доставляет" сигналы тактирования к ядру микроконтроллера, памяти (это как FLASH, так EEPROM и RAM) и модулю DMA Direct Memory Access — модуль прямого доступа к памяти), системному таймеру. Также, в семействе STM32F4 на эту шину "посажены" и все порты ввода/вывода GPIO .

## APB1, APB2 (Advanced Peripheral Bussess)

матрицы шин периферии. Соответственно, к остальным периферийным модулям тактовая частота распределяется уже через эти шины.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved				PLL12S RDY	PLL12S ON	PLLRDY	PLLON	Reserved				CSS ON	HSE BYP	HSE RDY	HSE ON
				r	rw	r	rw					rw	rw	r	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
HSICAL[7:0]								HSITRIM[4:0]				Res.	HSI RDY	HSION	
r	r	r	r	r	r	r	r	rw	rw	rw	rw		r	rw	

### Bits 13:11 PPREG2[2:0]

Делитель частоты шины APB2. Это устанавливается и очищается программно.

- 0xx: AHB **100**: AHB/2 **101**: AHB/4 **110**: AHB/8 **111**: AHB/16

### Bits 10:8 PPREG1[2:0]

Делитель частоты шины APB1. Это устанавливается и очищается программно.

- 0xx: AHB **100**: AHB/2 **101**: AHB/4 **110**: AHB/8 **111**: AHB/16

### Bits 7:4 HPREG[3:0]

Делитель частоты шины AHB.

- 0xxx: SYSCLK **1000**: SYSCLK/2 **1001**: SYSCLK/4 **1010**: SYSCLK/8 **1011**: SYSCLK/16 **1100**: SYSCLK/64 **1101**: SYSCLK/128 **1110**: SYSCLK/256 **1111**: SYSCLK/512

## 8.8. Алгоритм настройки частоты

- Определить какие источники частоты нужны
  - Например, PLL нужен для USB
- Включить нужный источник
  - Используя Clock Control register (RCC::CR)
- Дождаться стабилизации источника

- Используя соответствующие биты (.RDY) Clock Control register (RCC::CR)
- Назначить нужный источник на системную частоту
  - Используя Clock Configuration Register (RCC::CFGR)
- Дождаться пока источник не переключиться на системную частоту
  - Используя Clock Configuration Register (RCC::CFGR)

## 9. Контрольные вопросы

1. Что такое POD типы данных?
2. Назовите все виды типов в языке C++
3. Что такое пользовательский тип?
4. Назовите модификаторы типов.
5. Назовите правило установки размеров типов
6. Что делает оператор sizeof()?
7. Что характеризует тип std::size\_t
8. Назовите фиксированные типы целых в библиотеке std
9. Что такое псевдоним типа?
10. Что такое явное и неявное преобразование типа?
11. Какие явные преобразования типов вы знаете?
12. Что делает reinterpret\_cast?
13. Чем static\_cast отличается от reinterpret\_cast?
14. Что такое ОЗУ и ПЗУ?
15. Каков размер памяти ARM Cortex микроконтроллеров.
16. По какой архитектуре разработан ARM Cortex микроконтроллер?
17. В чем отличие Гарвардской архитектуры от Архитектура ФонНеймана?
18. Где располагаются локальные переменные?
19. Где располагаются статические переменные?
20. Где располагаются глобальные переменные?
21. Что такое стек?
22. Что такое указатель?
23. Что такое разыменовывание указателя?
24. Что означает взятие адреса?
25. Какие операции можно выполнять над указателями?
26. Что такое константный указатель?
27. Что такое указатель на константу?

28. Что такое ссылка? В чем её отличие от указателя?
29. Что такое регистр?
30. Что такие регистры общего назначения?
31. Что такие регистры специального назначения?
32. Как можно установить бит в регистре специального назначения?
33. Объясните как вызывается функция.
34. Что такое трансляция?
35. Что такое компоновка?
36. Как лучше организовывать структуру проекта и почему?
37. Что такое операторы?
38. Какие арифметические операторы вы знаете?
39. Какие логические операторы вы знаете?
40. Какие побитовые операторы вы знаете?
41. Приведите пример переопределения оператора
42. Какие еще операторы вы знаете?
43. Как сбросить бит с помощью битовых операторов?
44. Как установить бит с помощью битовых операторов?
45. Как поменять значение бита с помощью битовых операторов?
46. Какой микроконтроллер на отладочной плате XNUCLE ST32F411?
47. Какие блоки входят в состав микроконтроллера STM32F411?
48. В чем отличие ядра CortexM4 от CortexM3?
49. Назовите основные характеристики микроконтроллера STM32F411.
50. Назовите дополнительные характеристики микроконтроллера STM32F411.
51. Какие источники тактирования есть у микроконтроллера STM32F411
52. Назовите алгоритм подключения системной частоты к источнику тактирования микроконтроллера STM32F411.
53. Что такое ФАПЧ?
54. Что делает следующий код?

```

int main()
{
    int StudentUdacha = 10;
    int PrepodUdachca = 0 ;

    StudentUdacha = StudentUdacha ^ PrepodUdachca ;
    PrepodUdachca = StudentUdacha ^ PrepodUdachca ;
    StudentUdacha ^= PrepodUdachca ;
}

```

## 10. Порты общего назначения

### 10.1. Основные характеристики

- 5 портов общего назначения
- 16 линий ввода вывода
- Режимы входа:
  - цифровой с подтяжкой к 1 и к 0
  - аналоговый
- Возможность работы в альтернативном режиме

### 10.2. Различные режимы работы портов

- Плавающий цифровой вход (Input floating)
- Цифровой вход с подтяжкой к 1 (Input pull-up)
- Цифровой вход с подтяжкой к 0 (Input-pull-down)
- Аналоговый (Analog)
- Цифровой выход с открытым коллектором с подтяжкой к 1 или к 0 (Output open-drain with pull-up or pull-down capability)
- Цифровой двухтактный выход с подтяжкой к 1 или к 0 (Output push-pull with pull-up or pull-down capability)
- Альтернативная функция с открытым коллектором с подтяжкой к 1 или к 0 (Alternate function push-pull with pull-up or pull-down capability)
- Альтернативная функция двухтактный выход с подтяжкой к 1 или к 0 (Alternate function open-drain with pull-up or pull-down capability)

### 10.3. Цифровой режим

Когда мы говорим, что порт работает в цифровом режиме, то обычно подразумеваем, что порт имеет два состояния 1(true) и 0(false) или говоря на языке электроники **High** и **Low**.

Эти сигналы соответствуют уровню питания микроконтроллера, для нашего микроконтроллера обычно **High** соответствует 3-3.3В, а **Low** - 0 В.

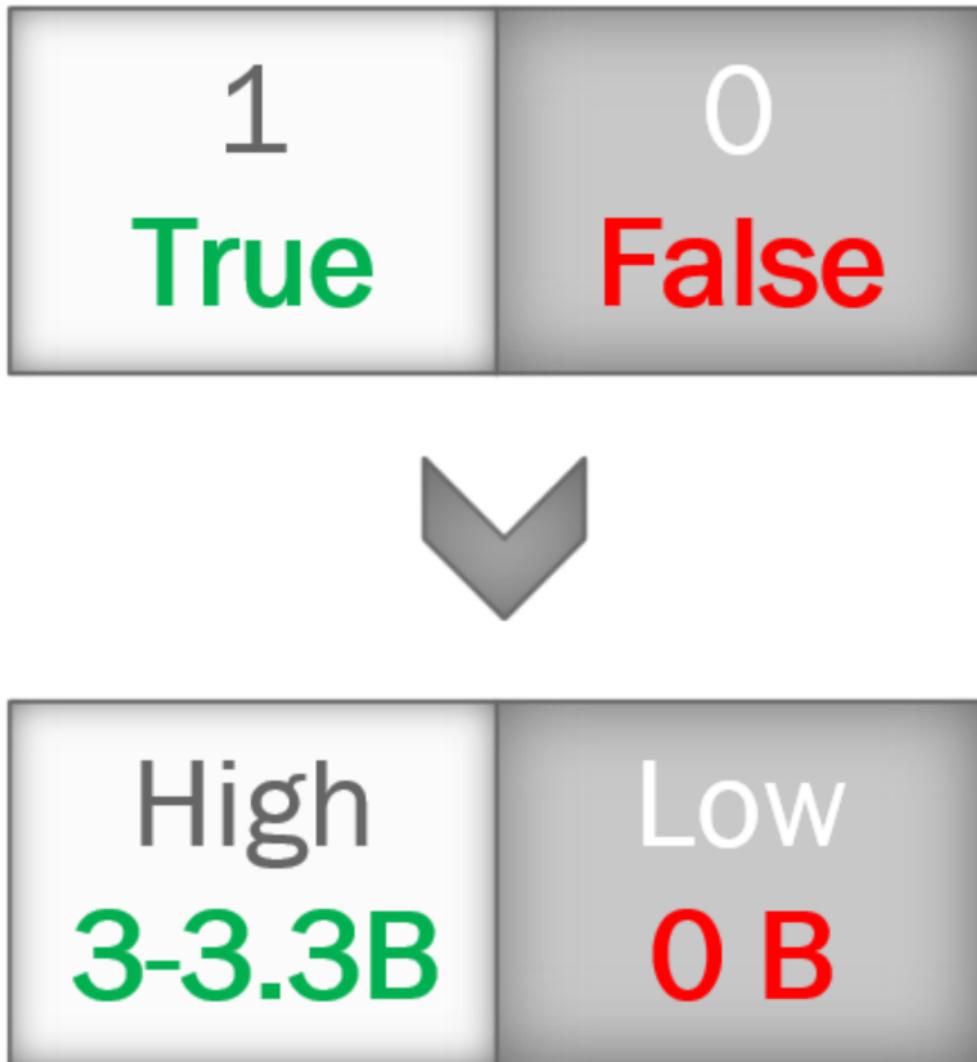


Рисунок 36. Цифровой режим

## 10.4. Работа в цифровом режиме

С помощью портов можно управлять работой других устройств.

Например, можно управлять режимом работы светодиода. На рисунке [Работа в цифровой режиме] показан источник питания, резистор, который ограничивает ток и определяет яркость светодиода. Светодиод подключен к питанию, а микроконтроллер подключают вторую часть к земле в итоге ток течет от + к - и светодиод горит.

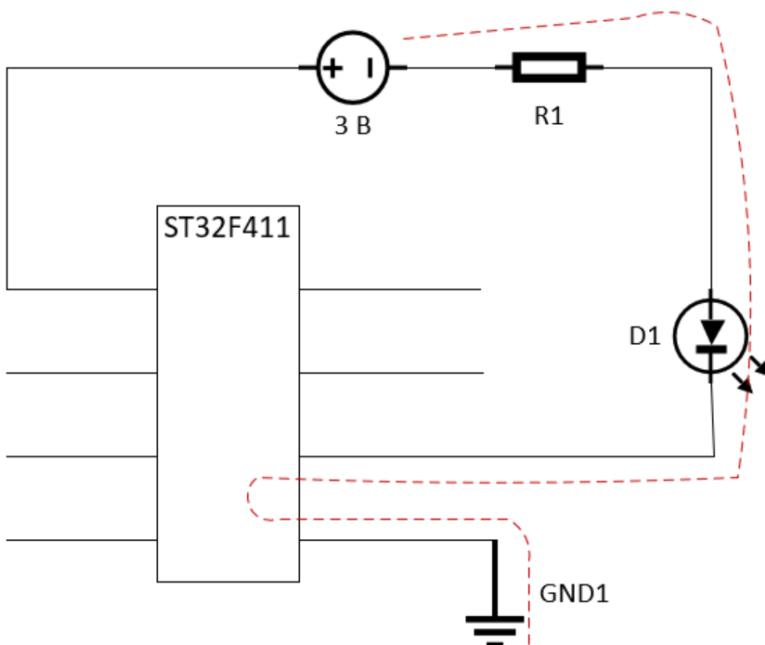
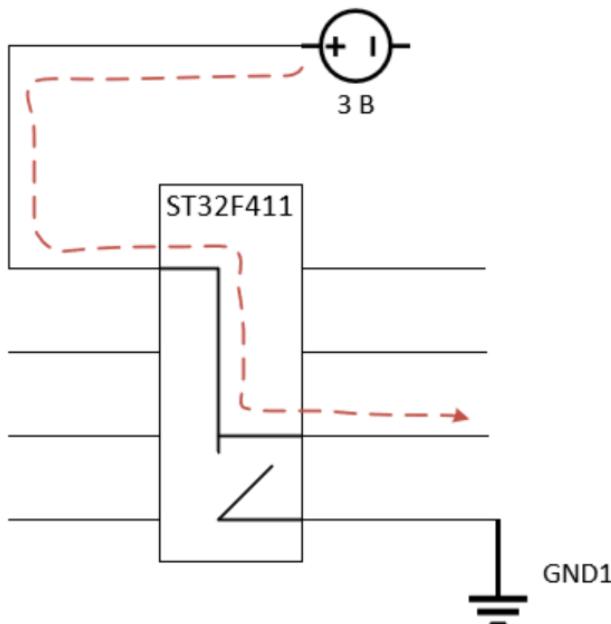


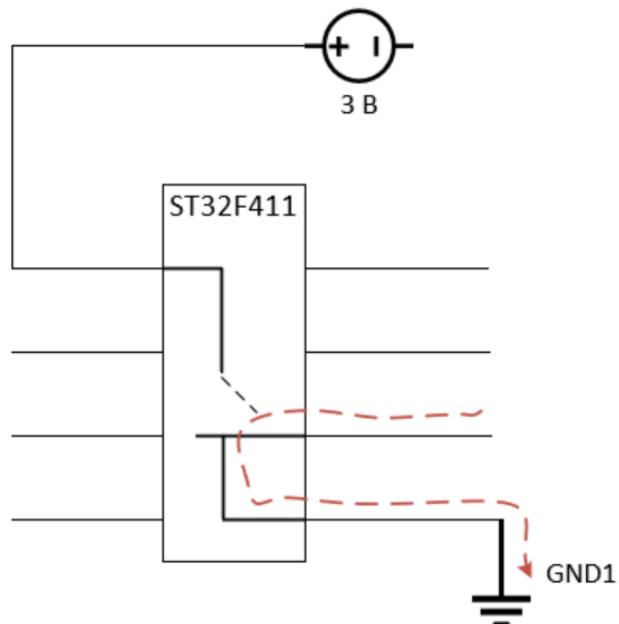
Рисунок 37. Работа в цифровой режиме

## 10.5. Цифровой выход

Когда порт настроен как цифровой выход им можно управлять. Например, если вы задали состояние порта High, то порт подключается к питанию, в итоге на ножке порта появляется высокий уровень напряжения. В случае, если вы задали Low, на ножке порта появляется низкий уровень напряжения или 0.



**Output High**



**Output Low**

Рисунок 38. Цифровой выход

## 10.6. Цифровой вход

Когда порт настроен как цифровой вход его сопротивление бесконечно, контакт никуда не

подключен ни к земле ни к питанию, поэтому ток никуда не течет. Любое напряжение на такой ножке будет интерпретирована как 1 или 0, в зависимости от уровня напряжения высокого или низкого. В таком случае это называется "подвешенная" или плавающая ножка и наводка или шум на этой ножке может быть интерпретирован как 1 или 0 в зависимости от уровня шума. Таким образом такая плавающая "ножка" не очень хорошо, так как могут генерироваться ложные переходы

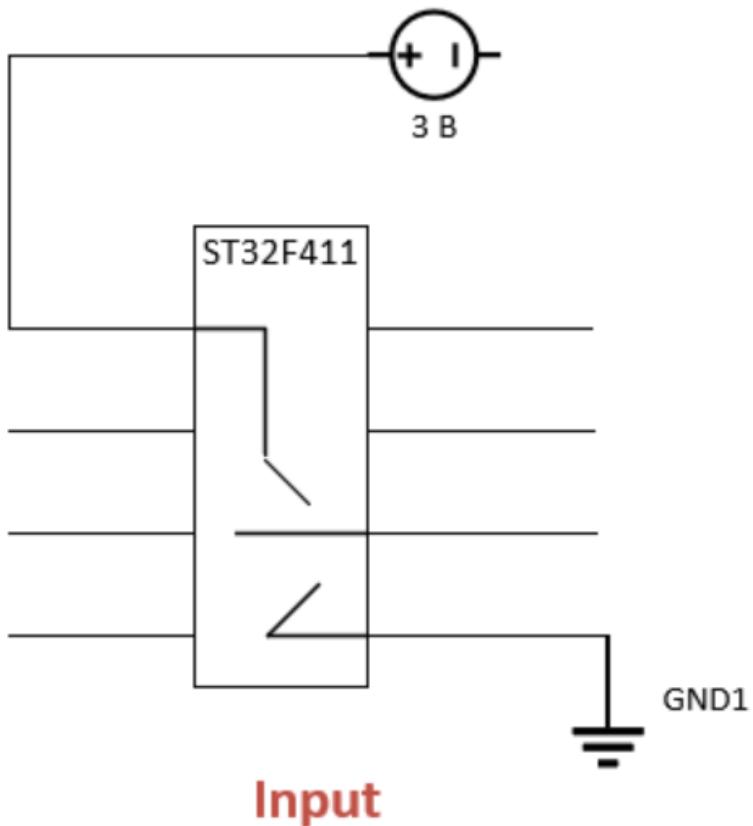


Рисунок 39. Цифровой вход

## 10.7. Цифровой вход с подтяжкой

Плавающий сигнал на подвешенной ножке может быть причиной следующих проблем

- Разное значение при считывании (1 или 0) в разные моменты времени
- Ложные переходы (если настроено прерывание, то вы постоянно будете входить в обработчик)
- Повышенное потребление из-за того, что схема входного буфера для ножки потребляет ток когда сигнал на ножке не полностью High или Low

Чтобы избавиться от плавающего сигнала на ножке обычно её подтягивают к 0 или 1. Обычно эта опция уже есть внутри микроконтроллера и может быть настроена

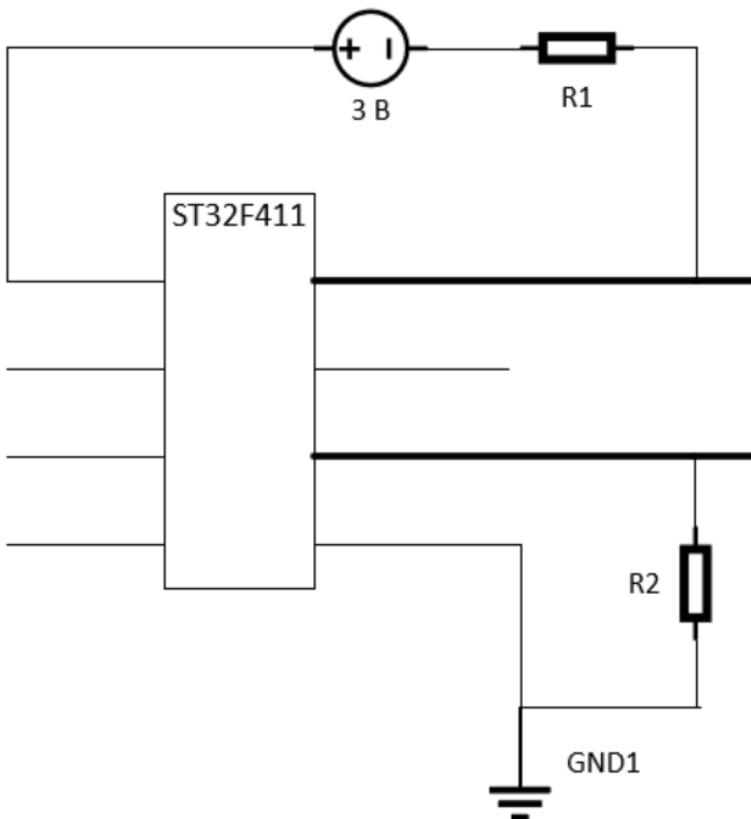


Рисунок 40. Цифровой вход с подтяжкой

## 10.8. Цифровой вход с подтяжкой к 1

Вход с подтяжкой к 1.

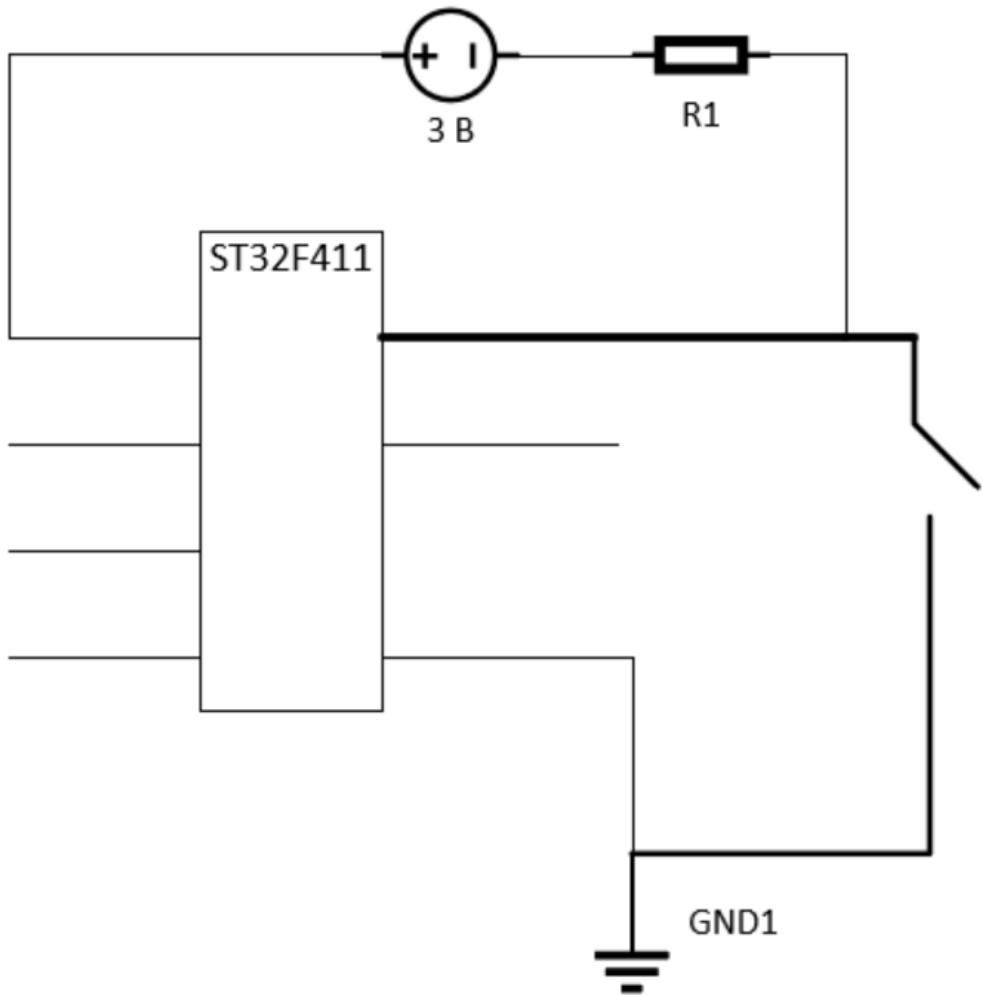


Рисунок 41. Цифровой вход с подтяжкой к 1

## 10.9. Регистры портов общего назначения

- **GPIOx\_MODER (port mode register).** Задает режимы работы индивидуально каждого из вывода порта.
  - Каждый из выводов GPIO может быть настроен как вход, выход, работать в аналоговом режиме, или подключен к одной из альтернативных функций.
- **GPIOx\_OSPEEDR (port output speed register).** Задает скорость работы порта:
  - 400кГц, 2МГц, 10МГц и 40Мгц.
- **GPIOx\_PUPDR (port pull-up/pull-down register).** Задает подключение подтягивающих резисторов
  - Без подтягивающего резистора, с подтяжкой к «+» питания, с подтяжкой к «gnd» земле.
- **GPIOx\_IDR (input data register).** регистр входных данных, из которого считывается состояние входов порта.
- **GPIOx\_ODR (output data register).** регистр выходных данных. Запись числа в младшие 16 бит, приводит к появлению соответствующих уровней на выводах порта.
- **GPIOx\_OTYPER (port output type register).** В режиме выхода или альтернативной функции, соответствующий бит регистра устанавливает тип выхода.

- Push-Pull (двуихтактный) или Open Drain (выход с открытым коллектором).

## 10.10. Регистры портов общего назначения

- **GPIOx\_BSRR (port bit set/reset register)**. Это регистр побитовой установки/сброса данных на выходных линиях порта.

Этот регистр дает возможность выполнения «атомарных» операций побитового управления выходными линиями порта. При этом нет риска возникновения прерывания между операциями чтения и модификации при записи числа в выходной регистр **GPIOx\_ODR**. Атомарные операции с регистром **GPIOx\_BSRR** выполняются за один цикл записи. При этом операции установки/сброса имеют однократный эффект. Предыдущее состояние модифицируемого бита регистра **GPIOx\_BSRR** совершенно неважно, можно сколько угодно «пишать» туда единицы и каждый раз регистр **GPIOx\_ODR** будет реагировать соответствующим образом.

- 32 разряда этого регистра позволяют индивидуально установить или сбросить каждый из 16 младших разрядов регистра **GPIOx\_ODR**.
- Младшие 16 разрядов регистра **GPIOx\_BSRR** отвечают за установку соответствующего бита регистра **GPIOx\_ODR** в «1», старшие 16 разрядов сбрасывают этот бит. Установка/сброс осуществляются записью «1» в соответствующий разряд. Запись «0» никак не воздействует на состояние соответствующего бита выходного регистра данных. При одновременной записи двух единиц в биты установки и сброса, приоритет имеет операция установки бита.
  - **GPIOxLCKR (port configuration lock register)**. Позволяет «заморозить», то есть защитить от изменения текущую настройку конфигурации. Можно запретить модификацию следующих регистров управления: **GPIOx\_MODER**, **GPIOx\_OTYPER**, **GPIOx\_OSPEEDR**, **GPIOx\_PUPDR**, **GPIOx\_AFRL**, **GPIOx\_AFRH**.

## 10.11. Работа с портами в режиме общего назначения

- Определить какой порт нужно использовать
- Подключить нужный порт к источнику частоты
  - Через регистр **RCC → AHB1ENR**
- Определить нужна ли какая-то специфическая скорость для конкретного порта и если да, настроить её
  - Через регистр **GPIOx\_OSPEEDR**
- Определить нужна ли подтяжка и какой тип выводов надо использовать
  - **GPIOx\_PUPDR** и **GPIOx\_OTYPER**
- Определить какие выводы портов нужно использовать как выход, а какие как вход
- Настроить нужные вывода порта на вход или выход
  - Через регистр **GPIOE → MODER**

# 11. Задания

Задания, кто не успеет в лабораторной, завершить дома.

## 11.1. Содержание отчета

- Описать процесс записи в регистр по его адресу
- Описать полученный результат записи в регистры MODER и ODR
- Описать процесс вызова функции в IAR
- Описать регистры общего назначения для семейства Cortex-m4
- Описать все виды источников тактирования параметры их настройки
- Описать процесс получения заданной по варианту частоты тактирования
- Описать ошибки, сделанные при выполнении работы
- Ответить на контрольные вопросы
- Сделать выводы

## 11.2. Задание 1

1. Создать проект в соответствии с Заданием 1 Лекции 1
2. Написать программу в main.cpp

```
#include "rccregisters.hpp" //for RCC
int main() {
    RCC::AHB1ENR::GPIOCEN::Enable::Set() ;
    for(;;) {
        //код лабораторной здесь.
    }
    return 0 ;
}
```

1. Открыть спецификацию на микроконтроллер [STM32F411](#) на странице
  - На странице 38 узнать на каком адресе расположен модуль **GPIOC**
  - На странице 157, узнать смещение регистра **GPIOC\_MODER** относительно адреса **GPIOC** и вычислить адрес регистра **GPIOC\_MODER**
2. Записать по адресу регистра **GPIOC\_MODER** биты 10,16,18 в 1, а биты 11,17,19 в 0.
3. Открыть спецификацию на микроконтроллер [STM32F411](#).
  - На странице 159, узнать смещение регистра **GPIOC\_ODR** относительно адреса **GPIOC** и вычислить адрес регистра **GPIOC\_ODR**
4. Записать по адресу регистра **GPIOC\_ODR** биты 5,8,9 в 1
5. Написать функцию задержки используя цикл **void Delay()**. И вызвать ей после

установки битов

6. После задержки Записать по адресу регистра **GPIOC\_ODR** биты 5,8,9 в 0
7. Вызвать функцию сброса битов
8. Запустить программу, в пошаговой отладке в окне Register, посмотреть, что происходит с регистрами **GPIOC\_MODER** и **GPIOC\_ODR**.
9. Посмотреть видео <https://www.youtube.com/watch?v=hukr8ZqS5Ys>

## 11.3. Задание 2

1. Создать указатель типа **volatile int\***, которая будет содержать адрес регистра **GPIOC\_MODER**
2. Создать переменную типа **int** и записать туда значение, которое содержится по этому адресу
3. Запустить отладку, запустить окно Memory и проверить, что по этому адресу лежит это значение
4. В отладке открыть окно регистры и проверить, что значение регистра **GPIOC\_MODER**, совпадает со значением в переменной типа **int**
5. Проделать тоже самое с произвольным адресом в ОЗУ.
6. Посмотреть видео <https://www.youtube.com/watch?v=M53lJlcFOZQ>

## 11.4. Задание 3

1. Ознакомиться с техническим описанием регистров тактирования микропроцессора
2. Произвести настройку тактирования микропроцессора по варианту см. [Варианты для системы тактирования]
3. Выполнить пошаговую отладку

Таблица 8. Варианты для системы тактирования

Номер варианта	Источник тактирования	Частота тактирования
0	HSI	1 МГц
1	HSE	
2	PLL	
3	HSI	
4	HSE	2 МГц
5	PLL	
6	HSI	
7	HSE	4 МГц
8	PLL	

## 11.5. Задание 4

- Сделать программу, которая при нажатии кнопки UserButton на отладочной плате <http://www.waveshare.com/xnucleo-F411RE.htm> меняет состояние всех 4 светодиодов

# Лекция 3

## 1. Таймеры

Одна из основных задач таймеров в микроконтроллерах это отсчитывать точные интервалы времени. Но, помимо этого таймеры могут использоваться для измерения частоты, периодов, генерации ШИМа и переменных сигналов различной формы. Всего

**TIM9-TIM11** Самые простые 16 битные таймеры.

**TIM2-TIM5** Таймеры общего назначения. (TIM2 и TIM5 32 битные) (TIM3 и TIM4 16 битные)

**TIM1** Расширенный 16 битный таймер

- Также существуют системный таймер **SysTick** таймер и Watchdog таймер.

В данном курсе мы рассмотрим только 32 битные таймеры общего назначения. Самостоятельно можно будет ознакомиться с TIM1.

### 1.1. Таймеры TIM2 и TIM5, основные особенности

- Таймеры 32 битные (то есть могут считать до  $2^{32}$ ), умеют работать:
  - с инкрементальными энкодерами и датчиками Холла,
  - несколько таймеров можно синхронизировать между собой.
- Таймеры могут использоваться для:
  - Захвата сигнала (Защелкивать значение, когда на выводе порта например 0 сменился на 1)
  - Сравнения (Считать до значения в регистре сравнения и установить/бросить/переключить вывод порта)
  - Генерации ШИМ (Генерировать прямоугольный сигнал с различной скважностью на вывод порта)
  - Генерации одиночного импульса

## 1.2. Таймеры TIM2 и TIM5

- Таймеры могут генерировать следующие события:
  - Переполнение
  - Захват сигнала
  - Сравнение
  - Событие-триггер

## 1.3. Таймеры TIM2 и TIM5 начальная запуск

- Таймеры тактируются от шины APB1.

Поэтому для каждого отсчет таймера по умолчанию происходит на частоте шины, т.е. если шина **APB1** работает на частоте 1 МГц, то один отсчет таймера произойдет через 1 мкс. Таким образом можно организовать измерение времени с разрешением в 1 мкс. Чтобы таймер заработал, его нужно подключить к системе тактирования, т.е. к шине **APB1**.

- Подключение к системе тактирование выполняется через регистр **APB1ENR** модуля **RCC**.
- Входную частоту таймера можно поделить, записав делитель частоты в регистр **PSC**.
- Включение таймера производится с помощью бита **CEN** в регистре **CR1** модуля таймера (TIM2 или TIM5)

## 1.4. Таймеры TIM2 и TIM5 переполнение

Как только таймер начал считать, его счетчик будет увеличиваться с каждым тактом подающейся на таймер частоты. Т.е. если входная частота таймера 1 МГц, то через секунду таймер достчитает до 1 000 000.

- Значение счетчика таймера можно прочитать из регистра **CNT**.
  - Поскольку таймеры **TIM2** и **TIM5** 32 битных, то переполнение наступит когда в регистре **CNT** будет значение **0xFFFFFFFF**, нетрудно посчитать, что при частоте работе таймера 1 МГц он переполнится через примерно 71.5 минуты.
  - При переполнении таймера, он сгенерирует событие (запрос на прерывание).
- Проверить случилось ли переполнение можно, считав бит **UIF** в регистре **CR**.

## 1.5. Таймеры TIM2 и TIM5 режим счета до значения

Допустим, нам нужно раз в 71.5 минуты моргнуть светодиодом. Мы можем запустить таймер и постоянно проверять значение бита **UIF**, как только оно установится в 1, моргнуть светодиодом.

- Используя переполнение невозможно задать таймером произвольный интервал времени.
- Задать производльный интервал можно, используя регистр автоперезагрузки **ARR**. В

этот регистр записывается число, до которого будет идти счет. При достижении этого значения, содержимое счетчика CNT обнуляется и формируются прерывание или запрос DMA (если они разрешены).

**Например:** мы хотим раз в 1 секунду моргать светодиодом. Частота работы таймера 1 МГц. Чтобы таймер генерировал запрос на прерывание каждые 1 секунду, нужно записать число 1 000 000 в регистр ARR и число 0 в регистр CNT и после этого запустить таймер. Как только таймер досчитает до 1 000 000 он выставит флаг UIF.

## 1.6. Таймеры TIM2 и TIM5 регистры для режима счета

### TIMx::CNT

Счетный 16/32 разрядный регистр таймера суммирующий, с приходом каждого тактового импульса инкрементирует свое содержимое. На вычитание работать не может.

### TIMx::PSC

16 разрядный регистр - делитель частоты для таймера. Коэффициент деления задается в 16-разрядном регистре, этот коэффициент можно задать в пределах от 1 до 65536.

### TIMx::ARR

16/32 разрядный регистр автоперезагрузки. В этот регистр записывается число, до которого будет идти счет. При достижении этого значения, содержимое счетчика TIMx\_CNT обнуляется и формируются прерывание или запрос DMA (если они разрешены).

### TIMx::SR

Регистр статуса. Можно узнать о всех возможных запросах на прерывания от таймера

## 1.7. Таймеры TIM2 и TIM5. Управляющий регистр (CR1)

Основные настройки таймера производятся через регистр CR1. Нам понадобятся всего несколько бит.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved				CKD[1:0]		ARPE	CMS		DIR	OPM	URS	UDIS	CEN		
				rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Рисунок 42. Регистр CR1

**Bit 2: URS** Источник генерации прерываний

- **0:** Любые из следующих событий будут генерировать прерывание или запрос DMA, если они включены:
  - Переполнение счетчика или установлен UG бит
- **1:** Только после переполнения счетчика может сгенерировать прерывание или запрос DMA

**Bit 1: UDIS** Отключить событие по изменению (Update Event)

- **0:** UEV включен. Событие по изменению(UEV) генерируются следующими событиями:
  - Переполнение счетчика или установлен UG бит
- **1:** UEV отключен.

**Bit 0 CEN** Включить счетчик

- **0:** Counter выключен
- **1:** Counter включен

## 1.8. Таймеры TIM2 и TIM5. Регистр статуса (SR)

Регистр SR хранит статусы запросов на прерывания

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved		CC4OF	CC3OF	CC2OF	CC1OF	Reserved	TIF	Res	CC4IF	CC3IF	CC2IF	CC1IF	UIF		

Рисунок 43. Регистр SR

**Bit0: UIF** Флаг прерывания по событию обновления. Бит устанавливается аппаратно, сбрасываться должен программно

- **0:** Флаг прерывания сброшен
- **1:** Флаг прерывания установлен

## 1.9. Работа с таймером в качестве счетчика

Для организации задержки

- Подать тактирование на модуль таймера
- Установить делитель частоты для таймера в регистре PSC
- Установить источник генерации прерываний по событию переполнение с помощью

бита **URS** в регистре **CR1**

- Установить значение до которого счетчик будет считать в регистре перезагрузке **ARR**
- Скинуть флаг генерации прерывания **UIF** по событию в регистре **SR**
- Установить начальное значение счетчика в 0 в регистре **CNT**
- Запустить счетчик с помощью бита **EN** в регистре **CR1**
- Проверять пока не будет установлен флаг генерации прерывания по событию **UIF** в регистре **SR**
- Как только флаг установлен остановить счетчик, сбросить бит **EN** в регистре **CR1**, Сбросить флаг генерации прерывания **UIF** по событию в регистре **SR**

## 1.10. Задание 1. Простое

- Светодиоды должны гореть раз в 500 мс
- Сделать задержку на 500, 1000, 1500 мс, вместо цикла `for(..)` с помощью таймера

# Лекция 5

## 1. Синхронный и асинхронный интерфейсы

**Асинхронный способ передачи данных** — такой способ передачи цифровых данных от передатчика к приемнику по последовательному интерфейсу, при котором данные передаются в любой момент времени. Синхронизация идет по времени — приемник и передатчик заранее договариваются о том на какой частоте будет идти обмен

**Синхронны способ передачи данных** - способ передачи цифровых данных по последовательному интерфейсу, при котором приемнику и передатчику известно время передачи данных, то есть, передатчик и приемник работают синхронно, в такт.

## 2. Асинхронный способ передачи

## Кодирование информации в RS-232C



## 3. Асинхронный интерфейс UART

**Universal Asynchronous Receiver Transmitter** – «универсальный асинхронный приемопередатчик».

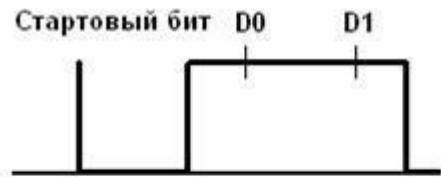
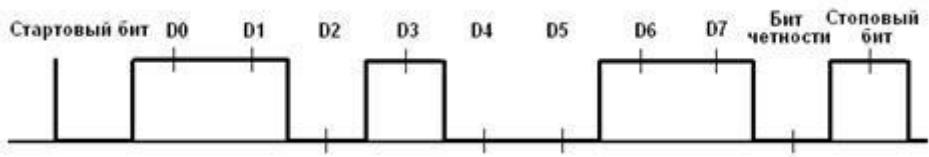
UART, ввиду своей простоты долгое время являлся стандартным последовательным интерфейсом для передачи данных, как в микроконтроллерных устройствах, так и в микропроцессорных.

Сейчас он больше распространен в микроконтроллерных устройствах, а в компьютерах заменен на USB.

Существуют микросхемы преобразования USB-UART для подключения устройств с UART интерфейсом к компьютерам

### 3.1. Протокол передачи данных

Для асинхронной передачи UART достаточно всего двух сигнальных линий – TX (Transmit) и RX (Receive). В начале передачи передатчик устанавливает линию в низкий уровень — это старт бит. Приемник, определив, что линия просела, интервал равный времени одного такта и считывает первый бит. С каждым тактом, передатчик выставляет новый бит, а приемник принимает их. Последний бит это стоп бит.



### 3.2. Прием и передача данных

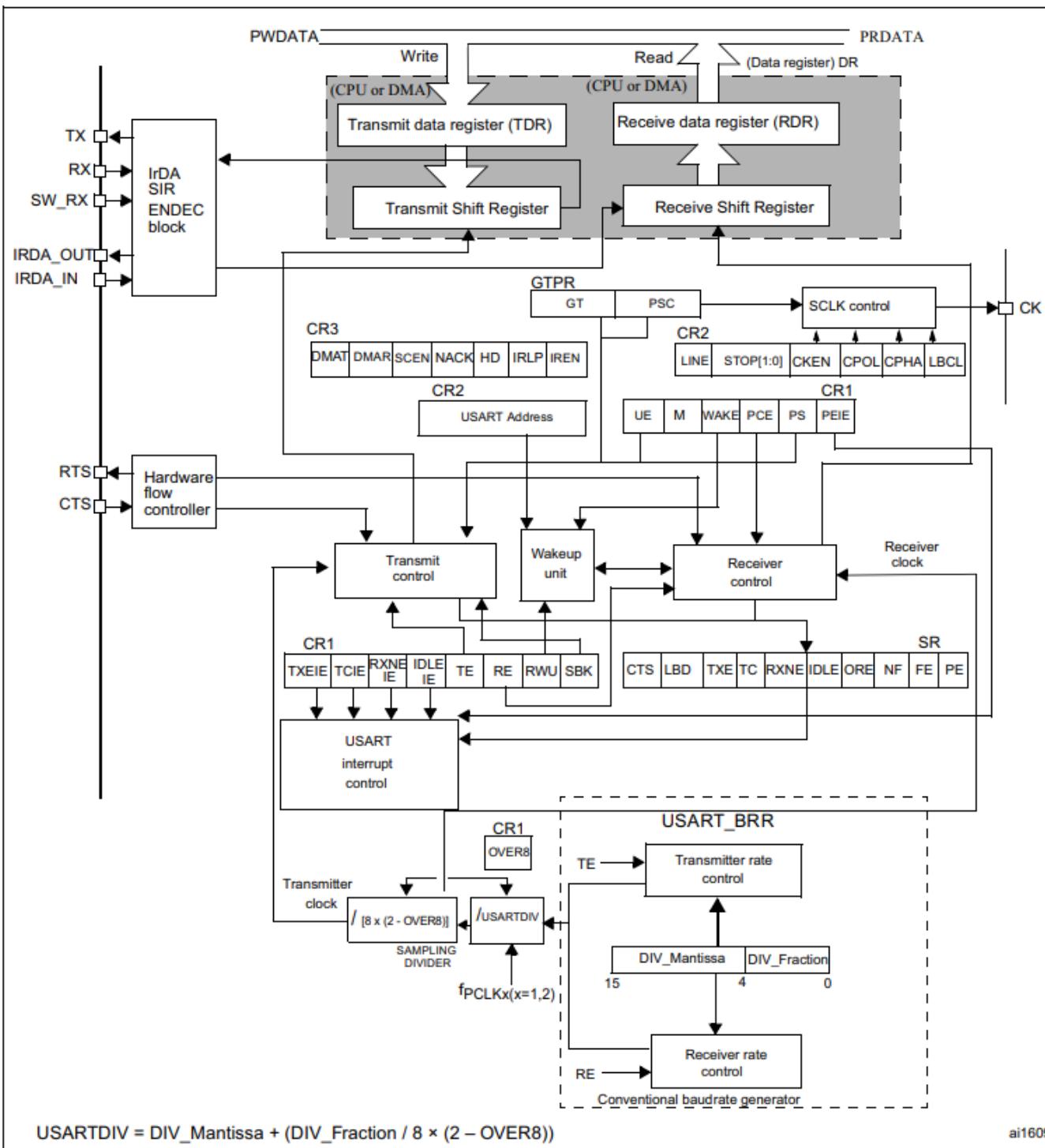
При приеме биты данных из линии с каждым тактом попадают в свивовый регистр. Как только сдвиговый регистр заполнился, данные переходят в регистр данных, откуда их можно считать программой.

[Image5] | *Image5.gif*

Тоже самое, но в обратном порядке происходит во время передачи.

## 4. Модуль UART в микроконтроллера STM32F411

3 Модуля USART - USART1, USART2 и USART6



$$\text{USARTDIV} = \text{DIV\_Mantissa} + (\text{DIV\_Fraction} / 8 \times (2 - \text{OVER8}))$$

ai1609

## 4.1. Общее описание модуля USART

- Гибкая система установки скорости передачи
- Программируемая длина слова (8 или 9 бит)
- Возможность конфигурации количества стоп битов (1 или 2)
- Контроль честности (четное количество 1 или нечетное)
- Независимое включение передатчика и приемника
- Конфигурируемый DMA для приема и передачи сообщений
- 4 Флага детектора ошибок: (Overrun error, Noise detection, Frame error, Parity error)

- 10 флагов прерываний:
  - Transmit data register empty
  - Transmission complete
  - Receive data register full
  - Overrun error, Framing error, Noise error, Parity error
  - CTS changes, LIN break detection, Idle line received
- Мульти процессорная коммуникация
- Поддержка LIN протокола, Поддержка ИК порта IrDA SIR(кодер и декодер), Поддержка SmartCard (возможность общения с SIM карты)

## 4.2. Особенности USART

UASRT STM микроконтроллера очень обширный, но мы рассмотрим только то, что относится к UART

В модуле USART можно настраивать следующие параметры:

- Скорость обмена до 4 мбит/с
- Контроль четности
- 1 или 2 стоповых битов
- 8 или 9 бит данных
- Запросы на детектирование ошибок приемо-передачи
- Прерывания по приему, передачи, ошибкам передачи

Для настройки и работы модуля UART нужны всего несколько регистра

- USART\_CR1/CR2/CR3 - регистр настройки 1
- USART\_DR - регистр принятого символа (регистр данных)
- USART\_BRR – регистр настройки скорости передачи
- USART\_SR - регистр состояния

## 4.3. Регистр CR1 - регистр управления

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OVER8	Reserved	UE	M	WAKE	PCE	PS	PEIE	TXEIE	TCIE	RXNEIE	IDLEIE	TE	RE	RWU	SBK
rw	Res.	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

**Bit15: OVER8** Режим дискретизации

- 0: 1/16
- 1: 1/8

**Bit13: UE** Включение модуля USART

- 0: Отключить
- 1: Включить

**Bit12: M** Длина символа

- 0: 1 Стартовый бит, 8 бит данных
- 1: 1 Стартовый бит, 9 бит данных

**Bit7: TXEIE** Разрешить прерывание по передаче

**Bit6 TCIE** Разрешить прерывание по концу передачи

**Bit5: RXNEIE** Разрешить прерывание по приему

**Bit3: TE** Разрешить передачу

**Bit2: RE** Разрешить прием

## 4.4. Регистр SR - регистр статуса

Reset value: 0x00C0 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved					CTS	LBD	TXE	TC	RXNE	IDLE	ORE	NF	FE	PE	
					rc_w0	rc_w0	r	rc_w0	rc_w0	r	r	r	r	r	

**Bit7: TXE** Регистр данных передачи пуст. Этот бит устанавливается аппаратно, когда содержимое регистра данных передачи перемещается в сдвиговый регистр. Установка этого бита может генерировать прерывание, если установлен TXEIE бит = 1 в регистре USART\_CR1. Этот бит очищается когда происходит запись в регистр данных UASRT\_DR.

- 0: Данные не перемещены в сдвиговый регистр
- 1: Данные перемещены в сдвиговый регистр

**Bit6: TC** Передача завершена. Этот бит устанавливается когда сдвиговый регистр тоже опустошился и стоит бит TXE. Установка этого бита также может генерировать прерывание если установлен бит TCIE=1 в регистре USART\_CR1. Очищается программно, путем записи 0

- 0: Передача не завершена
- 1: Передача завершена

**Bit5: RXNE** Регистр данных чтения не пуст. Этот бит устанавливается когда содержимое сдвигового регистра перемещается в регистр данных USART\_DR. Установка этого бита генерирует прерывание , если установлен бит RXNEIE=1 в регистре USART\_CR1. Этот бит очищается, сразу после чтения из регистра данных USART\_DR. Также этот бит может быть очищен посредством записи 0 в него

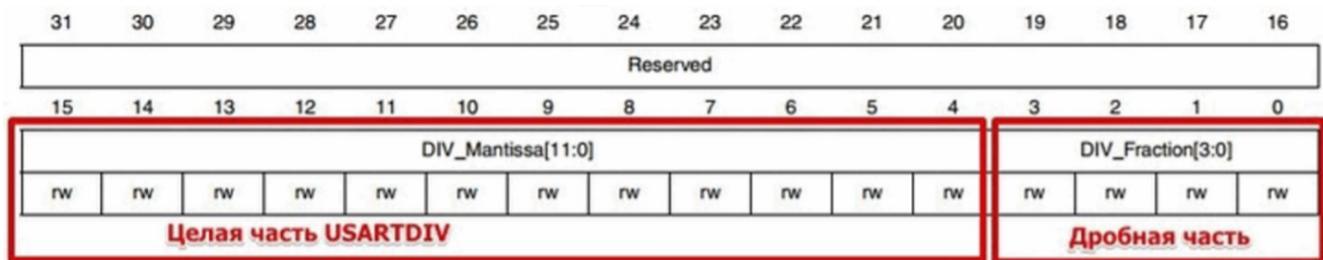
- 0: Данные не приняты
- 1: Данные готовы для чтения

## 4.5. Регистр данных

**USART\_DR** — Регистр данных. При передачи, посылаемый символ должен быть записан в этот регистр. При приеме, принятый символ нужно прочитать из этого регистра. Занимает 32 бита из которых используются только 9 (!) первых бит, остальные принудительно зануляются аппаратно.

## 4.6. Регистра настройки скорости передачи

**USART\_BRR** — Регистр настройки скорости передачи. Первые его два байта определяют частоту передачи. Вторые принудительно ноль.



Для вычисления скорости используется следующая формула

- USARTDIV = CLK/(BaudRate\*8\*(2 - OVER8))

## 4.7. Порядок запуска модуля UART

- Подключить USART к источнику тактирования – устанавливаем бит USART2EN в регистре APB1ENR (АЦП тактируется от матрицы шин APB1).

- Необходимо сконфигурировать порты. Настроить порты, на альтернативную функцию нужного модуля USART
- Настроить формат передачи байт, с помощью регистра CR1 и CR2
- Задать скорость передачи с помощью регистра BRR
- Разрешить передачу помошью бита TE и если надо прием, с помощью бита RE в модуле USART с помощью регистра CR1
- Включить сам модуль USART битом UE в регистре CR1
- Если работаем через прерывание, то разрешить глобальное прерывание для нужного USART, в регистре ISER[1] модуля NVIC
- Если работаем через прерывание, в зависимости от того, что нам нужно, разрешить прерывание по сигналу модуля UART (например, от сигнала регистра данных передачика пуст (бит TXEIE в регистре CR1))

## 5. Задание

Передавать раз в 0.5 секунды фразу Hello World!

- Подключиться к внешнему источнику тактирования
- Настроить таймер 2 на 0.5 секунды
- Подключить модуль UART2 к шине тактирования
- Настроить порты PORT A.2 как TX, Port A.3 как RX на альтернативную функцию работы с UART в режим Push-Pull(двуухтактный выход) + Pull Up(подтяжка к 1)
- Настроить USART2 на скорость 9600 бит/с, 1 стоп бит, 1 старт бит, без проверки четности, режим дискретизации 1/16, 8 бит данных.
- Включить USART2
- Включить переду данных
- Послать сообщение используя программный буфер

### 5.1. Дома сделать тоже самое с использованием прерывания

- Подключиться к внешнему источнику тактирования
- Настроить таймер 2 на 0.5 секунды
- Подключить модуль UART2 к шине тактирования
- Настроить порты PORT A.2 как TX, Port A.3 как RX на альтернативную функцию работы с UART в режим Push-Pull(двуухтактный выход) + Pull Up(подтяжка к 1)
- Настроить USART2 на скорость 19200 бит/с, 2 стоп бит, 1 старт бит, без проверки четности, режим дискретизации 1/8, 8 бит данных.
- Разрешить глобальное прерывание по USART

- Разрешить прерывание по передаче
- Включить USART2
- Включить переду данных

# Лекция 6

## 1. Аналогово-Цифровой преобразователь

Устройство, преобразующее входной аналоговый сигнал в дискретный код. Как правило, АЦП — электронное устройство, преобразующее напряжение в двоичный цифровой код. Тем не менее, некоторые неэлектронные устройства с цифровым выходом следует также относить к АЦП, например, некоторые типы преобразователей угол-код. Простейшим одноразрядным двоичным АЦП является компаратор.

— Wikipedia

Статья для общего представления есть на habre:

[Аналого-цифровой преобразователь для самых маленьких](#)

АЦП бывают **линейными** и нелинейными, мы будем работать только с **линейными** АЦП

### 1.1. Основные характеристики АЦП

- Разрешение
  - Разрядность
  - Эффективная разрядность
- Передаточная характеристика АЦП
- Точность
- Нелинейность
- Ошибки квантования
- Частота дискретизации

### 1.2. Разрешение

Разрешение АЦП — минимальное изменение величины аналогового сигнала, которое может быть преобразовано данным АЦП.

- В случае единичного измерения без учёта шумов разрешение напрямую определяется разрядностью АЦП.
- Разрядность АЦП характеризует количество дискретных значений, которые преобразователь может выдать на выходе.
- В двоичных АЦП измеряется в битах. Например, двоичный 8-разрядный АЦП способен выдать 256 дискретных значений (0...255),  $2^8=256$ .

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

— Wikipedia

Пример :

- Диапазон входных значений = от 0 до 3 вольт
- Разрядность двоичного АЦП 12 бит:  $2^{12} = 4096$  уровней квантования
- Разрешение двоичного АЦП по напряжению:  $(3-0)/4096 = 0,0007324$  вольт = 0,7324 мВ

## 1.3. Эффективная разрядность

На практике разрешение АЦП ограничено отношением сигнал/шум входного сигнала. При большой интенсивности шумов на входе АЦП различие соседних уровней входного сигнала становится невозможным.

При этом реально достижимое разрешение описывается эффективной разрядностью (англ. effective number of bits, ENOB), которая меньше, чем реальная разрядность АЦП.

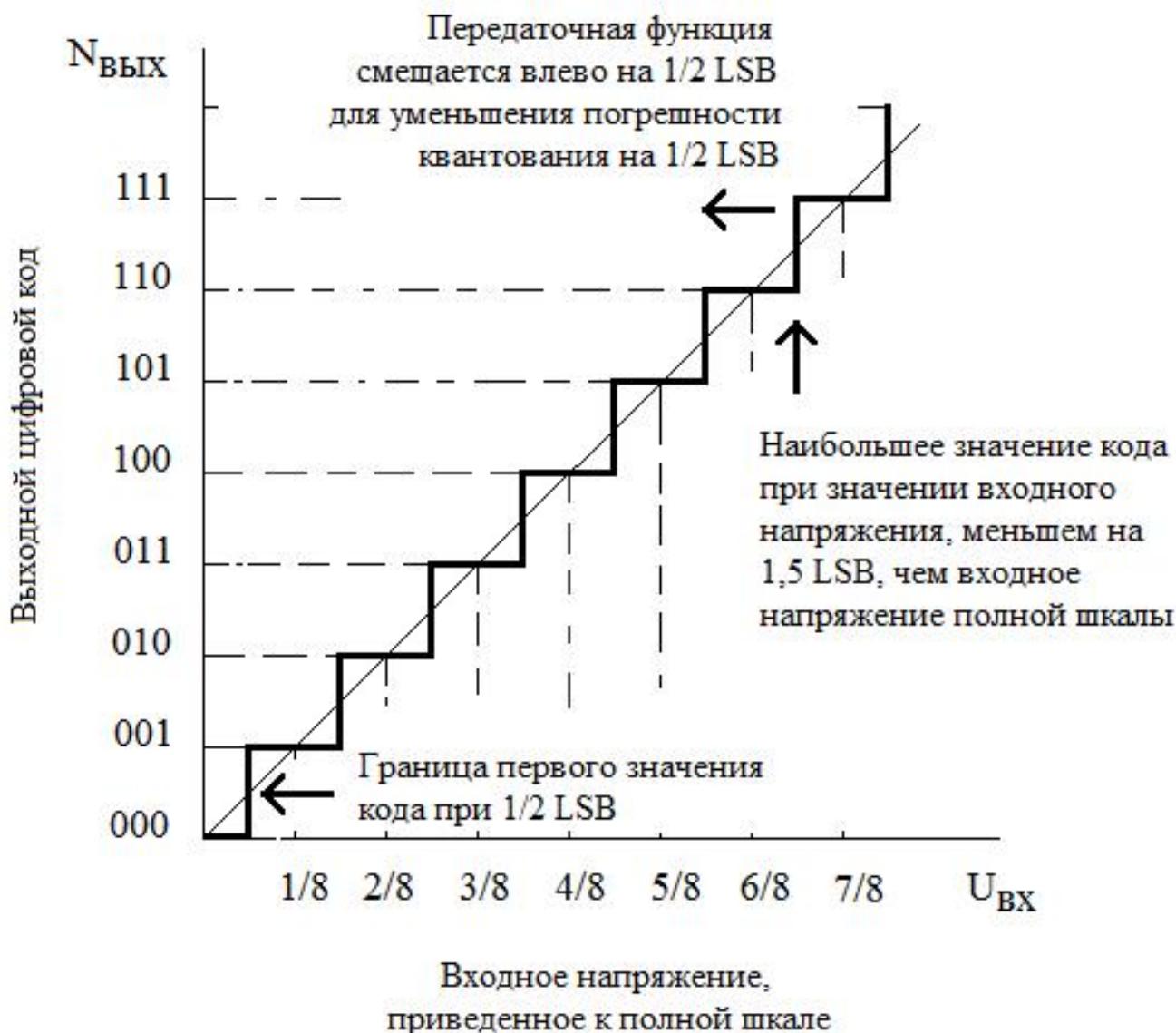
При преобразовании сильно зашумлённого сигнала младшие разряды выходного кода практически бесполезны, так как содержат шум. Для достижения заявленной разрядности отношение сигнал/шум входного сигнала должно быть примерно 6 дБ на каждый бит разрядности (6 дБ соответствует двукратному изменению уровня сигнала).

— Wikipedia

## 1.4. Передаточная характеристика АЦП

Передаточная характеристика АЦП — зависимость числового эквивалента выходного двоичного кода от величины входного аналогового сигнала.

Для линейных АЦП всегда возможно провести такую прямую линию, чтобы все точки передаточной характеристики, соответствующие входным значениям  $\delta \cdot 2^k$  где  $\delta$  — шаг дискретизации,  $k$  лежит в диапазоне 0..N, где N — разрядность АЦП, были от неё равноудалены.



## 1.5. Точность

Имеется несколько источников погрешности АЦП.

- Ошибки квантования и нелинейности присущи любому аналого-цифровому преобразованию.
- Апертурные ошибки, которые являются следствием джиттера (англ. jitter) тактового генератора, они проявляются при преобразовании сигнала в целом (а не одного отсчёта).

Эти ошибки измеряются в единицах, называемых МЗР — младший значащий разряд (LSB англ.).

— Wikipedia

Для 12-битного двоичного АЦП ошибка в 1 МЗР составляет  $1/4096$  от полного диапазона сигнала, то есть 0,0244 %.

## 1.6. Ошибка квантования

- Ошибки квантования являются следствием ограниченного разрешения АЦП. Этот недостаток не может быть устранён ни при каком типе аналого-цифрового преобразования.
- Абсолютная величина ошибки квантования при каждом отсчёте находится в пределах от нуля до половины МЗР. В общем случае можно считать, что ошибка квантования равна половине МЗР.

— Wikipedia

## 1.7. Нелинейность

Всем АЦП присущи ошибки, связанные с нелинейностью, которые являются следствием физического несовершенства АЦП.

Это приводит к тому, что передаточная характеристика (в указанном выше смысле) отличается от линейной (точнее от желаемой функции, так как она не обязательно линейна). Такие ошибки могут быть уменьшены путём калибровки.

— Wikipedia

## 1.8. Частота дискретизации

Аналоговый сигнал является непрерывной функцией времени, в АЦП он преобразуется в последовательность цифровых значений. Следовательно, необходимо определить частоту

выборки цифровых значений из аналогового сигнала.

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

Непрерывно меняющийся сигнал с ограниченной спектральной полосой подвергается оцифровке (то есть значения сигнала измеряются через интервал времени  $T$  — период дискретизации), и исходный сигнал может быть точно восстановлен из дискретных во времени значений путём интерполяции. Точность восстановления ограничена ошибкой квантования. Однако в соответствии с теоремой Котельникова — Шеннона точное восстановление амплитуды возможно, только если частота дискретизации выше, чем удвоенная максимальная частота в спектре сигнала.

Поскольку реальные АЦП не могут произвести аналого-цифровое преобразование мгновенно, входное аналоговое значение должно удерживаться постоянным 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.

## 2. Типы АЦП

- АЦП прямого(параллельные АЦП) преобразования
- АЦП последовательного приближения
- Сигма-дельта АЦП
- АЦП дифференциального кодирования
- АЦП сравнения с пилообразным сигналом
- АЦП с уравновешиванием заряда
- Оптические АЦП

### 2.1. АЦП последовательного преобразования

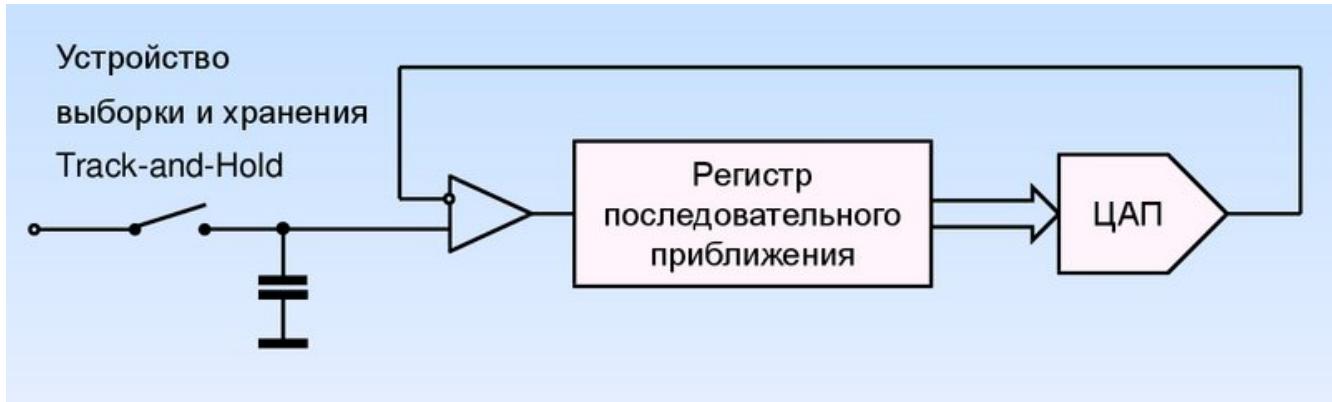
АЦП последовательного приближения работает методу половинного деления.

Пример для 8 битного двоичного АПЦ:

- На компаратор подается значение вначале равное половине опорного напряжения( $U_{оп}/2$ ) (соответствующее установленном старшем бите 1000 0000 $b$ )
- Если компаратор сработал старший бит скидывается, выставляется 1/4 опорного напряжения ( $U_{оп}/4$ ) ( 0100 0000 $b$ )
- Если компаратор не сработал старший бит остается, и выставляется 3/4 опорного

напряжения ( $U_{op}/4$ ) ( 1100 0000b)

- И так далее до самого младшего бита.



## 2.2. Сигма Дельта АЦП

Дома, подготовить доклад

## 2.3. АЦП прямого действия

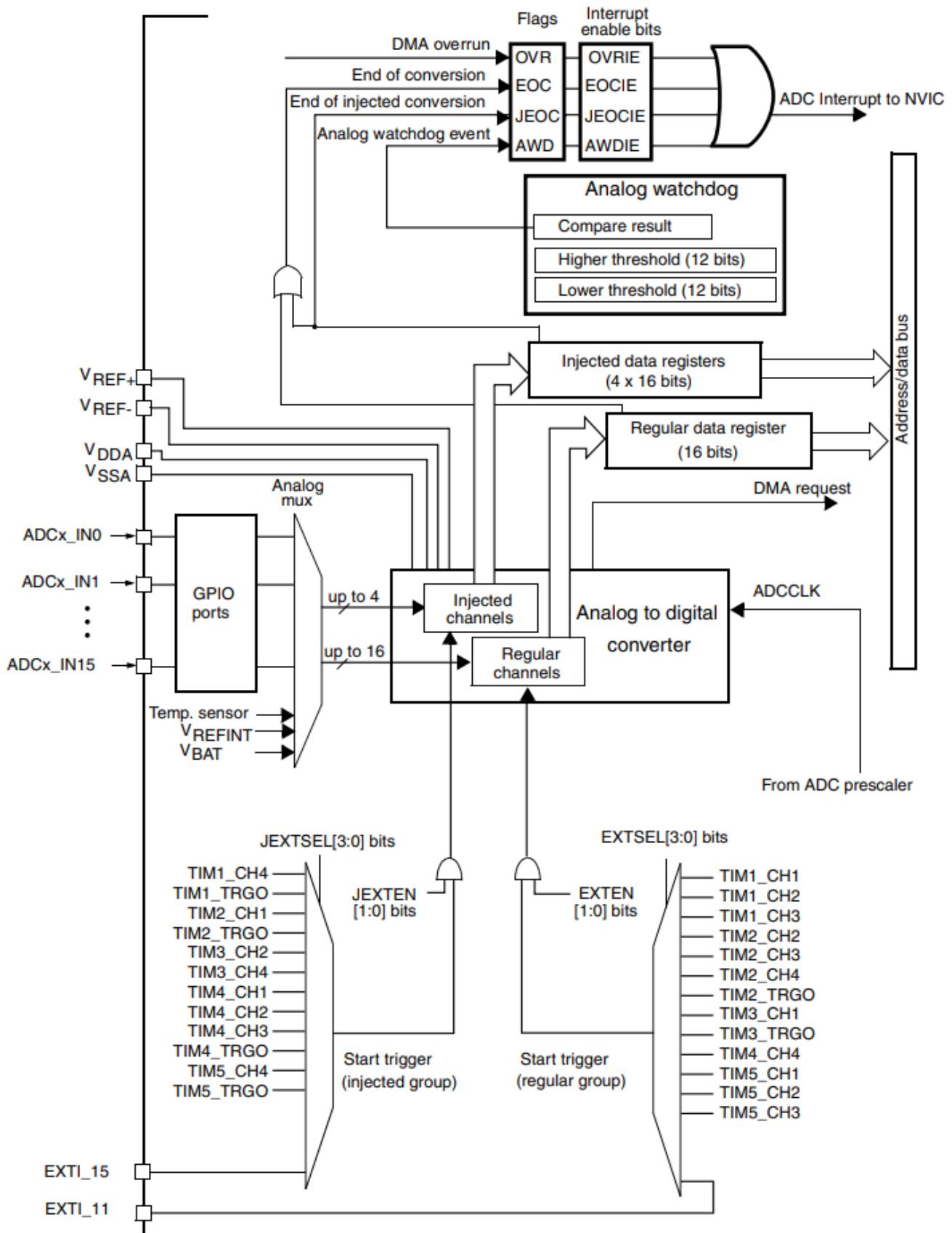
Дома, подготовить доклад

# 3. АПП микроконтроллера STM32F411

АЦП(макс 12 разрядов) микроконтроллера STM32F411 работает по принципу последовательного приближения.

- Основные элементы АЦП:
  - Наличие регулярных и инжектированных каналов – отличие только в том, что инжектированные каналы могут писать данные в 4 регистра с 4 каналов сразу, а регулярный только в один регистр
  - 19 аналоговых каналов, 16 из которых могут сконфигурированы на работу от внешних источников или 3 внутренних.
  - Внешние каналы поступают на мультиплексор, где выбирается только один из них. Т.е. в один момент времени может быть измерено напряжение только с одного канала.
  - Результат преобразования сохраняется в регистрах данных. Для регулярных каналов это только один 16 битный регистр. Для инжектированных – 4.
  - Запуск преобразования может быть как программным, так и внешним. Внешний запуск может происходить от таймеров или с двух внешних входов.

## 3.1. Схема АЦП микроконтроллера STM32F411



### 3.2. Особенности АЦП микроконтроллера STM32F411

- Разрядность АЦП можно изменять
  - 6, 8, 10, или 12 разрядов.

- Для одного канала можно задать разные режимы:
  - однократно измерить аналоговую величину
  - запустить канал в режиме непрерывного измерения.
- Режим сканирования
  - Можно задать группу каналов и порядок следования каналов в группе. Тогда измерения будут идти последовательно друг за другом, входной мультиплексор будет подключать внешние каналы к АЦП по очереди, в соответствии с запрограммированным порядком.
- Функция внешнего запуска для регулярных и инжектированных каналов.
- Режим “прерывистых” преобразований.
- Время одного преобразования зависит от частоты тактирования АЦП и времени скорости дискретизации , которое можно настроить.
  - $T_{conv} = Sampling\ time + 12\ cycles$ . При 1 МГц, а время дискретизации 3 циклам, то полное время преобразование будет 15 тактов или 15 мкс.
- Размах входного сигнала не должен выходить за пределы опорного напряжения Vref.
- Возможность введения временной задержки, автоматически вставляемой между преобразованиями. Длительность задержки программируется.
- Генерация запроса для прямого доступа к памяти (режим DMA) во время преобразования в регулярном канале.

### **3.3. Режим одиночного преобразования**

В этом режиме АЦП находится сразу после сброса. Бит CONT регистра ADC\_CR2 равен 0.

- Для начала работы с АЦП в этом режиме нужно
  - Настроить нужный порт, подключенный к нужному каналу АЦП на аналоговый вход
  - Подать тактирование на АЦП
  - Выбрать нужный канал для измерения
  - Настроить канал АЦП на нужную частоту преобразования
  - Включить АЦП
  - Начать преобразование
  - Дождаться флага готовности преобразования
  - Считать преобразованное значение

### **3.4. Режим сканирования**

В этом режиме опрашивается группа каналов.

- Режим выбирается установкой бита SCAN в регистре ADC\_CR1.
  - АЦП опрашивает все каналы, выбранные в регистрах ADC\_SQRx (регулярные каналы)

- Для каждого канала группы выполняется одиночное преобразование. После окончания каждого преобразования следующий канал в группе опрашивается автоматически.
- Если установлен режим непрерывного преобразования (CONT = 1), то после последнего преобразования в группе, преобразования не прекращаются, а заново начинаются от первого выбранного канала в группе.
- Если установлен бит DMA, то данные из регистра ADC\_DR пересылаются в память после каждого преобразования.
- В регулярных каналах флаг окончания преобразования EOC может устанавливаться либо в конце каждого преобразования, либо только по окончании всей последовательности. Для этого в регистре ADC\_CR2 предусмотрен управляющий бит EOCS
  - EOCS = 0 – бит окончания преобразования EOC устанавливается после завершения всей последовательности регулярных преобразований.
  - EOCS = 1 – бит окончания преобразования EOC устанавливается после завершения каждого регулярного преобразования

### 3.5. Регистр статуса SR (ADC status register)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved								OVR	STRT	JSTRT	JEOC	EOC	AWD		
								rc_w0	rc_w0	rc_w0	rc_w0	rc_w0	rc_w0		

#### Bit5: OVR

Переполнение. Бит указывает, что данные преобразования регулярного канала были потеряны, т.е. программа не успела считать регистр данных регулярного канала, до следующего преобразования.

- **0:** было переполнение
- **1:** переполнения не было

#### Bit4: STRT

Флаг начала преобразования регулярного канала

- **0:** преобразование начато
- **1:** преобразование не начато

#### Bit1: EOC

Преобразование регулярных каналов закончено. Можно считывать регистр данных.

- **0:** преобразование не закончено
- **1:** преобразование закончено

### 3.6. Регистр управления CR1 (ADC control register 1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	
Reserved				OVRIE	RES		AWDEN	JAWDEN	Reserved							
				rw	rw	rw	rw	rw								
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
DISCNUM[2:0]			JDISCN	DISCEN	JAUTO	AWDSGL	SCAN	JEOCIE	AWDIE	EOCIE	AWDCH[4:0]					
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

#### Bits25..24: RES[1:0]

разрядность АЦП

- **00:** разрядность 12 бит (время преобразования  $12 + 4 = 16$  тактов)
- **01:** разрядность 10 бит (время преобразования  $11 + 4 = 15$  тактов)
- **10:** разрядность 8 бит (время преобразования  $9 + 4 = 13$  тактов)
- **11:** разрядность 6 бит (время преобразования  $7 + 4 = 11$  тактов)

#### Bit8: SCAN

Включение режима сканирования. В этом режиме опрашивается группа каналов. АЦП опрашивает все каналы, выбранные в регистрах ADC\_SQRx (регулярные каналы). Для каждого канала группы выполняется одиночное преобразование. После окончания каждого преобразования следующий канал в группе опрашивается автоматически. Если установлен режим непрерывного преобразования (CONT = 1), то после последнего преобразования в группе, преобразования не прекращаются, а заново начинаются от первого выбранного канала в группе.

- **0:** режим сканирования выключен
- **1:** режим сканирования включен

#### Bit5: EOCIE

Разрешает прерывание по установке флага окончание преобразования EOC

- **0:** прерывание запрещено
- **1:** прерывание разрешено

### 3.7. Регистр управления CR2 (ADC control register 2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
reserved	SWSTART	EXTEN		EXTSEL[3:0]				reserved	JSWSTART	JEXTEN		JEXTSEL[3:0]			
	rw	rw	rw	rw	rw	rw	rw		rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
reserved				ALIGN	EOCS	DDS	DMA	Reserved				CONT	ADON	rw	rw
				rw	rw	rw	rw								

<b>Bit30: SWSTART</b>	<b>Bit9: DDS</b>
начать преобразование регулярного канала. Устанавливается программно, скидывает аппаратно.	режим выключения DMA <ul style="list-style-type: none"> <li>• <b>0:</b> DMA запросы не будут выполняться после последнего переданного данного</li> <li>• <b>1:</b> DMA будут выполняться всегда, как только данные подготовлены и пока включен бит DMA</li> </ul>
<b>Bit10: EOCS</b>	<b>Bit8: DMA</b>
Выбор типа окончания преобразования	Включение DMA <ul style="list-style-type: none"> <li>• <b>0:</b> DMA выключен</li> <li>• <b>1:</b> DMA включен</li> </ul>
	<b>Bit1: CONT</b>
	Включение режима непрерывного преобразования <ul style="list-style-type: none"> <li>• <b>0:</b> Режим единичного преобразования</li> <li>• <b>1:</b> Режим непрерывного преобразования</li> </ul>
	<b>Bit0: ADON</b>
	Включение АЦП <ul style="list-style-type: none"> <li>• <b>0:</b> Отключить АЦП и перейти в режим энергопреобразования</li> <li>• <b>1:</b> Включить АЦП</li> </ul>

### 3.8. Регистр настройки времени дискретизации АЦП SMPRx(ADC sample time register)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved				SMP18[2:0]			SMP17[2:0]			SMP16[2:0]			SMP15[2:1]		
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SMP15_0	SMP14[2:0]			SMP13[2:0]			SMP12[2:0]			SMP11[2:0]			SMP10[2:0]		
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved	SMP9[2:0]			SMP8[2:0]			SMP7[2:0]			SMP6[2:0]			SMP5[2:1]		
	rw	rw	rw												
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SMP5_0	SMP4[2:0]			SMP3[2:0]			SMP2[2:0]			SMP1[2:0]			SMP0[2:0]		
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

## Bits0..26 SMPx[2:0]

Выбор времени дискретизации для канала от x.

- **000**: 3 cycles
- **001**: 15 cycles
- **010**: 28 cycles
- **011**: 56 cycles
- **100**: 84 cycles
- **101**: 112 cycles
- **110**: 144 cycles
- **111**: 480 cycles

## 3.9. Регистр настройки последовательности преобразований SQR (ADC regular sequence register 1 )

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved								L[3:0]				SQ16[4:1]			
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SQ16_0	SQ15[4:0]						SQ14[4:0]				SQ13[4:0]				
rw	rw	rw	rw	rw	rw	rw	rw					rw	rw	rw	rw

## Bits20..23: L[3:0]

Длина последовательности преобразований

- **0000**: 1 преобразование
- **0001**: 2 преобразований
- .....
- **1111**: 16 преобразований

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved		SQ12[4:0]					SQ11[4:0]					SQ10[4:1]			
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SQ10_0	SQ9[4:0]						SQ8[4:0]				SQ7[4:0]				
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

## SQx

Номер канала для x преобразования

## 3.10. Регистр данных DR (ADC data register)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DATA[15:0]															
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

#### Bits0..15: DATA[15:0]

Данные преобразования регулярного канала

### 3.11. Общий регистр управления CCR (ADC common control register ADC\_Common)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved								TSVREFE	VBATE	Reserved				ADCPRE	
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved															

#### Bits23: TSVREFE

Подключить сенсор температуры и Vref

- 0: Отключить сенсора температуры и Vref
- 1: Включить сенсор температуры и Vref

#### Bits16:17: ADCPRE

установить частоту работы АЦП

- 00: частота равна PCLK2/2
- 01: частота равна PCLK2/4
- 10: частота равна PCLK2/6
- 11: частота равна PCLK2/8

## 4. Порядок запуска одиночного АЦ преобразования

- Подключить АЦП к источнику тактирования – устанавливаем бит ADC1EN в регистре RCC::APB2ENR (АЦП тактируется от шины APB2).
- Сконфигурировать порты. Определиться по каким каналам будут проводиться измерения, затем соответствующие выводы портов настроить для работы в аналоговом режиме.
- Сконфигурировать АЦП.
  - Установить разрядность в регистре ADC::CR1
  - Установить режим одиночного преобразование в регистре ADC::CR1 (биты CONT и EOCS установить в нужное значение)

- Установить количество измерений 1 в регистре ADC1::SQR1 бит L
- Выбрать канал для первого преобразования в регистре ADC1::SQR3 биты SQ1
- Установить скорость дискретизации в регистре SMPRx для нужного канала
- Включить АЦП. Это делается установкой бита ADON в регистре ADC::CR2.
- Запустить АЦП на преобразование установкой бита SWSTART в регистре ADC::CR2 для регулярных каналов
- Дождаться готовности бита EOC в регистре ADC::SR
- Считать данные из регистра ADC::DR

## 5. Задание

Измерить температуру микроконтроллера с помощью встроенного датчика температуры.

- Прочитать все АЦП в библии все про встроенный датчик температуры на странице 225
- Включить измерение датчика температуры
- Сконфигурировать АЦП
  - 12 бит
  - Одиночное преобразование
  - Регулярные каналы
  - Время дискретизации 84 цикла
  - Установка ЕОС после каждого измерения регулярного канала
  - Установить первое измерение с канала куда подключен датчик температуры

## 6. Библиография

- ① Недяк С.П., Шаропин Ю.Б. Лабораторный практикум по микроконтроллерам семейства Cortex-M: Методическое пособие по проведению работ на отладочных платах фирмы "Миландр"- Томск. гос. ун-т систем упр. и радиоэлектроники, 2017. - 110 с.
- ② Волков В.Л. Программное обеспечение измерительных процессов. Учебное пособие для студентов технических специальностей дневной, заочной, и заочной форм обучения. /АПИ НГТУ. Арзамас, 2008 – 120 с.
- ③ Руководство по оформлению кода на C++ Стэнфордского университета: <http://stanford.edu/class/archive/cs/cs106b/cs106b.1158/styleguide.shtml>
- ④ Объектно-ориентированное программирование: <https://ru.wikipedia.org/wiki>
- ⑤ Можно ли использовать С++ вместо Си для небольших проектов в микроконтроллерах: <https://habr.com/post/347980/>
- ⑥ AsciiDoc шпаргалка: <https://powerman.name/doc/asciidoc>
- ⑦ Reveal.js: <https://github.com/hakimel/reveal.js#full-setup>
- ⑧ Asciidoctor Reveal.js: <https://asciidoctor.org/docs/asciidoctor-revealjs/#node-javascript-setup>

- ⑨ Где хранятся ваши константы на микроконтроллере CortexM: <https://habr.com/ru/post/453262/>
- ⑩ Обзор одной российской RTOS, часть 4. Полезная теория: <https://habr.com/post/337476/>
- ⑪ Начинаем изучать STM32: Что такое регистры? Как с ними работать? <https://habr.com/ru/post/407083/>
- ⑫ Справочное руководство на микроконтроллер STM32F411 [https://www.st.com/resource/en/reference\\_manual/dm00119316.pdf](https://www.st.com/resource/en/reference_manual/dm00119316.pdf)
- ⑬ Безопасный доступ к полям регистров на C++ без ущерба эффективности (на примере CortexM) <https://habr.com/ru/post/459642/>