**Что такое CMSIS[[1]](#footnote-1)**

1. **Назначение и структура СМSIS**

Итак, **CMSIS** (*Cortex Microcontroller Software Interface Standard*) − это:   
 а) стандарт программного интерфейса микроконтроллеров с ядром Cortex  
 б) стандартная библиотека для всех микроконтроллеров с ядром Cortex.

Т.е. это единый стандарт и средство описаний (на языке С) ресурсов *процессоров Cortex* («кортексов») − центрального узла различных *микроконтроллеров на базе ARM Cortex процессоров* («армов»).

Благодаря этой библиотеке легко переносить код с одного вида ARM Cortex-контроллеров на другой. Конечно периферия у всех «ARM-контроллеров» разная, даже в пределах одной линейки, но вот доступ к регистрам периферии из Си стандартизирован и описан в CMSIS. А то, что едино для всех Cortex M3 /M4 — ядро, контроллер прерываний и системный таймер, то едино и во всех CMSIS и при переносе с контроллера на контроллер правки не требует вообще.

Библиотека состоит из нескольких файлов:

а) Описание ядра стандартные для всех Cortex M3 / Cortex M4

* **core\_cm3.c** / **core\_cm4.c –** файл описаний системных ресурсов
* **core\_cm3.h** / **core\_cm3.h –** ссылочный файл

б) Описание конкретного контроллера (**device**) или (семейства), лежит в CMSIS библиотеке конкретного семейства и «качается» с официального сайта производителя контроллера. Основными являютсят файлы:

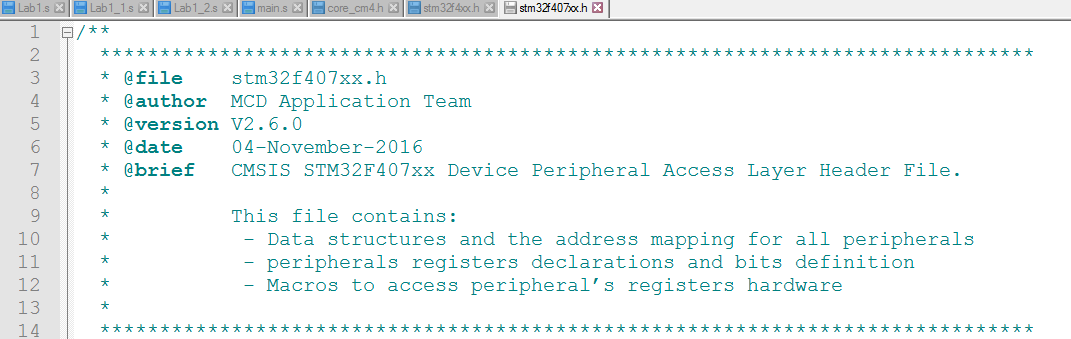
* <**device**>**.h** — файлы описаний периферии, а также структуры доступа к ним   
   (Все самое нужное и важное лежит тут);
* **System\_**<**device**>**.с** файлы функций CMSIS  
  Их немного (**SystemInit**, **SystemCoreClockUpdate**, **SystemCoreClock**);  
  Они имеют одинаковую структуру для всех CMSIS семейств МСU на основе Сortex- M3/- M4 , касаются они только стартовой инициализации контроллерной периферии и работе с установками тактовой частоты;
* **System\_**<**device**>**.h** — заголовочные файлы для указанных функций CMSIS

Для микроконтроллеров **STM32** эти файлы имеют названия:

* **stm32f10x.h** / **stm32f40x.h** ;
* **system\_stm32f10x.c** / **system\_stm32f40x.c** ;
* **system\_stm32f10x.h** / **system\_stm32f40x.h**.

1. **Обзор содержания файлов описания периферии <device>.h**

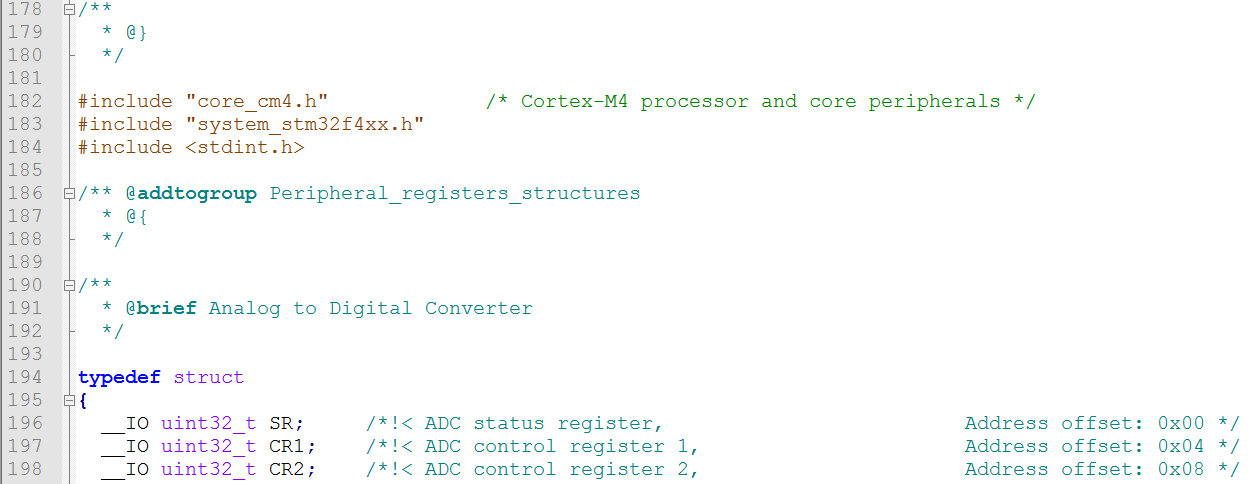
Стуктуру файлов типа <device>.h рассмотрим на примере файла **stm32f4xx.h** и **stm32f407xx.h**[[2]](#footnote-2):

****

Файл начинается с описания типов.

Все регистры периферии в файлах **stm32f4xx.h**   увязаны в конструкцию типа **struct** (структура).

Описание регистров периферии после общего заголовка дано в алфавитном порядке и начинается со структуры с описанием регистров аналого-цифрового преобразователя ADC



Тут в нескольких строках в виде полей структуры указаны все регистры аналого-цифрового преобразователя

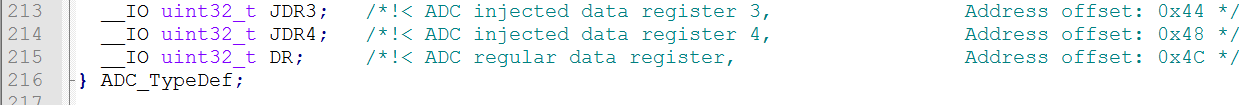


Рис 1

Таких и похожих записей там добрая половина файла. Рекомендую открыть этот файл каким-нибудь NotePad++ (или моим «любимым» Koedit), и оглядеть, что там вообще есть. Пригодится, когда будете работать с разной периферией, чтобы знать, как зовутся те или иные группы периферии.

Прокрутив порядка 4 сотен строк найдем описание типа структуры для регистров общего применения GPIO:

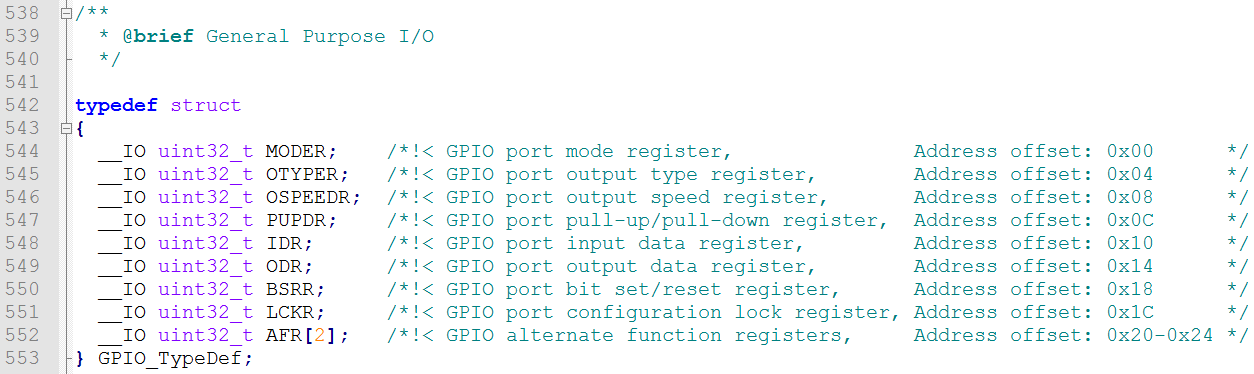


Рис.2

Потом даны значения адресов из **Memory map**. Вначале задается базовые адреса памяти и пространства адресов IO (рис.3), а потом адреса отдельных ресурсов задаются с помощью смещений относительно базы (рис.4):

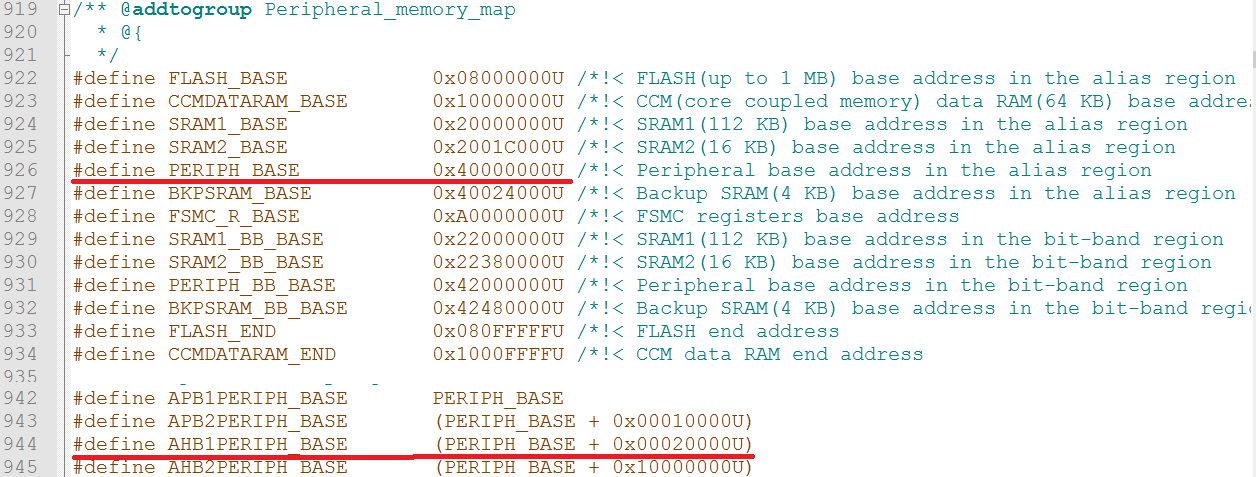


Рис.3 В результате адрес **AHB1PERIPH\_BASE = 0x40020000**

А тут пошли «дефайны» базовых адресов для конкретной периферии AHB1:

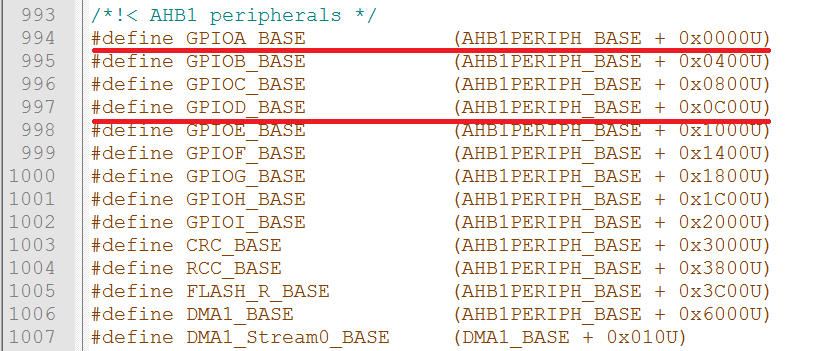


Рис.4

Их там очень много — вся периферия которая есть в STM32 семействе. А ее ОЧЕНЬ много.

После присвоения баз идет уже создание непосредственно указателей на структуры, к которым мы и будем обращаться. Идет прорва записей вида (рис.5):

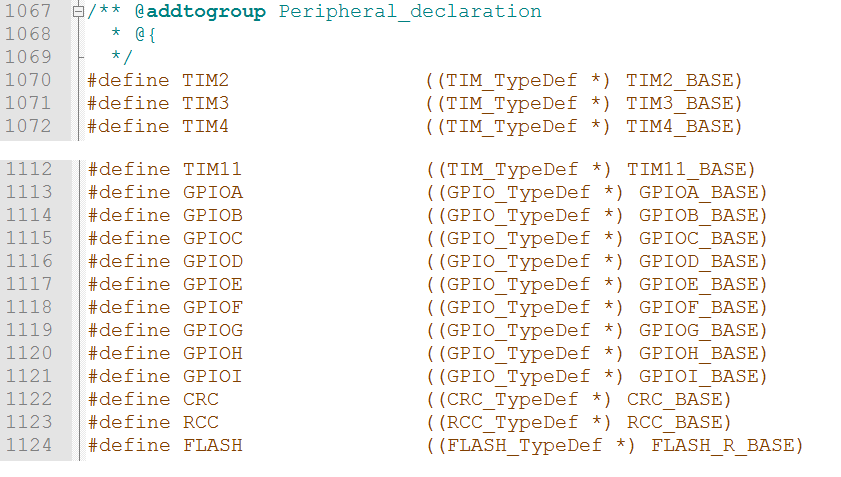


Рис.5

Как видим, тут уже фигурируют имена периферийных блоков с привязкой указателя к базовому адресу блока (а они все, в свою очередь, идут как смещение от базового адреса I/O-пространства).

И что же мы теперь имеем? Рассмотрим на примере порта GPIO.

Есть общая для всех портов структура типа (рис.2, рис.6):

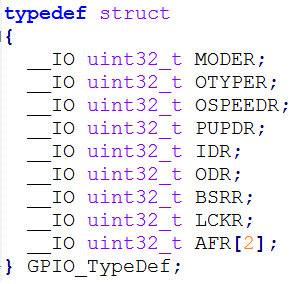


Рис.6

И есть указатели на нее, с привязкой к конкретным GPIO (рис.7).

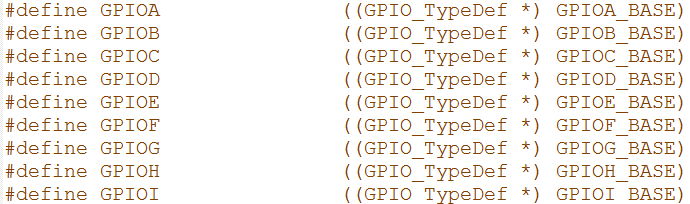


Рис. 7

И если теперь нам надо записать «1» в 5-й бит регистра BSRR порта A, то это обращение к нему будет выглядеть так:

|  |  |
| --- | --- |
| 1 | GPIOA->BSSR = 0x0020; |

Как видите, не надо думать о том, где какой адрес и по какому смещению лежит. Все сгруппировано и собрано в удобные связки. Конечно, у разных процессоров все имена зовутся по-разному.   
Например, у LPC13хх в CMSIS прописано чуток иначе:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | typedef struct  {  union {  \_\_IO uint32\_t MASKED\_ACCESS[4096];  struct {  uint32\_t RESERVED0[4095];  \_\_IO uint32\_t DATA;  };  };  uint32\_t RESERVED1[4096];  \_\_IO uint32\_t DIR;  \_\_IO uint32\_t IS;  \_\_IO uint32\_t IBE;  \_\_IO uint32\_t IEV;  \_\_IO uint32\_t IE;  \_\_IO uint32\_t RIS;  \_\_IO uint32\_t MIS;  \_\_IO uint32\_t IC;  } LPC\_GPIO\_TypeDef; |
| 1  2  3  4 | #define LPC\_GPIO0 ((LPC\_GPIO\_TypeDef \*) LPC\_GPIO0\_BASE )  #define LPC\_GPIO1 ((LPC\_GPIO\_TypeDef \*) LPC\_GPIO1\_BASE )  #define LPC\_GPIO2 ((LPC\_GPIO\_TypeDef \*) LPC\_GPIO2\_BASE )  #define LPC\_GPIO3 ((LPC\_GPIO\_TypeDef \*) LPC\_GPIO3\_BASE ) |

Но система та же самая. И доступ будет такой же. Например, к регистру DIR порта 1:

|  |  |
| --- | --- |
| 1 | LPC\_GPIO1->DIR = 0x0002; |

Названия иные конечно, но суть не меняется. Это существенно упрощает написание слоя HAL[[3]](#endnote-1)

**Попробуйте** определить константу для сброса бита 5 порта А STM32F40x с помощью BSRR.

|  |  |
| --- | --- |
| 1 | GPIOA->BSSR = 0x????; |

В разных CMSIS от разных производителей есть свои фишки. Т.е. в базовой части они совпадают друг с другом, но вот дальше: - кто во что горазд. Впрочем, это уже не имеет значения и делается для нашего удобства.

Например, в CMSIS от STM32 после обязательной части идет произвольная программа вида (рис.8):

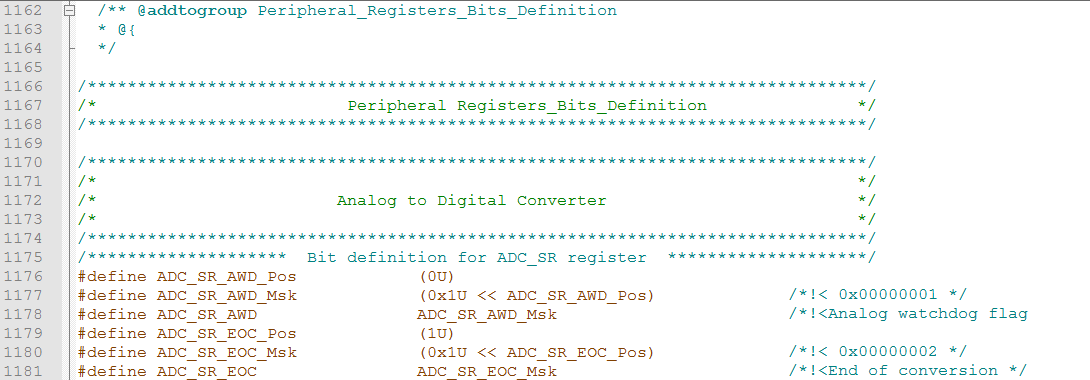


Рис.8

Начинается этот раздел, как и прежде с ADC и далее в алфавитном порядке для каждого бита каждого регистра каждого ресурса. И таких записей почти на пол мегабайта. Крутим файл до GPIO (рис.9).

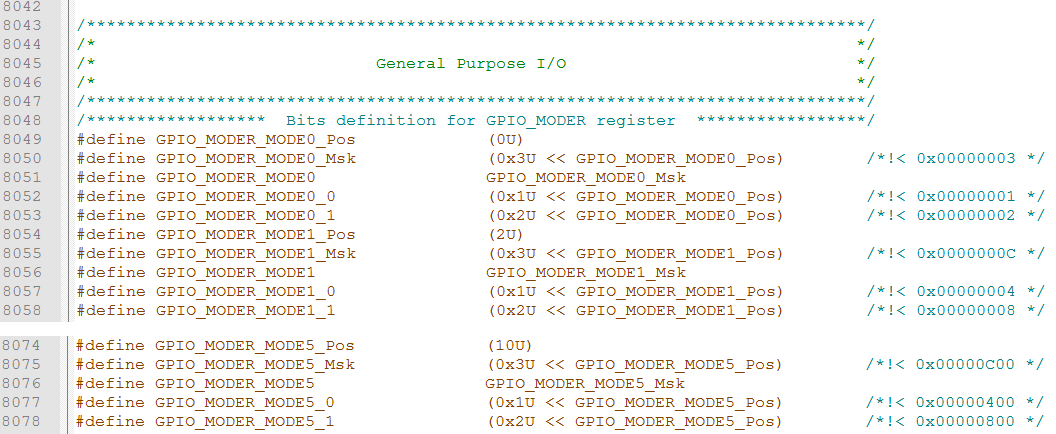


Рис.9

Что это? А это расшифровка отдельных битов каждого из регистров по функциям! Готовые бит-маски. Т.е. производитель уже заранее позаботился о том, чтобы нам не пришлось ломать голову над режимами и помнить, где какие биты стоят. Например, мы хотим настроить pin5 порта А (GPIOA.5) на вывод, чтобы, к примеру, управлять состоянием светодиода (LED 2) на плате NUCLEO. Для этого, как мы уже знаем из Лаб2, нужно выставить для порта GPIOA.5 биты режима MODE[1:0] = 01. Согласно RM0368 [1] или Лаб 2 [2] это биты 11 и 10 регистра GPIOA\_MODER. Так нам даже биты этого регистра не придется считать. И никаких магических чисел в коде!

Делаем так:

|  |  |
| --- | --- |
| 1 | GPIOA->MODER |= GPIO\_MODER\_MODE5\_0; |
| 2 | GPIOA->MODER &= ~ GPIO\_MODER\_MODE5\_1; *//Битте!* |

Причем имя дефайна уже содержит полный путь куда его надо вписать, так что не ошибешься.

Если у нас плата STM32F4 Discovery, то на ней, согласно [2], диодами управляют выводы 12 ÷ 15 порта D. Настроим их на вывод. Причем попробуем это сделать сразу для четырех выводов порта D.

|  |  |
| --- | --- |
| 1 | GPIOD->MODER |= GPIO\_MODER\_MODE12\_0 | GPIO\_MODER\_MODE13\_0 |   GPIO\_MODER\_MODE14\_0 | GPIO\_MODER\_MODE15\_0; |
| 2 | GPIOD->MODER &= ~ GPIO\_MODER\_MODE15\_1 | GPIO\_MODER\_MODE14\_1 |  GPIO\_MODER\_MODE13\_1 | GPIO\_MODER\_MODE12\_1; *//Битте Зер!* |

A заглянув в STM32F401xx.h несколькими страницами ниже увидим нужную бит-маску для записи «1» в разряд 5 регистра вывода GPIOA\_ODR (рис.10):

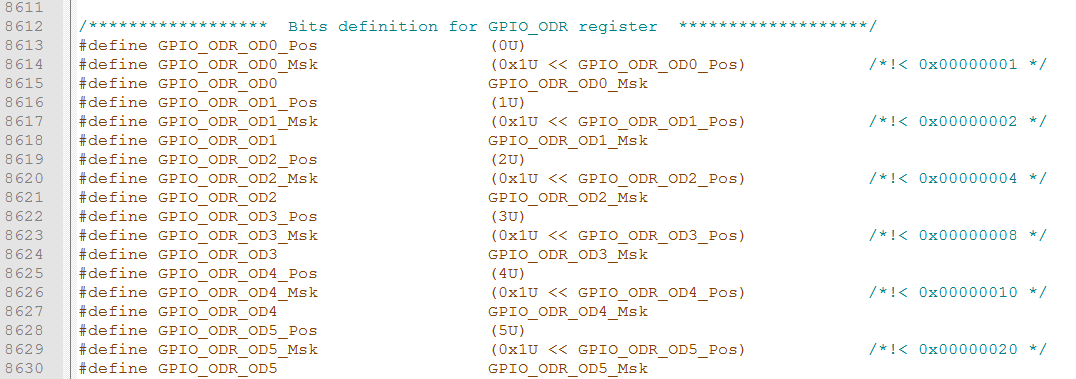
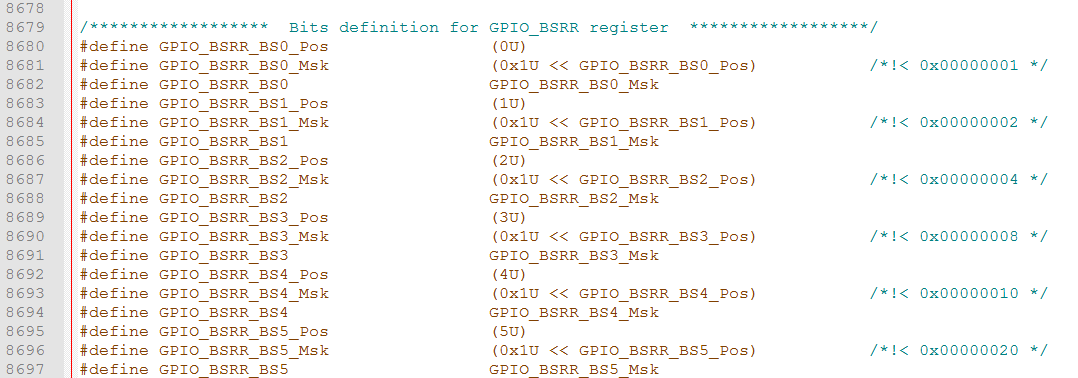


Рис.10

Теперь засветить светодиод на плате NUCLEO c помощью ODR сможем так:

**GPIOA->ODR |= GPIO\_ODR\_OD5**;

Не сложно сделать это же и через регистр BSRR, если подсмотреть, как строятся маски на биты этого регистра (рис.11):  
Рис. 11  
Але-Оп!  
**GPIOA->BSRR |=GPIO\_BSRR\_BS5;**

и светодиод на плате NUCLEO засветился.

Напомним, что для сброса в «0» вывода **y** порта нужно установить в «1» соответствующий бит старшей половины регистра BSRR. Теперь «умственные страдания» типа **у+16** не понадобятся:

**GPIOA->BSRR |=GPIO\_BSRR\_BR5**;

Попробуйте самостоятельно сконструировать команды для «засветки» и «тушения» всех четырех светодиодов платы Discovery посредством регистров ODR и BSRR

Не трудно догадаться, что на плате Discovery светодиод RED, подключенный выходу PD14, засветится после команды:

**GPIOD->ODR |= GPIO\_ODR\_OD14**;

Если нужно поменять состояние светодиода на противоположное, то тут поможет операция XOR “^” - «исключаюшее или», которая инвертирует значение бита, указанного в маске GPIO\_ODR\_OD13

**GPIOD->ODR ^= GPIO\_ODR\_OD14**;

Наконец, вспомним, что для работы с портами GPIO (как, впрочем, и с большинством ресурсов STM32), их необходимо предварительно «включить», а точнее разрешить их тактирование. Это делается через соответствующий регистр группы RCC, а именно **RCC\_AHB1ENR** (см. [1] или [2][[4]](#footnote-3)). Напомним, что тут **AHB1** – название шины по которой ядро МСU обращается к портам GPIO, а **EN** – сокращение от Enable (разрешить). Безусловно эту операцию следовало выполнить в первую очередь. Мы же подошли к ее программированию только сейчас, так как осуществляли «навигацию» по файлу STM32F40xx.h в порядке алфавита ресурсов микроконтроллера.

Разрешить тактирование порта А можно операцией прямого обращения к регистру RCC\_AHB1ENR, как мы это делали в лабораторной работе №2. Либо, используя описание структуры RCC в CMSIS.

«Открутим» файл в начало и найдем описание соответствующего типа (рис.12):

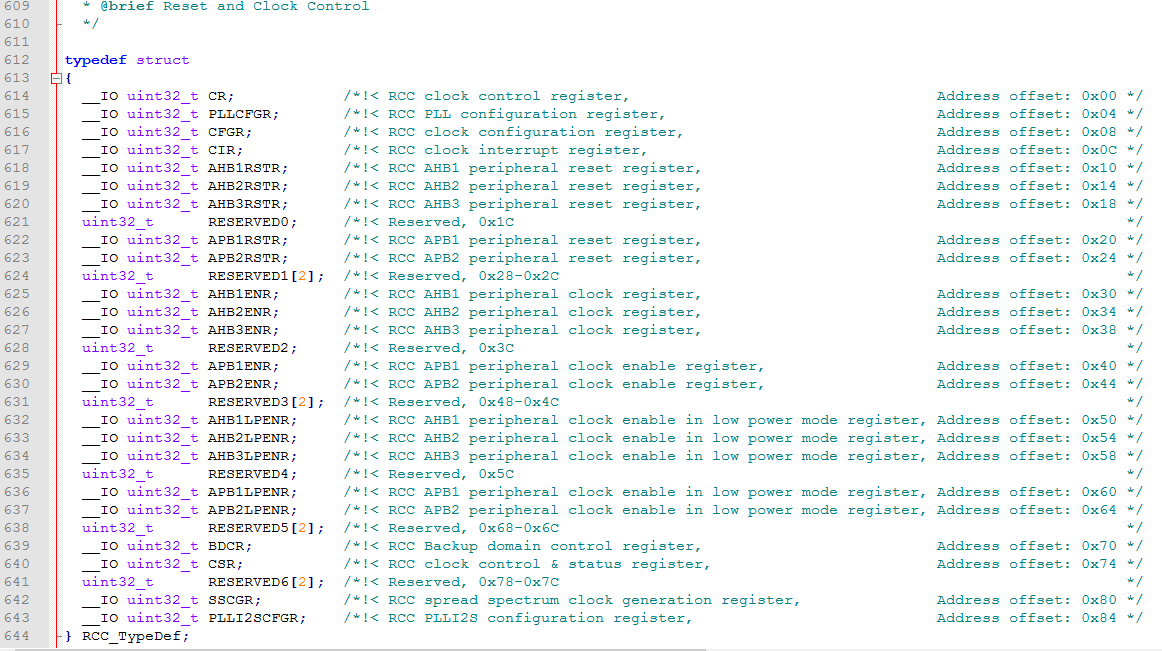


Рис. 12

Теперь «прокрутим» файл вперед на несколько тісяч строк (!) и найдем битовые маски разрядов регистра RCC\_AHB1ENR (рис.13):

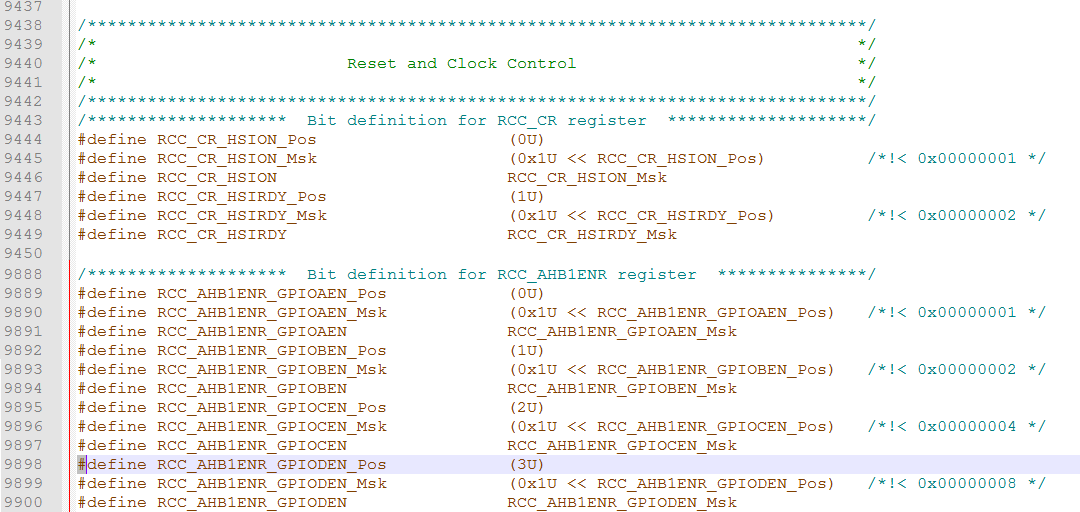


Рис.13

Теперь, учитывая то, что необходимые определения адресов (см. рис 3 и рис.4), указателей структур (рис. 5) и битових полей RCC (рис.13) имеются в файле CMSIS Stm32f40xx.h, разрешим тактирование портов A и D :

**RCC->AHB1ENR|=RCC\_AHB1ENR\_GPIOAEN**;

**RCC->AHB1ENR|=RCC\_AHB1ENR\_GPIODEN**;

Ниже приведен текст программы, выполняющей поочередное включение светодиодов на плате Discovery.

Напомним схему включения светодиодов на плате Discovery (рис.14).

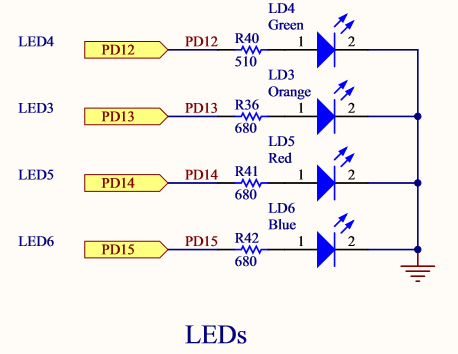
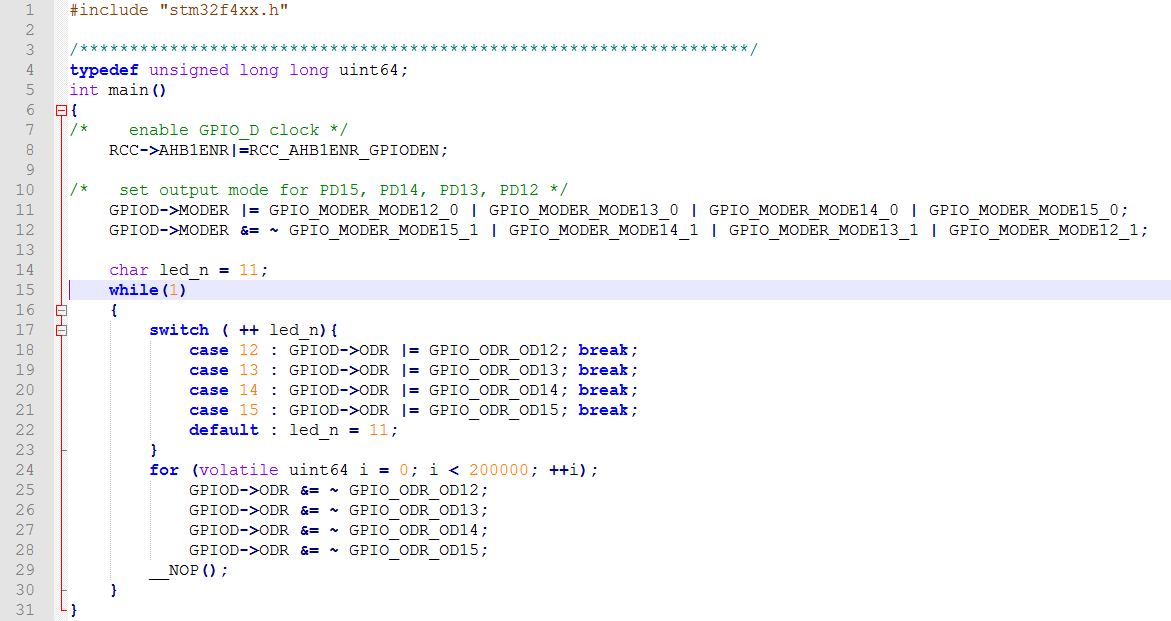


Рис. 14 – Схема подключения светодиодов на плате Discovery

**Подключаем CMSIS к проекту.**  
Технология создания проекта в Keil µVision и написания программы c использованием біблиотеки CMSIS рассмотрена файле-презентации Less 8 CMSIS .  
Практическому освоению программирования c использованием біблиотеки CMSIS посвящена лабораторная работа №3.

**Пример.** Поочередное включение светодиодов на плате **Discovery**

Текст программы **main4LED.c**

****

1. Проект никуда не убежит, потому сделаю небольшое отступление и опишу, что есть CMSIS. [↑](#footnote-ref-1)
2. Номера строк в разных версиях файла могут отличаться. Тут даны примеры фрагментов файла stm32f407хx.h v2.6.0   
   из CMSIS Pack v2.11.0  
   C:\Keil\_v5\ARM\PACK\Keil\STM32F4xx\_DFP\2.11.0\Drivers\CMSIS\Device\ST\STM32F4xx\Include [↑](#footnote-ref-2)
3. **Hardware Abstraction Layer** (**HAL**, [укр.](https://uk.wikipedia.org/wiki/%D0%A3%D0%BA%D1%80%D0%B0%D1%97%D0%BD%D1%81%D1%8C%D0%BA%D0%B0_%D0%BC%D0%BE%D0%B2%D0%B0) *Шари апаратних абстракцій*) — шар абстрагування, реалізований у [програмному забезпеченні](https://uk.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BD%D0%B5_%D0%B7%D0%B0%D0%B1%D0%B5%D0%B7%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%BD%D1%8F), розташованому між фізичним рівнем [апаратного забезпечення](https://uk.wikipedia.org/wiki/%D0%90%D0%BF%D0%B0%D1%80%D0%B0%D1%82%D0%BD%D0%B5_%D0%B7%D0%B0%D0%B1%D0%B5%D0%B7%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%BD%D1%8F) і програмним забезпеченням, що запускаються на цьому [комп'ютері](https://uk.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%27%D1%8E%D1%82%D0%B5%D1%80). HAL призначений для приховування відмінностей в апаратному забезпеченні від основної частини [ядра операційної системи](https://uk.wikipedia.org/wiki/%D0%AF%D0%B4%D1%80%D0%BE_%D0%BE%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D1%96%D0%B9%D0%BD%D0%BE%D1%97_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B8), таким чином, щоб більша частина коду, що працює в режимі ядра, не мала потреби в зміні при її запуску на системах з різним апаратним забезпеченням. На персональних комп'ютерах HAL, по суті, може розглядатися як [драйвер](https://uk.wikipedia.org/wiki/%D0%94%D1%80%D0%B0%D0%B9%D0%B2%D0%B5%D1%80) [материнської плати](https://uk.wikipedia.org/wiki/%D0%9C%D0%B0%D1%82%D0%B5%D1%80%D0%B8%D0%BD%D1%81%D1%8C%D0%BA%D0%B0_%D0%BF%D0%BB%D0%B0%D1%82%D0%B0" \o "Материнська плата), що дозволяє взаємодіяти інструкціям високорівневих [мов програмування](https://uk.wikipedia.org/wiki/%D0%9C%D0%BE%D0%B2%D0%B0_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F" \o "Мова програмування) з низькорівневими компонентами, такими як апаратне забезпечення.

   HAL є навіть нижчим рівнем в комп'ютерних мовах програмування, ніж [API](https://uk.wikipedia.org/wiki/API), тому що він взаємодіє безпосередньо з апаратним забезпеченням замість ядра системи, отже HAL вимагає менше процесорного часу ніж API. Мови високого рівня часто використовують HAL і API для взаємодії з компонентами нижчого рівня.

   Операційні системи, що мають HAL, легше портуються на різне обладнання. Це особливо важливо для [вбудованих систем](https://uk.wikipedia.org/wiki/%D0%92%D0%B1%D1%83%D0%B4%D0%BE%D0%B2%D0%B0%D0%BD%D0%B0_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0), які повинні працювати на великій кількості різних платформ [↑](#endnote-ref-1)
4. Лабораторная работа №2 [↑](#footnote-ref-3)