**Файлът с кода на програмата е main.c , който се намира \embedded\_systems\_exp\bootup\_GPIOD\_leds\Core\Src\main.c**

Цел да направим нещо подобно на библиотеките, която са ни дадени във HAL за да работим с LED 6,3,4 и 5 намиращи се на PD 12,13,14 и 15.

За целта ще поглеждаме и разясняваме кодът даден от HAL

**Стъпка I :**

Трябва да направим функция, която ще:

1. Енейбалне клока на порт D - \_\_HAL\_RCC\_GPIOD\_CLK\_ENABLE()

Картина, която съдържа текст, екранна снимка, Шрифт, линия

Описанието е генерирано автоматично

Тук използваме do{}while(0u), понеже искаме **кодът да бъде изпълнен веднъж** на извикване . Тази структура се използва главно за инициализиране

Регистърът RCC->AHB1ENR или (\*RCC).AHB1 (Advanced High-performance Bus 1 Peripheral Clock Enable Register) e част от struct RCC , който е отговорен за всички системни клокове и интеръпти .

Set\_bit се използва за да напишем 1 на бита от регистъра , който енейбалва клока на d порта (а именно 3тиа бит от регистъра след маска) .

**#define** SET\_BIT(REG, BIT) ((REG) |= (BIT)).

Този бит е дефинирам като GPIO D(port) EN (enable).

**#define** RCC\_AHB1ENR\_GPIODEN\_Pos (3U)

**#define** RCC\_AHB1ENR\_GPIODEN\_Msk (0x1UL << RCC\_AHB1ENR\_GPIODEN\_Pos) /\*!< 0x00000008 \*/

**#define** RCC\_AHB1ENR\_GPIODEN RCC\_AHB1ENR\_GPIODEN\_Msk

READ\_bit – тази команда прави AND само с бита който сме посочили да видим неговото състояние (а именно RCC\_AHB1ENR\_GPIODEN). Необходимо е да се направи за синхронизация и да сме сигурни че дори да е имало забавяне регистърът е променен.

Нека разясним регистъра AHB1ENR и какви са неговите настройки (за какво са отговорни отделните битове): Той е отговорен за енейбала на доста от периферните устройства, като например GPIO A,B,C,D и E , DMA 1 и 2 , CRC и различните видове RAM, които се намират на Advanced High-performance Bus 1.

Картина, която съдържа текст, линия, екранна снимка, диаграма

Описанието е генерирано автоматично

RCC base address = 0x4002 3800

AHB1ENR address = RCC base address + 0x30 = 0x4002 3830

GPIODEN 🡺 RCC->AHB1enr | = (1<<3)

**Стъпка II :**

Да направим в началото на функцията reset или 0 всеки един от пиновете, които използваме , това се прави за да предотврати грешки или нежелано поведение при включване т.е. :

HAL\_GPIO\_WritePin(GPIOD, LD4\_Pin|LD3\_Pin|LD5\_Pin|LD6\_Pin, *GPIO\_PIN\_RESET*);

За целта трябва да разгледаме функцията HAL\_GPIO\_WritePin :

Ще напишем функция която ще трябва да получава (порт , пин, ниво):

**void** **write\_pin**(GPIO\_TypeDef\* GPIOx, uint16\_t GPIO\_Pin, GPIO\_PinState PinState);

в тяловто на функцията трябва да проверим дали получените данни са валидни:

assert\_param(IS\_GPIO\_PIN(GPIO\_Pin));

assert\_param(IS\_GPIO\_PIN\_ACTION(PinState));

assert\_param - работи като връща void за правилни данни и при грешни поинтер към файла с грешни данни и int които показва на кой ред е грешката

**#ifdef** USE\_FULL\_ASSERT

**#define** assert\_param(expr) ((expr) ? (**void**)0U : assert\_failed((uint8\_t \*)\_\_FILE\_\_, \_\_LINE\_\_))

**void** assert\_failed(uint8\_t\* file, uint32\_t line);

**#else**

**#define** assert\_param(expr) ((**void**)0U)

**#endif** /\* USE\_FULL\_ASSERT \*/

Парчето код се изпълнява само ако USE\_FULL\_ASSERT е дефинирано преди това. Тук също е интересен подхода със функцията **void** assert\_failed() тук тя е използвана като функция prototype, казваме на компилатора, че тази функция трябва да се изпълни, но селд като ръннем кода (съответно ако ще я използваме трябва и да я дефинираме #define).

**#define** IS\_GPIO\_PIN\_ACTION(ACTION) (((ACTION) == GPIO\_PIN\_RESET) || ((ACTION) == GPIO\_PIN\_SET))

**#define** IS\_GPIO\_PIN(PIN) (((((uint32\_t)PIN) & GPIO\_PIN\_MASK ) != 0x00U) && ((((uint32\_t)PIN) & ~GPIO\_PIN\_MASK) == 0x00U))

IS\_GPIO\_PIN\_ACTION- проверява дали сме пратили 0 или 1 .

IS\_GPIO\_PIN(PIN) проверява дали не сме дали число по голямо от 16 бита това е важно понеже пиновете са 16 , отбелязани в първите 16 бита също проверява дали не пращаме 0 което ще е отново грешка няма пин 0x0 .

Промяна на физическото нивото на пин, става като се променят стойностите в регистър BSRR (port set/reset reggister).

**if**(PinState != *GPIO\_PIN\_RESET*)

{

GPIOx->BSRR = GPIO\_Pin;

}

**else**

{

GPIOx->BSRR = (uint32\_t)GPIO\_Pin << 16U;

}

Нека обсъдим какво правят отделните регистри и кои са те в една структура GPIOx :

За да съставим структурата трябва да видим, с какви регистри разполага нашият процесор в неговия memory mapped Peripheral Space .

Картина, която съдържа текст, екранна снимка, Шрифт

Описанието е генерирано автоматично

Картина, която съдържа текст, Шрифт, екранна снимка

Описанието е генерирано автоматично

С цел да променим STATE на лед пиновете, трябва да използваме BSRR register .

Как работи той ?

За 16-те пина 1 на битове (от регистъра) между 0-15 отговаря на 1 за дадения пин и

за 16-те пина 1 на битове(от регистъра) между 31-16 отговаря на 0 за дадения пин.

ИЗКЛЮЧЕНИЕ : Ако напишем 1(set) на бит 0 и 1(reset) на бит 16 , съответният пин ще бъде във състояние BS, понеже BS е с приоритет.

Картина, която съдържа текст, екранна снимка, Шрифт, номер

Описанието е генерирано автоматично

Ето я и връзката между BSRR и ODR:

Картина, която съдържа текст, диаграма, План, схематичен

Описанието е генерирано автоматично

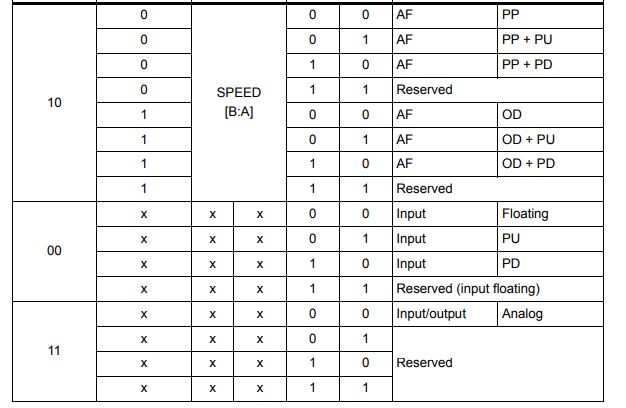
Той е директно свързан с ODR който предава информацията към output control .

Output control e конфигурация която настройваме чрез тези 4 регистъра MODER, OTYPER, OSPEEDR и PUPDR. Като, 2 бита от moder, 1 bit от otyper, 2 бита от Speedr и 2 бита от pupdr отговарят за 1 пин .

Пример: ако искаме да настоим пин 12 трябва да работим с [22; 23] бит от moder, [11] bit от otyper, [22; 23] бит от Speedr и [22; 23] бит от pupdr.

Картина, която съдържа текст, екранна снимка, номер, Шрифт

Описанието е генерирано автоматично



1. GP = general-purpose, PP = push-pull, PU = pull-up, PD = pull-down, OD = open-drain, AF = alternate function

**Знаейки тази информация подходът, който можем да предприемем е да , направим функция, която да приема PIN, MODE, PULL, SPEED и ALTERNATE и след изпълнение да ни настой пина . Също така важно нещо от което се нуждаем е пойнтер към началния адрес на структурата GPIOx. Вече имаме дефиниран такъв :**

**#define** GPIOD\_BASE (AHB1PERIPH\_BASE + 0x0C00UL)

И свършва във :

**#define** GPIOE\_BASE (AHB1PERIPH\_BASE + 0x1000UL)

За да намерим реалната адресна клетка трябва да проследим началото на периферното пространство :

**#define** PERIPH\_BASE 0x40000000UL /\*!< Peripheral base address in the alias region

**#define** AHB1PERIPH\_BASE (PERIPH\_BASE + 0x00020000UL)



Moder otyper …..



0х40020C00 0x40021000

( Тък е важно да се отбележи, че в тази архитектура 1 аддрес сочи към 1 байт, т.е. понеже работим с 32 битови регистри – 1 регистър се разполага на 4 адресни клетки !!!)

Така нека да се върнем на функцията HAL\_GPIO\_Init (), тя трябва да получи ((поинтер към желания порт), (PIN , MODE, PULL, SPEED и ALTERNATE ) ). С тези данни тя трябва да настрой съответните регистри за дадения пин .