1. Лабораторная работа №1.

Изучение основных принципов программирования микроконтроллера STM32F303VCT6. Учебное время 6 часов.

Цель работы : Привитие практических навыков по работе с интегрированной средой Keil µVision V5 и технической документацией.

Ход работы:

1. Изучить и практически выполнить примеры методического руководства.
2. Написать и отладить работу двух программ на ассемблере. В первую программу входит:
   a) объявление стека и области памяти с неупорядоченным хранением данных ('*heap*') размерами согласно варианта задания и заполнением этих областей значениям 0xA и 0xB соответственно;
   b) настройка согласно номера варианта поочередного переключения двух выводов микроконтроллера обычным способом, без использования технологии побитовой адресации и регистров GPIOx_BSRR и GPIOx_BRR.

   Во вторую программу входит:
   a) объявление стека и '*heap*' размерами согласно варианта задания и заполнение этих областей нулями;
   b) настройка согласно номера варианта поочередного переключения двух выводов микроконтроллера используя технологию побитовой адресации и регистры GPIOx_BSRR и GPIOx_BRR;
3. Найти согласно схемы два вывода на плате и измерить эпюры напряжения осциллографом по двум каналам одновременно.
4. Используя второй вариант программы подобрать значение счётчика задержки под заданную вариантом частоту следования переключений выводов.
5. Оформить отчёт.

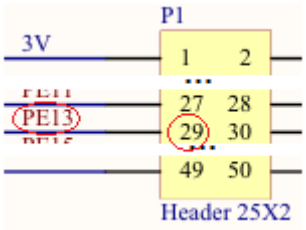
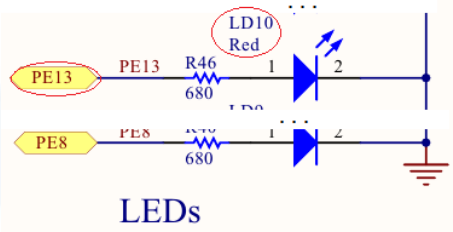
Варианты заданий:

Номер варианта	Размер стека, байт	Размер '*heap*', байт	Номера выводов (ножек-pin)	Частота переключений, Гц
1	16	120	10, 81	5
2	20	116	15, 68	8
3	24	112	33, 95	11
4	28	108	11, 16	14
5	32	104	23, 58	17
6	36	100	35, 86	20
7	40	96	17, 96	23
8	44	92	53, 84	26
9	48	88	62, 63	29
10	52	84	34, 36	32
11	56	80	18, 19	35
12	60	76	80, 82	38
13	64	72	54, 55	41
14	68	68	24, 73	44
15	72	64	60, 79	41
16	76	60	51, 67	38
17	80	56	65, 91	35
18	84	52	25, 88	32
19	88	48	37, 56	29
20	92	44	47, 78	26
21	96	40	26, 87	23
22	100	36	27, 57	20
23	104	32	48, 64	17

24	108	28	29, 59	14
25	112	24	52, 83	11
26	116	20	66, 85	8
27	120	16	61, 69	5

Содержание отчёта:

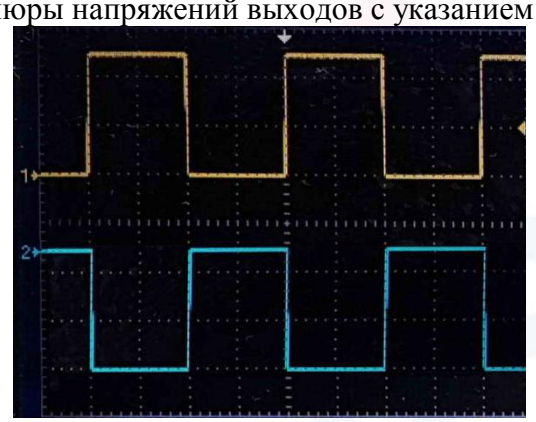
1) Два варианта программы.
2) Таблица трассировки заданных выводов. Пример:

Номер вывода	Обозначение согласно *DS9118* стр. 34	Номера разъёмов и выводов на отладочной плате
44	PE13	 29 вывод на двухрядном штыревом разъёме P1  присоединен к красному светодиоду LD10 через резистор R46

3) Таблица используемых регистров с расчётом адресов (с указанием на документацию) и управляемые биты. Пример:

Регистр	Расчёт адреса и ссылки на документацию	Биты и их назначение согласно документации
RCC_AHBENR	0x4002 1000+0x0000 0014 = 0x4002 1014 *DS9118* стр. 54 0x4002 1000 - 0x4002 13FF | 1 K | RCC *RM0316* стр.148 Address offset: 0x14	Bit 21 IOPEEN - разрешает работу GPIOE
GPIOE_MODER	0x4800 1000+0x0000 0000 = 0x48001000 *DS9118* стр. 54 0x4800 1000 - 0x4800 13FF | 1 K | GPIOE *RM0316* стр.237 Address offset:0x00	Bits 26,27 MODER[1:0] - управляет режимом работы 13 линии ПВВ
GPIOE_BSRR	0x4800 1000+0x0000 0018 = 0x4800 1018 *DS9118* стр. 54 0x4800 1000 - 0x4800 13FF | 1 K | GPIOE *RM0316* стр.240 Address offset: 0x18	Bits 13 BS - устанавливает в единицу 13 бит регистра GPIOE_ODR

4) Принцип расчёта адреса (технология побитовой адресации) для изменения бита в регистре RCC_AHBENR.
5) Эпюры напряжений выходов с указанием амплитуды, частоты, периода каждого сигнала:



Контрольные вопросы:

1. Быть готовым пояснить любую букву и цифру в отчёте и программе.
2. Понятие стека, назначение, принцип работы.
3. Понятие сегмента памяти с неупорядоченным хранением данных ('*heap*'), назначение.
4. Карта памяти, найти в документации и пояснить по основным сегментам.
5. Понятие сигналов логического нуля и единицы, понятие логической линии ввода/вывода.
6. Понятие логического цифрового порта ввода/вывода.
7. Понятие работы с микроконтроллером в режиме отладки, основные инструменты отладки.
8. Понятие регистров управления подсистемами микроконтроллера, назначение, принцип работы.
9. Понятие технологии побитовой адресации. Быть готовым пояснить в режиме отладки в области стека или '*heap*' заполненной нулями.
10. Названия основных регистров управления логическими портами ввода/вывода и их назначение.

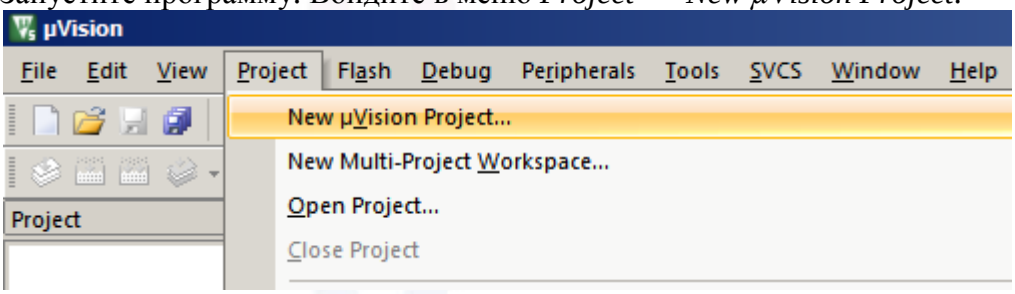
Санкт-Петербургский государственный университет аэрокосмического приборостроения

Методическое руководство по проведению лаб. раб.

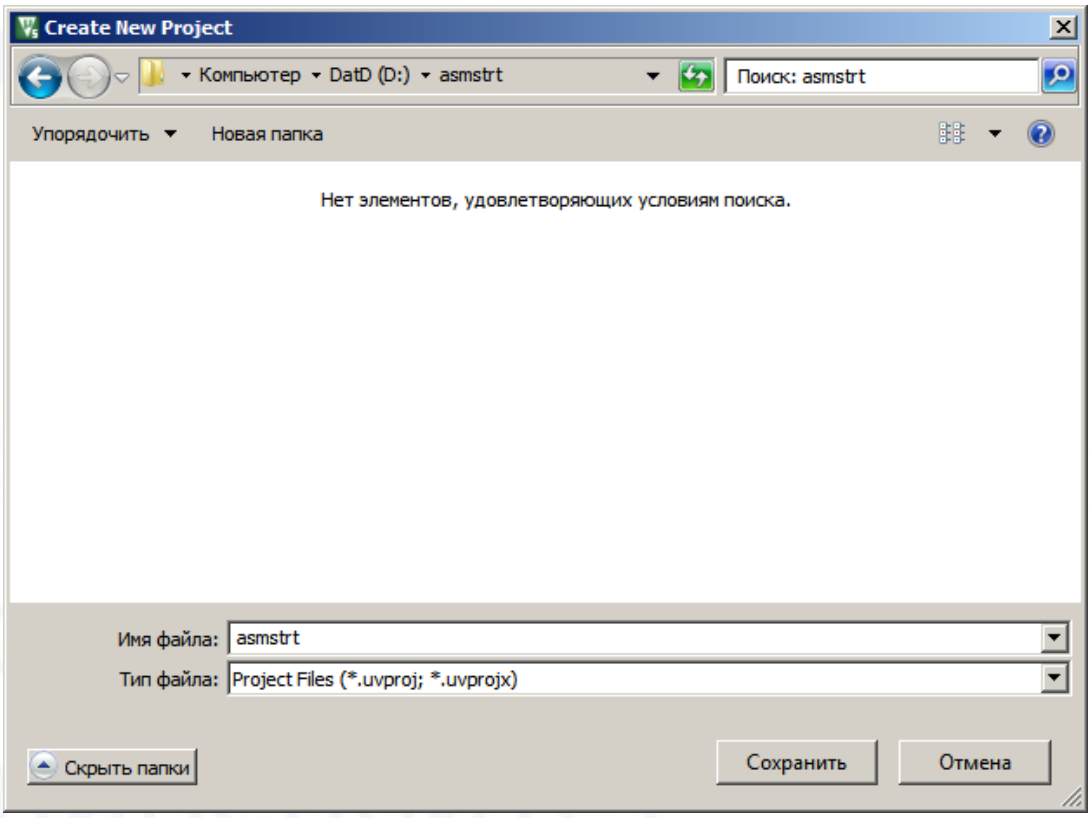
1.1 Порядок создания проекта в Keil µVision V5

(*более детально см. UM1562 User manual Getting started with software and firmware environments for the STM32F3DISCOVERY Kit* https://www.st.com/resource/en/user_manual/dm00062662.pdf)

1.1.1 Запустите программу. Войдите в меню *Project >> New µVision Project*.

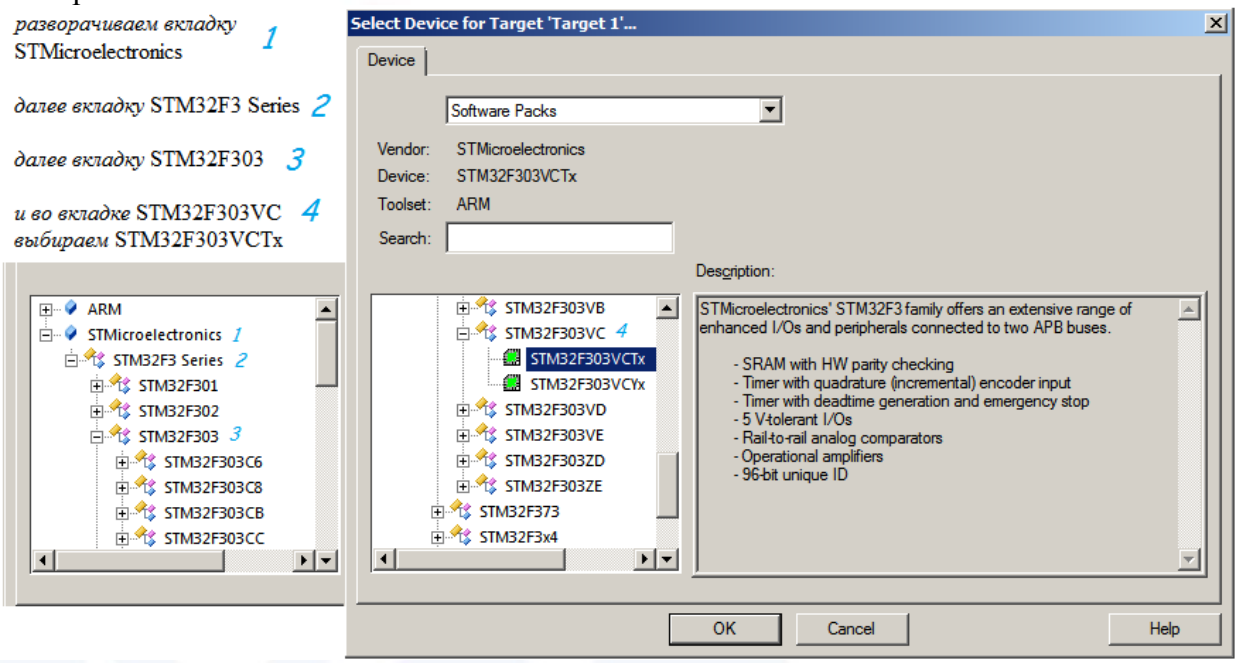


1.1.2 Программа попросит указать папку на диске, где в дальнейшем будут находиться файлы вашего проекта. Выберите (создайте папку) и введите имя проекта. Например, '*asmstrt*'.

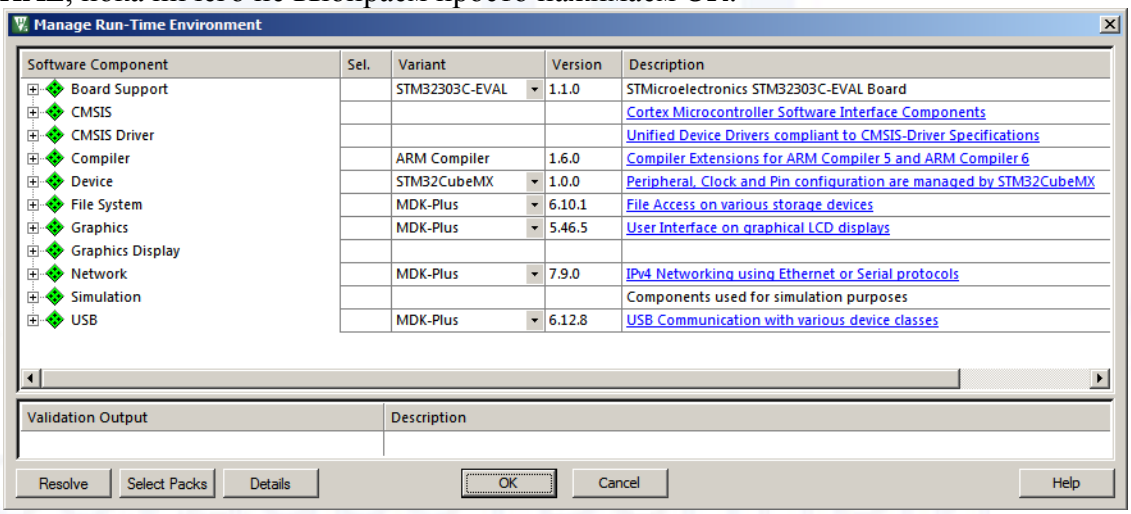


Нажмите кнопку *Сохранить*

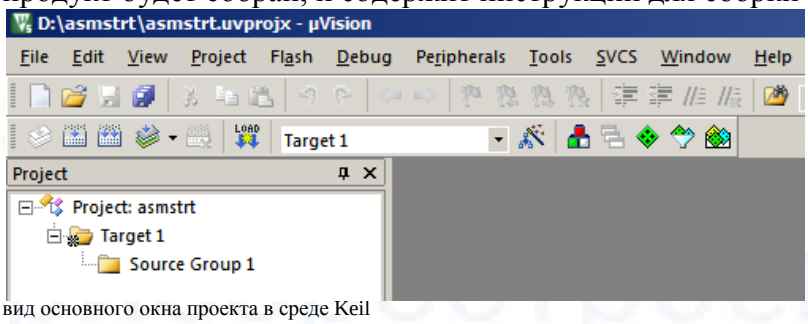
1.1.3 В следующем окне будет предложено выбрать тип СБИС, под которую создаётся проект. Выбираем STM32F303VCTx. Нажимаем *OK*.



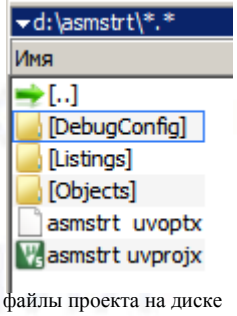
1.1.4 В следующем окне настройка работы с библиотеками ARM MDK-Professional и STM HAL, пока ничего не выбираем просто нажимаем *OK*.



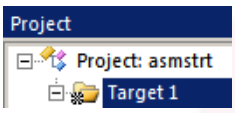
1.1.5 Автоматически создан проект '*asmstrt*' включающий цель проекта '*Target 1*', а в нём группа файлов '*Source Group 1*'. (*Project* — это репозиторий для файлов и ресурсов, необходимых для сборки программного продукта; *Target* — точно определяет, какой продукт будет собран, и содержит инструкции для сборки проекта из набора файлов)



вид основного окна проекта в среде Keil

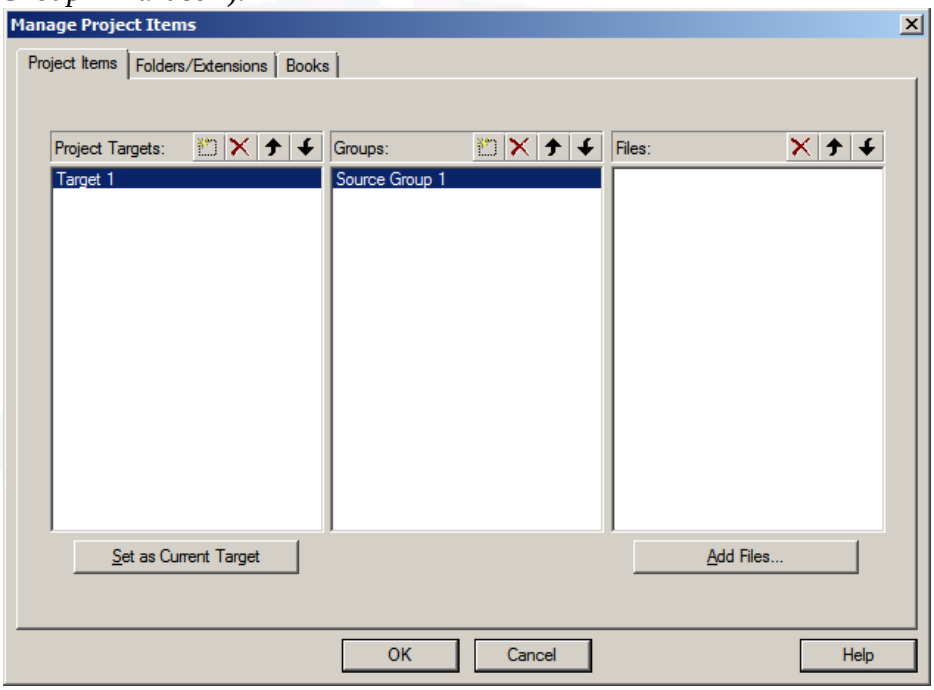


файлы проекта на диске

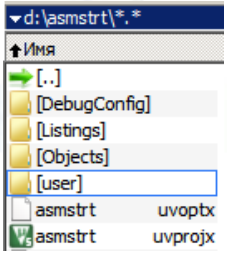
1.1.6 Сделаем свои названия созданных папок проекта. Для этого наводим указатель на папку проекта 

(*'Target 1'*) и нажимаем правую клавишу мышки. Далее в контекстном меню выбираем опцию

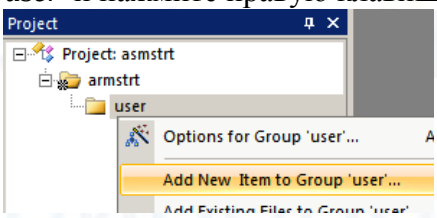
(*Manage Project Items...*) и в появившемся окне меняем названия на свои (например *'Target 1'* на *'asmstrt'*, а *'Source Group 1'* на *'user'*).

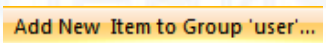


Также в этом окне можно создавать группы, добавлять и удалять файлы из проекта по этим группам. С группами файлов (особенного когда их много) намного удобнее работать в этом окне чем непосредственно через контекстное меню в основном окне (см. п. 1.1.5).

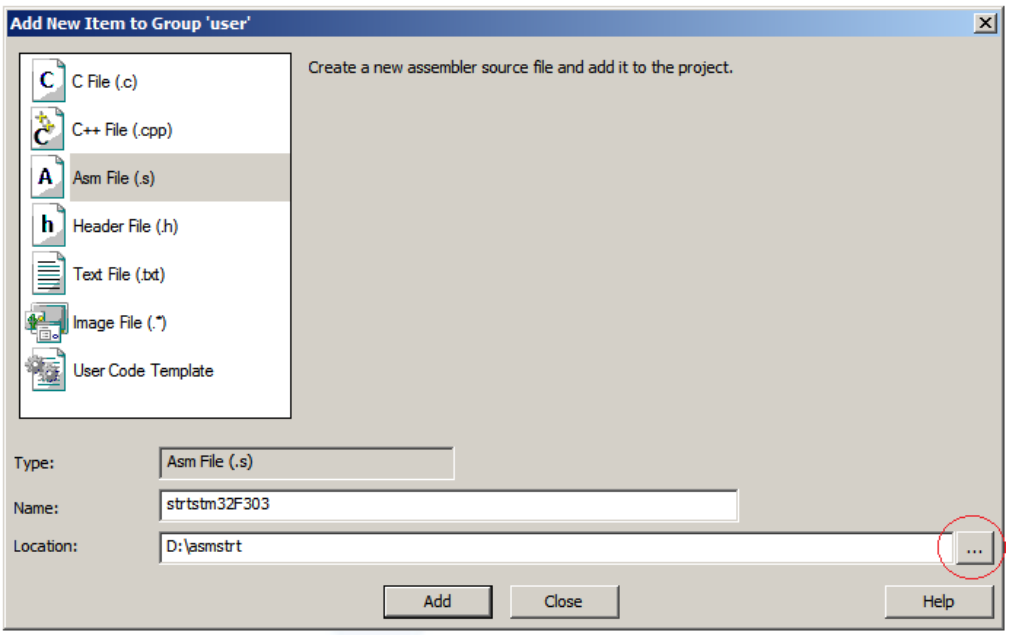
В дальнейшем для исключения путаницы в файлах при работе со средой Keil лучше согласовывать папки проекта с реальным расположением папок на диске. Поэтому на диске в папке проекта создадим директорию *'user'* 

1.1.7 Теперь перейдем к наполнению своего проекта файлами. Для начала, это будет один единственный ассемблерный файл. Для создания файла наведите курсор на имя группы *'user'* и нажмите правую клавишу мышки.

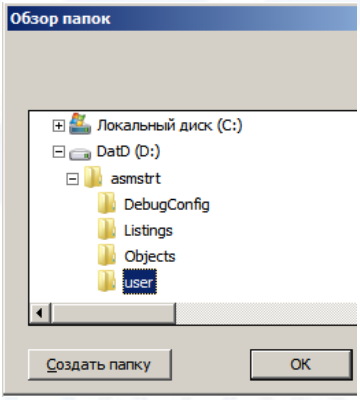


Далее в контекстном меню выбираем опцию 

(*Add New Item to Group 'user' ...*).

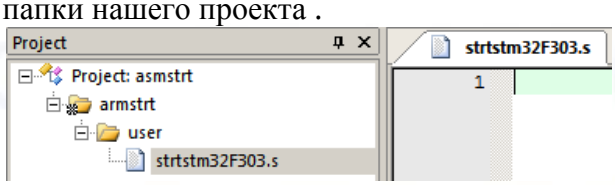


В появившемся окне выбираем опцию *Asm File (.s)* и внизу в строке *Name:* вводим имя создаваемого файла (например '*strtstm32F303*'). Далее ещё ниже в строке *Location:* нажимаем кнопку с тремя точками и в появившемся окне дерева папок находим и выбираем папку '*user*'



, нажимаем *OK* и в строке *Location:* прописался путь

для создаваемого нами файла. И по нажатию кнопки *Add* создаём наш файл который физически будет расположен в директории '*user*' папки нашего проекта .



1.2 Программирование на ассемблере и отладка.

1.2.1 Минимальная программа на ассемблере.

Наполним наш созданный ассемблерный файл следующим кодом:

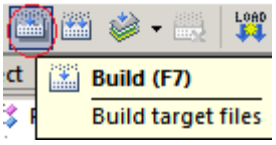
strtstm32F303.s

```
1   Stack_Size      EQU 0x00000030
2               AREA    STACK, NOINIT, READWRITE, ALIGN=3
3   Stack_Mem       SPACE   Stack_Size
4   __initial_sp
5   Heap_Size       EQU 0x00000040
6               AREA    HEAP, NOINIT, READWRITE, ALIGN=3
7   __heap_base
8   Heap_Mem        SPACE   Heap_Size
9   __heap_limit
10              PRESERVE8
11              THUMB
12              AREA    RESET, DATA, READONLY
13              EXPORT  __Vectors
14  __Vectors       DCD __initial_sp
15              DCD Reset_Handler
16  __Vectors_End
17  __Vectors_Size  EQU __Vectors_End - __Vectors
18              AREA    |.text|, CODE, READONLY
19              ENTRY
20              EXPORT  Reset_Handler
21  Reset_Handler   PROC
22              B   .
23              ENDP
24      END
```

Этот же код с возможностью копирования для вставки в проект:

```
Stack_Size          EQU     0x00000030
                    AREA    STACK, NOINIT, READWRITE, ALIGN=3
Stack_Mem           SPACE   Stack_Size
__initial_sp
Heap_Size           EQU     0x00000040
                    AREA    HEAP, NOINIT, READWRITE, ALIGN=3
__heap_base
Heap_Mem            SPACE   Heap_Size
__heap_limit
                    PRESERVE8
                    THUMB
                    AREA    RESET, DATA, READONLY
                    EXPORT          __Vectors
__Vectors           DCD     __initial_sp
                    DCD     Reset_Handler
__Vectors_End
__Vectors_Size  EQU     __Vectors_End - __Vectors
                    AREA    |.text|, CODE, READONLY
                    ENTRY
                    EXPORT          Reset_Handler
Reset_Handler   PROC
                    B       .
                    ENDP
    END
```

При наборе кода следует учитывать что в строках где символы (метки адресов) идут с начала строки, то так и должно быть. Не допускается вставка даже пробела между ними и началом строки. В остальных случаях пробелы в начале строки обязательны, в противном случае компилятор примет команду за метку. (http://www.keil.com/support/man/docs/ARMASM/armasm_dom1359731141352.htm)

После ввода кода нажатием клавиши **F7** скомпилируем программу (также возможно запустить сборку кнопкой '*Build*' 

на главной панели или через меню '*Project >> Build Target*').

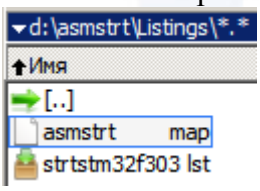
После сборки окне *Build Output* получаем

```
Build Output
*** Using Compiler 'V5.06 update 6 (build 750)', folder:
Rebuild target 'asmstrt'
assembling strtstm32F303.s...
linking...
Program Size: Code=4 RO-data=8 RW-data=0 ZI-data=48
".\Objects\asmstrt.axf" - 0 Error(s), 0 Warning(s).
Build Time Elapsed:  00:00:00
```

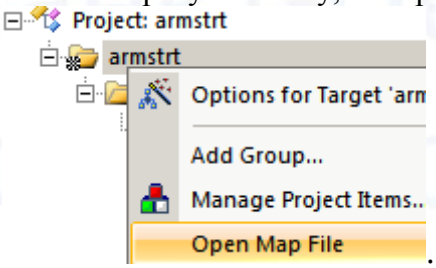
где *Code* - размер кода, *RO-data* - константы в коде (*Read Only Data*), *RW-data* - переменные (*Read Write Data*), *ZI-data* - переменные инициализируемые нулями (*Zero-Initialized Data*). Исходя из этих данных можно посчитать сколько оперативной (RAM) и постоянной (ROM) памяти нужно нашему проекту: RAM = RW+ZI = 0+48 = 48 байт и ROM (FLASH память) = Code+RO+RW = 4+8+0 = 12 байт.

1.2.2 Файл отображения адресов - *map*-файл.

Так же данные по размеру памяти необходимой исполняемому коду можно посмотреть в текстовом файле, в котором находится таблица адресов (файл с расширением *map*). Физически он расположен в папке '*Listings*' проекта:



Для открытия его в среде Keil наведите курсор на вкладку '*armstrt*' дерева проекта и нажмите правую кнопку, выберете вкладку '*Open Map File*':



В открывшемся map-файле, промотав в конец, последние две строки как раз и показывают необходимые RAM и ROM:

```
Total RW  Size (RW Data + ZI Data)              48 (   0.05kB)
Total ROM Size (Code + RO Data + RW Data)       12 (   0.01kB).
```

В этом файле в разделе '*Memory Map of the image*' показаны адреса по которым расположены стек и область памяти с неупорядоченным хранением данных ('*heap*'), а также адреса по которым происходит загрузка исполняемого кода в ROM.

```
0x20000000        -        0x00000030   Zero   RW          1    STACK              armstrt.o
```

адрес расположения стека размер стека

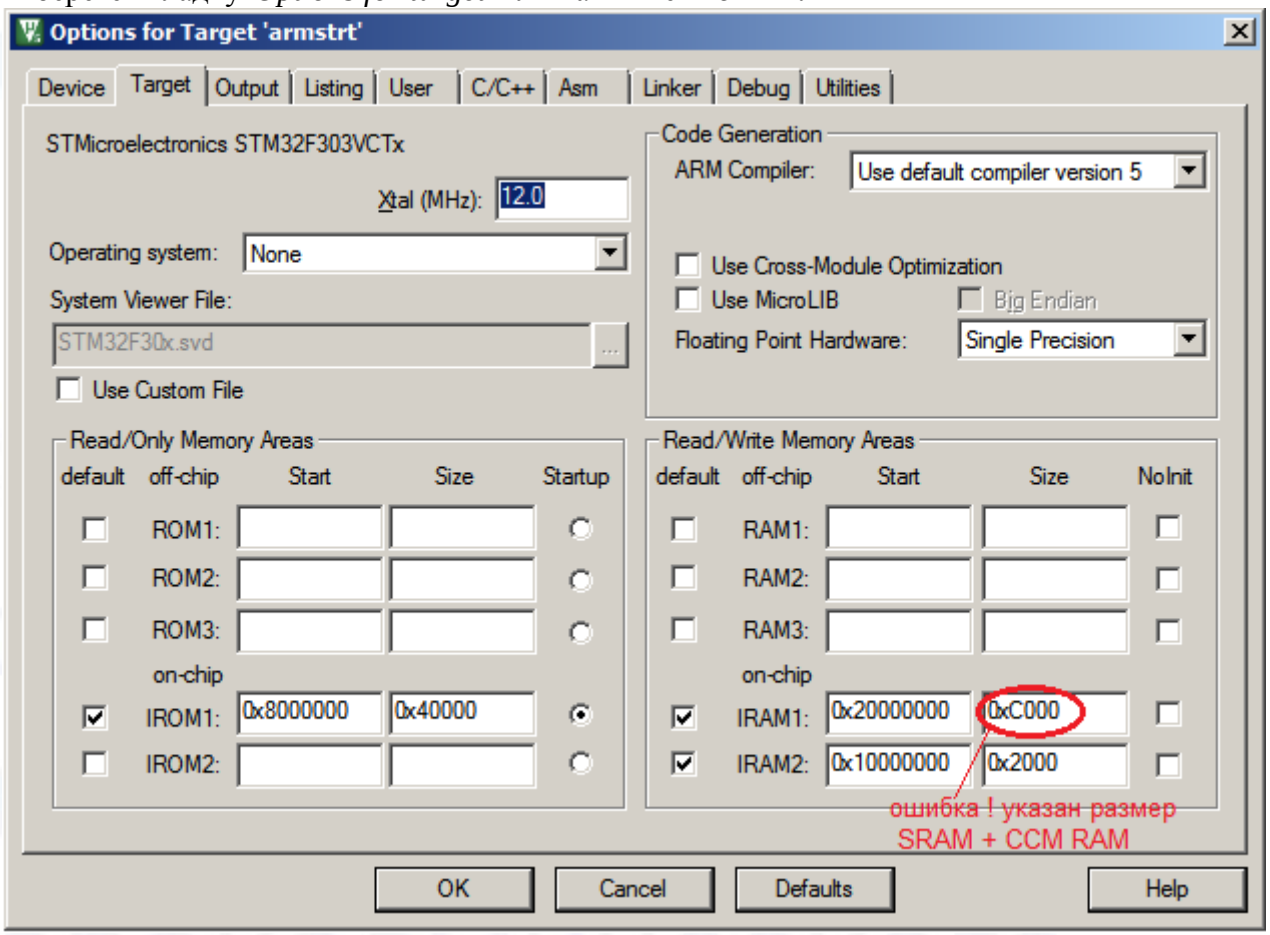
Раздел '*heap*' пока отсутствует, так как компилятор убрал его, посчитав что раз не используется значит не нужен, о чём и оповестил в начале *map*-файла:

```
==============================================================

Removing Unused input sections from the image.

    Removing armstrt.o(HEAP), (64 bytes).

1 unused section(s) (total 64 bytes) removed from the image.

==============================================================
```

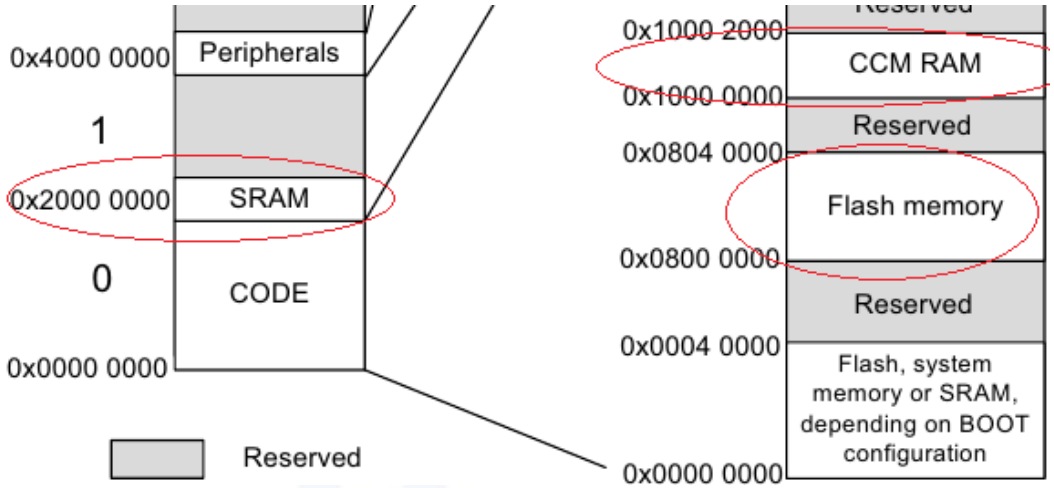
В разделе '*Image Symbol Table*' показаны значения адресов и их мнемонические обозначения которые мы указали в программе.
(*более детально см. DUI0474M - ARM® Compiler Version 5.06 User Guide* http://infocenter.arm.com/help/topic/com.arm.doc.dui0474m/DUI0474M_armlink_user_guide.pdf ; http://www2.keil.com/coresight/#etm)

1.2.3 Настройка среды Keil.

Прежде чем продолжить писать программу на ассемблере, настроим среду (*более детально см.* http://www.keil.com/support/man/docs/uv4/uv4_dg_options.htm).
Для этого наведите курсор на вкладку '*armstrt*' дерева проекта и нажмите правую кнопку, выберете вкладку '*Options for target*' или нажмите **Alt + F7.**



На открывшейся вкладке *Target* указан начальный адрес флеш-памяти микроконтроллера IROM1. Он начинается с 0x80000000 и имеет размер 0x40000 (256 Кб). Адреса оперативной памяти: IRAM1 соответствует SRAM 0x20000000 размер которой 0xA000 (40Кб); IRAM2 соответствует CCM RAM 0x10000000 размером 0x2000 (8Кб). Эти данные выставляются при выборе типа микроконтроллера (см. п. 1.1.3). Также их можно узнать из технического описания на STM32F303VCT6 (*STM32F303xC Datasheet - production data* https://www.st.com/resource/en/datasheet/stm32f303vc.pdf). В котором на странице 53 показано:



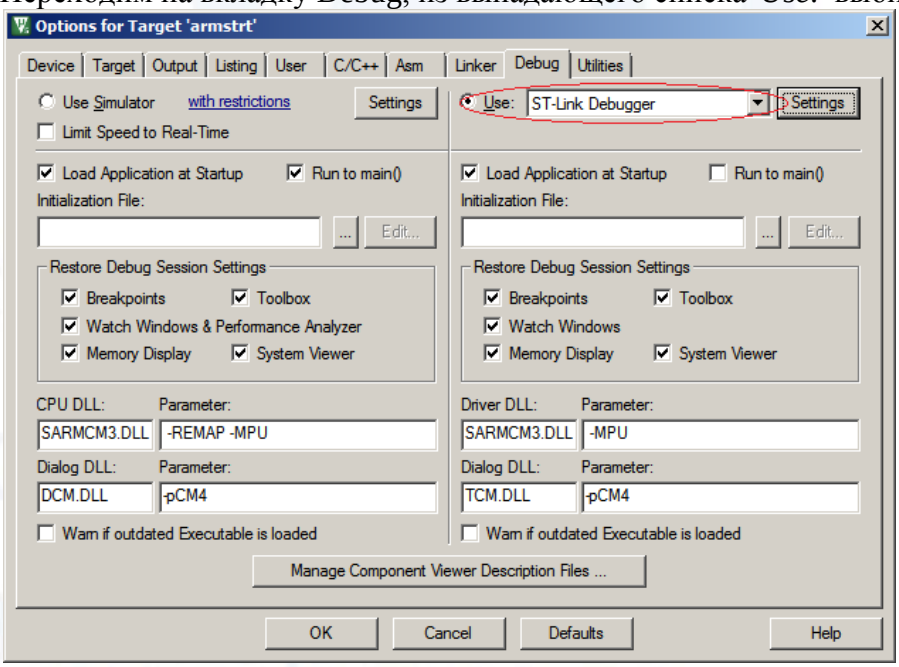
и на странице 12 в таблице указан размер этих областей:

Table 2. STM32F303xB/STM32F303xC family device features and peripheral counts

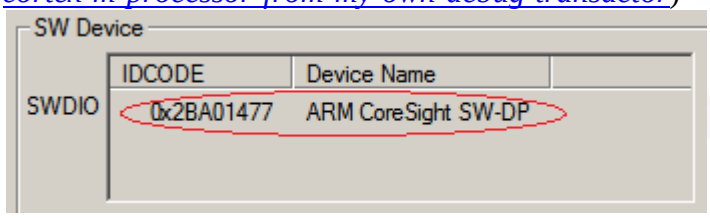
Peripheral	STM32F303Cx		STM32F303Rx		STM32F303Vx	
Flash (Kbytes)	128	256	128	256	128	256
SRAM (Kbytes) on data bus	32	40	32	40	32	40
CCM (Core Coupled Memory) RAM (Kbytes)	8					

Менять в этой вкладке ничего не надо.

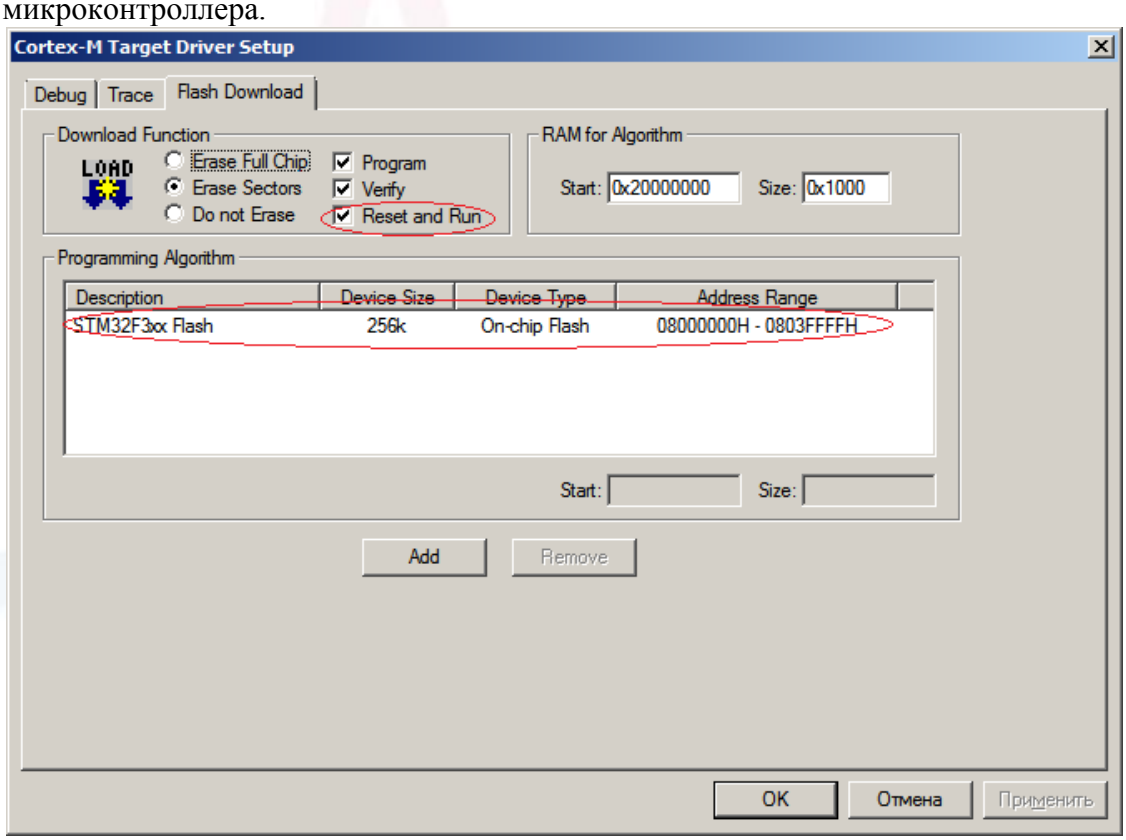
Переходим на вкладку Debug, из выпадающего списка '*Use:*' выбираем *ST-Link Debugger*.



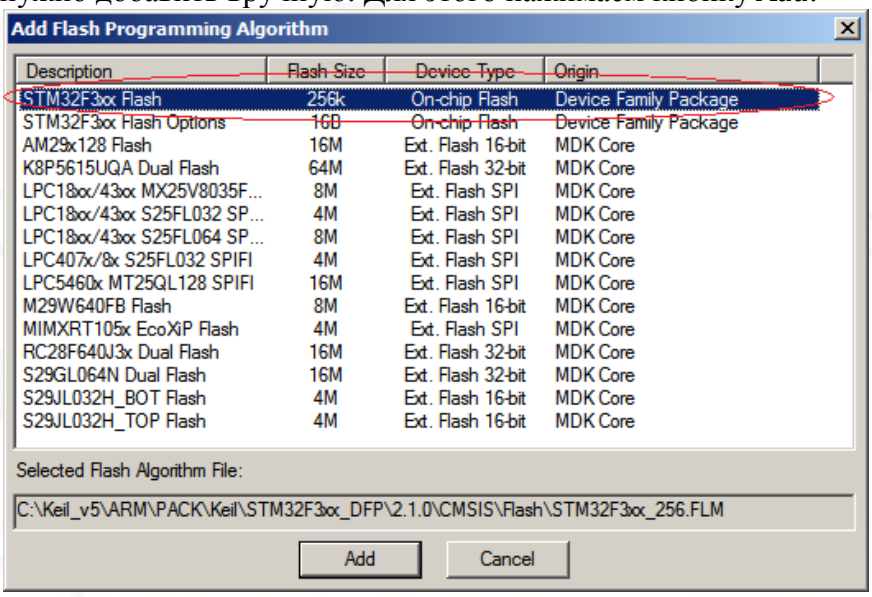
Далее нажимаем рядом со списком кнопку *Settings* и в открывшемся окне на вкладке *Debug* проверяем что есть связь с микроконтроллером. В разделе SW Device будет написан код изготовителя микроконтроллера и тип протокола обмена (*подробнее тут: https://developer.arm.com/docs/103489837/latest/how-do-i-access-the-memory-system-of-a-cortex-m-processor-from-my-own-debug-transactor*)



Если связи нет программатора с микроконтроллером то будет надпись '*ST-LINK connection error*', если связи нет компьютера с программатором '*NO ST-LINK detected*'.
Далее переходим к вкладке *Flash Download*. Отмечаем пункт *Reset and Run*. Это обеспечит автоматический запуск программы после ее прошивки во flash-память микроконтроллера.



Если *KEIL* не смог правильно распознать ваш микроконтроллер, то окно *Programming Algorithm* будет пустым. В этом случае тип микроконтроллера с адресом flash памяти нужно добавить вручную. Для этого нажимаем кнопку *Add*:



и в появившемся окне выбираем тип нашего микроконтроллера STM32F3xx Flash 256k. И нажимаем кнопку *Add*. После чего он отобразиться в окне *Programming Algorithm*.

1.2.4 Разбор основных конструкций нашей программы на ассемблере (см. п. 1.2.1).
Наша программа имеет следующую структуру:
1) Объявление области стека (stack).

2) Объявление области памяти с неупорядоченным хранением данных (heap).
3) Таблица векторов прерываний.
4) Код обработчика сброса (reset handler).

1) Объявление области стека (stack) состоит из следующих директив:

```
1  Stack_Size      EQU 0x00000030
2              AREA    STACK, NOINIT, READWRITE, ALIGN=3
3  Stack_Mem       SPACE   Stack_Size
4  __initial_sp
```

В первой строке объявляем метку Stack_Size со значение 0x30 (48 байт) - размер стека который нам нужен. Здесь директива EQU подобна директиве препроцессора #define в языке си. Во второй строке объявление секции STACK. Директивой AREA создается отдельная секция в памяти с названием STACK. За названием следуют атрибуты: NOINIT — данные в секции либо остаются, как есть, без изменений, либо инициализируются только нулями; READWRITE — секция доступна для чтения и записи; ALIGN=3 — выравнивание секции по границе кратной четырём байт (дополнение до четырёх). Т.е. адрес (вершины стека + 1), размер и адрес дна стека будут кратны четырём. Требования стандарта AAPCS (вызова процедур для архитектуры ARM), что разумно так как размер регистров четыре байта. Более подробно см.: http://www.keil.com/support/man/docs/ARMASM/armasm_dom1361290000455.htm
В третьей строке выделяется пространство в памяти заданного размера (Stack_Size) для области стека. Директива SPACE просто резервирует место в памяти. В метке Stack_Mem прописывается начальный адрес резервируемой области памяти (дно стека), а в метке __initial_sp следующий адрес после этой области (вершина стека). В дальнейшем используется в таблице векторов. Так как стек растет вниз, она будет служить начальным значением указателя стека.
2) Объявление области памяти с неупорядоченным хранением данных (heap) начинается с 5ой строки и по структуре идентична объявлению стека:

```
5  Heap_Size       EQU 0x00000040
6              AREA    HEAP, NOINIT, READWRITE, ALIGN=3
7  __heap_base
8  Heap_Mem        SPACE   Heap_Size
9  __heap_limit
```

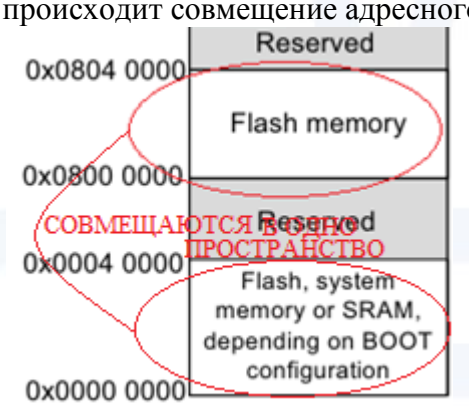
Объявляем метку Heap_Size равной 0x40 (64 байта) - размер '*heap*' которая нам нужна. Объявление секции HEAP атрибуты имеют те же значения что и при объявлении стека. __heap_base — начальный адрес '*heap*', __heap_limit — адрес следующий после области '*heap*'.
Названия меток для стека и '*heap*' такие как: Stack_Size; Stack_Mem; __initial_sp; Heap_Size; __heap_base; __heap_limit; Heap_Mem, можно заменить на свои, но если в дальнейшем вы планируете работать с операторами языка си, то возникнут ошибки, поскольку библиотеки компилятора настроены на восприятие именно таких названий меток этих областей.
Далее в 10ой строке PRESERVE8 указывает компоновщику сохранять 8-байтное выравнивание стека требование стандарта AAPCS, а THUMB указывает ассемблеру интерпретировать последующие инструкции как THUMB-инструкции.
3) Таблица векторов прерываний является неотъемлемой составляющей для запуска процессора, а также работы стека. Это связано с тем, что процессор при запуске начинает выполнение кода с адреса 0x0000 0000, а именно читает первые четыре байта (значение вершины стека) и записывает их в регистр R13(SP) (Stack Pointer - указатель стека). Затем считывает следующие четыре байта начиная с адреса 0x0000 0004. Считанные четыре байта являются адресом по которому процессор далее выполняет программный код и соответствующим образом меняет регистр R15(PC) (Program Counter - счётчик команд, указывает, какую команду нужно выполнять следующей). Именно

такой порядок работы микроконтроллера указан на странице 62 справочника RM0316 (*Reference manual STM32F303xB/C/D/E* доступного по ссылке: https://www.st.com/resource/en/reference_manual/dm00043574.pdf).

After this startup delay has elapsed, the CPU fetches the top-of-stack value from address 0x0000 0000, then starts code execution from the boot memory at 0x0000 0004. Depending on the selected boot mode, main Flash memory, system memory or SRAM is accessible as follows:

- Boot from main Flash memory: the main Flash memory is aliased in the boot memory space (0x0000 0000), but still accessible from its original memory space (0x0800 0000). In other words, the Flash memory contents can be accessed starting from address 0x0000 0000 or 0x0800 0000.

А так как загрузка исполняемого кода происходит из памяти ПЗУ (Flash) располагающейся по адресу 0x0800 0000 (см. п. 1.2.3), то микроконтроллер отражает весь исполняемый код на пространство памяти начальной загрузки (0x0000 0000), т.е. происходит совмещение адресного пространства двух регионов.



Сама же таблица векторов изложена на странице 285 этого руководства. В нашей программе мы объявили использование только первых двух векторов из этой таблицы:

Table 82. STM32F303xB/C/D/E, STM32F358xC and STM32F398xE vector table

Position	Priority	Type of priority	Acronym	Description	Address
-	-	-	-	Reserved	0x0000 0000
-	-3	fixed	Reset	Reset	0x0000 0004
-	-2	fixed	NMI	Non maskable interrupt. The RCC Clock Security System (CSS) is linked to the NMI vector.	0x0000 0008
-	-1	fixed	HardFault	All class of fault	0x0000 000C

```
12                AREA    RESET, DATA, READONLY
13                EXPORT   __Vectors
14  __Vectors         DCD  __initial_sp
15                    DCD  Reset_Handler
16  __Vectors_End
17  __Vectors_Size  EQU  __Vectors_End - __Vectors
```

Поясним порядок объявления таблицы векторов:

Сначала создаём секцию RESET с атрибутами: DATA — секция содержит данные, а не инструкции; READONLY — секция только для чтения. Далее в строке 13 директивой EXPORT указываем компилятору что метка __Vectors является глобальной и видна за пределами нашего файла. В строке 14 директивой DCD распределяем в ПЗУ четыре байта памяти (DCW - 2 байта, DCB - 1 байт) под метку __initial_sp (адрес вершины стека), а метке __Vectors сопоставляется адрес начала таблицы векторов прерываний 0x0800 0000 который указан в *map*-файле в разделе *Image Symbol Table* (см. п. 1.2.2):

```
__Vectors                          0x08000000   Data          4   strtstm32f303.o(RESET)
```
имя метки | адрес | тип секции | размер байт | название секции

так же в этом разделе видно что секция RESET расположена по адресу 0x0800 0000 и имеет размер 8 байт:

```
Symbol Name                          Value      Ov Type        Size  Object(Section)

user\strtstm32F303.s                 0x00000000   Number         0  strtstm32f303.o ABSOLUTE
RESET                                0x08000000   Section        8  strtstm32f303.o(RESET)
```

Далее в строке 15 распределяем следующие четыре байта под обработчик прерывания сброса (Reset) - указатель на процедуру Reset_Handler. Код этой процедуры будет выполняться всегда после сброса процессора.

Таблица векторов содержит только адреса обработчиков и указатель вершины стека, причём обработчики прерываний располагаются строго в порядке указанным в таблице *Table 82. STM32F303xB/C/D/E, STM32F358xC and STM32F398xE vector table* руководства *RM0316*. Если прерывания не используются, таблицу можно сократить до двух строк. Поэтому на этом наша таблица заканчивается и далее идёт метка __Vectors_End - адрес конца таблицы векторов прерываний. Далее вычисляем размер таблицы векторов прерываний __Vectors_Size просто вычитая из конечного адреса начальный.

4) Код обработчика сброса (reset handler) по сути то место с которого начинается выполнение программы пользователя. Сначала в строке 18 объявляем секцию |.text| с атрибутами: CODE — секция содержит инструкции (исполняемый код); READONLY — секция только для чтения. Название секции |.text| используется общепринятое, но может быть любым другим. Так как мы не используем общепринятую в языке си функцию main, то компоновщику необходимо указать точку входа (адрес в программе, по которому находится команда, которая будет выполнена первой при старте программы), для этого мы используем директиву ENTRY непосредственно перед объявлением нашей процедуры как глобальной с помощью директивы EXPORT:

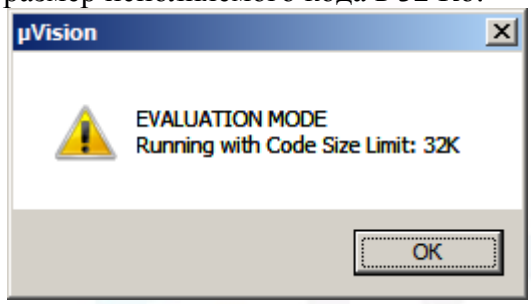
```
18              AREA    |.text|, CODE, READONLY
19              ENTRY
20              EXPORT  Reset_Handler
21  Reset_Handler   PROC
22              B    .
23              ENDP
24      END
```

Строка 21 начинается с объявления метки Reset_Handler функцией с помощью директивы PROC и далее следует исполняемый этой функцией код до директивы ENDP. Весь наш код состоит из одной команды безусловного перехода к самой себе (B .), т.е. бесконечный цикл. И заканчивается наш файл директивой END которая информирует компилятор об окончании файла.

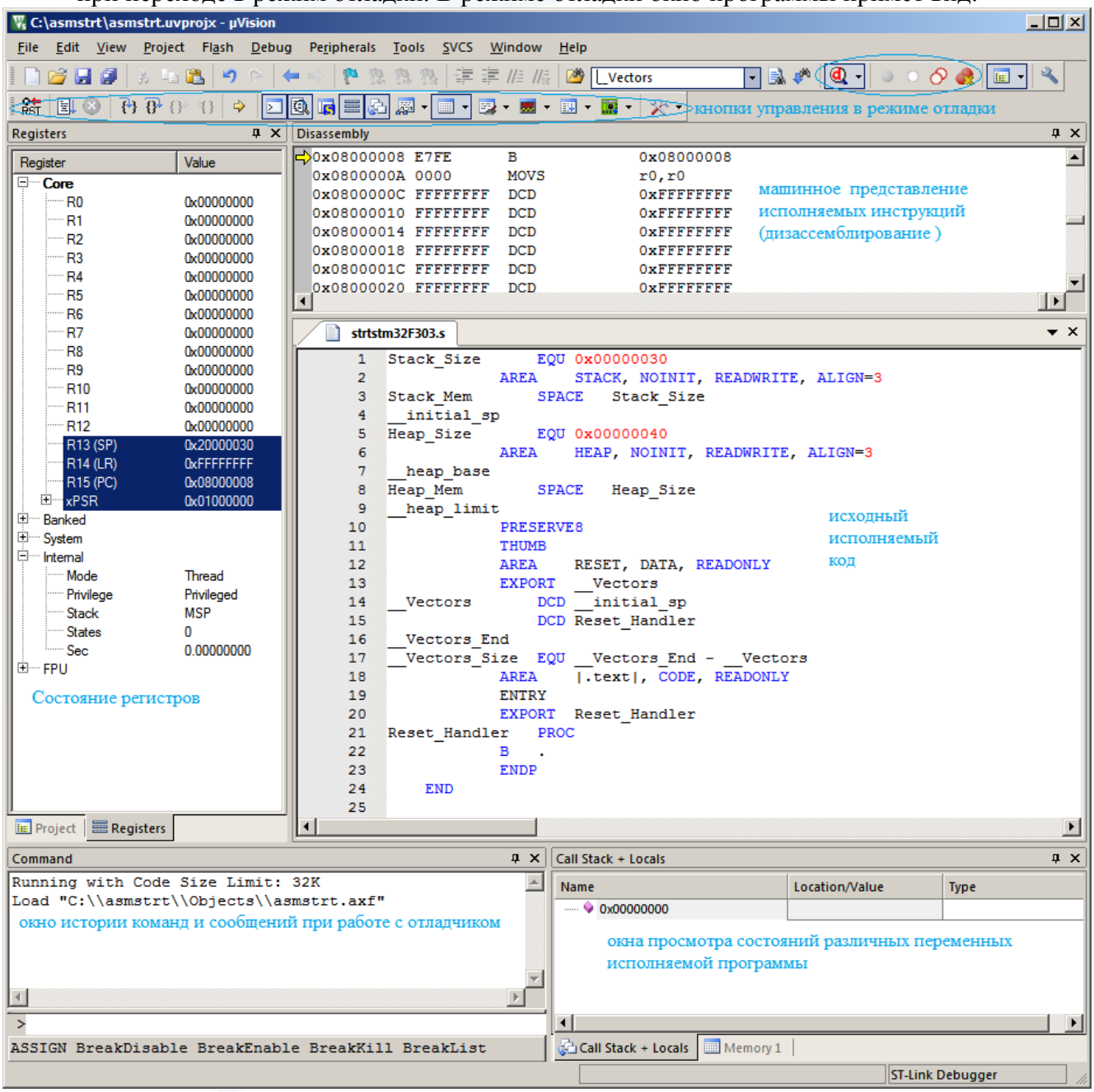
Разобравшись с программой перейдём к возможностям отладки в среде Keil.

1.2.5 Ядро микроконтроллера ARM Cortex-M4 оснащено технологией отладки и трассировки ARM CoreSight (подробнее тут *http://www2.keil.com/coresight/*), которая позволяет осуществлять доступ к памяти «на лету», то есть во время исполнения рабочей программы, без использования каких-либо дополнительных программных модулей. При выполнении программы разработчик непрерывно получает информацию об актуальном состоянии памяти и переменных контроллера. Существует возможность устанавливать точки останова как триггер для какой-либо переменной. Переход в режим отладки запускается нажатием **Ctrl + F5**, либо через главное меню *Debug >> Start/StopDebug Session*, либо значком на панели программных инструментов. После того как

Keil нас любезно предупредит о неполноценности нашей версии с ограничением на размер исполняемого кода в 32 Кб:



нажимаем OK и продолжаем работу. Данное напоминание будет появляться постоянно при переходе в режим отладки. В режиме отладки окно программы примет вид:



1) Состояние регистров. Как видно из окна *Registers* метка __initial_sp уже загружена в R13(SP) , а в регистр R15(PC) уже загружен адрес следующей исполняемой команды функции Reset_Handler (см.1.2.4 п.3). Регистр R14(LR - регистр связи, используется для запоминания адреса возврата при вызове подпрограмм) инициализирован значением

0xFFFFFFFF. Регистр xPSR содержит флаги результатов выполнения арифметических и логических операций, состояние выполнения программы и номер обрабатываемого в данный момент прерывания. В документации об этом регистре пишут во множественном числе. Это сделано потому, что к трём его частям можно обращаться как к отдельным регистрам. Эти «подрегистры» называются: APSR — регистр состояния приложения (тут как раз хранятся флаги), IPSR — регистр состояния прерывания (содержит номер обрабатываемого прерывания) и EPSR — регистр состояния выполнения. О чём в руководстве программиста PM0214 *Programming manual* (https://www.st.com/resource/en/programming_manual/dm00046982.pdf) на странице 18 написано:

General-purpose registers

R0-R12 are 32-bit general-purpose registers for data operations.

Stack pointer

The *Stack Pointer* (SP) is register R13. In Thread mode, bit[1] of the CONTROL register indicates the stack pointer to use:

- 0: *Main Stack Pointer* (MSP). This is the reset value.
- 1: *Process Stack Pointer* (PSP).

On reset, the processor loads the MSP with the value from address 0x00000000.

Link register

The *Link Register* (LR) is register R14. It stores the return information for subroutines, function calls, and exceptions. On reset, the processor loads the LR value 0xFFFFFFFF.

Program counter

The *Program Counter* (PC) is register R15. It contains the current program address. On reset, the processor loads the PC with the value of the reset vector, which is at address 0x00000004. Bit[0] of the value is loaded into the EPSR T-bit at reset and must be 1.

Program status register

The *Program Status Register* (PSR) combines:

- *Application Program Status Register* (APSR)
- *Interrupt Program Status Register* (IPSR)
- *Execution Program Status Register* (EPSR)

Здесь необходимо пояснить что регистров R13(SP) вообще два, а не один. Первый называется MSP — указатель основного стека, а второй PSP — указатель стека процесса. Однако в каждый момент доступен только один из этих регистров. Что определяется в одном из регистров специального назначения CONTROL (PM0214 стр.24):

CONTROL register

The CONTROL register controls the stack used and the privilege level for software execution when the processor is in Thread mode and indicates whether the FPU state is active. See the register summary in *Table 2 on page 17* for its attributes.

Table 10. CONTROL register bit definitions

Bits	Function
Bits 31:3	Reserved
Bit 2	**FPCA:** Indicates whether floating-point context currently active: 0: No floating-point context active 1: Floating-point context active. The Cortex-M4 uses this bit to determine whether to preserve floating-point state when processing an exception.
Bit 1	**SPSEL:** Active stack pointer selection. Selects the current stack: 0: MSP is the current stack pointer 1: PSP is the current stack pointer. In Handler mode this bit reads as zero and ignores writes. The Cortex-M4 updates this bit automatically on exception return.
Bit 0	**nPRIV:** Thread mode privilege level. Defines the Thread mode privilege level. 0: Privileged 1: Unprivileged.

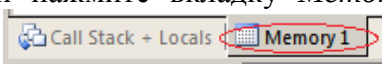
В дальнейшем необходимо самостоятельно изучить все регистры специального назначения и их флаги (http://radiohlam.ru/?p=1345).

2) В окне *Disassembly* наблюдаем несколько столбцов. Первый столбец это адреса по которым хранятся исполняемые команды, второй столбец это машинные коды этих команд, третий столбец сами команды на ассемблере вместе с операндами:

```
Disassembly
0x08000008 E7FE        B                    0x08000008
адреса команд  машинные коды  команды ассемблере  операнды
```

жёлтая стрелка указывает на текущую команду подлежащую исполнению.

Как видно в регистре R15(PC) загружен адрес 0x08000008 как раз нашей первой команды бесконечный цикл (в разделе 3.9.5 PM0214 на стр. 141 можно подробно изучить эту команду).

3) Для просмотра областей памяти нажмите вкладку *Memory 1* в окне просмотра состояний различных переменных 

. В появившемся окне *Memory 1* в поле *Address:* введите адрес начала флэш памяти 0x08000000 (см. п. 1.2.3) и нажмите ввод. Получим текущий отпечаток памяти:

```
Memory 1
Address: 0x08000000
0x08000000: 30 00 00 20 09 00 00 08 FE E7 00 00 FF FF FF FF
0x08000010: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
```

из которого видно что первые четыре байта это указатель вершины стека - адрес 0x20 00 00 30 (архитектура little-endian), вторые четыре байта это адрес функции Reset_Handler 0x08 00 00 09, третье это два байта кода команды бесконечного цикла 0xE7 FE и последние два нулевых байта дополнение до четырёх. Итого 12 байт кода записанного в ROM (см. п. 1.2.2). Далее сравним эту область с памятью по нулевому адресу. Изменим адрес на 0x00000000:

```
Memory 1
Address: 0x00000000
0x00000000: 30 00 00 20 09 00 00 08 FE E7 00 00 FF FF FF FF
0x00000010: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
```

Как видно значения абсолютно одинаковые, так как эти адресные регионы совмещены (см. п.1.2.4 пп.3).

Теперь введём адрес дна стека 0x2000 0000 и видим что всё 48 байт стека забиты случайными значениями. Давайте изменим нашу программу, для этого выйдем из режима отладки также как в него вошли (см. п.1.2.5), нажатием **Ctrl + F5** например.

1.2.6 Воспользуемся командой LDR (PM0214 стр.70) - загружает в регистр содержимое ячеек памяти. Загрузим в регистры адреса по нашим меткам.

```
21  Reset_Handler   PROC
22              LDR     R0, =Stack_Mem          ; загружаем в регистр R0 Stack_Mem
23              LDR     R1, =Stack_Size         ; загружаем в регистр R1 Stack_Size
24              LDR     R2, =__initial_sp       ; загружаем в регистр R2 __initial_sp
25              LDR     R3, =__heap_base        ; загружаем в регистр R3 __heap_base
26              LDR     R4, =Heap_Mem           ; загружаем в регистр R4 Heap_Mem
27              LDR     R5, =Heap_Size          ; загружаем в регистр R5 Heap_Size
28              LDR     R6, =__heap_limit       ; загружаем в регистр R6 __heap_limit
29              LDR     R7, =__Vectors          ; загружаем в регистр R7 __Vectors
30              LDR     R8, =__Vectors_Size     ; загружаем в регистр R8 __Vectors_Size
31              LDR     R9, =__Vectors_End      ; загружаем в регистр R9 __Vectors_End
32              LDR     R10, =Reset_Handler     ; загружаем в регистр R10 Reset_Handler
```

Далее поместим содержимое регистров в стек командой PUSH (*PM0214* стр.77) .

```
33              PUSH    {R0,R1,R2}
34              PUSH    {R3,R4,R5,R6}
35              PUSH    {R7,R8,R9,R10}
36              B    .
37              NOP
38              ENDP
```

Команда NOP (нет операции) добавлена для выравнивания пула кода до границы четыре байта, что бы избежать сообщения *warning: A1581W*, в противном случае с этим сообщением компилятор сам дополнит уже нулями два байта.

Запускаем режим отладки. И далее несколько раз нажимаем **F11**, либо значок в панели инструментов, либо через меню *Debug >> Step*. И наблюдаем как поочерёдно загружаются в регистры значения наших меток. При этом обратите внимание как меняется регистр счётчик команд R15(PC). При достижении строки 33 получим такую картину состояния наших регистров:

Registers

Register	Value
Core	
R0	0x20000040
R1	0x00000030
R2	0x20000070
R3	0x20000000
R4	0x20000000
R5	0x00000040
R6	0x20000040
R7	0x08000000
R8	0x00000008
R9	0x08000008
R10	0x08000009
R11	0x00000000
R12	0x00000000
R13 (SP)	0x20000070
R14 (LR)	0xFFFFFFFF
R15 (PC)	0x08000028
xPSR	0x01000000

и сравним значения регистров и секцию *Image Symbol Table map*-файла.

```
__Vectors_End                              0x08000008
HEAP                                       0x20000000
Heap_Mem                                   0x20000000
__heap_base                                0x20000000
STACK                                      0x20000040
Stack_Mem                                  0x20000040
__heap_limit                               0x20000040
__initial_sp                               0x20000070

Global Symbols

Symbol Name                                Value     C

BuildAttributes$$THM_ISAv4$E$P$D$K$B$S$7EM$VFPi3$EXT
__Vectors                                  0x08000000
Reset_Handler                              0x08000009
```

Как видим всё совпадает. Обратите внимание как теперь расположены в памяти:

1) стек и 'heap'

```
Exec Addr     Load Addr     Size          Type    Attr      Idx     E Section Name

0x20000000        -         0x00000040    Zero    RW          2     HEAP
0x20000040        -         0x00000030    Zero    RW          1     STACK
```

2) совмещение адресов 0x0800 0000 и 0x0000 0000 в единый регион в памяти

Memory 1

Address: 0x08000000

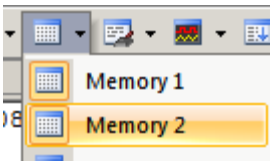
```
0x08000000: 70 00 00 20 09 00 00 08 0A 48 4F F0 30 01 0A 4A
0x08000010: 0A 4B 0B 4C 4F F0 40 05 0A 4E 0B 4F 4F F0 08 08
0x08000020: DF F8 28 90 DF F8 28 A0 07 B4 78 B4 2D E9 80 07
0x08000030: FE E7 00 BF 40 00 00 20 70 00 00 20 00 00 00 20
0x08000040: 00 00 00 20 40 00 00 20 00 00 00 08 08 00 00 08
0x08000050: 09 00 00 08 FF FF FF FF FF FF FF FF FF FF FF FF
0x08000060: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
```

Memory 2

Address: 0

```
0x00000000: 70 00 00 20 09 00 00 08 0A 48 4F F0 30 01 0A 4A
0x00000010: 0A 4B 0B 4C 4F F0 40 05 0A 4E 0B 4F 4F F0 08 08
0x00000020: DF F8 28 90 DF F8 28 A0 07 B4 78 B4 2D E9 80 07
0x00000030: FE E7 00 BF 40 00 00 20 70 00 00 20 00 00 00 20
0x00000040: 00 00 00 20 40 00 00 20 00 00 00 08 08 00 00 08
0x00000050: 09 00 00 08 FF FF FF FF FF FF FF FF FF FF FF FF
0x00000060: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
0x00000070: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
```

Второе окно *Memory 2* открывается либо через меню *View >> Memory Windows >> Memory 2*, либо через кнопку меню на панели инструментов:



3) расположение исполняемого кода в памяти:

с нулевого адреса идёт таблица векторов

0x2000 0070 вершина стека __initial_sp

0x0800 0009 адрес обработчика сброса Reset_Handler

```
Memory 2
0
0x00000000: 70 00 00 20
0x00000004: 09 00 00 08
```

далее исполняемые инструкции функции Reset_Handler

```
48 0A        LDR   R0, =Stack_Mem
F0 4F 01 30  LDR   R1, =Stack_Size
4A0A         LDR   R2, =__initial_sp
4B0A         LDR   R3, =__heap_base
```

и т.д.

```
0x00000008: 0A 48
0x0000000A: 4F F0
0x0000000C: 30 01
0x0000000E: 0A 4A
0x00000010: 0A 4B
```

после тела функции идут значения наших меток

```
0x00000034: 40 00 00 20   Stack_Mem
0x00000038: 70 00 00 20   __initial_sp
0x0000003C: 00 00 00 20   __heap_base
0x00000040: 00 00 00 20   Heap_Mem
0x00000044: 40 00 00 20   __heap_limit
0x00000048: 00 00 00 08   __Vectors
0x0000004C: 08 00 00 08   __Vectors_End
0x00000050: 09 00 00 08   Reset_Handler
0x00000054: FF FF FF FF
```

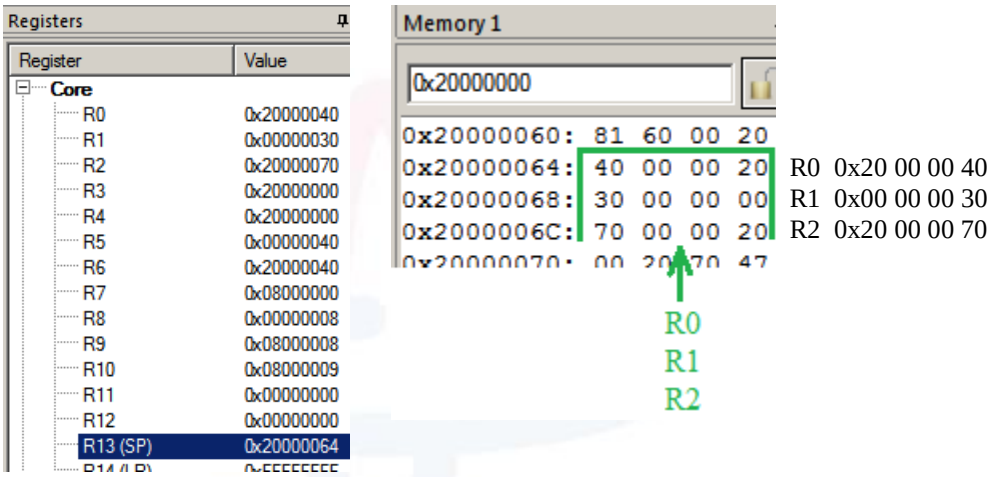
адреса по которым считываются значения меток указаны в окне *Disassembly*:

```
Disassembly
    22:                          LDR     R0, =Stack_Mem
0x08000008 480A      LDR            r0,[pc,#40]  ; @0x08000034
    23:                          LDR     R1, =Stack_Size
0x0800000A F04F0130  MOV            r1,#0x30
    24:                          LDR     R2, =__initial_sp
0x0800000E 4A0A      LDR            r2,[pc,#40]  ; @0x08000038
    25:                          LDR     R3, =__heap_base
0x08000010 4B0A      LDR            r3,[pc,#40]  ; @0x0800003C
    26:                          LDR     R4, =Heap_Mem
0x08000012 4C0B      LDR            r4,[pc,#44]  ; @0x08000040
    27:                          LDR     R5, =Heap_Size
```

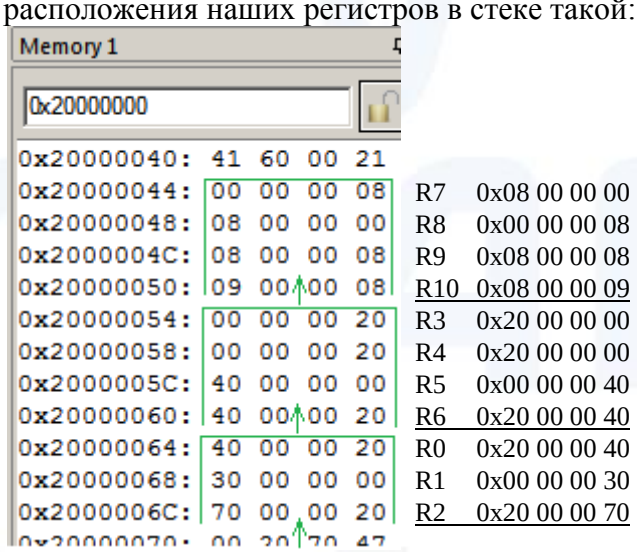
(напоминаю что регионы в памяти по адресам 0x0800 0000 и 0x0000 0000 - совмещены в один, т.е. на один байт данных приходиться два адреса и не играет роли какой использовать)

Итого весь код занимает регион с 0x0000 0000 по 0x0000 0053 адреса, всего 84 байта (*Total ROM Size (Code + RO Data + RW Data) 84 (0.08kB)*)

Далее рассмотрим как работает стек. Перейдём в окне *Memory 1* по адресу 0x2000 0000 и при выполнении строки 33 (нажимаем F11), команды PUSH {R0,R1,R2}, у нас начиная с адреса вершины стека R13(SP)-1 будут записаны значения регистров:



При этом указатель вершины стека R13(SP) измениться и станет равным 0x2000 0064. Следующими командами мы поместим в стек остальные регистры. Общий вид расположения наших регистров в стеке такой:



Т.е. надо иметь ввиду что извлечение данных из стека должно происходить в обратном порядке и такими же порциями как и при помещении в стек.

Покажем это. Остановим отладку. Допишем следующий код:

```
36          POP     {R10,R9,R8,R7}
37          POP     {R6,R5,R4,R3,R2,R1,R0}
```

Инструкция POP извлекает данные из стека (*PM0214* стр.77). Откомпилируем и снова запустим отладку. При прохождении 36 строки регистры R7, R8, R9, R10 останутся без изменений, т.е. значения считанные из стека их не изменят. А вот при прохождении 37 строки регистры с R0 по R6 поменяют значения:

до

Registers

Register	Value
Core	
R0	0x20000040
R1	0x00000030
R2	0x20000070
R3	0x20000000
R4	0x20000000
R5	0x00000040
R6	0x20000040
R7	0x08000000
R8	0x00000008
R9	0x08000008
R10	0x08000009
R11	0x00000000
R12	0x00000000
R13 (SP)	0x20000054

, после

Registers

Register	Value
Core	
R0	0x20000000
R1	0x20000000
R2	0x00000040
R3	0x20000040
R4	0x20000040
R5	0x00000030
R6	0x20000070
R7	0x08000000
R8	0x00000008
R9	0x08000008
R10	0x08000009
R11	0x00000000
R12	0x00000000
R13 (SP)	0x20000070

Также надо иметь ввиду что если продолжать помещать значения в стек, то можно запросто опуститься ниже дна стека при этом будут затираться данные которые расположены ниже стека, в наше случае область памяти '*heap*'. Т.е. переполнение (срыв) стека нужно отслеживать самому программисту. Тоже самое нужно сказать и в отношении '*heap*'. ОЗУ всего 40 Кб, это всего лишь двумерный массив 100 на 100 элементов типа int. Но как мы видим нам нужна ещё память и под стек, и под другие переменные. Поэтому при переполнении '*heap*' мы начинаем затирать данные в стеке и других переменных. На языке более высокого уровня си, существуют библиотечные функции которые контролируют данные нюансы системы, но необходимо чёткое понимание как они работают и в каких случаях.

1.2.7 Затирание области стека и '*heap*'. Выйдем из режима отладки и напишем новую программу заполняющую область стека кодом 0xAA, а область '*heap*' кодом 0xBB:

```
21  Reset_Handler   PROC
22                  LDR     R0, =Stack_Mem
23                  LDR     R1, =Stack_Size
24                  LDR     R2, =0xAAAAAAAA
25  fillstack       STR     R2, [R0], #4
26                  SUBS    R1,#4
27                  BNE     fillstack
28
29                  LDR     R0, =Heap_Mem
30                  LDR     R1, =Heap_Size
31                  LDR     R2, =0xBBBBBBBB
32  fillheap        STR     R2, [R0], #4
33                  SUBS    R1,#4
34                  BNE     fillheap
```

Разобраться в программе нужно самостоятельно используя руководство программиста *PM0214* страницы 72 (STR), 82(SUBS), 141, 66(BNE).

Скомпилируйте и запустите в режиме отладки.

1.3 Работа с регистрами периферийных устройств в памяти микроконтроллера

1.3.1 Как уже стало ясно, при работе с микроконтроллером STM32, часто нужно заглядывать в документацию, чтобы найти какие-либо сведения. Основной перечень документации на микроконтроллер STM32F303VC доступен на сайте компании ST Microelectronics по ссылке https://www.st.com/content/st_com/en/products/microcontrollers/stm32-32-bit-arm-cortex-mcus/stm32-mainstream-mcus/stm32f3-series/stm32f303/stm32f303vc.html

Перечень разделён на разделы, выделим полезные для нас из раздела Technical Documentation - техническая документация:

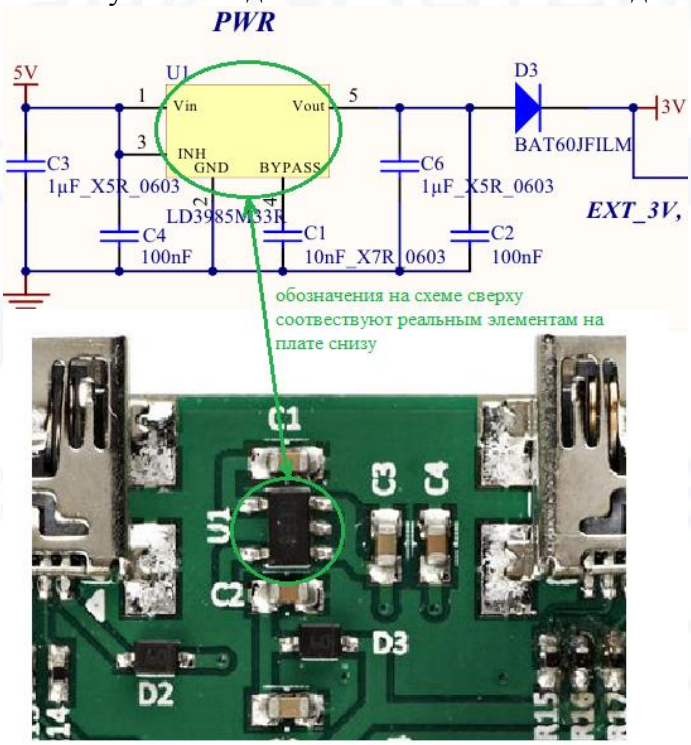
– Product Specifications - Спецификация изделия. Здесь один файл спецификации *DS9118* в котором подробно описаны аппаратные и программные особенности микроконтроллера STM32F303VC. Дополняет справочное руководство *RM0316*.
– Application Notes - Заметки по применению. Рекомендации по проектированию и настройке устройств с STM32;
– Reference Manuals - справочники. Здесь одно справочное руководство *RM0316* по семейству STM32F303xB/C/D/E (более 1000 страниц) с подробнейшей информацией по этой серии. В котором описаны архитектура, регистры, их биты, функции и периферия микроконтроллера.
– Programming Manuals - руководства программиста. Здесь одно руководство *PM0214* для программирования систем с ядром STM32 Cortex®-M4, в котором описывается программирование на уровне машинных команд и ассемблера, регистры общего и специального назначения.
– Errata Sheets - Описание аппаратных ошибок и "шероховатостей".

Далее следует раздел TOOLS AND SOFTWARE - инструментальные средства и программное обеспечение. В этом разделе можно найти и скачать все необходимые программные пакеты и библиотеки для комфортной работы с микроконтроллером STM32F303VC. Проявив усердие возможно самостоятельно разобраться во всём представленном разнообразии.

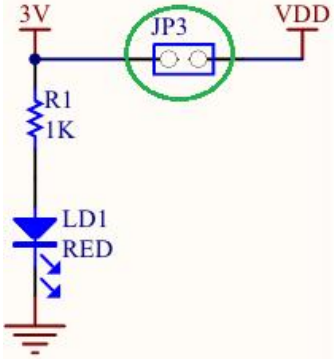
Так как мы работаем с отладочной платой STM32F3DISCOVERY то нам необходима схема этой платы, для этого найдём это название в перечне и перейдём по ссылке https://www.st.com/content/st_com/en/products/evaluation-tools/product-evaluation-tools/mcu-eval-tools/stm32-mcu-eval-tools/stm32-mcu-discovery-kits/stm32f3discovery.html. И в разделе User Manuals - руководство пользователя, скачаем документ *UM1570* - руководство по отладочной плате STM32F3DISCOVERY.

Итак основной перечень необходимых документов для дальнейшей работы в том числе и самостоятельной состоит всего из четырёх документов: *DS9118, RM0316, PM0214, UM1570.*

1.3.2 Разобравшись с документацией давайте разберемся со схемой. Питание нашего микроконтроллера осуществляется тремя вольтами (см. *UM1570*, стр. 33). Напряжение 5V поступает с подключаемого USB кабеля и далее преобразуется в 3V.



Далее через перемычку JP3 3V поступают на микроконтроллер. Питание микроконтроллера обозначается V_{DD}, а V_{SS} - земля ⏚ (можно считать ноль вольт).



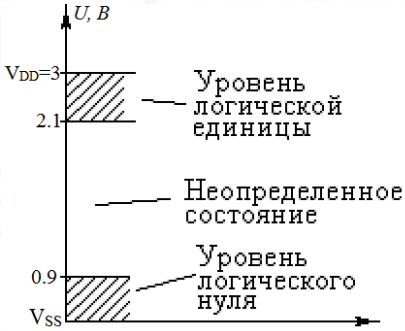
Таким образом перемычка JP3 должна быть закорочена, в противном случае питание не будет поступать на STM32F303VCT6.

Теперь разберём понятие логических входов/выходов. Логический (вход)/(выход) - это (вход)/(выход) устройства (на который подается)/(с которого поступает) сигнал логической единицы или логического нуля. Сигнал логического нуля или единицы – это сигнал, принимающий два возможных значения или «низкое» состояние «0», или «высокое» «1». Если объяснять совсем просто: когда мы подадим на светодиод 3 вольта (сигнал логическая единица), он начнет светиться. Если же мы подадим 0 вольт (сигнал логического ноля), то светодиод перестанет светиться. Сигналы логического нуля и единицы в зарубежных источниках обозначают как low level - низкий уровень и high level -высокий уровень соответственно.

Теперь откроем *DS9118* на странице 87 и посмотрим таблицу 54 в которой показаны уровни напряжений для логических входов/выходов нашего микроконтроллера (смотрим для всех входов/выходов All I/Os):

– логический ноль на входе (V_{IL}) соответствует напряжению до $0.3*V_{DD}$, т.е. до 0.3*3=0.9 V;
– логическая единица на входе (V_{IH}) соответствует напряжению свыше $0.7*V_{DD}$, т.е. свыше 0.7*3=2.1 V.

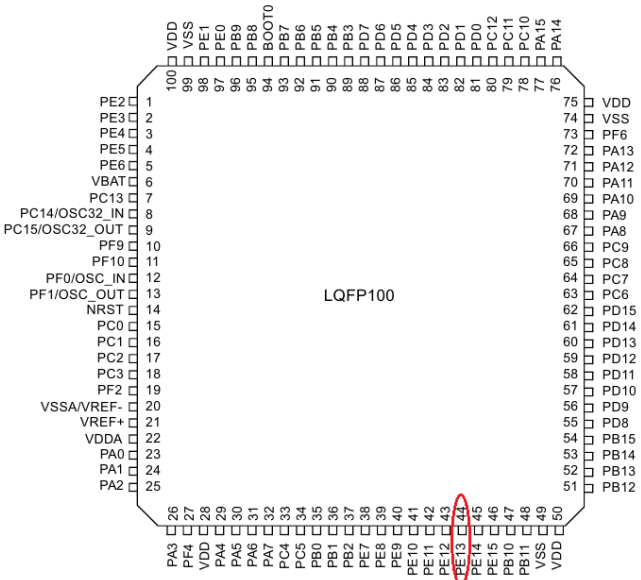
Что графически можно изобразить так:



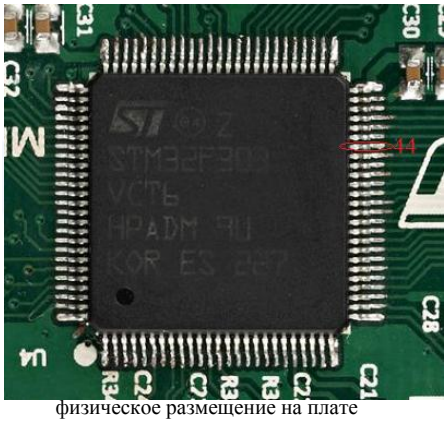
Поясним, что входное напряжение может быть и больше питания V_{DD}=3В и его микроконтроллер будет интерпретировать как логическую единицу, но чем больше напряжение на входе, напряжения V_{DD}, тем выше вероятность что вход микроконтроллера перегорит.

Обычно в процессоре логические сигнальные линии объединяются в порты ввода/вывода (general-purpose I/O port - GPIO). Каждый порт состоит из 8 или 16 сигнальных линий, стандартное обозначение которых выглядит как Pnx, где n – обозначение порта (прописными буквами латинского алфавита - A,B,C,D и т.д.); x – номер бита (линии) в порту (цифры от 0 до 15).

Разберёмся как установить сигнал логической единицы на 44 ножке нашего микроконтроллера.

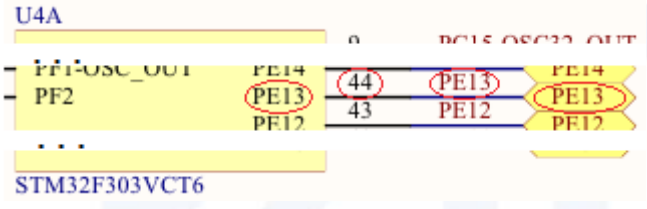


DS9118 стр.34 рисунок 6

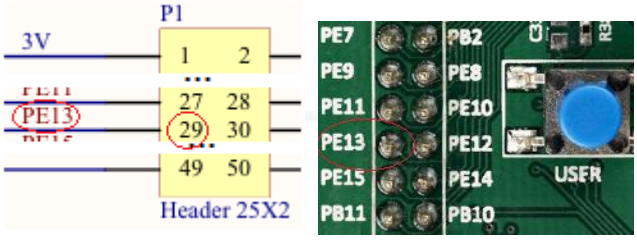


физическое размещение на плате

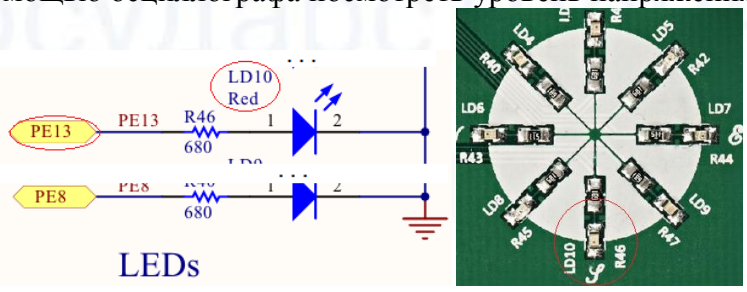
Рассмотрев схему (см. *UM1570*, стр. 32÷35) подробнее проследим путь сигнала с 44 ножки микроконтроллера:



1) выясняем что логически она входит в архитектуру микроконтроллера как порт в/в E и является 13 линией в этом порту, обозначение на схеме PE13.



2) вывод PE13 имеется на двухрядном штыревом разъёме P1. Таким образом удобно с помощью осциллографа посмотреть уровень напряжения на этом выходе.



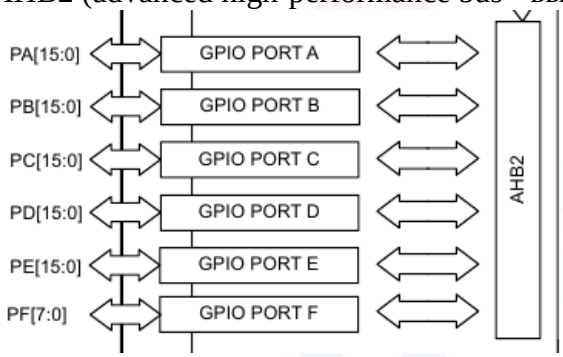
3) также вывод PE13 присоединен к красному светодиоду LD10 через резистор R46, что означает свечение светодиода при высоком уровне на 44 выводе процессора.

1.3.3 Включение в работу порта в/в (ПВВ) E.

Будем иметь ввиду что после сброса микроконтроллера по умолчанию выставляется: частота тактирования 8МГц от внутреннего генератора тактовой частоты; все порты (за исключением PB3÷4, PA13÷15) устанавливаются в состояние плавающий вход (*RM0316* стр.231 подраздел 11.3.1) и тактирование всех портов отключено.

Поэтому перед использованием ПВВ E необходимо разрешить его синхронизацию и вывести из состояния сброса. Управляет тактированием в процессоре подсистема RCC

(Reset and clock control - Сброс и управление тактированием, подробнее см. *RM0316* стр.123). Для управления этой подсистемой имеется перечень регистров. На рисунке 1, страницы 13 спецификации *DS9118* видно, что все ПВВ (GPIO) подключены к шине AHB2 (advanced high-performance bus - высокопроизводительная шина)



Для управления тактированием шины AHB2 используется регистр RCC_AHBENR (*RM0316* стр.148 подраздел 9.4.6) в котором имеются следующие управляющие биты для ПВВ:

24	23	22	21	20	19	18	17	16
CEN	IOPG EN(1)	IOPF EN	IOPE EN	IOPD EN	IOPC EN	IOPB EN	IOPA EN	IOPH EN(1)
rw	rw	rw	rw	rw	rw	rw	rw	rw

где указано :
Bit 21 **IOPEEN**: I/O port E clock enable(STM32F303xB/C and STM32F358xC devices only)\
(Бит 21 **IOPEEN** : разрешает тактирование ПВВ E)
Set and cleared by software. (Устанавливается и очищается программно)
0: I/O port E clock disabled. (0: Тактирование порта ввода/вывода E запрещено)
1: I/O port E clock enabled. (1: Тактирование порта ввода/вывода E разрешено)
Таким образом для включения в работу ПВВ E необходимо установить в единицу 21-ый бит регистра RCC_AHBENR.
Давайте выясним адрес этого регистра и поменяем наш бит. Для этого сначала определим область памяти в которой находиться этот регистр. Заходим в раздел номер 5 Memory mapping (распределение памяти) спецификации *DS9118* и на странице 54 находим область памяти подсистемы RCC:

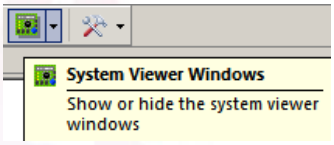
AHB1	0x4002 2000 - 0x4002 23FF	1 K	Flash interface
	0x4002 1400 - 0x4002 1FFF	3 K	Reserved
	0x4002 1000 - 0x4002 13FF	1 K	RCC
	0x4002 0800 - 0x4002 0FFF	2 K	Reserved

Как видно из таблицы адрес начала области памяти подсистемы RCC начинается с адреса 0x4002 1000. Далее снова вернёмся к описанию регистра RCC_AHBENR (*RM0316* стр.148 подраздел 9.4.6), нас интересует поле Address offset: 0x14 (Адрес смещения). Зная смещение вычислим адрес по которому располагается регистр RCC_AHBENR: 0x4002 1000+0x0000 0014 = 0x4002 1014. Итак кодируем:

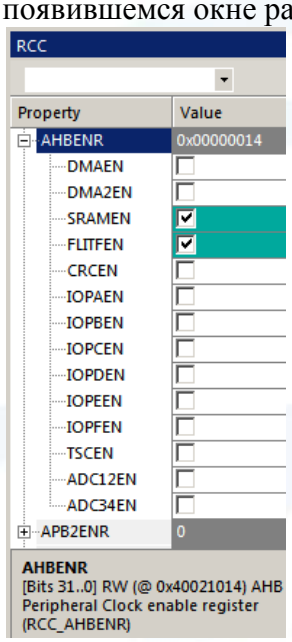
```
LDR     R0, =0x40021014
LDR     R1, =0x00200000
LDR     R2, [R0]
ORR     R1, R1, R2
STR     R1, [R0]
```

Сначала в регистр R0 загружаем адрес регистра RCC_AHBENR , далее в регистр R1 загружаем нашу единицу в 21 бите. Затем в регистр R2 загружаем значения битов регистра RCC_AHBENR. Далее командой ORR производим операцию "поразрядного или" для значений регистров R1 и R2, а результат сохраняем в R1. Далее пишем

значение регистра R1 в регистр RCC_AHBENR. На языке си это можно записать в одну строчку: `*(uint32_t*)(0x40021014)|=0x00200000;`

Теперь скомпилируем и запустим в режиме отладки. Для контроля откроем регистры подсистемы RCC, через главное меню *View* >> *System Viewer* >> *RCC*, через панель инструментов значок: 

и в выпавшем списке выбрать RCC. В появившемся окне развернём регистр AHBENR.



Напротив самого регистра мы видим его текущее значение 0x00000014. В самом низу показано что в регистре 32 бита [Bits 31..0], доступен для чтения и записи (RW), его физический адрес в памяти 0x40021014, и указание на то что это регистр RCC_AHBENR. Также наглядно показано текущее состояния управляющих битов регистра. Их состояние можно поменять левой кнопкой мыши, причем эти изменения произойдут и в микроконтроллере.

При пошаговом исполнении инструкций ( ) проходя строку 24 наблюдаем как в R2 загрузилось значение R2 0x00000014, после команды ORR в R1 загрузилось значение R1 0x00200014, и последняя команда изменила значение регистра AHBENR (напротив теперь значение 0x00200014 вместо 0x00000014) , плюс появилась галочка напротив бита IOPEEN: IOPEEN ☑.

1.3.4 Теперь включив в работу ПВВ E установим высокий уровень на 13 линии (PE13) или 44ой ножке процессора.

Открываем справочное руководство *RM0316* стр.228 раздел 11 и внимательно изучаем. Выясняется что из множества режимов работы ПВВ нам нужен только один (табл. 72):

Table 72. Port bit configuration table(1)

MODER(i) [1:0]	OTYPER(i)	OSPEEDR(i) [1:0]	PUPDR(i) [1:0]		I/O configuration	
01	0	SPEED [1:0]	0	0	GP output	PP
	0		0	1	GP output	PP + PU
	0		1	0	GP output	PP + PD
	0		1	1	Reserved	
	1		0	0	GP output	OD
	1		0	1	GP output	OD + PU
	1		1	0	GP output	OD + PD
	1		1	1	Reserved (GP output OD)	

режим двухтактного выхода (General-Purpose output Push-Pull). Для настройки в этот режим необходимо:

1) В регистре GPIOE_MODER (*RM0316* стр.237 подраздел 11.4.1) в 26ой бит записать единичку, а в 27ой бит нолик, т.е. MODER13[1;0] =0x01

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
MODER15[1:0]		MODER14[1:0]		MODER13[1:0]		MODER12[1:0]		MODER11[1:0]		MODER10[1:0]		MODER9[1:0]		MODER8[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

2) В регистре GPIOE_OTYPER в 13й бит записать ноль, но так как после сброса микроконтроллера он обнуляется мы этот шаг пропустим.

3) Регистр GPIOE_OSPEEDR влияет на скорость переключения линии, нам это не важно, поэтому его тоже не трогаем.
4) В регистре GPIOE_PUPDR записать нули в биты 26 и 27, но так как после сброса он устанавливается в ноль мы этот шаг пропускаем.
5) В регистр GPIOE_ODR (*RM0316* стр.239 подраздел 11.4.6) в 13ый бит установить единичку.
Таким образом нам необходимо всего две операции: записать 0x04000000 в регистр GPIOE_MODER, записать 0x00002000 в регистр GPIOE_ODR. Осталось вычислить адреса этих регистров по схожей схеме пункта 1.3.3.
Находим адрес области памяти GPIOE (*DS9118* стр. 54):

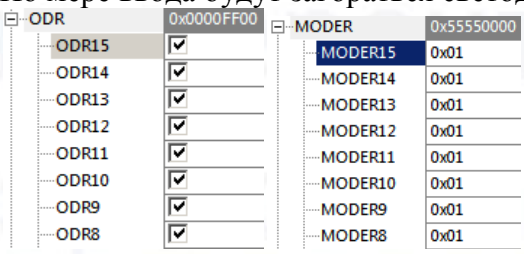
Table 20. STM32F303xB/STM32F303xC memory map, peripheral register boundary addresses(1)

Bus	Boundary address	Size (bytes)	Peripheral
AHB3	0x5000 0400 - 0x5000 07FF	1 K	ADC3 - ADC4
	0x5000 0000 - 0x5000 03FF	1 K	ADC1 - ADC2
	0x4800 1800 - 0x4FFF FFFF	~132 M	Reserved
	0x4800 1400 - 0x4800 17FF	1 K	GPIOF
	0x4800 1000 - 0x4800 13FF	1 K	GPIOE

И учитывая смещение (Address offset) находим адреса:
– регистр GPIOE_MODER 0x48001000 + 0x00=0x48001000;
– регистр GPIOE_ODR 0x48001000 + 0x14=0x48001014;
Кодируем:

```
27          LDR     R0, =0x48001000
28          LDR     R1, =0x04000000
29          STR     R1, [R0]
30          LDR     R0, =0x48001014
31          LDR     R1, =0x00002000
32          STR     R1, [R0]
```

Здесь всё по схеме как и предыдущем подразделе 1.3.3, за исключением операции "поразрядного или", так как можно не сохранять состояние регистров. В режиме отладки для наглядности производимых изменений откроем окно регистров ПВВ E: *View >> System Viewer >> GPIO >> GPIOE*. При отладочном проходе 29 строки в регистре MODER появиться 0x01 напротив MODER13. Далее при проходе 32 строки в регистре ODR появиться галочка напротив ODR13 и зажжется красный светодиод.
Поскольку управлять состоянием регистров можно и через систему отладки, зажжём остальные светодиоды. Для этого в регистре ODR поставим галочки напротив ODR8;9;10 до 15 включительно. Но горит пока только наш светодиод, потому что ещё не настроен регистр MODER, раскроем его и пропишем 0x01 напротив MODER8;9;10 ÷15. По мере ввода будут загораться светодиоды.



Соответствие светодиодов и линий ПВВ установите сами с помощью схемы *UM1570*.
Так как программно у нас зажигается только один светодиод, то нажатие клавиши сброса

обнулит все изменения внёсенные через отладку.

1.3.5 Более сложная программа управления светодиодами.

Для удобства управления ПВВ добавлены регистры GPIOx_BSRR (Address offset: 0x18; *RM0316* стр.240 подраздел 11.4.7) и GPIOx_BRR(Address offset: 0x28; *RM0316* стр.242 подраздел 11.4.11), с помощью которых можно управлять состоянием регистра GPIOx_ODR. При записи единицы в биты младшего полуслова регистра GPIOE_BSRR, либо GPIOE_BRR происходит либо установка единицы, либо сброс в ноль соответствующего бита в регистре GPIOE_ODR.

Определим адреса этих регистров:

– GPIOE_BSRR 0x48001000 + 0x18=0x48001018;
– GPIOE_BRR 0x48001000 + 0x28=0x48001028.

Начнём с создания процедуры паузы (задержки). Для этого создадим ассемблерный файл delay.s (см. пп. 1.1.7). Наполним его следующим кодом:

strtstm32F303.s | delay.s | asmstrt.map

```
1          PRESERVE8
2          THUMB
3          AREA    subrout, CODE, READONLY
4  delay       PROC
5      EXPORT delay
6                  POP     {R6}
7  delay_loop      SUBS    R6, R6, #1
8                  BNE delay_loop
9                  BX      LR
10             ENDP
11     END
```

Создаём секцию для подпрограмм subrout (имя может быть любое). В этой секции объявляем процедуру delay с глобальной видимостью. Далее в строке 6 достаём из стека в регистр R6 значение пустых циклов задержки. Далее начинается пустой цикл delay_loop в котором уменьшаем регистр R6 на единицу, и далее если R6 не ноль переходим снова к метке delay_loop.

Как только значение регистра R6 станет равным нулю выходим из подпрограммы командой BX LR (R14 (LR)).

Основная программа выглядит так:

```
21  Reset_Handler   PROC
22              IMPORT  delay
23              LDR     R0, =0x40021014
24              LDR     R1, =0x00200000
25              LDR     R2, [R0]
26              ORR     R1, R1, R2
27              STR     R1, [R0]
28              LDR     R0, =0x48001000
29              LDR     R1, =0x55550000
30              STR     R1, [R0]
31              LDR     R0, =0x48001018
32              LDR     R1, =0x48001028
33              LDR     R2, =1<<8 :OR: 1<<10 :OR: 1<<12 :OR: 1<<14
34              LDR     R3, =1<<9 :OR: 1<<11 :OR: 1<<13 :OR: 1<<15
35  loopb       LDR     R5, =0x000411AA*3 ;0.1*3=0.3 сек
36  loopa       STR     R2, [R0]
37              STR     R3, [R1]
38              PUSH    {R5}
39              BL      delay
40              STR     R3, [R0]
41              STR     R2, [R1]
42              PUSH    {R5}
43              BL      delay
44              SUBS    R5, R5, #0x8000
45              BGT loopa
46              B loopb
47              ENDP
48      END
```

Строкой 22 указываем что delay внешняя функция.

В строках:

– 23÷27 разрешаем работу GPIOE;
– 28÷30 выставляем линии PE8÷15 в режим на вывод сигналов;
– 31÷34 подготавливаем регистры для работы в цикле. В R0 загружаем адрес 0x48001018 регистра GPIOE_BSRR, в R1 загружаем адрес 0x48001028 регистра GPIOE_BRR, в R2 загружаем константу для управления 8,10,12,14 линиями, в R3 загружаем константу для управления 9,11,13,15 линиями;
– 35 начало внешнего цикла loopb, загружаем в регистр R5 значение счётчика задержки;
– 36 начало цикла loopa, пишем в регистр GPIOE_BSRR значение регистра R2, тем самым устанавливаем в единицы биты 8,10,12,14 в регистре GPIOE_ODR, т.е зажигаем светодиоды LD4,5,9,8;
– 37 пишем в регистр GPIOE_BRR значение регистра R3, тем самым сбрасываем в ноль биты 9,11,13,15 в регистре GPIOE_ODR, т.е выключаем светодиоды LD 3,7,10,6;
– 38÷39 помещаем в стек счётчик задержки из регистра R5, вызываем подпрограмму delay (задержка);
– 40÷41 здесь наоборот выключаем светодиоды LD4,5,9,8 и включаем LD 3,7,10,6;
– 42÷43 снова помещаем в стек счётчик задержки из регистра R5, вызываем подпрограмму delay (задержка);
– 44 уменьшаем значение счётчика задержки регистра R5 на 0x8000;
– 45 повторяем цикл loopa пока R5>0;
– 46 повторяем цикл loopb.

В нашей программе можно сократить код воспользовавшись технологией побитовой адресации Bit-banding (PM0214 стр.31). Регистр RCC_AHBENR входит в регион адресов с побитовым доступом:

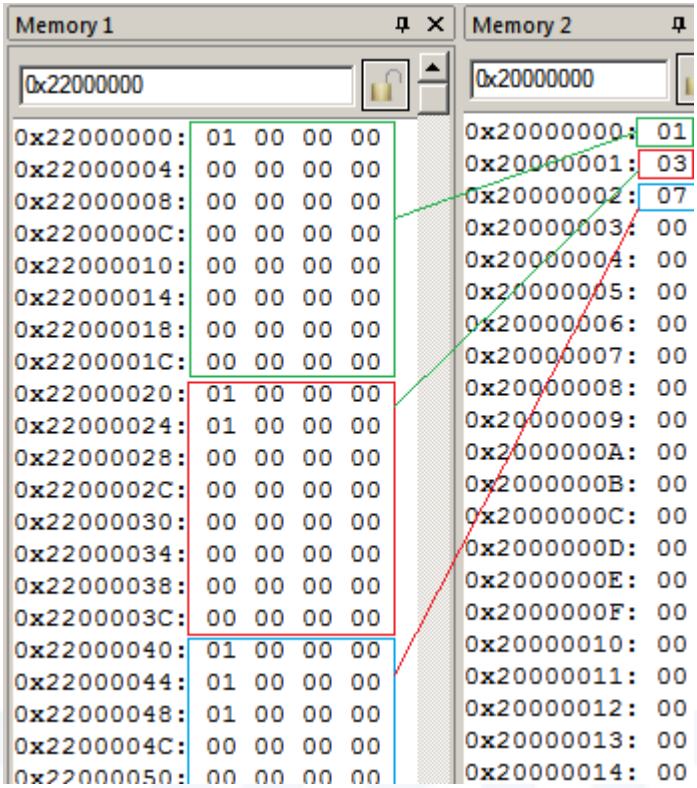
Table 14. Peripheral memory bit-banding regions

Address range	Memory region	Instruction and data accesses
0x40000000-0x400FFFFF	Peripheral bit-band region	Direct accesses to this memory range behave as peripheral memory accesses, but this region is also bit addressable through bit-band alias.
0x42000000-0x43FFFFFF	Peripheral bit-band alias	Data accesses to this region are remapped to bit-band region. A write operation is performed as read-modify-write. Instruction accesses are not permitted.

```
23      LDR R10,=0x1
24              ;bit_band_base + (     byte_offset          * 32) + (bit_number * 4)
25      LDR R0,=0x42000000     + (0x40021014 - 0x40000000) * 32  +      21      * 4
26      STRB        R10, [R0]
```

поэтому воспользовавшись формулами пересчёта адреса доступа мы можем изменить 21й бит в регистре RCC_AHBENR за две команды. В 25 строке загружаем в R0 адрес отображения 21го бита в области Peripheral bit-band alias, и далее в 26 строке записываем единицу по этому адресу. Протестируйте этот код самостоятельно и убедитесь что он также работает.

Разобраться как работает Bit-banding поможет код из подраздела 1.2.7. Заполнив области стека и '*heap*' нулями запишите по адресу 0x20000000 единицу и посмотрите как она отобразилась в области SRAM bit-band alias в режиме отладки:



Санкт-Петербургский
государственный
университет
аэрокосмического
приборостроения