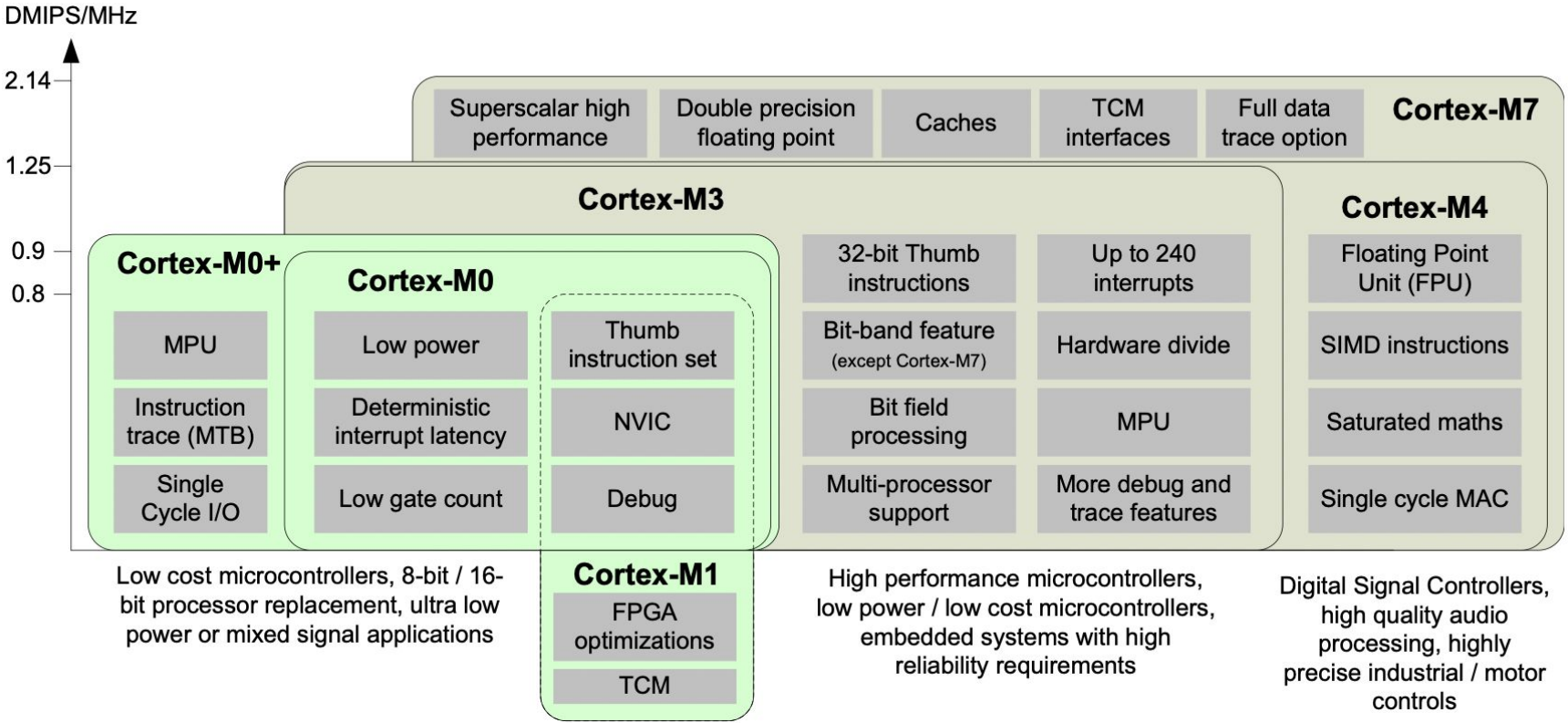
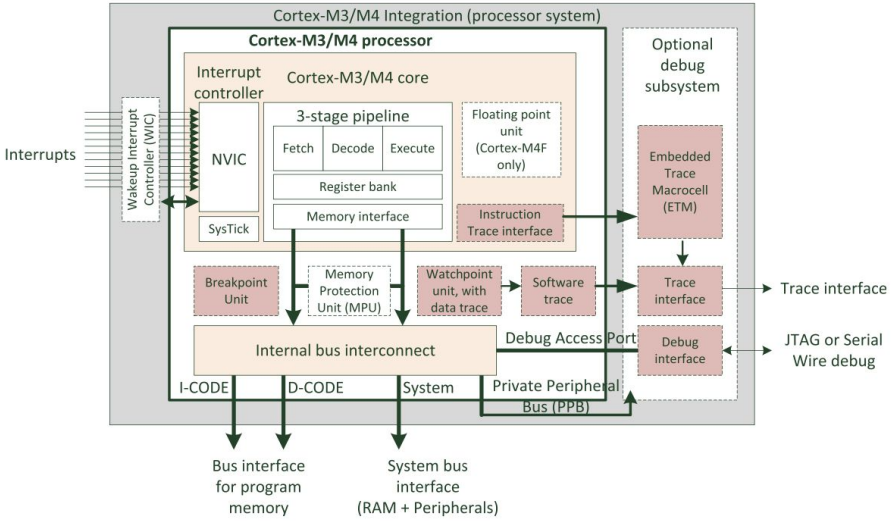
Архитектура ARM Cortex-M и ее сравнение с другими ARM архитектурами

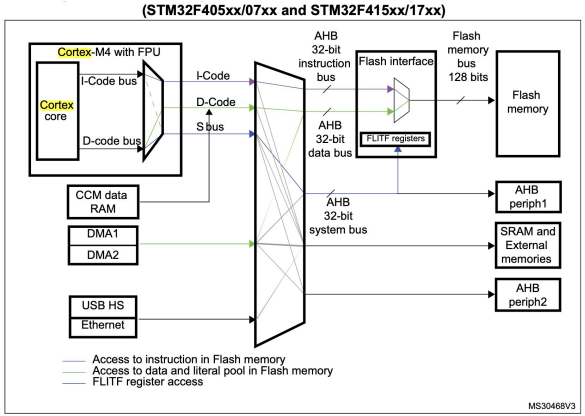
Part 3

- Cache vs TCM memory
- How Cache works
- Cache Coherence Protocols
- NVIC

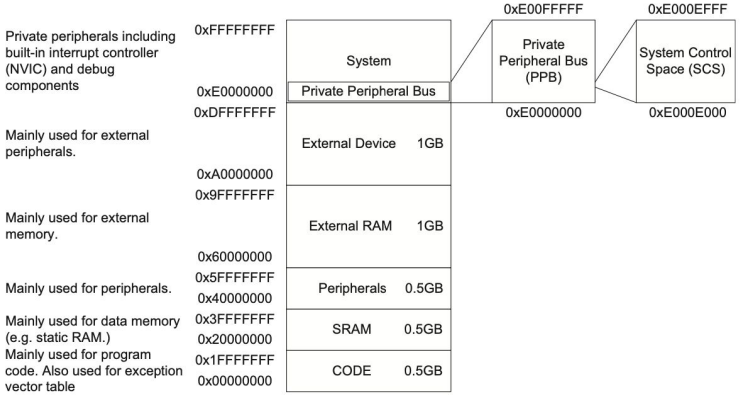

DMIPS/MHz
2.14
1.25
0.9
0.8
Superscalar high performance
Double precision floating point
Caches
TCM interfaces
Full data trace option
Cortex-M7
Cortex-M3
Cortex-M4
Cortex-M0+
Cortex-M0
MPU
Low power
Thumb instruction set
32-bit Thumb instructions
Up to 240 interrupts
Floating Point Unit (FPU)
Instruction trace (MTB)
Deterministic interrupt latency
NVIC
Bit-band feature (except Cortex-M7)
Hardware divide
SIMD instructions
Single Cycle I/O
Low gate count
Debug
Bit field processing
MPU
Saturated maths
Multi-processor support
More debug and trace features
Single cycle MAC
Cortex-M1
FPGA optimizations
TCM
Low cost microcontrollers, 8-bit / 16-bit processor replacement, ultra low power or mixed signal applications
High performance microcontrollers, low power / low cost microcontrollers, embedded systems with high reliability requirements
Digital Signal Controllers, high quality audio processing, highly precise industrial / motor controls

Возвращаясь к Cortex M3

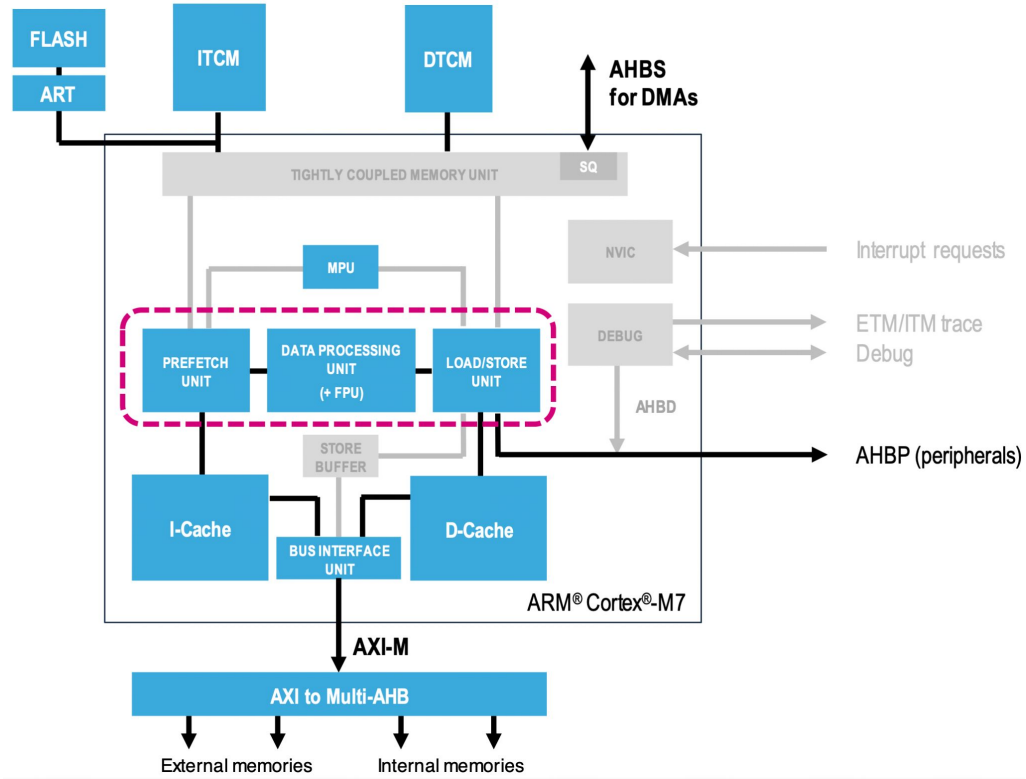




- NVIC - контроллер прерываний(*Nested Vector Interrupt Controller*);
- SysTick - системный таймер;
- MPU - модуль защиты памяти (*Memory Protection Unit*);
- I-CODE - AHB Lite шина для доступа(fetch) к коду 0x0-0x1FFFFFFF;
- D-CODE - AHB Lite шина для доступа к данным 0x0-0x1FFFFFFF;
- System - AHB Lite шина для доступа к RAM памяти 0x20000000-0xFFFFFFFF и периферии;



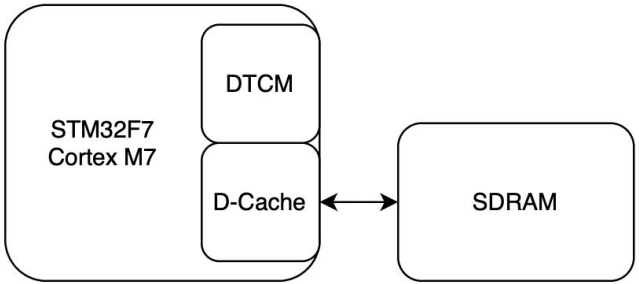
Полная схема Cortex M3



- ITCM/DTCM - Instruction/Data *Tightly-Coupled Memory* - это быстрая память (такая же быстрая как cache), но замапленная по определенным адресам. Обращения к этой памяти не кэшируется т.к. скорость доступа к кэшу = скорости доступа к этой памяти. Код и данные, которые требуют быстрого исполнения, могут быть помещены в эту область памяти (например слинкованы туда через linker скрипт).

- I/D Cache - Instruction/Data Cache;
- ART - Adaptive real-time memory accelerator - специальный cache (не I/D Cache) для flash памяти;

https://developer.arm.com/documentation/ddi0489/f/memory-system/about-the-memory-system




```
uint32_t array_sdram[NUM_SAMPLES]
for (n = 0; n < NUM_SAMPLES; n++)
{
  acc += array1[n];
}

// 2nd Run

for (n = 0; n < NUM_SAMPLES; n++)
{
  acc += array1[n];
}
```

Data placement	D-Cache	Cycles for 500 iterations
DTCM	N/A	1018
SDRAM	Disabled	6604
SDRAM	Enabled 1st run	2954
SDRAM	Enabled 2nd run	1017

Conditions: System clock = 200MHz

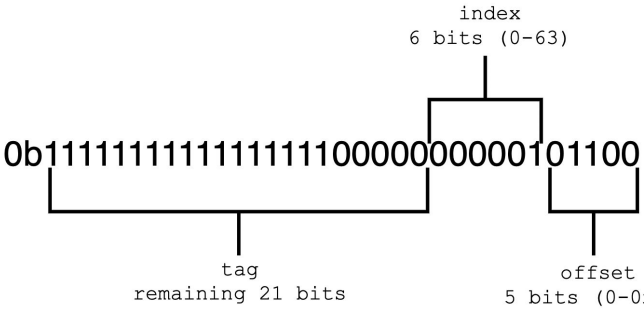
- **array_sdram** еще не находится в D-Cache;
- **array_sdram** уже находится в D-Cache т.к. был подтянут в кэш при первом обращении (1st run);

D-Cache: 4-way set-associative cache, line length = 32-bytes, cache size = 4kb for stm32f746
I-Cache: 2-way set-associative cache, line length = 32-bytes, cache size = 4kb for stm32f746

Example for I-Cache:

Total I-Cache lines = Cache Size / Line Length / Way Number = 4kb / 32b / 2way = 64 lines

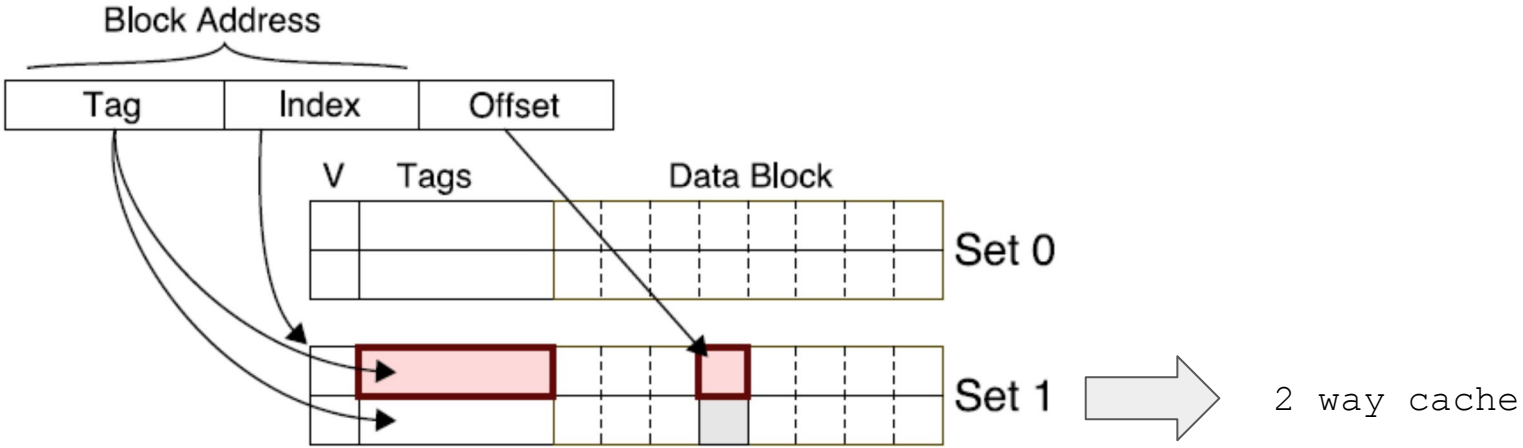
Read instruction from 0x1FFF002C (0b11111111111111111000000000101100)



V (Valid) bit - Данные этой линии уже в кэше

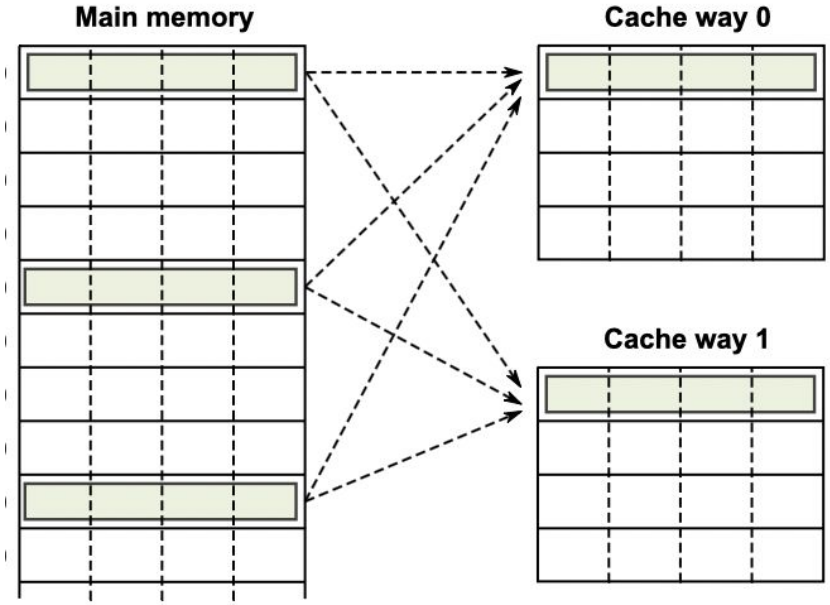
7	6	5	4	3	2	1	0	index	tag	V
0x7FC	0x7F8	0x7F4	0x7F0	0x7EC	0x7E8	0x7E4	0x7E0	[63]		
...	...	...	...	...	...	...	...	...		
0x03C	0x038	0x034	0x030	0x02C	0x028	0x024	0x020	[1]	0b111111111111100000	1
0x01C	0x018	0x014	0x010	0x00C	0x008	0x004	0x000	[0]		

https://blog.feabhas.com/2020/10/introduction-to-the-arm-cortex-m7-cache-part-1-cache-basics/



- Cache сверяет Tag, Index и V(Valid бит) для проверки находится ли данные по Block Address уже в кэше; Если данные находятся в кэше, offset указывает смещение в кэш линии откуда нужно читать данные для Block Address;

https://blog.feabhas.com/2020/10/introduction-to-the-arm-cortex-m7-cache-part-2-cache-replacement-policy/



NOTE: Для определения из какого cache way вытеснять кэш линию, в Cortex M7 используется **pseudo-random** подход.
Другие подходы: Least Recently Used (LRU), FIFO, LIFO, Most recently used (MRU)

Ограниченный размер кэша не позволяет кэшировать всю память сразу, поэтому возникает необходимость вытеснять/заменять кэш линии.

Example:
0x402C, 0x202C, 0x002C имеют один и тот же Index но разные tag. Если у нас 2way cache, последовательное чтение 0x402C, 0x202C, 0x002C:

- 0x402C попадает в **Cache Way 0;**
- 0x202C попадет в **Cache Way 1**, т.к. линия с нужным индексом в **Cache Way 0** занята(уже содержит данные для 0x402C);
- 0x002C - нужная линия в **Cache Way 1** и **Cache Way 0** уже занята. Необходимо вытеснить кэш линию из Cache Way 1 или 2 в память и записать туда новые данные для 0x002C.

Когда подтягивать в кэш данные - только при чтении из памяти ?
Или при записи в память тоже ?

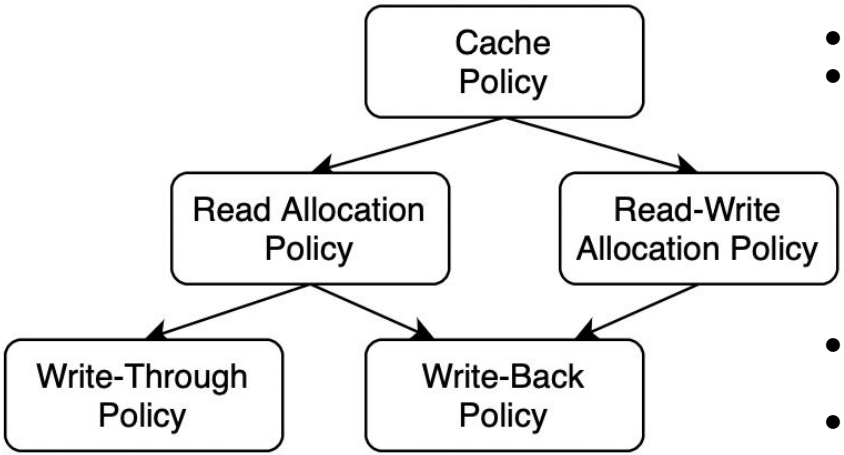
NOTE: Рассматривается пример когда линия в которую идет запись/чтение не находится в кэше (только в памяти) - нужно ли подтягивать ее в кэш ?

Read Allocation Policy	Подтягивает данные из памяти в кэш (аллоцирует кэш линию) только при запросе на чтение (LDR и др.); При запросе на запись - данные записываются сразу в память (кэш линия не аллоцируется);
Read-Write Allocation Policy	Данные подтягиваются в кэш при как при записи так и при чтении.

```
int x = data[0]; // data подтянется в кэш при любой Allocation Policy
buf[0] = y;      // buf подтянется в кэш только при Read-Write Allocation Policy
```

Если нужный адрес уже находится в кэше, как происходит запись в память ?

Write-Back Policy	Запись идет только в кэш. В памяти остается не обновленное значение, до момента сброса кэша в память, которое произойдет только при вытеснении строки кэша, другой строкой.
Write-Through Policy	Запись идет сразу и в память и в кэш.

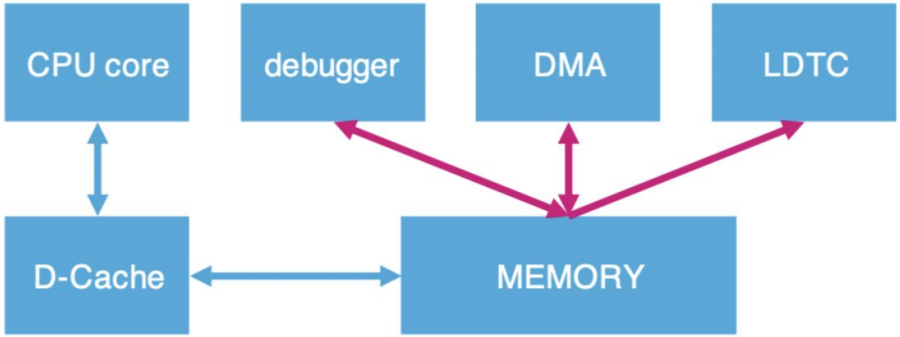


- Плюсы Write-Back: Более быстрые операции записи;
- Минусы Write-Back: Реальная память может находится в несогласованном с кэшом состоянии. Это может приводить к проблемам, например когда вы записали в память некие данные и хотите их скопировать в другую область с помощью DMA (см. след. слайд);

- Плюсы Write-Through: Память всегда согласована с кэшом;
- Минусы Write-Through: Медленные операции записи;

LDTC - LCF Display Controller
DMA - Data Memory Access Controller

Они могут автоматически (без участия процессора) читать/писать память.



Проблема: DMA и LDTC подключены напрямую к памяти и они не видят данные в кэше процессора. Что может привести к следующей несогласованности данных в памяти и кэше:

- CPU подготовил некоторые данные (записал) и хочет чтобы DMA скопировал их, например в периферию или в другую область памяти. Но данные которые подготовил CPU до сих пор находятся в кэше (они не согласованы с памятью), а значит DMA будет копировать неправильные (устаревшие) данные.
  NOTE: Данная проблема существует только для *Write-Back Policy*.

- DMA скопировал некоторые данные в память по адресу *ADDR1*, а CPU хочет прочитать эти данные. НО, CPU имеет в кэше закэшированные данные для адреса *ADDR1* и CPU не знает что данные в памяти обновились чз DMA. А значит CPU считаетнеправильные (устаревшие) данные из кэша.

- **Issue 1**
  CPU подготовил некоторые данные (записал) и хочет чтобы DMA скопировал их, например в периферию или в другую область памяти. Но данные которые подготовил CPU до сих пор находятся в кэше (они не согласованы с памятью), а значит DMA будет копировать неправильные (устаревшие) данные.
  NOTE: Данная проблема существует только для *Write-Back Policy*.

  Решается операцией сброса кэша (Clean Cache) до старта DMA транзакции. Clean Cache принудительно сбрасывает кэш в память.

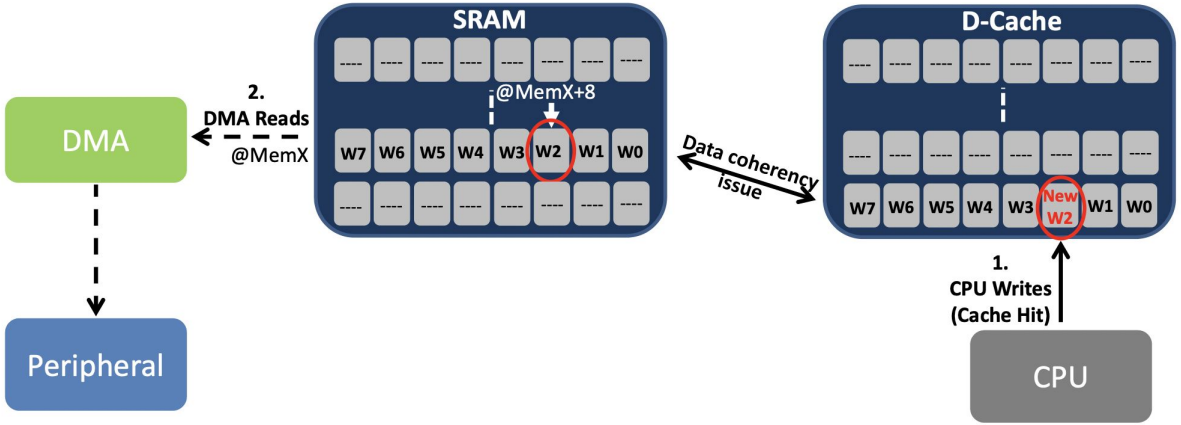
- **Issue 2**
  DMA скопировал некоторые данные в память по адресу *ADDR1*, а CPU хочет прочитать эти данные. НО, CPU имеет в кэше закэшированные данные для адреса *ADDR1* и CPU не знает что данные в памяти обновились чз DMA. А значит CPU считает неправильные (устаревшие) данные из кэша.

  Решается операцией инавлидации кэша (Invalidate Cache) до чтения данных из ADDR1. Invalidate Cache уничтожает кэш - кэш не сбрасывается в память, просто сбрасывается V(Valid) бит в кэше и все последующие операции не будут видеть данные в кэше а будут читать напрямую из памяти.

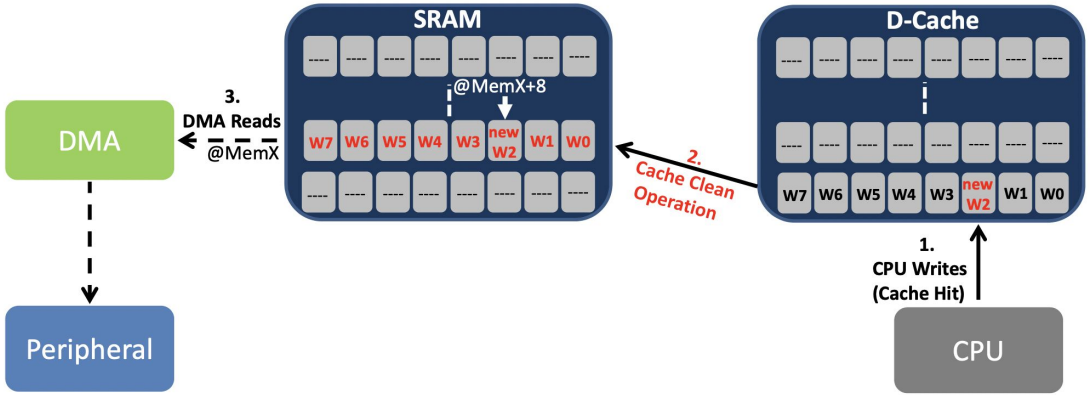
http://ww1.microchip.com/downloads/en/DeviceDoc/Managing-Cache-Coherency-on-Cortex-M7-Based-MCUs-DS90003195A.pdf

ISSUE 1

Without Cache Clean

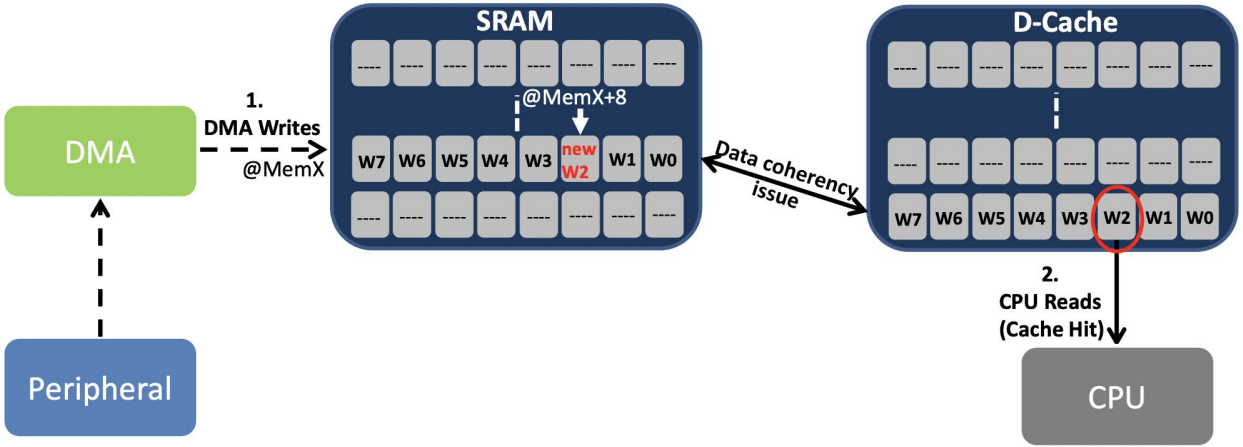


With Cache Clean

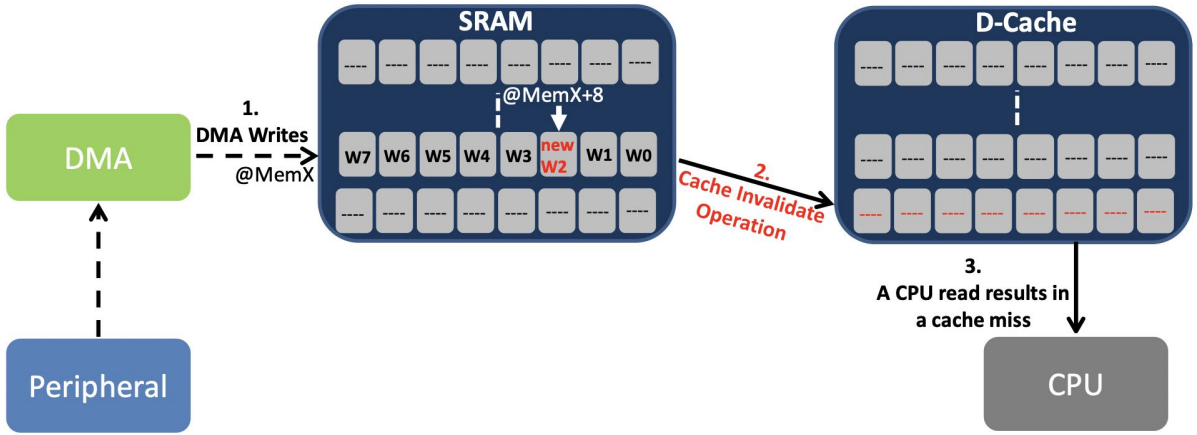


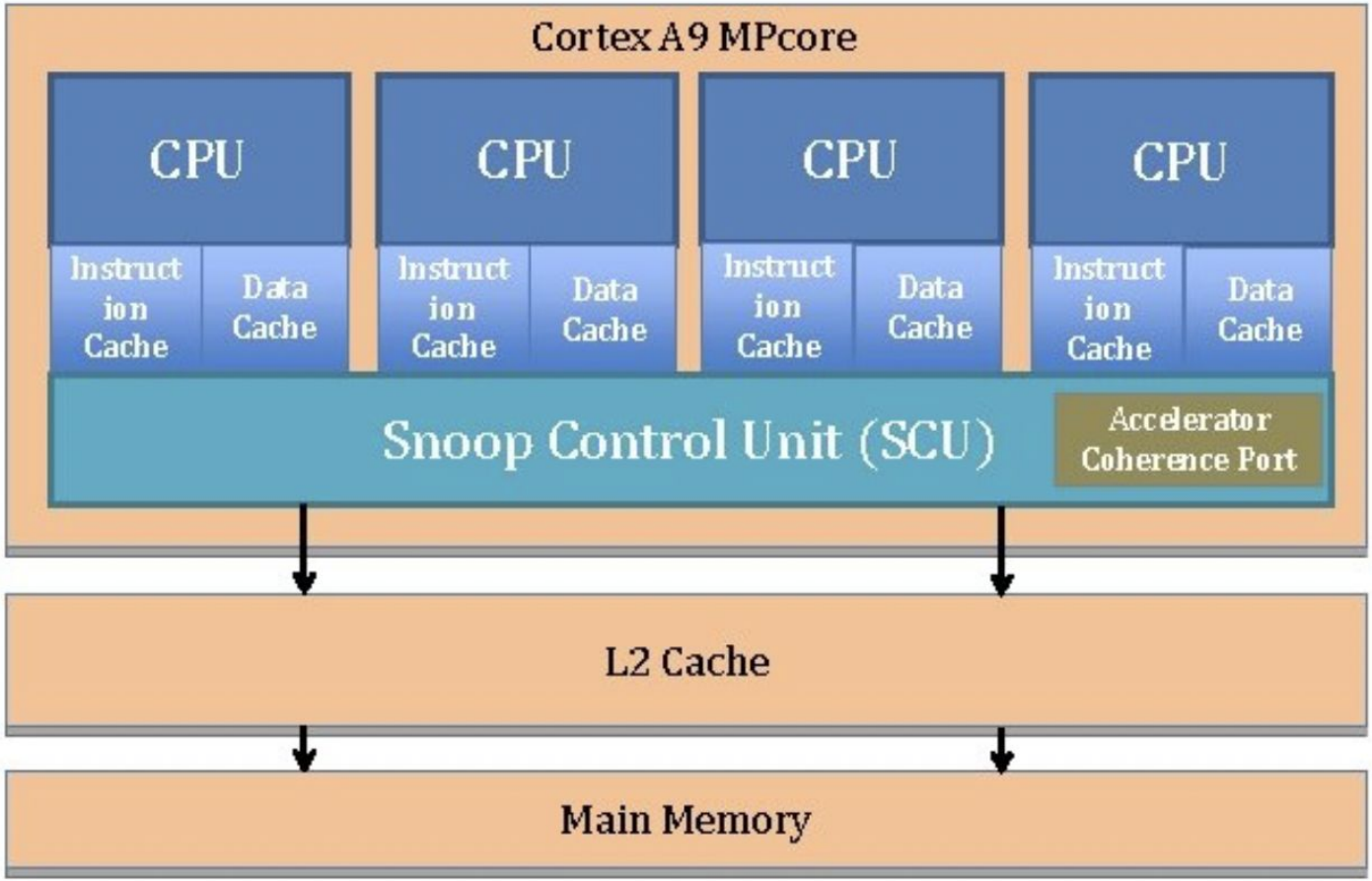
ISSUE 2

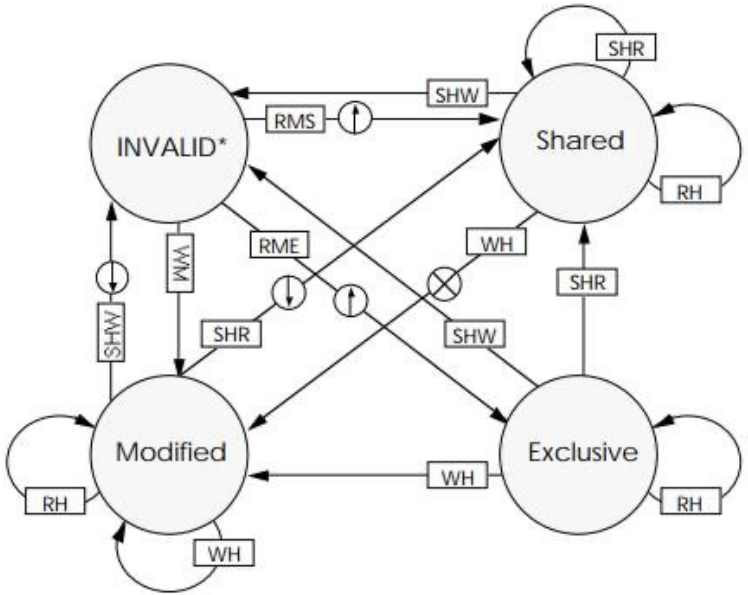
Without Cache Invalidate



With Cache Invalidate



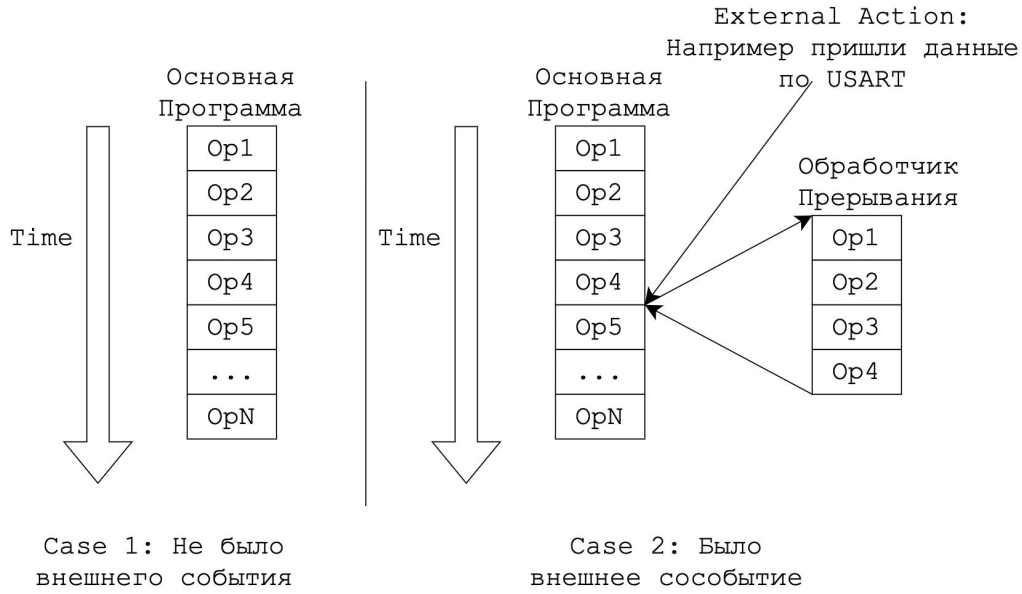

Cortex A9 MPcore
CPU
CPU
CPU
CPU
Instruct ion Cache
Data Cache
Instruct ion Cache
Data Cache
Instruct ion Cache
Data Cache
Instruct ion Cache
Data Cache
Snoop Control Unit (SCU)
Accelerator Coherence Port
L2 Cache
Main Memory

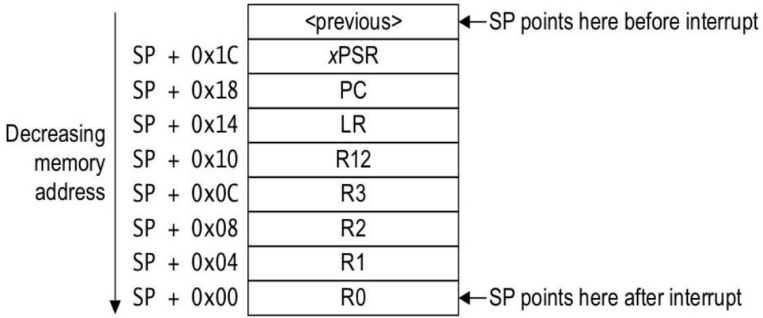


MESI cache coherency protocol

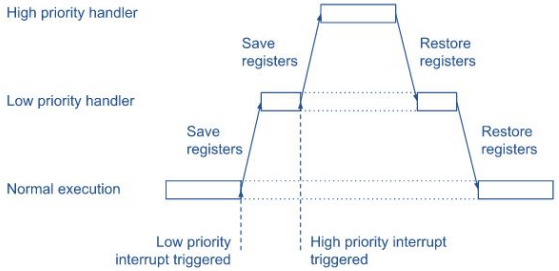
1. **Invalid to Shared.**Read miss and issue read request. If there are other cores hold the block and the state is shared, transfer to shared state.
2. **Invalid to Exclusive.**Read miss and issue read request. If there are no other cores hold the block, transfer to exclusive state.
3. **Invalid to Modified.**Write miss and issue read request. Broadcast message to invalidate exclusive and shared state blocks. May incur dirty write back for other modified block on other core.
4. **Shared to Modified.**Write hit. Broadcast message to invalidate shared state blocks.
5. **Shared to Invalid.**Snoop hit on a write.
6. **Exclusive to Modified.**Write hit. **No need to broadcast any messages.**
7. **Exclusive to Invalid.**Snoop hit on a write.
8. **Modified to Shared.**Snoop hit on a read. Dirty write back and other core read the updated copy and transfer from invalid to shared.
9. **Modified to Invalid.**Snoop hit on a write. Dirty write back and other core read the update copy and transfer from invalid to modified.

https://www.scss.tcd.ie/Jeremy.Jones/VivioJS/caches/MESI.htm





- При прерывании, на текущий стэк (MSP или PSP) сохраняются PC, PSR, R0-R3,R12 и LR регистры. Затем *"вызывается"* обработчик прерывания из таблицы(вектора) прерываний.
- При выходе из обработчика прерывания, PC, PSR, R0-R3,R12 и LR регистры восстанавливаются со стэка.
- Сохранение регистров сделано для того чтобы можно было писать обработчик прерывания на C/C++. PC, PSR, R0-R3,R12 и LR это caller save регистры и обработчик их может поменять, поэтому их нужно восстановить перед выходом из обработчика.



SCB->VTOR

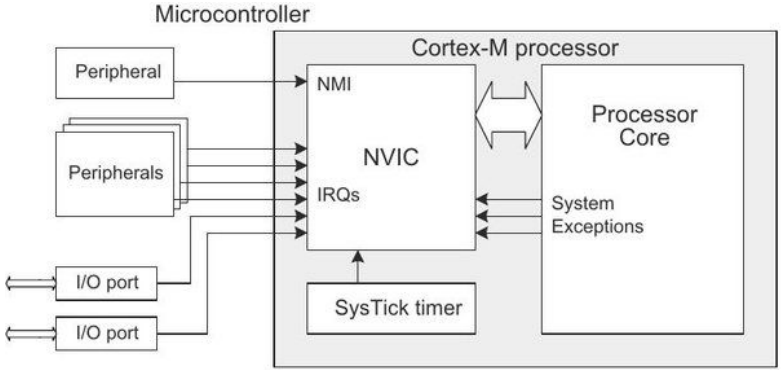
Interrupt Number = HW Interrupt Priority

No.	Exception type	Priority	Description
0	*Stack*	*N/A*	*Initial main stack pointer*
1	Reset	-3 (fixed)	Reset vector
2	NMI	-2 (fixed)	Non mask-able interrupt
3	Hard fault	-1 (fixed)	Hard fault
4	Memmanage fault	Programmable	MPU violation or access to illegal locations
5	Bus fault	Programmable	Bus error
6	Usage fault	Programmable	Program errors like trying to access coprocessor
7-10	Reserved	N/A	Reserved
11	SVC	Programmable	Supervisor call
12	Debug Monitor	Programmable	Break-point, watch-points, external debug requests
13	Reserved	N/A	Reserved
14	PendSV	Programmable	Pendable service call
15	SysTick	Programmable	System Tick timer
16	ExtInt0	Programmable	External interrupt #0
17	ExtInt1	Programmable	External interrupt #1
...	...	...	...
256	Interupt240	Programmable	Interrupt #240

Interrupt Number	Interrupt Handler Address
1	Reset Handler

NMI: if the HSE clock happens to fail, the CSS generates an interrupt, which causes the automatic generation of an NMI.

Hard/Mem/Bus/Usage Fault: https://badembed.ru/hard-fault-memmanage-fault-usage-fault-bus-fault-cortex-m3/

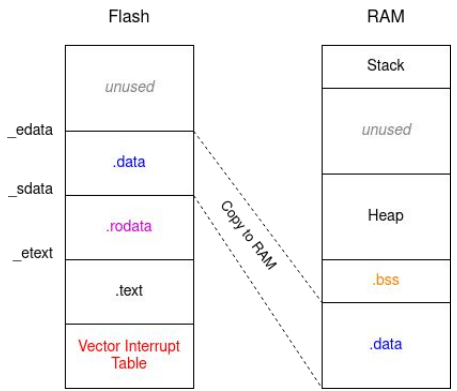


STM32F746NGHx_FLASH.ld:

```
/* Specify the memory areas */
MEMORY
{
RAM (xrw)      : ORIGIN = 0x20000000, LENGTH = 320K
FLASH (rx)      : ORIGIN = 0x8000000, LENGTH = 1024K
}

/* Define output sections */
SECTIONS
{
  /* The startup code goes first into FLASH */
  .isr_vector :
  {
    . = ALIGN(4);
    KEEP(*(.isr_vector)) /* Startup code */
    . = ALIGN(4);
  } >FLASH

  /* The program code and other data goes into FLASH */
  .text :
  {
    . = ALIGN(4);
    *(.text)           /* .text sections (code) */
    *(.text*)          /* .text* sections (code) */
    *(.glue_7)         /* glue arm to thumb code */
    *(.glue_7t)        /* glue thumb to arm code */
    *(.eh_frame)

    KEEP (*(.init))
    KEEP (*(.fini))

    . = ALIGN(4);
    _etext = .;        /* define a global symbols at end of code */
  } >FLASH

  /* used by the startup to initialize data */
  _sidata = LOADADDR(.data);

  /* Initialized data sections goes into RAM, load LMA copy after code */
  .data :
  {
    . = ALIGN(4);
    _sdata = .;        /* create a global symbol at data start */
    *(.data)           /* .data sections */
    *(.data*)          /* .data* sections */

    . = ALIGN(4);
    _edata = .;        /* define a global symbol at data end */
  } >RAM AT> FLASH


  /* Uninitialized data section */
  . = ALIGN(4);
  .bss :
  {
    /* This is used by the startup in order to initialize the .bss secion */
    _sbss = .;         /* define a global symbol at bss start */
    __bss_start__ = _sbss;
    *(.bss)
    *(.bss*)
    *(COMMON)

    . = ALIGN(4);
    _ebss = .;         /* define a global symbol at bss end */
    __bss_end__ = _ebss;
  } >RAM
}
```

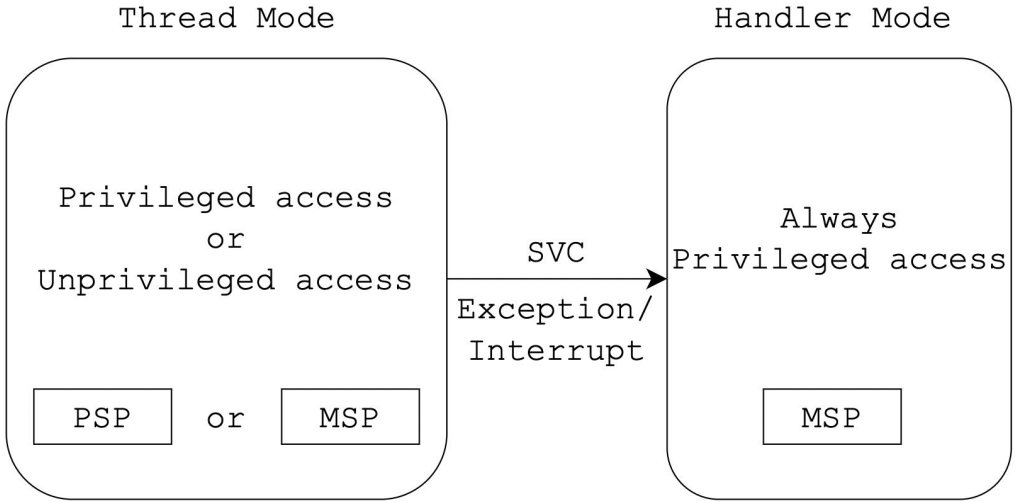
startup_stm32f746xx.s:

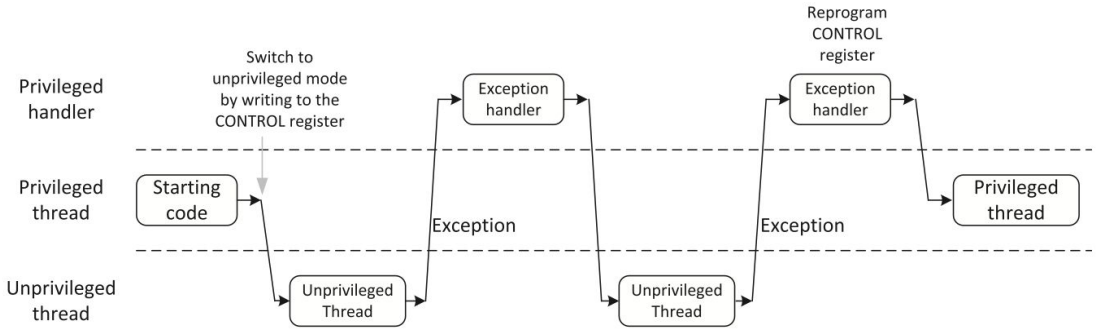
```
/*******************************************************************************
*
* The minimal vector table for a Cortex M7. Note that the proper constructs
* must be placed on this to ensure that it ends up at physical address
* 0x0000.0000.
*
*******************************************************************************/
  .section  .isr_vector,"a",%progbits
.type  g_pfnVectors, %object
.size  g_pfnVectors, .-g_pfnVectors


g_pfnVectors:
.word  _estack
.word  Reset_Handler

.word  NMI_Handler
.word  HardFault_Handler
.word  MemManage_Handler
.word  BusFault_Handler
.word  UsageFault_Handler
.word  0
.word  0
.word  0
.word  0
.word  SVC_Handler
.word  DebugMon_Handler
.word  0
.word  PendSV_Handler
.word  SysTick_Handler
```




```
/**
 * @brief  This is the code that gets called when the processor first
 *          starts execution following a reset event. Only the absolutely
 *          necessary set is performed, after which the application
 *          supplied main() routine is called.
 * @param  None
 * @retval : None
*/

    .section  .text.Reset_Handler
  .weak  Reset_Handler
  .type  Reset_Handler, %function
Reset_Handler:
  ldr   sp, =_estack      /* set stack pointer */

/* Copy the data segment initializers from flash to SRAM */
  movs  r1, #0
  b  LoopCopyDataInit

CopyDataInit:
  ldr  r3, =_sidata
  ldr  r3, [r3, r1]
  str  r3, [r0, r1]
  adds  r1, r1, #4

LoopCopyDataInit:
  ldr  r0, =_sdata
  ldr  r3, =_edata
  adds  r2, r0, r1
  cmp  r2, r3
  bcc  CopyDataInit
  ldr  r2, =_sbss
  b  LoopFillZerobss
/* Zero fill the bss segment. */
FillZerobss:
  movs  r3, #0
  str  r3, [r2], #4

LoopFillZerobss:
  ldr  r3, = _ebss
  cmp  r2, r3
  bcc  FillZerobss

/* Call the clock system initialization function.*/
  bl  SystemInit
/* Call static constructors */
    bl __libc_init_array
/* Call the application's entry point.*/
  bl  main
  bx  lr
.size  Reset_Handler, .-Reset_Handler
```

https://developer.arm.com/documentation/dui0774/i/armclang-Integrated-Assembler-Directives/Section-directives





2 Режима:

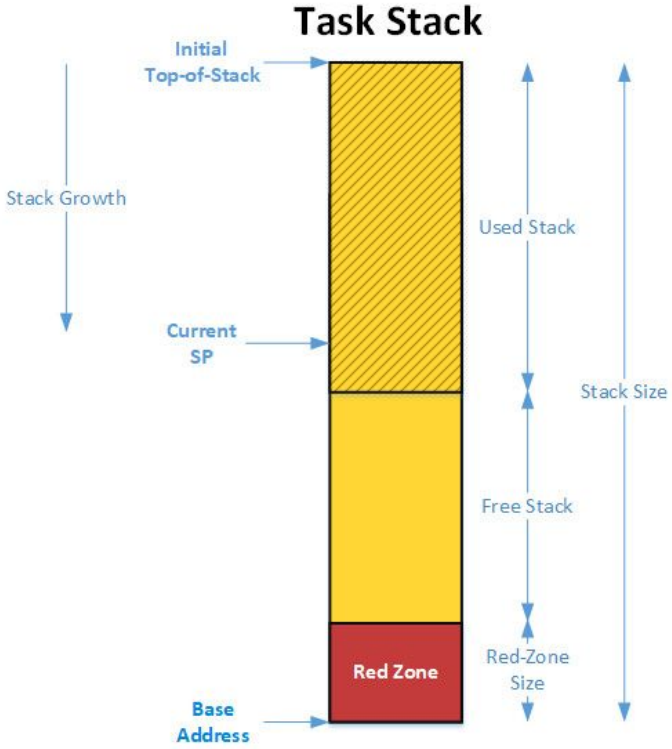
- Handler Mode:
  Вход по прерыванию/exception;
  В обработчике прерывания/exception всегда Thread Mode;
  Всегда имеет привилегированный доступ;
  Всегда использует MSP (Main Stack Pointer);
- Thread Mode:
  Основной режим работы процессора;
  Может использовать MSP или PSP (Process Stack Pointer);
  При старте использует MSP;
  Может иметь или не иметь привилегированный доступ;
  Стартует с привилегированным доступом;
  Переключение привилегированного доступа возможно только в Handler Mode;

Unprivileged

- Has limited access to the MSR and MRS instructions, and cannot use the CPS instruction;
- Cannot access the system timer, NVIC, or system control block;
- Might have restricted access to memory(MPU) or peripherals;

R0-R12	RW	Either
MSP	RW	Privileged
PSP	RW	Either
LR	RW	Either
PC	RW	Either
PSR	RW	Privileged
ASPR	RW	Either
IPSR	RO	Privileged
EPSR	RO	Privileged
PRIMASK	RW	Privileged
FAULTMASK	RW	Privileged
BASEPRI	RW	Privileged
CONTROL	RW	Privileged

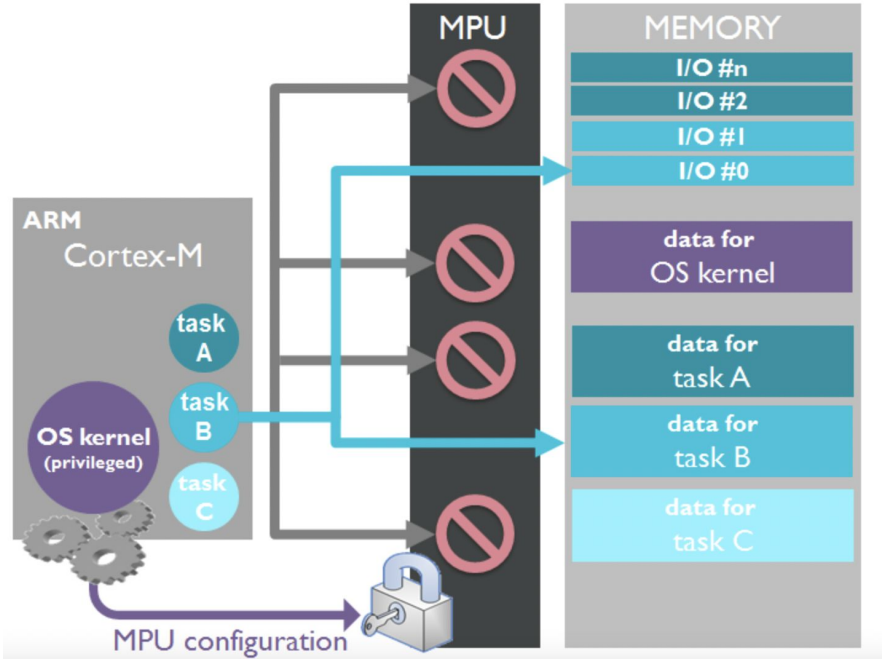
https://developer.arm.com/documentation/dui0552/a/the-cortex-m3-processor/programmers-model/processor-mode-and-privilege-levels-for-software-execution



A **Memory Protection Unit** (MPU) is hardware that limits the access to memory and peripheral devices to only the code that needs to access those resources. If a task attempts to access a memory location or a peripheral device outside of its allotted space, then a CPU exception is triggered, and depending on the application, corrective actions must be taken.

NOTE: Cortex M33 (armv8-m) может автоматически проверять переполнение стэка без использования MPU - stack limit registers (*MSPLIM_S* and *PSPLIM_S*). При переполнение стэка - usage fault exception.

https://www.embeddedcomputing.com/application/industrial/industrial-computing/using-a-memory-protection-unit-with-an-rtos



Limitation:

- До 16 регионов;
- Размер региона должен быть кратен степени 2;
- Адрес региона должен быть выровнен по размеру данного региона. Например, если размер региона 64KB (0x00010000), то адрес должен может быть 0x00010000, 0x00020000 и т.д.

NOTE: Cortex M33 (armv8-m) не имеет ограничений на размер и адрес региона.