EMBSYS 110, Spring 2017 Week 8

- Week 8
  - DMA
  - Bulk Transfer
  - Cache
  - Non-blocking vs blocking.
  - Others.

Speed - DMA

- DMA is very efficient in transferring a large amount of data between memory and peripheral (via data register).
- It can work with I2C, SPI, UART or GPIO.
- It can work in both directions (to and from peripherals).
- There is an overhead to set up each transfer, so it may not be efficient to send a small amount of data.

  For the example of UART output, take a look at:

```
HAL_UART_Transmit_DMA( );
   HAL_DMA_Start_IT( );
      DMA_SetConfig( );
```

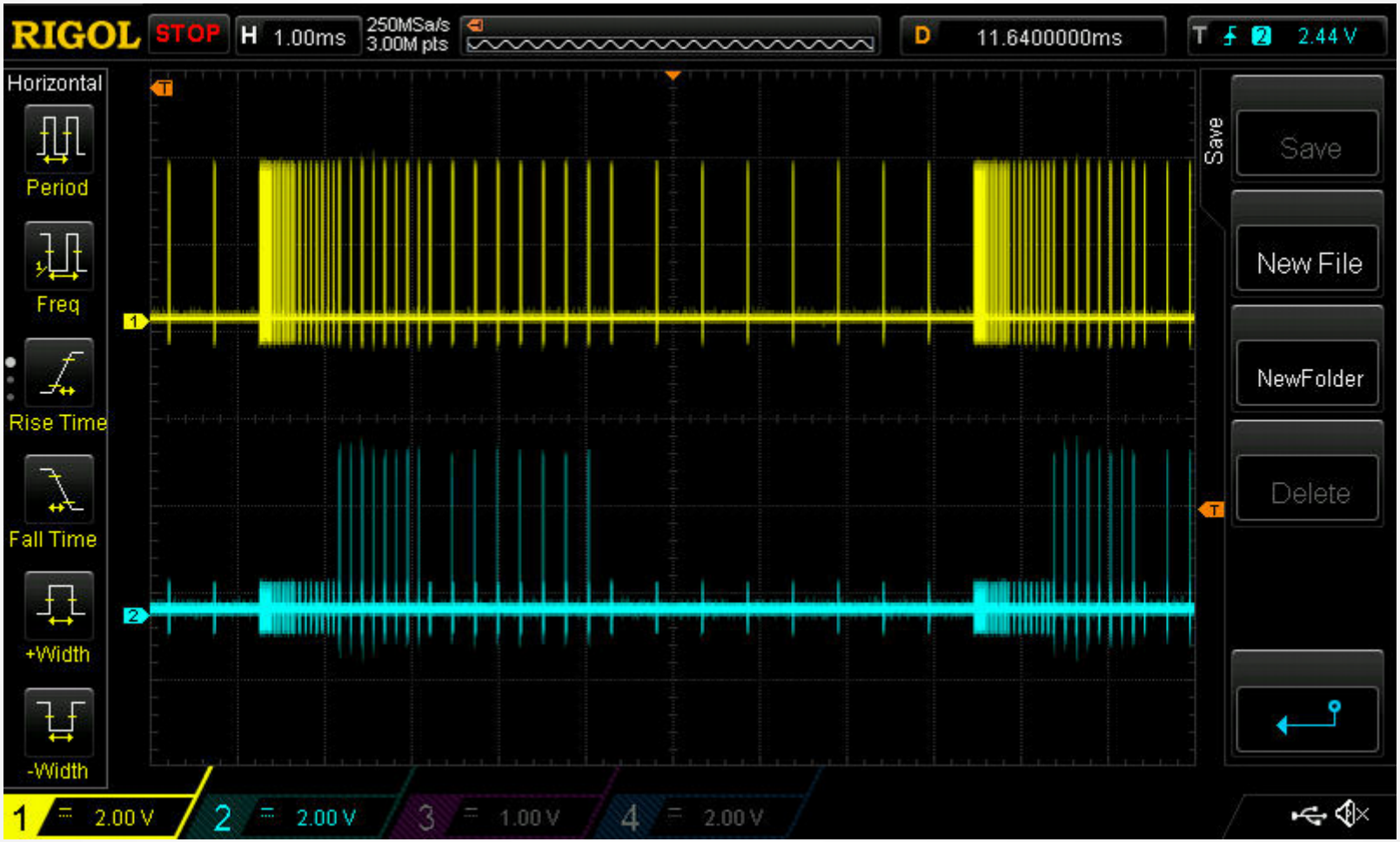
Speed - DMA

- It can be triggered by timer interrupt to achieve precise timing when writing to GPIO.
  - For example, we can use DMA to trigger writing to a GPIO port on every risinig/falling edge of a PWM timer.
  - PWM is used a clock signal and the GPIO port outputs data synchronously to the peripheral. This is how the LED matrix panel works.

Speed - DMA

Channel 1 – Clock signal generated by PWM
Channel 2 – Blue color signal output by DMA triggered by falling edge of clock.

Speed - DMA

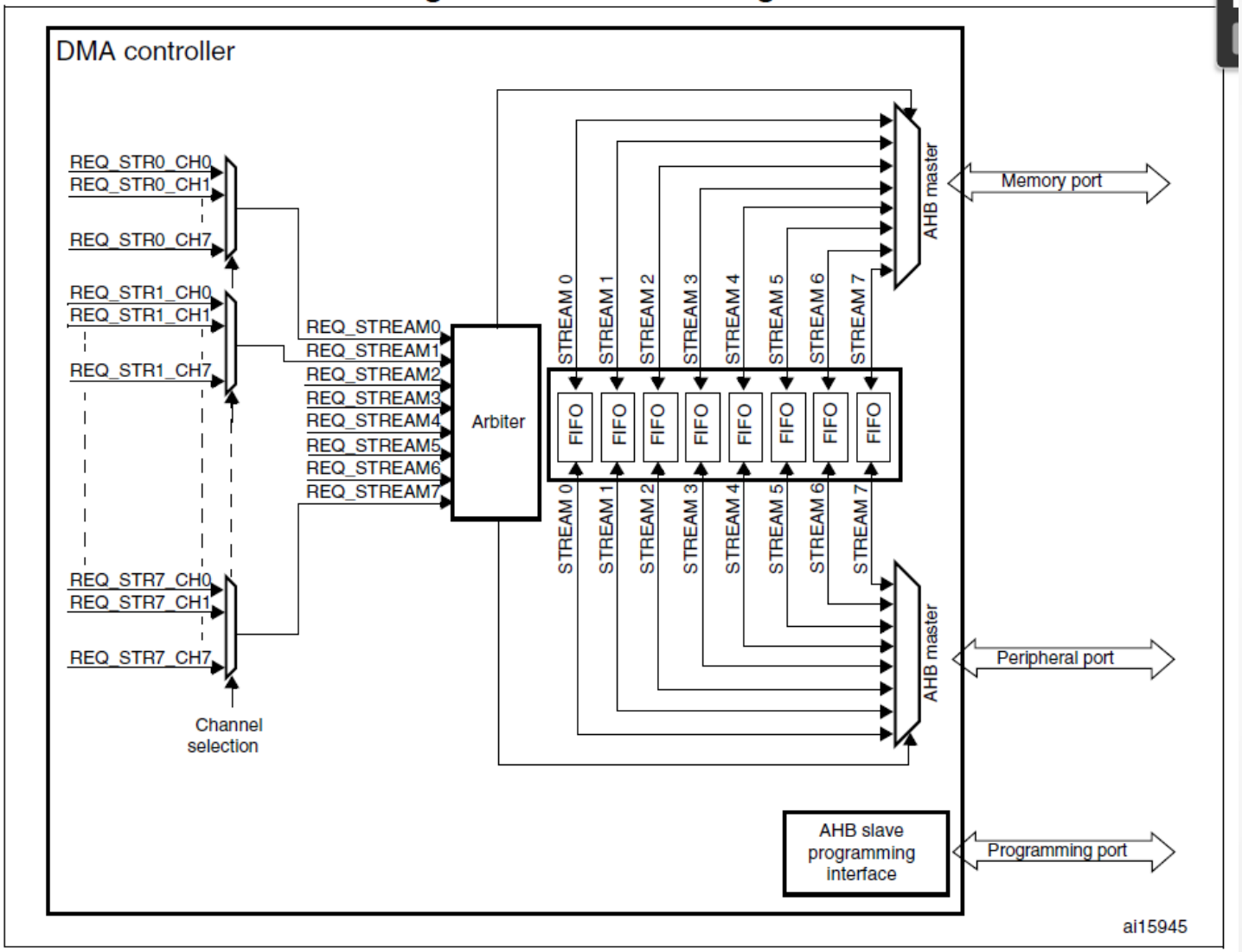


Speed - DMA

- For UART (or SPI or I2C) output, DMA is triggered when the data register is available for writing.
  - This is much more efficient than triggering an interrupt each time when the transmit buffer is empty.
  - Note that STM32F4 does not have data FIFO for UART, which makes DMA essential. If a deep FIFO is available, one may get by without DMA.
- For UART (or SPI or I2C) input, DMA is triggered when the receive buffer is not empty.
  - This is much more efficient than triggering an interrupt to copy each byte in ISR.
  - In earlier demo project, we only detect single character so DMA was not used.
  - A tricky point. When setting up RX DMA, we need to tell HW how many bytes are to be received. For UART, there is a problem. How do we solve it?

Speed - DMA

Figure 22. DMA block diagram



Speed - DMA

- See STM32F401 Reference Manual Section 9.3.3 for DMA channel/stream selection.

Table 28. DMA1 request mapping (STM32F401xB/C and STM32F401xD/E)

Peripheral requests	Stream 0	Stream 1	Stream 2	Stream 3	Stream 4	Stream 5	Stream 6	Stream 7
Channel 0	SPI3_RX		SPI3_RX	SPI2_RX	SPI2_TX	SPI3_TX		SPI3_TX
Channel 1	I2C1_RX	I2C3_RX				I2C1_RX	I2C1_TX	I2C1_TX
Channel 2	TIM4_CH1		I2S3_EXT_RX	TIM4_CH2	I2S2_EXT_TX	I2S3_EXT_TX	TIM4_UP	TIM4_CH3
Channel 3	I2S3_EXT_RX	TIM2_UP TIM2_CH3	I2C3_RX	I2S2_EXT_RX	I2C3_TX	TIM2_CH1	TIM2_CH2 TIM2_CH4	TIM2_UP TIM2_CH4
Channel 4						USART2_RX	USART2_TX	
Channel 5			TIM3_CH4 TIM3_UP		TIM3_CH1 TIM3_TRIG	TIM3_CH2		TIM3_CH3
Channel 6	TIM5_CH3 TIM5_UP	TIM5_CH4 TIM5_TRIG	TIM5_CH1	TIM5_CH4 TIM5_TRIG	TIM5_CH2	I2C3_TX	TIM5_UP	
Channel 7			I2C2_RX	I2C2_RX				I2C2_TX

Speed - DMA

- See demo project UartOut for UART TX DMA. Key points:
  - DMA transfers data out of FIFO directly. No extra copy.
  - STM32Cube driver (stm32f4xx_hal_uart.c) is a good place to start, but may not be optimized.
    - The original driver always enables the DMA half complete interrupt which we don't care here. (We would need it for the RX path. Why?)
    - It also triggers the UART TX Complete interrupt upon DMA complete interrupt. (To share common code.)
    - The result is for every DMA TX completion, 3 interrupts are generated.

Speed – Bulk Transfer

- The second optimization technique for speed is to use bulk transfer as much as possible.
- A small number of large transfer is more efficient than a larger number of small transfer.
- We are familiar with this: Reading multiple bytes in a single fread() is more efficient han reading one byte at a time.
- Similarly, when reading data from the ST IMU (LSM6DS0.pdf), it is much more efficient to read all available samples in a single I2C or SPI transfer.

Speed – Bulk Transfer

- See Section 3.3 Multiple reads (burst) (page 20) of LSM6DS0.pdf.
- When using high ODR, enable FIFO continuous mode.
- Note how the ST hardware automatically wrap around the register address to enable reading out multiple sets of gyro XYZ and accel XYZ in a single transfer.
- Recall overhead of I2C read:

  device address, register address write, data read…
- For single byte read, 3X of data is transferred.
- SPI is better at the expense of additional wires.

Speed – Other techniques

- RTOS is a good source to get ideas since they are highly optimized.
- How does QP find the highest priority ready task to run?
  - On Cortex-M3 or above, use CLZ instruction (count leading zeros) for fast LOG2 (see qpset.h or qf_port.h):

    ```
    #define QF_LOG2(n_) ((uint_fast8_t)(32U - __CLZ(n_)))
    ```

  - Otherwise, use table-lookup (see QF_log2Lkup[ ] in qf_act.cpp). (Tradeoff between speed and code size).
- In fw_pipe.h, why does it force size to order of 2.
- Check out handy macros in fw_macro.h

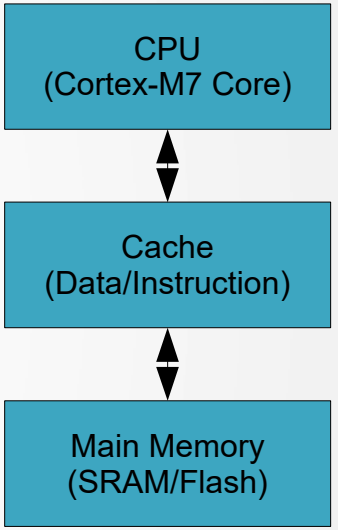
Speed - Cache

- Cortex-M0/M3/M4 cores do not have cache. We don’t need to worry about cache with our Nucleo board.
- Cortex-M7 (e.g. STM32F769I-Nucleo board) has cache.
- See STM32F769 Programmer's Manual page 30 Section 2.2 (next page).
  - STM32F769 has 16K Data Cache and 16K Instruction Cache.
- Cache is very fast memory located between processor and main memory (e.g. internal Flash/SRAM, or external memory).
- It makes use of temporal locality principle, i.e. memory location accessed is likely to be accessed again in near future (next page).

Speed - Cache

Table 12. STM32F76xxx/STM32F77xxx Cortex®-M7 configuration

Features	STM32F76xxx/STM32F77xxx
Floating Point Unit	Double and single precision floating point unit
MPU	8 regions
Instruction TCM size	Flash TCM: 2 Mbytes RAM ITCM: 16 Kbytes
Data TCM size	128 Kbytes
Instruction cache size	16 Kbytes
Data cache size	16 Kbytes
Cache ECC	Not implemented
Interrupt priority levels	16 priority levels
Number of IRQ	110
WIC, CTI	Not implemented
Debug	JTAG & Serial-Wire Debug Ports 8 breakpoints and 4 watchpoints.
ITM support	Data Trace (DWT), and instrumentation trace (ITM)
ETM support	Instruction Trace interface



Speed - Cache

- Cache memory is partitioned into slots, each called a *cache line.*
- In STM32F769, cache line size is 8 words, i.e. 32-bytes. It makes use of spatial locality principle, i.e. nearby memory location is likely to be accessed at the same time. Why not make a cache line too large?
- See STM32F769 Programmer's Manual page 219 Section 4.5.3. In Table 79, see the field LineSize. (next page)

  Note line size is 2 ^(LineSize + 2). If LineSize is 1, line size is 8 words.
- In Table 80, what do the terms NumSets and Associativity mean?

  Note values shown are one less than actual value.

  If Associativity shows 0x3, it means “4-way set associative mapping” (explained later.)

Speed - Cache

Table 80. CCSIDR encodings

CSSELR	Cache	Size	Complete register encoding	Register bit field encoding						
				WT	WB	RA	WA	NumSets	Associativity	LineSize
0x0	Data cache	4 Kbytes	0xF003E019	1	1	1	1	0x001F	0x3	0x1
		8 Kbytes	0xF007E019					0x003F		
		16 Kbytes	0xF00FE019					0x007F		
		32 Kbytes	0xF01FE019					0x00FF		
		64 Kbytes	0xF03FE019					0x01FF		
0x1	Instruction cache	4 Kbytes	0xF007E009	1	1	1	1	0x003F	0x1	0x1
		8 Kbytes	0xF00FE009					0x007F		
		16 Kbytes	0xF01FE009					0x00FF		
		32 Kbytes	0xF03FE009					0x01FF		
		64 Kbytes	0xF07FE009					0x03FF		

Speed - Cache

Tag	Cache Line (32 bytes)
	cache line 0
	cache line 1
	cache line 2
	cache line 3
	cache line 4
	cache line 5
	cache line 6
	cache line 7
	...
	...
	cache line N-1

Speed - Cache

- Since cache size is much smaller than main memory size, we need an algorithm to map from a main memory block to cache line.
- There are three algorithms:
  - Direct mapping.
  - Associative mapping.
  - Set associative mapping.

Speed - Cache

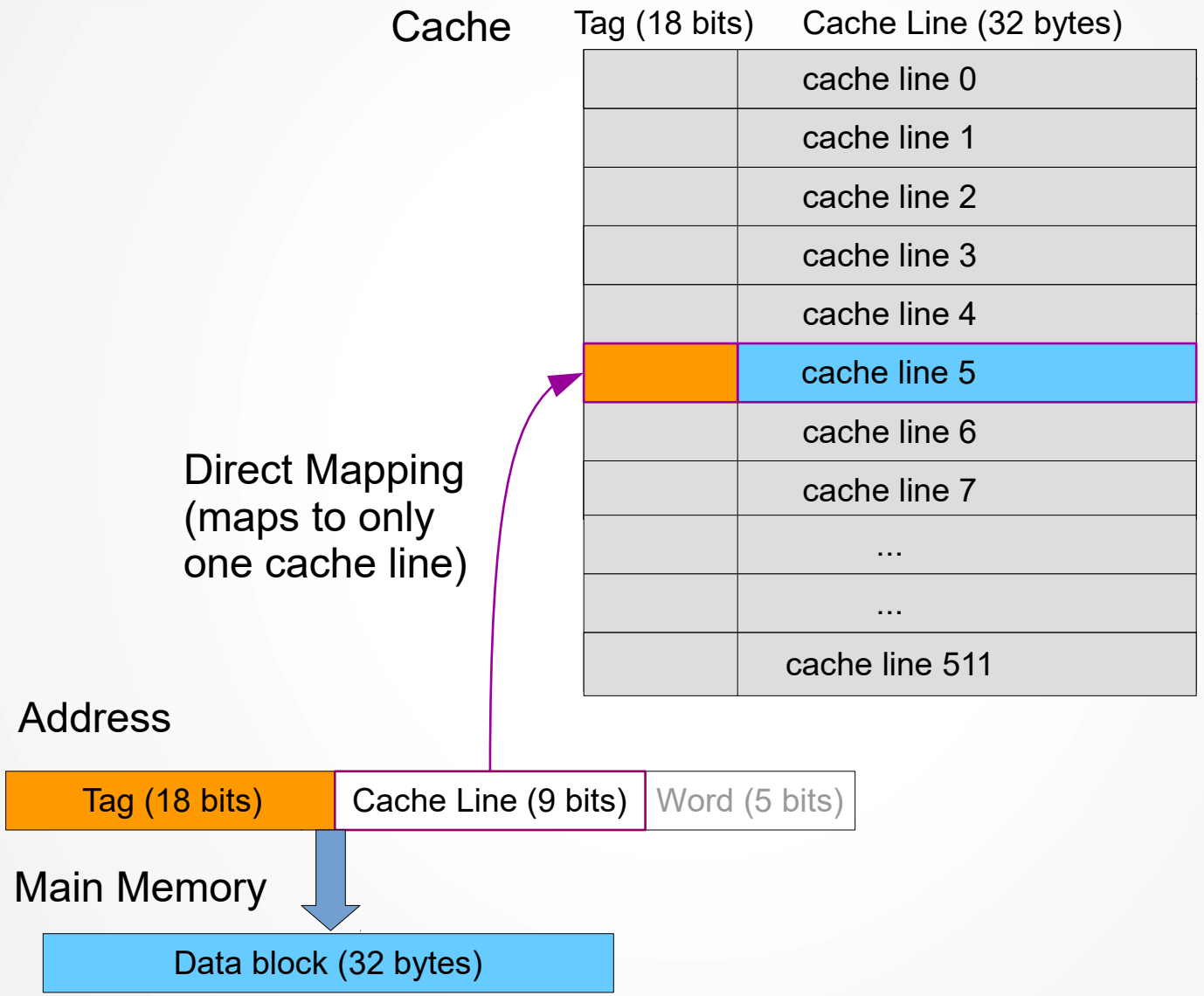
- Direct mapping – Each main memory block is mapped to a single cache line.
  - We divide a 32-bit address into 3 parts:

    Memory address = Tag | Cache Line | Word
  - Since each cache line is 32 bytes, the *Word* part needs 5 bits (2^5 = 32).
  - Since there are 512 cache lines (in STM32F769) (16KB / 32B), the *Cache Line* part needs 9 bits (2^9 = 512).
  - The *Tag* part will have the remaining 18 bits (32 – 5 – 9).

Speed - Cache

- *Tag* is stored along with each cache line so it knows which main memory block is currently stored in a cache line (many-to-1 mapping).
- *Cache Line* (slot) selects a cache line to use.
- *Word* identifies the word within a cache line being addressed.
- Direct mapping is simple to implement.
- However it suffers a low hit-to-miss ratio. (*Hit* is when a memory block is found in the cache. *Miss* is when it is not found.)

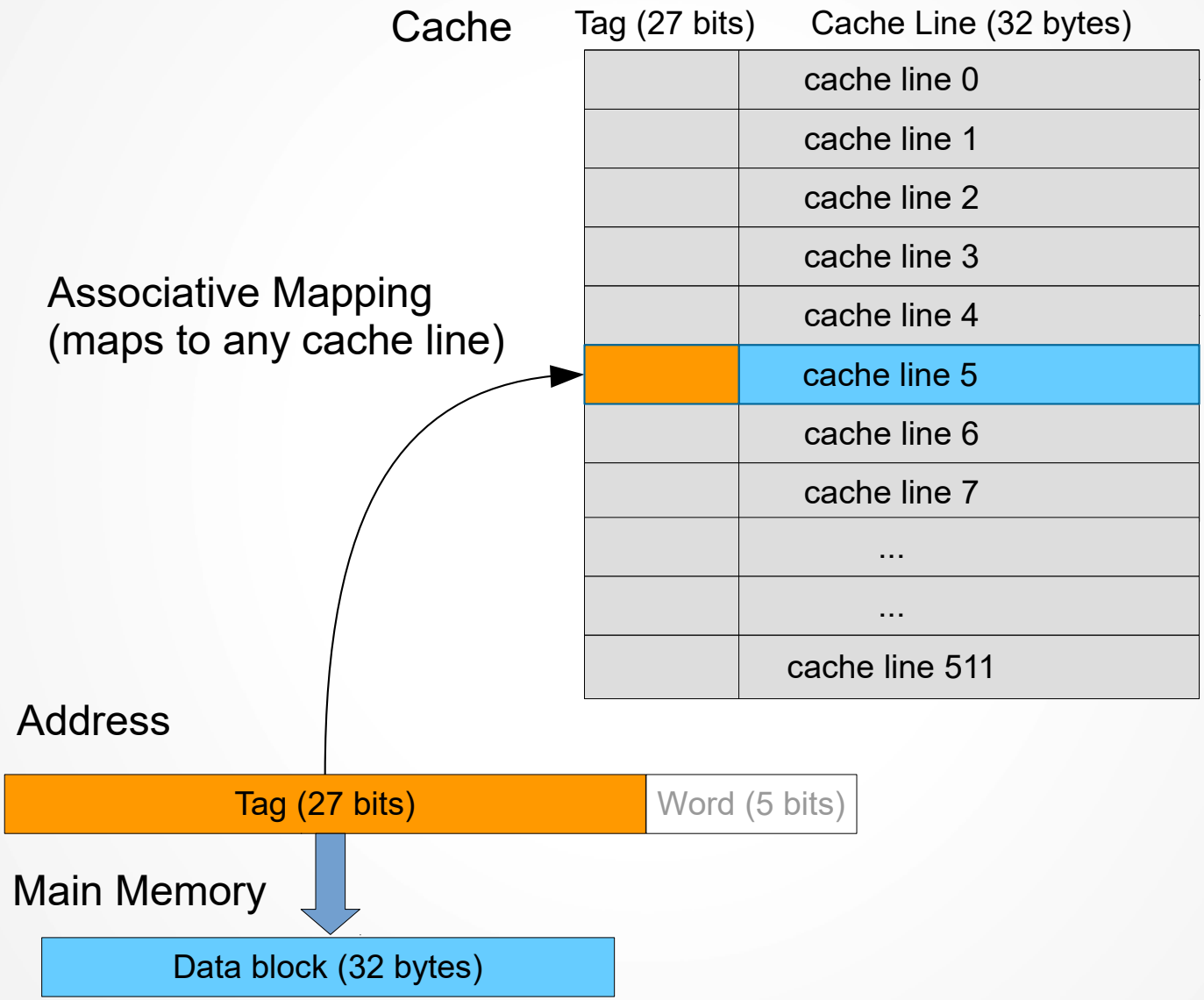
Speed - Cache



Speed - Cache

- Associative mapping – Each main memory block can be mapped to any cache lines.
  - Memory address = Tag | Word
  - *Tag* is stored along with each cache line so it knows which main memory block is currently stored in a cache line (many-to-1 mapping).
  - *Word* identifies the word within a cache line being addressed.
  - Most flexible, since any cache line can be used to hold a main memory block. Maximum hit-to-miss ratio.
  - However it is expensive to implement, as the hardware logic needs to examine tags of all cache lines in parallel.

Speed - Cache



Speed - Cache

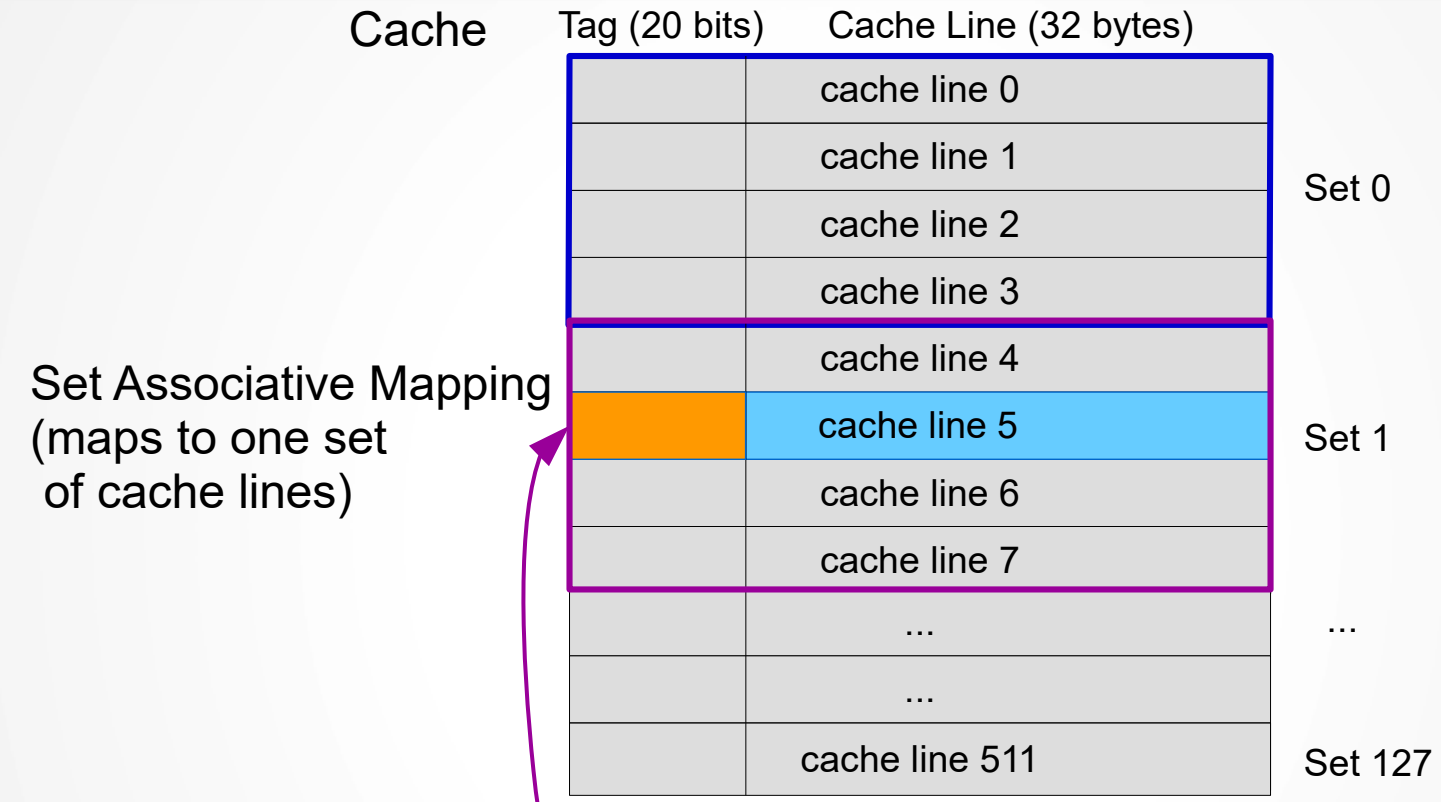
- Set associative mapping – Compromise between direct and associative mapping. Each main memory block is mapped to a set of cache lines, i.e. it can be stored in *any* one of the cache lines in the set it is mapped to.
- Memory address = Tag | Cache Set | Word
  - *Cache Set* is the set number the memory block is mapped to.
  - *Tag* is stored along with each cache line so it knows which main memory block is currently stored in a cache line (many-to-1 mapping).
  - *Word* identifies the word within a cache line being addressed.

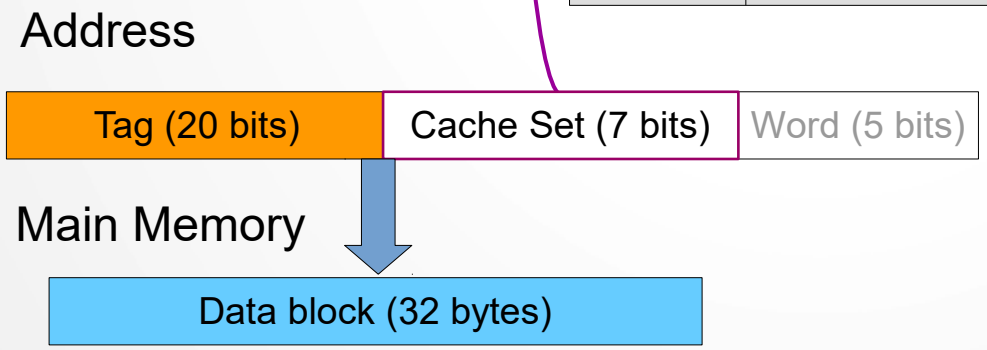
Speed - Cache

- The number of cache lines in each set is referred to as *ways*. A 2-way set associative mapping means that there are two cache lines in each set.
- Studies show 2-way set associative mapping yields significant improvement of hit ratio over direct mapping. 4-way mapping yields modest improvement.

  (Stallings, William. Computer Organization and Architecture. 4th Edition. 1996. Prentice Hall)

Speed - Cache





Speed - Cache

- What is the additional cost of set associative over direct mapping?
- Refer back to STM32F769 Programmer's Manual page 220 Section 4.5.3 Table 80.
- We now understand what it means by *NumSets* and *Associativity* along with *LineSize.*
- Let’s verify if the numbers match up:

  For STM32F769, the cache is 4-way set associative (0x3 + 1), with 128 sets (0x7F + 1). Each cache line has 8 words (2^(0x1 + 2)), i.e. 32 bytes.
- The total cache size is 4 * 128 * 32 = 16KB (correct).

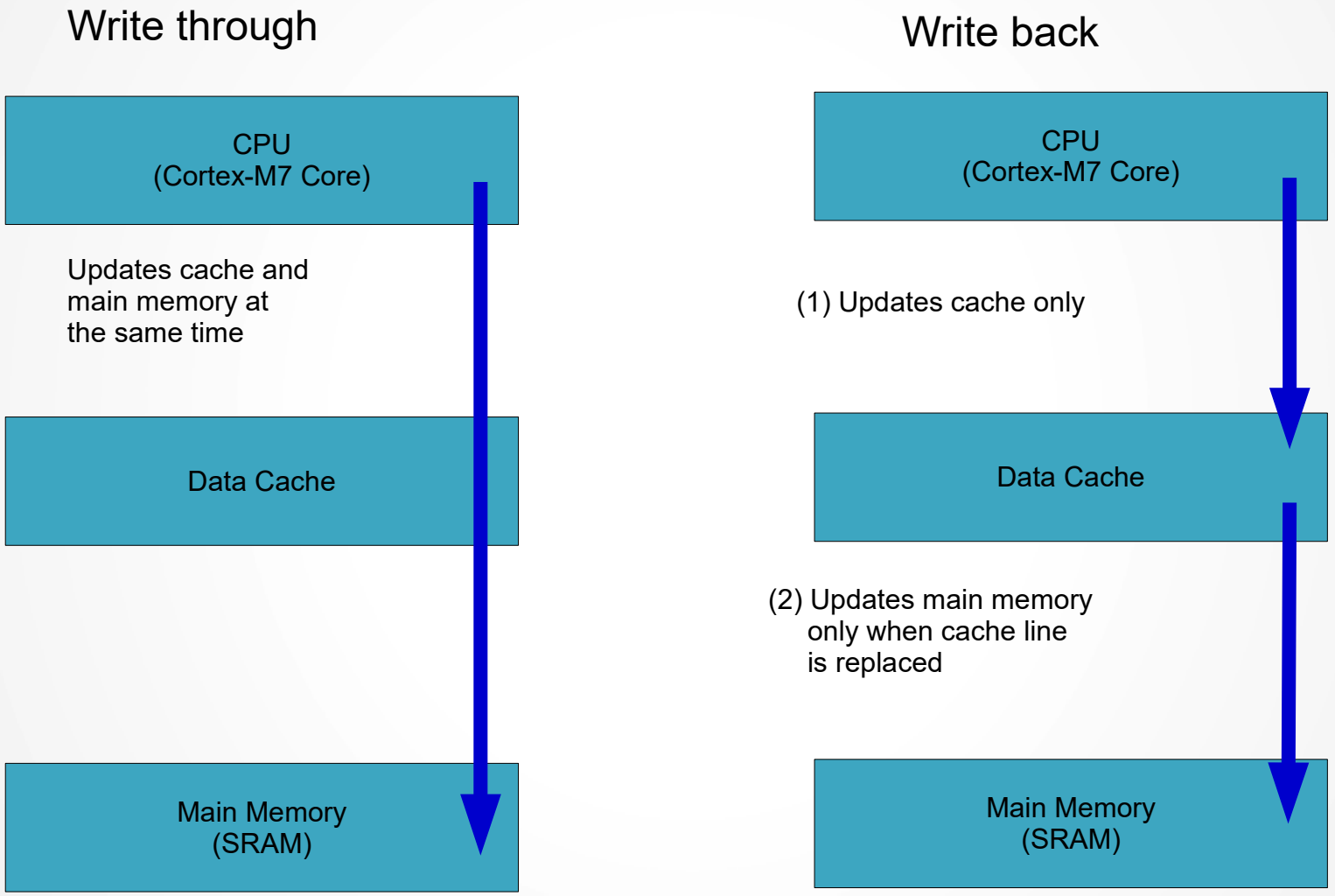
Speed – Cache

- For set associative mapping, we need a cache replacement algorithm to select a cache line in a set to be replaced when it needs to make room for a new block.
  - Common algorithms:
    - Least recently used (LRU).
    - Least frequently used (LFU).
    - First-in-first-out (FIFO).
    - Random (only slightly inferior).
  - Internal to hardware implementation.

Speed – Cache

- Write policy :
  - *Write through*.
    - Memory writes update cache and main memory at the same time.
    - More bus traffic, but ensures cache and main memory always remain in sync.
  - *Write back*.
    - Memory writes only update cache. Main memory is only updated when a *dirty* (updated) cache line is replaced.
    - Less bus traffic, but cache and main memory may be out of sync.
    - A problem with DMA. Why?

Speed - Cache



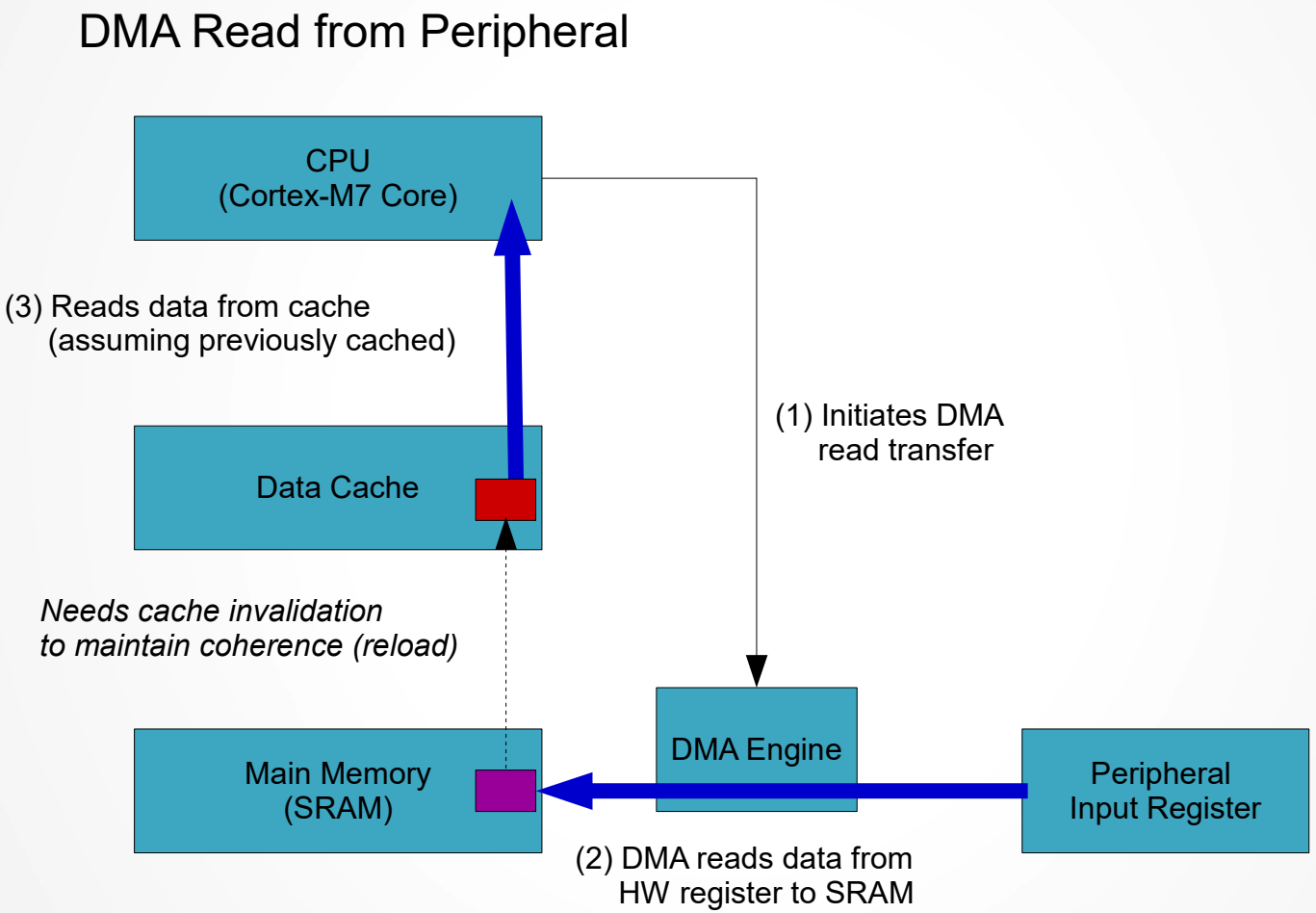
Speed - Cache

- Why do we care about cache (if the processor has one)?
- Cache miss is expensive (need to read from main memory, and potentially write-back a dirty line to be replaced).
  - How could we minimize cache miss?
  - The key is to optimize for locality, such as:
    - Use static arrays over dynamic memory allocation.
    - For two dimension arrays, looping over all columns of a row is more efficient than looping over a row element of all columns.
    - For arrays of objects, group data that are frequently processed together in one class and keep others in separate classes.
    - Keep data structure aligned to cache line size. Consider the case when a data structure spans over two cache line – may cause two replacements.
    - Avoid unnecessary multi-threading. Context switching may swap out the process stack causing many cache lines to be refilled.

Speed - Cache

- Another reason to care about cache is DMA.
- Cache coherence issue.
- Two main cases:
  - DMA read
  - DMA write
- For DMA read
  - Data is transferred by DMA from peripheral register(s) to a main memory buffer.
  - Some memory blocks of the buffer may have been cached.
  - When the processor reads from the buffer, out-of-date cached data likely will be returned.
  - How to fix? Invalidate cache lines (mapped from updated buffer) after DMA read completes. Alternatively, set the memory region containing the buffer as non-cacheable.

Speed - Cache

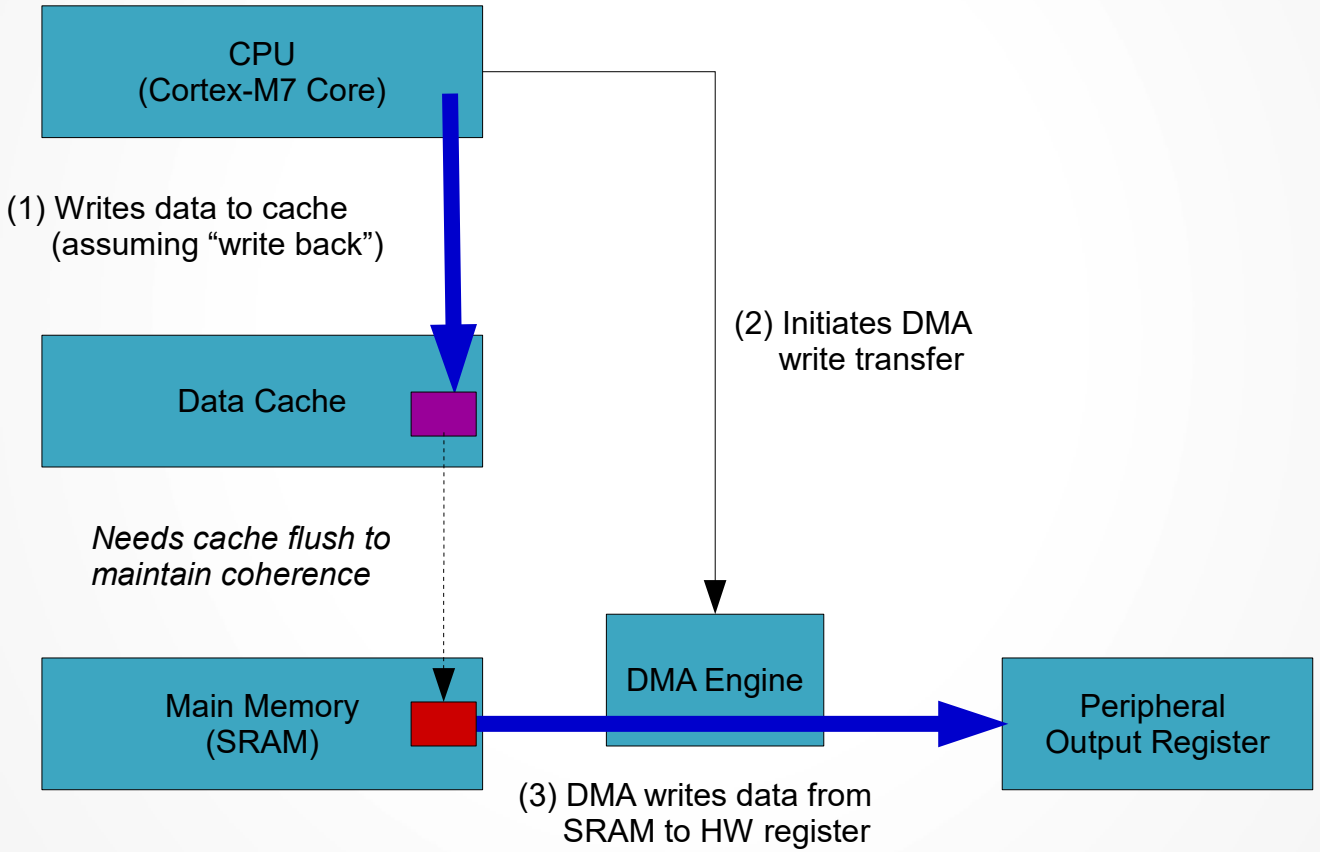


Speed - Cache

- For DMA write
  - Processor writes outgoing data to a memory buffer.
  - Assuming write-back policy, data are only updated in cache (unless replaced).
  - DMA is initiated to transfer data from main memory buffer to peripheral register(s).
  - Since main memory buffer may be out-of-sync with cache, outdated data are written to peripheral.
  - How to fix? Flush cache (mapped from updated buffer) before initiating DMA. Alternatively use write-through policy or set the memory region as non-cacheable.

Speed - Cache

DMA Write to Peripheral



Speed – Cache

- Cortex-M7 provides cache maintenance functions.
- See ARM Cortex-M7 Generic User Guide, Table 4-64 (page 292).
- See Eclipse STM32F7 project file “system\include\cmsis\core_cm7.h”.
- Examples of CMSIS data cache functions:

  void SCB_EnableDCache (void);

  void SCB_DisableDCache (void);

  void SCB_InvalidateDCache (void);

  void SCB_CleanDCache (void);

  void SCB_CleanInvalidateDCache (void);

  void SCB_InvalidateDCache_by_Addr (uint32_t *addr, int32_t dsize);

  void SCB_CleanDCache_by_Addr (uint32_t *addr, int32_t dsize);

  void SCB_CleanInvalidateDCache_by_Addr (uint32_t *addr, int32_t dsize);

Speed - Cache

- To invalidate cache after DMA read, call:

  void SCB_InvalidateDCache_by_Addr (uint32_t *addr, int32_t dsize);

- To flush cache before DMA write, call:

  void SCB_CleanDCache_by_Addr(uint32_t *addr, int32_t dsize);

  Example:

```
Fifo &fifo = *(me->m_fifo);
uint32_t addr = fifo.GetReadAddr();
uint32_t len = fifo.GetUsedCount();
if ((addr + len - 1) > fifo.GetMaxAddr()) {
    len = fifo.GetMaxAddr() - addr + 1;
}
FW_ASSERT((len > 0) && (len <= fifo.GetUsedCount()));
//SCB_CleanDCache();
SCB_CleanDCache_by_Addr((uint32_t *)(ROUND_DOWN_32(addr)),
                        ROUND_UP_32(addr + len - ROUND_DOWN_32(addr)));
HAL_UART_Transmit_DMA(&me->m_hal, (uint8_t*)addr, len);
```

Speed – Cache

- For DMA write, an alternative method is to use write-through policy. See Eclipse STM32F7 project file main.cpp (line 215) function MPU_Config().

```
static void MPU_Config(void) {
  MPU_Region_InitTypeDef MPU_InitStruct;
  HAL_MPU_Disable(); // Disable MPU
  // BaseAddress must be aligned to multiples of size. For the STM32F746 Discovery board
  // it has 320KB SRAM. For cover it all, we have to select 512KB and start from 0x20000000.
  // Note the original example use the base address of 0x20010000 and the size of 256KB. This is
  // incorrect since 0x20010000 is only aligned to 64KB and not 256KB. As a result, the range
  // covered is actually from 0x20000000 to 0x20040000 (256KB only, missing the last part).
  MPU_InitStruct.Enable = MPU_REGION_ENABLE;
  MPU_InitStruct.BaseAddress = 0x20000000;              // was 0x20010000.
  MPU_InitStruct.Size = MPU_REGION_SIZE_512KB;          // was MPU_REGION_SIZE_256KB.
  MPU_InitStruct.IsCacheable = MPU_ACCESS_CACHEABLE;
  MPU_InitStruct.IsShareable = MPU_ACCESS_NOT_SHAREABLE;// “not sharable” forces write-through.
  MPU_InitStruct.Number = MPU_REGION_NUMBER0;
  ...
  HAL_MPU_ConfigRegion(&MPU_InitStruct);
  HAL_MPU_Enable(MPU_PRIVILEGED_DEFAULT);  // Enable MPU
}
```

Code – Multiple Instances (HW)

- Now we move on to optimization for code space.
- More than often you would need more than one instance of the same class.
- For example in our demo project we have an active object class for the user button named UserBtn. (Similarly we have one for the user LED named UserLed.)
- It serves well to interface with the user button and user LED.
- What if we want to add more buttons and LED's?

Code – Multiple Instances (HW)

- One way is to create another active object class for the new button/LED, such as MenuBtn or StatusLed, etc.
- Let's check what will be different in each class. It turns out we will be duplicating a lot of code.
- A more optimized way is to parameterize the class such that the GPIO port/pin used for the button or LED are configuration parameters rather than hard-coded.
- The configuration parameters will be a members of the class, such as *m_port* and *m_pin*.
- Those parameters can either be passed in via the constructor (like UartAct), or contained in a static constant array of structures of configuration parameters.

Code – Multiple Instances (HW)

- What does the last point mean?
- Example:

```
typedef struct {
    uint8_t id;             // HSM ID
    GPIO_TypeDef *port;     // GPIO port
    uint32_t pin;           // GPIO pin
} LedConfig;

static const LedConfig CONFIG[] = {
    { USER_LED,    GPIOA, GPIO_PIN_5 },
    { USER_STATUS, GPIOC, GPIO_PIN_1 }
};
```

- Note that you may need more HW specific parameters (e.g. if PWM is used, you'll need to parameterize HW timer used).
- Note how a line in the structure array replaces an entire duplicated class.

Data and Speed – Non-blocking Architecture

- With QP we can encapsulate multiple HSM (regions) in a single active object.
- In essence it allows multiple HSM's to share a single *thread*.
- This is possible due to the fundamental non-blocking run-to-completion nature of HSM's.
- Question – Why is it not possible with blocking design?
- Question – What is the difference between making an HSM a region than creating its own active object?
- Less threads → Less memory overhead.
- Less threads → Less context switching overhead (faster)
- Compare to Node.js which also uses a non-blocking architecture.