COPYRIGHT

DISCLAIMER

This content is provided without warranty or representation of any kind. The use of the content is entirely at your own risk and Stellenbosch University (SU) will have no liability directly or indirectly as a result of this content.

The content must not be assumed to provide complete coverage of the particular study material. Content may be removed or changed without notice.

The video is of a recording with very limited post-recording editing. The video is intended for use only by SU students enrolled in the particular module.

UNIVERSITEIT
iYUNIVESITHI
STELLENBOSCH
UNIVERSITY

100
1918 · 2018

forward together · saam vorentoe · masiye phambili

Computer Systems / Rekenaarstelsels 245 - 2020

Lecture 22

Direct Memory Access (DMA)/ Direkte Geheue Toegang

Dr Rensu Theart & Dr Lourens Visagie

Reason for DMA
Rede vir DMA

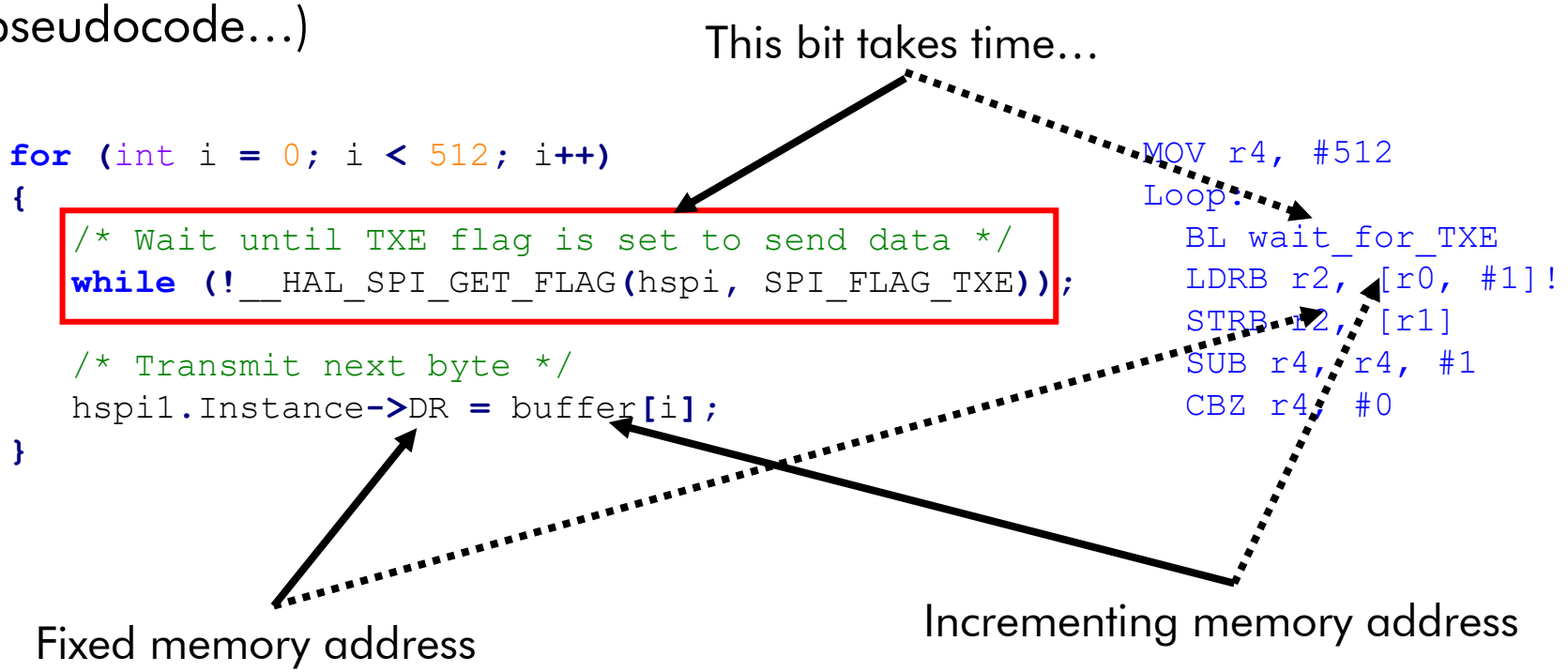
- Microcontroller programs tend to move around a lot of data
  - Continuously sample accelerometer, and store into array (in SRAM)
  - (Continuously sample data from the analog-to-digital (ADC) converter)
  - Write sampled data, from array in SRAM, to SD card
- Typical code loop will:
  - Read from memory at address A0 (remember reading a peripheral register is also just memory read)
  - Write to memory at address A1 (writing to a peripheral register is just a memory write)
  - Possibly increment A0 and/or A1
  - Until buffer is full, then possibly repeat

Reason for DMA
Rede vir DMA

- Example: write block of data (512 bytes) to SD card
- HAL_SPI_Transmit(&hspi1, buff, 512, 10);

- And in HAL_SPI_Transmit:

(pseudocode…)



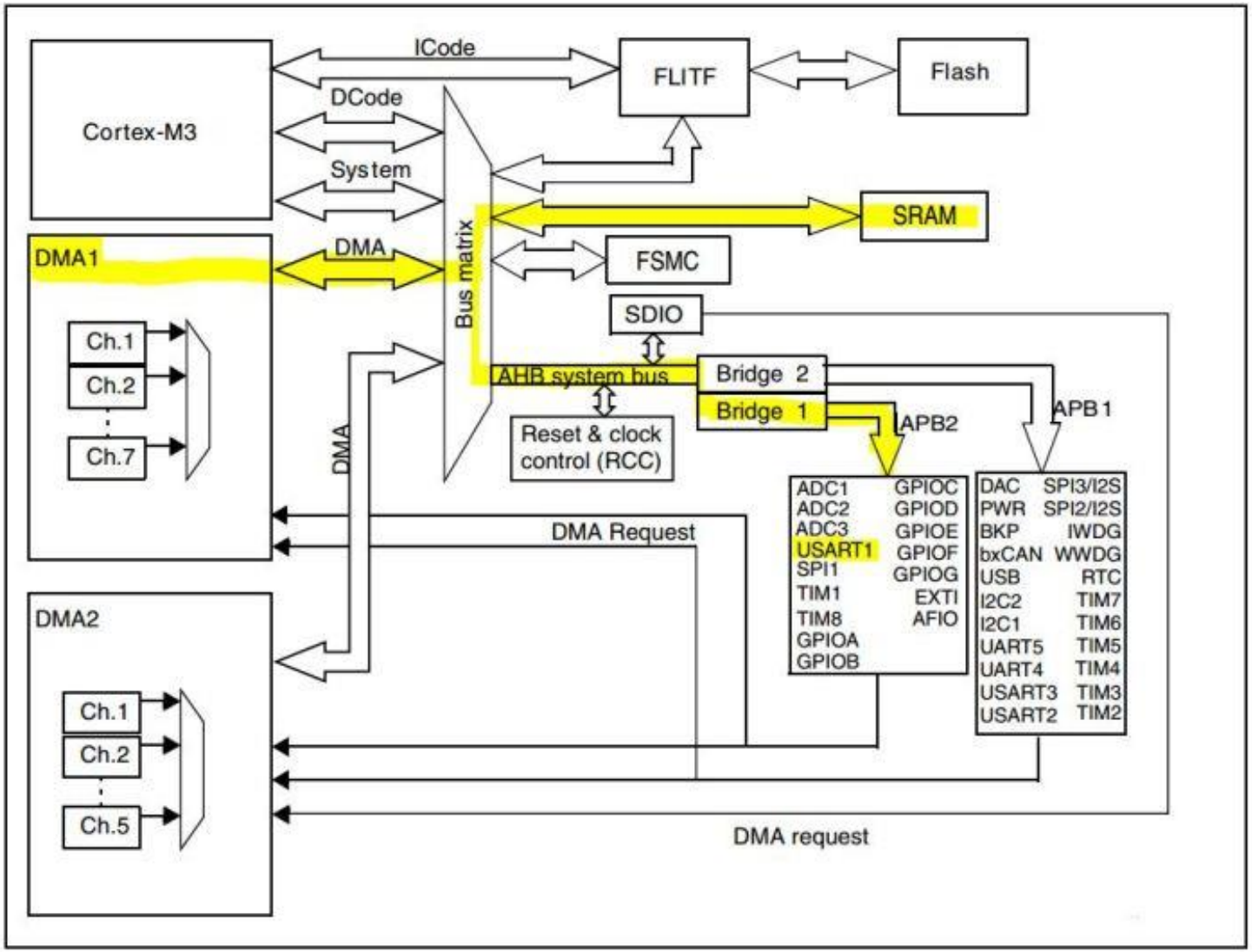
Reason for DMA
Rede vir DMA

- Blocking function calls wait for the action to complete before returning back to the code that called it
  - HAL_SPI_Transmit will only exit after all the bytes have been sent, and then the main loop code can continue
- For data transmission (SPI, UART, I2C) the blocking delay is directly proportional to the transmission data rate
  - UART sending/receiving data with 115200 baudrate, 8N1 configuration will result in a data rate of 11520 bytes per second. If you are sending/receiving 100kB of data it will take 8.9s
  - SPI at 8MHz will send/receive at a data rate of 1000000 bytes/s. 100kB of data will take 102ms.
- The time spent waiting for the transmission of each byte to complete is "wasted" CPU cycles
- Using interrupts can help a bit – while each byte/word is being transmitted the CPU can run other code, but it still has to intervene for every data element.

Reason for DMA
Rede vir DMA

- Wouldn't it be cool if there was a way of doing this in the background?
- That's exactly what DMA is!
- The DMA controller is a control unit in the MCU that moves data around, without the CPU involved in the transfer (the CPU is only needed to setup the transfer)
- By making use of DMA, the CPU can use the entire blocking time delay for other tasks

What is DMA?
Wat is DMA?



What is DMA?
Wat is DMA?

- DMA controller is accessed from the CPU through memory mapped registers
- PAR: Peripheral Address Register
- M0AR: Memory Address Register
- NDTR: Number of data register
- CR: Control Register.
  - DIR bits:
    - 00 = Peripheral-to-memory
    - 01 = Memory-to-peripheral
    - 10 = Memory-to-memory (PAR is also a memory address)
  - EN bit: start the transfer

What is DMA?
Wat is DMA?

- Depending on the transfer direction, load data from the source address, and store it to the destination address

Table 29. Source and destination address

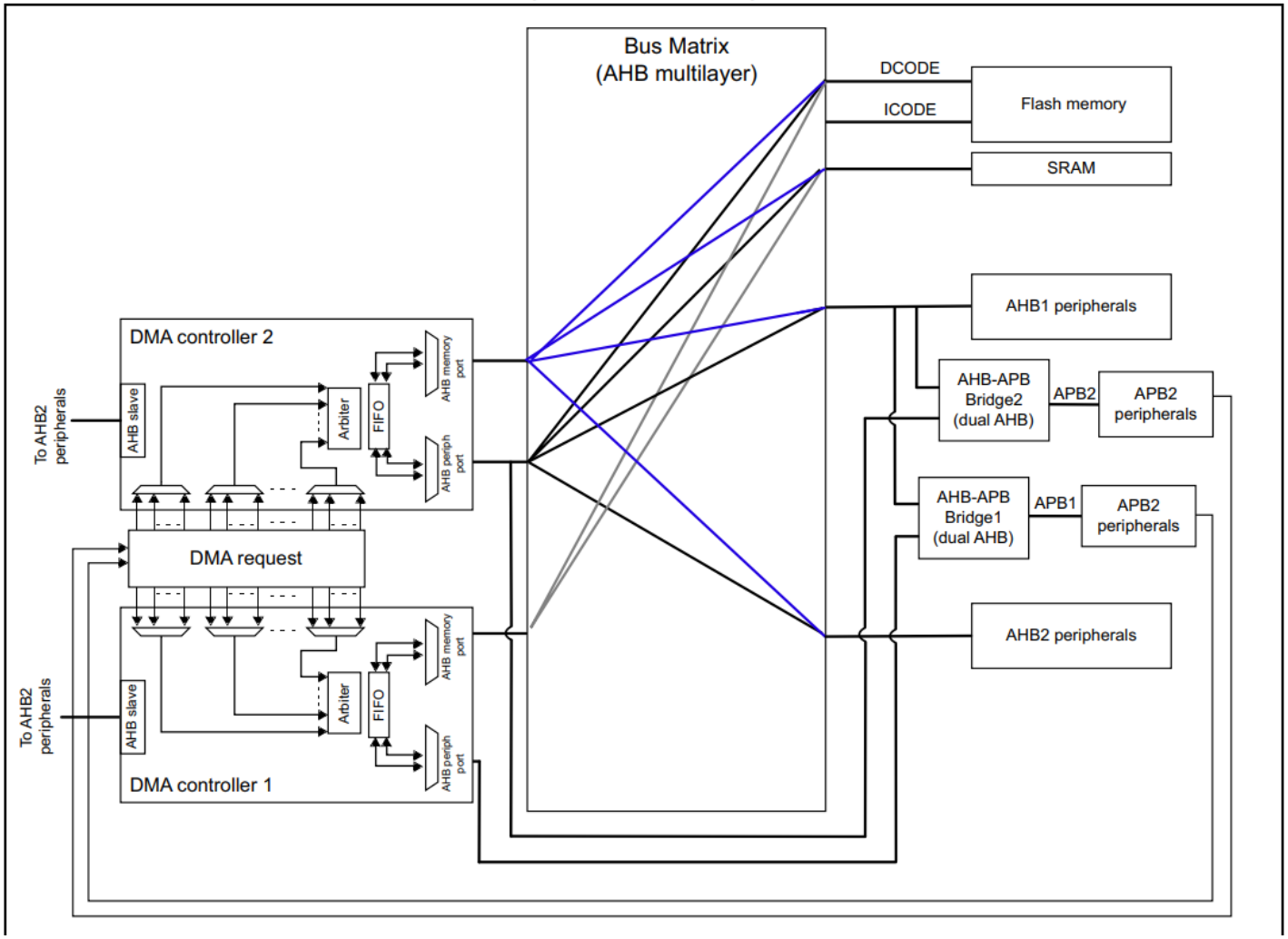
Bits DIR[1:0] of the DMA_SxCR register	Direction	Source address	Destination address
00	Peripheral-to-memory	DMA_SxPAR	DMA_SxM0AR
01	Memory-to-peripheral	DMA_SxM0AR	DMA_SxPAR
10	Memory-to-memory	DMA_SxPAR	DMA_SxM0AR
11	reserved	-	-

- Size of data to load/store is determined by bits in the Configuration Register (MSIZE, PSIZE)
- Decrement the value in the NDTR register (number of data register)
- Optionally increment the address source and data address registers (PAR, M0AR) – determined my MINC and PINC settings in the CR
- Transfer stops when NDTR reaches zero
- Optionally restart the entire transfer with original settings (Circular transfer mode)

STM32F4 DMA

- STM32F4 is complicated… It has 2x DMA controllers

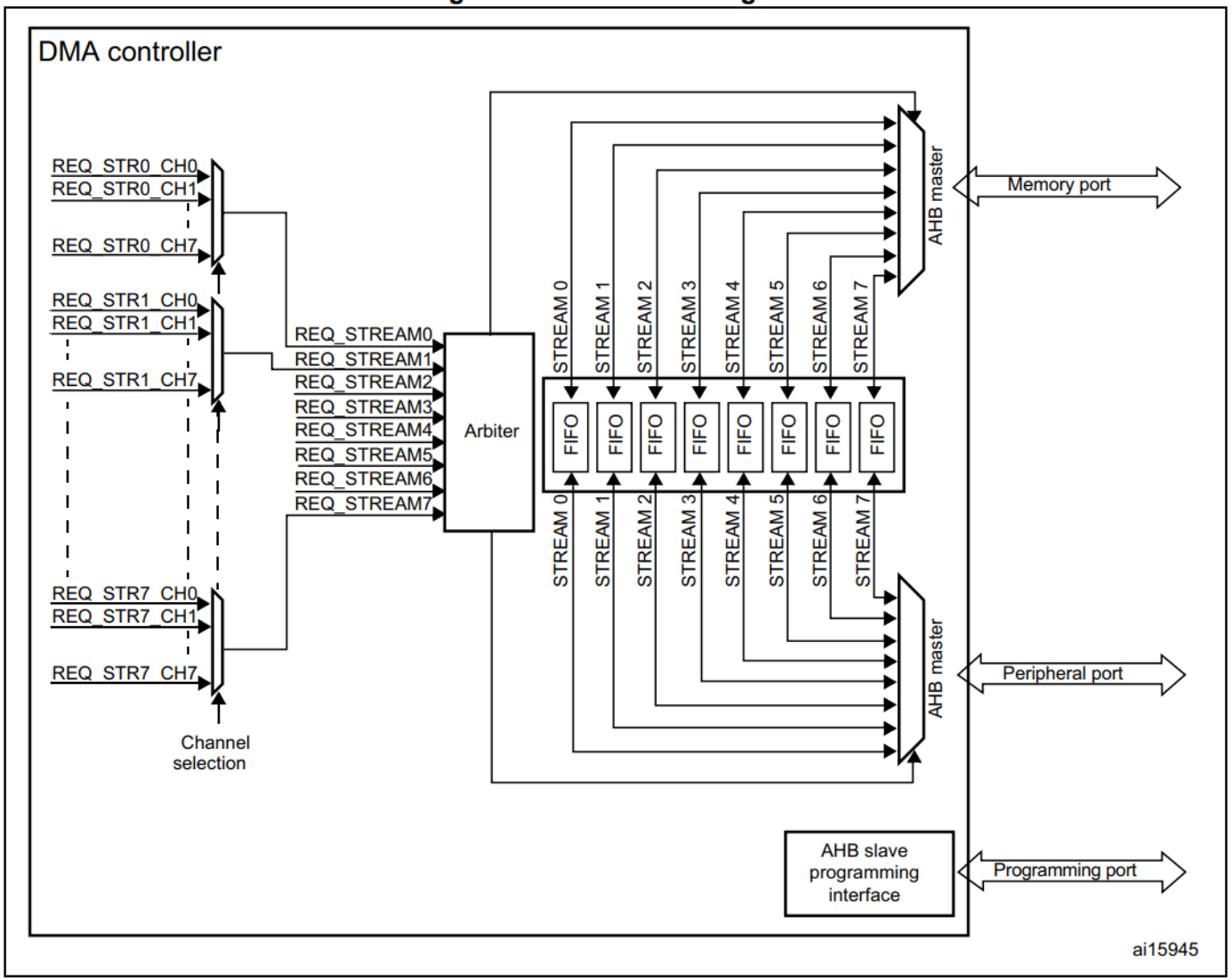
Figure 23. System implementation of the two DMA controllers (STM32F411xC/E)



STM32F4 DMA

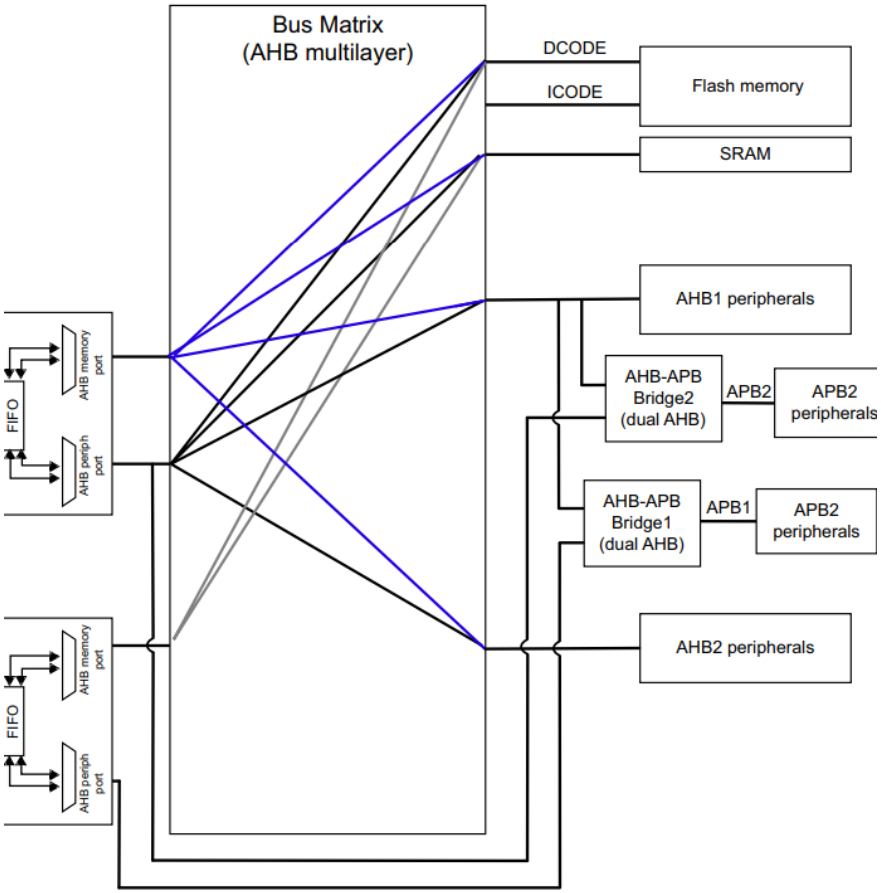
- And each DMA controller has 8 “streams”

Figure 22. DMA block diagram



STM32F4 DMA

- And each DMA controller has 8 “streams”
  - Each DMA controller can have 8 active transfers.
  - Memory and peripheral ports (AHB master ports) can only do one transfer at a time. Access of all 8x streams to port is determined by “arbiter” – makes use of priorities for each stream
- Two DMA controllers
  - DMA1 cannot do memory-to-memory transfers – it’s peripheral port is not connected to the flash or SRAM memories through the bus matrix
- Other complexities:
  - Each stream has a FIFO buffer
  - Transfers can happen in bursts (4, 8 or 16 beats) – AHB bus is “locked” during the burst
  - Double-buffer transactions: DMA switches between MA0R and MA1R in circular transfer mode



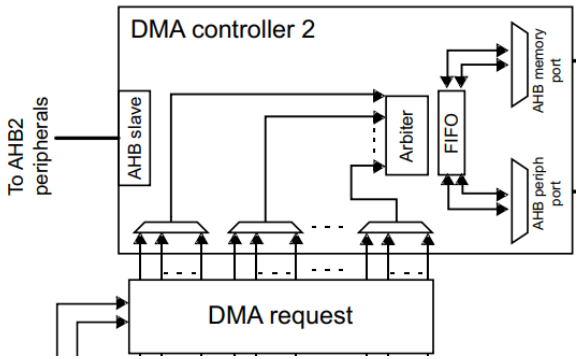
STM32F4 DMA

- But wait… what about the “wait”?

```
for (int i = 0; i < 512; i++)
{
   /* Wait until TXE flag is set to send data */
   while (!__HAL_SPI_GET_FLAG(hspi, SPI_FLAG_TXE));

   /* Transmit next byte */
   hspi1.Instance->DR = buffer[i];
}
```

- DMA transfer is “throttled” by something called a “DMA request”
- The source for the DMA request can be configured, to come from a peripheral event



Peripherals

STM32F4 DMA

- DMA Requests:
- Channel Select (CHSEL) option in Control Register determines where the DMA request comes from
- Mapping of channel request to peripheral events are given in the Reference Manual

Figure 24. Channel selection

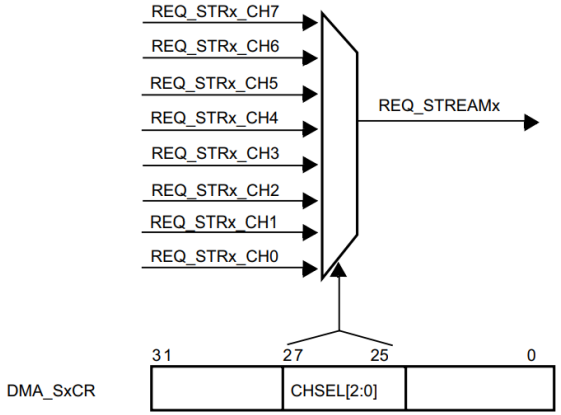


Table 27. DMA1 request mapping (STM32F411xC/E)

Peripheral requests	Stream 0	Stream 1	Stream 2	Stream 3	Stream 4	Stream 5	Stream 6	Stream 7
Channel 0	SPI3_RX	I2C1_TX	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	USART2_RX
Channel 7	-	-	I2C2_RX	I2C2_RX	-	-	-	I2C2_TX

STM32F4 DMA

- DMA Requests:
- For instance, for DMA1, stream 3, if I use CHSEL = 000_2, the DMA request is generated if SPI2 receives data
- A single DMA transfer (load from PAR/MA0R, store to MA0R/PAR) will happen if
  - DMA is enabled for the SPI2 peripheral, AND
  - The RXNE flag changes to 1 (receive data register not empty)
- And similar for other peripherals. A timer event can also generate a DMA request

Table 27. DMA1 request mapping (STM32F411xC/E)

Peripheral requests	Stream 0	Stream 1	Stream 2	Stream 3	Stream 4	Stream 5	Stream 6	Stream 7
Channel 0	SPI3_RX	I2C1_TX	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	USART2_RX
Channel 7	-	-	I2C2_RX	I2C2_RX	-	-	-	I2C2_TX

STM32F4 DMA

- DMA Requests:
- Does that mean, I can setup stream 1 to transfer from USART1 data register to memory, and select I2C1_TX as the DMA request source?
  - Yes
- Does it make sense to do that?
  - Probably not…

Table 27. DMA1 request mapping (STM32F411xC/E)

Peripheral requests	Stream 0	Stream 1	Stream 2	Stream 3	Stream 4	Stream 5	Stream 6	Stream 7
Channel 0	SPI3_RX	I2C1_TX	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	USART2_RX
Channel 7	-	-	I2C2_RX	I2C2_RX	-	-	-	I2C2_TX

STM32F4 DMA

- Memory-to-memory transfers
- Transfer from one part of memory to another – essentially performs a ‘memcpy’ function in non-blocking mode
- Don’t need a DMA request – no need to wait between transfers

Programming with the STM32F4 DMA
Programmering met die STM32F4 DMA

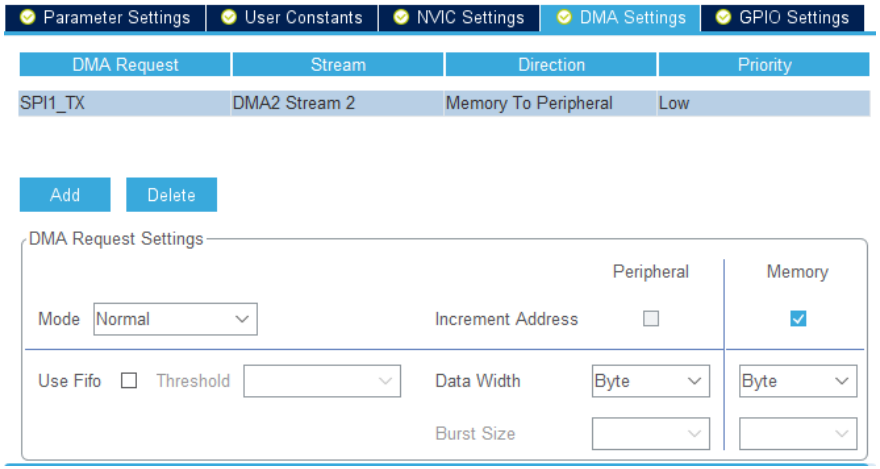
- As with most of the peripherals, we rather use the HAL framework, instead of directly configuring the DMA registers
- For USART, SPI, I2C, ADC peripherals, use the _DMA equivalent function

```
HAL_StatusTypeDef HAL_SPI_TransmitReceive_DMA(SPI_HandleTypeDef *hspi, uint8_t *pTxData, uint8_t
*pRxData, uint16_t Size);

HAL_StatusTypeDef HAL_UART_Transmit_DMA(UART_HandleTypeDef *huart, uint8_t *pData, uint16_t
Size);

HAL_StatusTypeDef HAL_UART_Receive_DMA(UART_HandleTypeDef *huart, uint8_t *pData, uint16_t Size);
```

- But, you will still have to enable DMA for that peripheral in the device configuration tool.



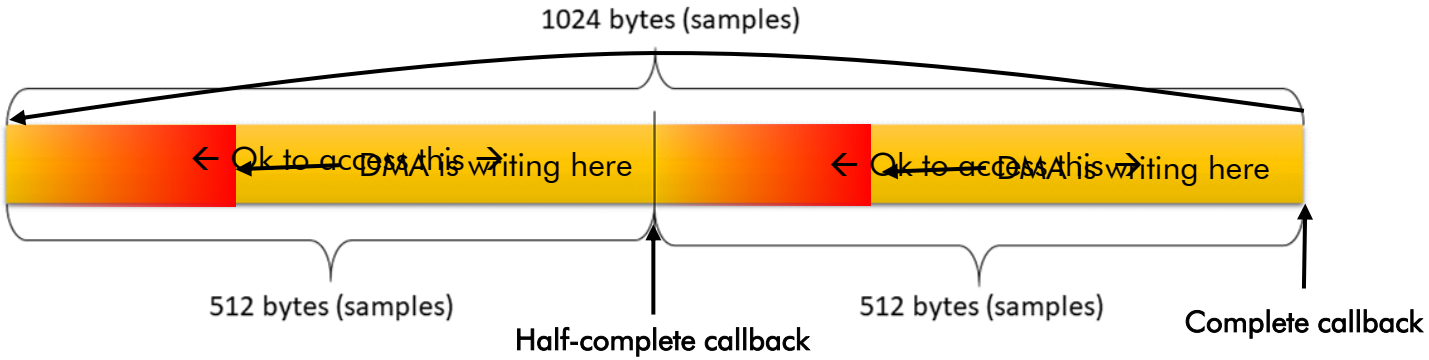
Programming with the STM32F4 DMA
Programmering met die STM32F4 DMA

- How do you (your program) know when the transfer is complete?
- Answer: **Interrupts**!
- DMA can generate two interrupts for transfer progress (also error interrupts, and FIFO underrun)
  - Transfer half-complete (NDTR is half of initial value)
  - Transfer complete (NDTR is zero)
- In your program, make use of the provided call-back functions for the particular peripheral

```
void HAL_SPI_TxCpltCallback(SPI_HandleTypeDef *hspi);
void HAL_SPI_RxCpltCallback(SPI_HandleTypeDef *hspi);
void HAL_SPI_TxHalfCpltCallback(SPI_HandleTypeDef *hspi);
void HAL_SPI_RxHalfCpltCallback(SPI_HandleTypeDef *hspi);
```

Programming with the STM32F4 DMA
Programmering met die STM32F4 DMA

- Double buffering:
- Used with circular transfers
- Either use the built-in double-buffer features of the DMA, or two side-by-side memory arrays (or one memory array twice as long)
- Example Scenario 1:
- Continuously sample temperature measurement from analog to digital converter and calculate an average over 512 samples
- Declare a 1024 element array
- Start circular DMA transfer for 1024 elements
- Wait until half-complete callback, then calculate average over first 512 samples
- Wait until complete callback, then calculate average over last 512 samples. Repeat.



Programming with the STM32F4 DMA
Programmering met die STM32F4 DMA

- Double buffering:
- Used with circular transfers
- Either use the built-in double-buffer features of the DMA, or two side-by-side memory arrays (or one memory array twice as long)
- Example Scenario 2:
- Continuously transmit digital audio to I2S peripheral, but CPU program generates audio (synthesizer)
- Declare a 1024 element array. Fill it with first 1024 audio samples
- Start circular DMA transfer for 1024 elements
- Wait until half-complete callback, then calculate next 512 samples
- Wait until complete callback, then calculate following 512 samples. Repeat.