5 Section: Configuring DMA peripherals (page 263)

- **Additional info**, page 263
  - Some helpful DMA references:
  - *Reference manual : STM32F76xxx and STM32F77xxx advanced Arm®-based 32-bit MCUs*
    - Section *8 Direct memory access controller (DMA)*
  - Blog-articles:
    - "STM32 DMA Cheat Sheet"
      - https://adammunich.com/stm32-dma-cheat-sheet/
      - The article starts with, "*STMicro's documentation about the subject is some of the most terse documentation I've ever seen in the business!*"
    - "STM32F2xx DMA Controllers – Frank's Random Wanderings"
      - https://blog.frankvh.com/2011/08/18/stm32f2xx-dma-controllers/

6 Section: A buffer-based driver with DMA (page 266)

- **Additional info**, page 266
  - The whole DMA set-up has to be performed after each full DMA transfer (here, it's 16 bytes).
  - `startReceiveDMA()` sets-up the DMA. `startReceiveDMA()` is called for each 16 bytes received via the DMA. (In `uartPrintOutTask()`, `startReceiveDMA()` is called for each iteration of the while-loop.) After the first call to `startReceiveDMA()`, could it be that subsequent calls don't have to do all of the DMA set-up, and just part of the set-up is needed, e.g., just issue `HAL_DMA_Start()`?
  - From experimenting, it appears that once the 16-byte DMA transfer is done, all of the DMA set-up has to be performed again, in order to get more data.
  - A blog article confirms this: *If circular buffer mode is not enabled, you must manually reload the source address, destination address, and number of bytes to be transferred, into the DMA stream's configuration registers prior to re-launching the stream. The DMA controller uses these registers for the address pointers and byte counter, with no separate mechanism to reload them in normal usage. This is pretty weak, as it wastes a bunch of clock cycles for short transfers.*
    - https://adammunich.com/stm32-dma-cheat-sheet/

- **Clarification**, page 268
  - Apparently the baud rate is in bits/sec, and there's 10 bits sent per byte (8 data bits and stop/start bits).
  - The time to send 16 bytes can be calculated mathematically:
    - At 9600 bits/sec, 16 bytes is sent in 0.0167 sec
    - At 256,400 bits/sec, 16 bytes is sent in 624 micro-sec
  - These calculated periods are the same as those shown empirically on page 268

7 Section: Stream buffers (FreeRTOS 10+) (page 269)

- **Added info**, page 269
  - In the FreeRTOS docs, stream-buffers are described here:
    - https://www.freertos.org/RTOS-stream-message-buffers.html

- **Clarification**, page 270
  - The `xStreamBuffer*` functions are FreeRTOS APIs, e.g.,
    - `xStreamBufferCreate()`
    - `xStreamBufferReceive()`
    - `xStreamBufferSendFromISR()`

- **Bug in the code** (`mainUartDMAStreamBufferCont.c`), page 270f
  - **Problem #1:**
    - Typo in `main()`: `rxStream = xStreamBufferCreate( 100, 1);`
  - **Solution #1:**
    - In the book it is: `rxStream = xStreamBufferCreate( 100, 2);`

  - **Problem #2:**
    - Buffer-overflow in `uartPrintOutTask()`, in the `memset()`:
      - `memset(rxBufferedData, 0, 20);`
  - **Solution #2:**
    - `memset(rxBufferedData, 0, maxBytesReceived);`

- **Additional info**, page 274
  - How to set-up the second DMA-buffer is discussed in the book. The code for it is in `startCircularReceiveDMA()`:

```
DMA1_Stream5->M1AR = (uint32_t) rxData2;
```

  - However, setting-up the first DMA-buffer isn't discussed, specifically. The code-comments state that the first buffer is specified in `DMA1_Stream5->M0AR`.
  - `DMA1_Stream5->M0AR` is set by `HAL_DMA_Start()` (it also sets the buffers' length).
  - `HAL_DMA_Start()` is called in `startCircularReceiveDMA()`: