KURS STM32

Wojciech Olech

CZĘŚĆ V: DIRECT MEMORY ACCESS (DMA)

Każdy mikrokontroler musi wymieniać się danymi między peryferiami i/lub pamięcią. Każdy dostęp do pamięci i peryferiów (które są reprezentowane przez rejestry) zajmuje procesorowi trochę czasu - w zależności od tego ile danych obsługuje i jak często. Wiąże się to z jego obciążeniem, ponieważ w czasie kiedy obsługuje kopiowanie danych nie może on robić nic innego.

W nowoczesnych systemach ten problem jest rozwiązany przez DMA. DMA, czyli moduł **Direct Memory Access**, pozwala na w pełni niezależną od procesora, a co za tym idzie - asynchroniczną - obsługę transferu danych między pamięcią i peryferiami.

JAK DZIAŁA DMA?

Każde peryferium w STMie musi wymieniać dane z rdzeniem Cortex-M. Peryferia są zmapowane jako adresy pamięci do których należy się odwołać w celu uzyskania dostępu do danych w ich rejestrach. Każdy dostęp do takiego rejestru wymaga interwencji procesora, w celu uzyskania dostępu i skopiowania danych z miejsca A do miejsca B. Dodatkowo, CPU musi obsłużyć wszystkie przerwania jakie po drodze wystąpią, oraz czekać na gotowość peryferiów jeśli nie można skopiować pamięci za jednym zamachem.

Wszystkie te operacje zgromadzone razem zaczynają zajmować stosunkowo dużą ilość czasu, który procesor mógłby lepiej i efektywniej wykorzystać. DMA pozwala odciążyć procesor z tych operacji, poprzez obsługę danego peryferium i powiadomienie procesora na przykład o zakończeniu transferu.

ARCHITEKTURA SZYN DANYCH W STM32

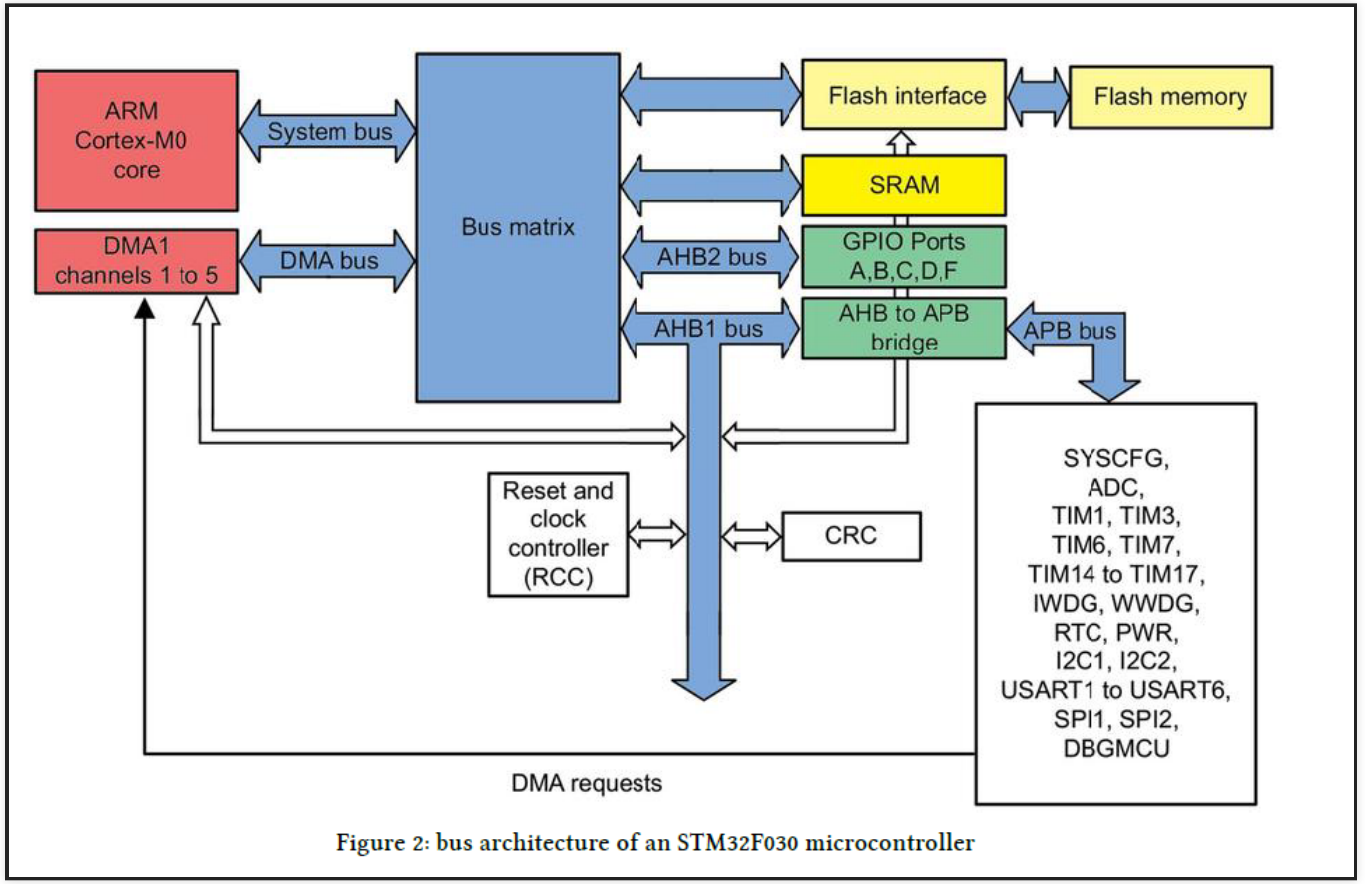


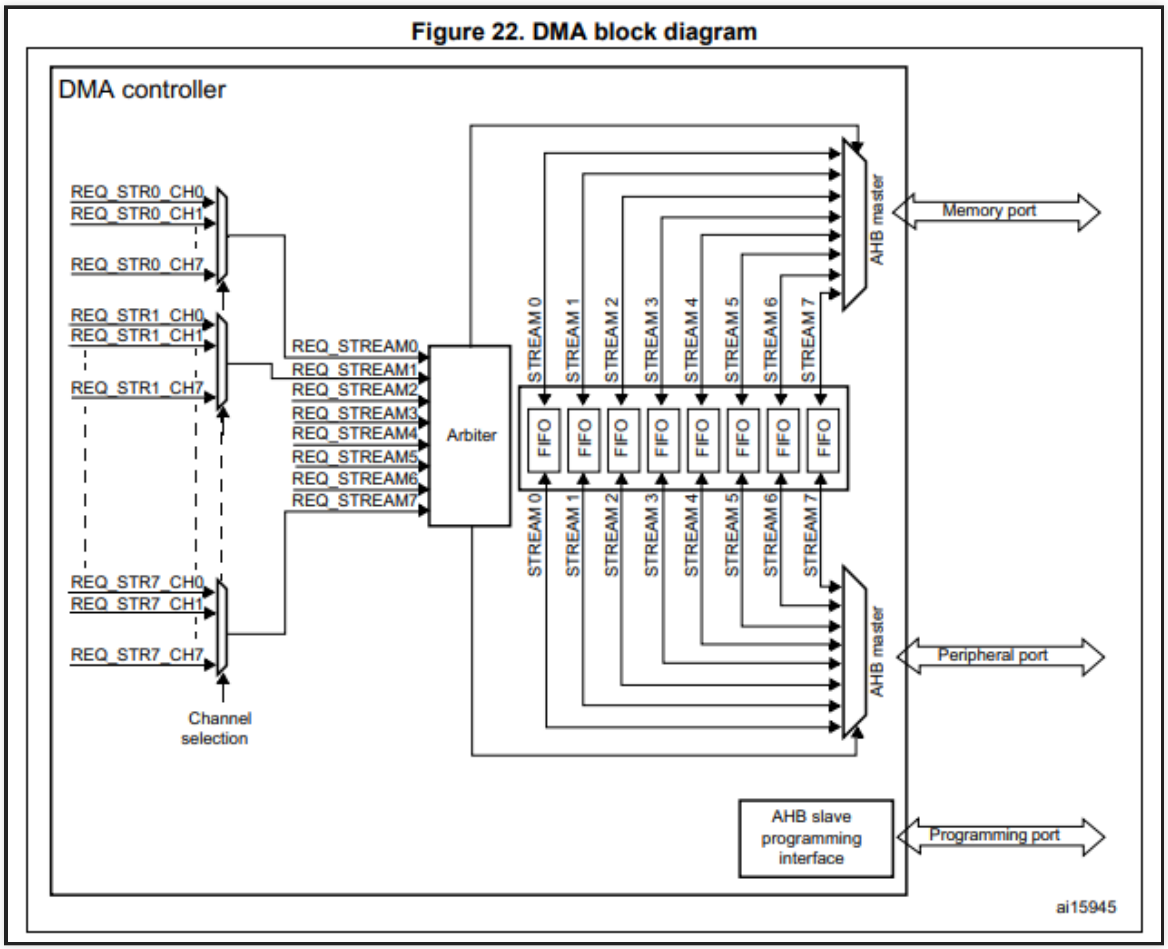
Figure 2: bus architecture of an STM32F030 microcontroller

źródło: Mastering STM32, sam diagram pochodzi z manuala do STM32F030

AHB to *Advanced High-performance Bus*, główna szyna danych po której peryferia i pamięci komunikują się z rdzeniem i DMA. Procesory STM32 zazwyczaj mają takie dwie.
APB to *Advanced Peripheral Bus*, po tej szynie rozmawiają ze sobą peryferia typu USART, timery, SPI, itd.

Cortex-M i DMA są masterami, co oznacza że mogą inicjować transfer między peryferiami a pamięcią.
Slave'ami są szyny AHB i pamięć Flash oraz SRAM. BusMatrix zarządza połączeniami między rdzeniem Cortex-M i DMA a peryferiami.
Jak widzimy na schemacie, peryferia mają bezpośrednie połączenie do DMA do którego mogą przekazywać przerwania. Dzięki temu DMA może bez udziału rdzenia je obsługiwać.

BUDOWA DMA



źródło: manual STM2F401

DMA zawiera w sobie

- Dwa *master* porty, nazywane *peripheral* i *memory*, połączone do AHB. Służą one do transferów danych między peryferiami a pamięcią. W niektórych kontrolerach DMA, port *peripheral* ma również dostęp do pamięci, co pozwala na transfery pamięć-pamięć z użyciem DMA.
- Jeden *slave* port, połączony do AHB który służy do programowania DMA przez CPU.
- Kilka kanałów (źródeł requestów) podłączonych do peryferiów, poprzez które DMA odbiera komunikaty od nich.

DMA pozwala przypisać różne priorytety danym kanałom oraz pozwala na dostęp w obie strony, zarówno z peryferium do pamięci jak i z pamięci do peryferium.

Strumienie są przypisane do danych peryferiów na etapie designu DMA. Jeden kanał DMA może obsługiwać maksymalnie jeden strumień jednocześnie, nie może on obsługiwać kilku strumieni na raz. Niektóre STMy mają dwa kontrolery DMA co pozwala w pewnym stopniu ominąć to ograniczenie. Request mapping jest dostępny w manualu danego mikrokontrolera, ale nie będzie nas on w tej chwili interesować ponieważ Device Manager w STM32CubeIDE pozwoli nam łatwo skonfigurować DMA.

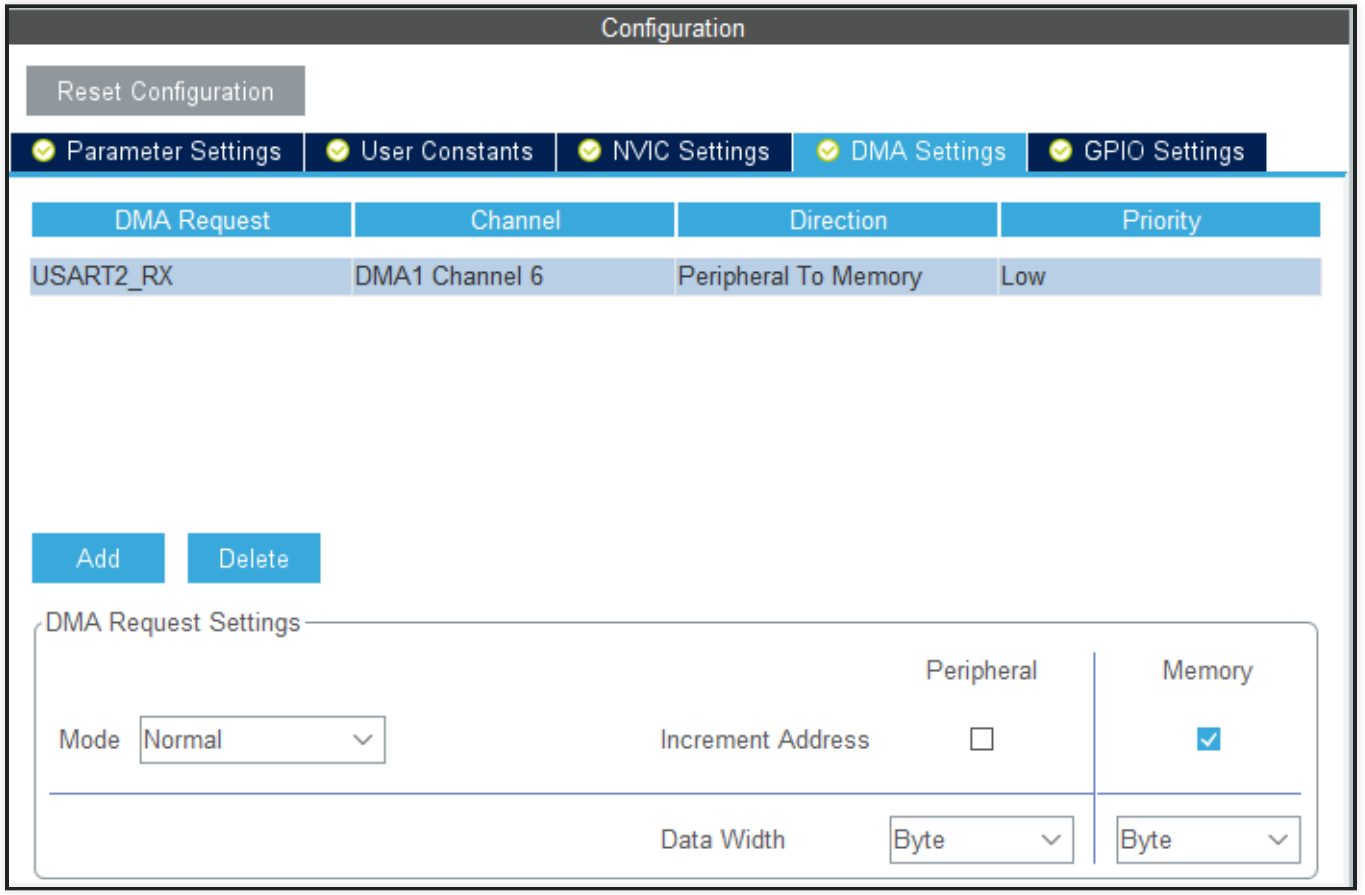
Dodatkowo, DMA będzie pracować nawet przy procesorze zatrzymanym przez debuger (ponieważ debuger zatrzymuje tylko rdzeń)

KONFIGURACJA PERYFERIÓW W TRYBIE DMA

Posłużymy się tutaj przykładem użycia USARTa z DMA do odbioru znaków w taki sam sposób jak zrobiliśmy to w poprzednim rozdziale, ale tym razem z użyciem DMA.

Zaczniemy od konfiguracji USARTa w Device Managerze. Wybieramy USART2 i przechodzimy do zakładki DMA. Klikamy przycisk "Add" i z listy wybieramy USART2_RX, ponieważ na razie będziemy chcieli obsłużyć asynchronicznie tylko odbiór.

Potem przechodzimy do zakładki *NVIC Settings* i **uruchamiany globalne przerwanie UARTa**. Bez tego, Device Manager nie wygeneruje kodu do obsługi przerwań UARTa, co utrudni nam obsługę.



Screenshot pochodzi z konfiguratora dla Nucleo-L476RG, w przypadku innych Nucleo kanał request USART_RX może być przypisany do innego kanału

Mamy tutaj dwie opcje które możemy skonfigurować

- **Mode** - tryb w którym DMA będzie obsługiwać peryferium.
  - W trybie normalnym, DMA odczytuje/zapisuje dane z/do podanego bufora i kończy działanie po odczytaniu/zapisaniu wszystkich bajtów
  - W trybie cyklicznym (*circular*), DMA nie przerywa działania po zakończeniu odczytu tylko wraca do początku bufora i ponownie z niego odczytuje lub do niego zapisuje dane.
- **Data Width** - tutaj możemy skonfigurować wielkość pojedynczej jednostki danych jaką DMA ma obsługiwać. W przypadku domyślnych ustawień USARTa, ramki są 8-bitowe więc bajt w zupełości wystarczy. Do dyspozycji są jeszcze półsłowa (*Half Word*) i pełne słowa (*Word*), odpowiednio o wielkościach 16 i 32 bitów.

Na razie skonfigurujmy DMA w trybie normalnym, potem przejdziemy do cyklicznego.

Po ustawieniu kanału DMA i zapisaniu projektu, Device Manager wygeneruje nam kod inicjalizacji USARTa z DMA. Znajduje się on w funkcjach `MX_DMA_Init` (z pliku `main.c` lub `dma.c`), gdzie inicjalizowany jest zegar DMA i konfigurowane jest przerwanie, oraz w `HAL_UART_MspInit` (z pliku `stm32f4xx_hal_msp.c` lub `usart.c`) gdzie znajduje się konkretna konfiguracja DMA dla UARTa i przypisanie (zlinkowanie) ustawionego strumienia do uchwytu UARTa.

Obsługa UARTa z HALem i wygenerowanym kodem DMA wygląda podobnie jak obsługa UARTa w trybie przerwań - do rozpoczęcia odbioru należy użyć funkcji `HAL_UART_Receive_DMA`, która przyjmuje trzy argumenty - uchwyt UARTa, adres bufora oraz ilość bajtów do odczytania, a po zakończeniu (i w połowie odbioru) zostaną wywołane te same handlery co w trybie *interrupt*, czyli `HAL_UART_RxCpltCallback` (i `HAL_UART_RxHalfCpltCallback`).

Możemy więc wykorzystać nasz kod w którym obsługiwaliśmy UARTa z użyciem przerwań i zastosować go w ten sam sposób dla DMA. Różnica będzie taka, że podczas odbierania danych z użyciem DMA procesor będzie mógł przeznaczyć większość swojego czasu na inne zadania, podczas gdy DMA zajmie się obsługą UARTa i kopiowaniem danych z rejestrów UARTa do pamięci.

Przepiszmy nasz echo server który napisaliśmy dla UARTa w poprzednich rozdziałach, tak żeby korzystał z DMA.

DMA, A CZAS ŻYCIA BUFORA

W języku C należy pamiętać o tym, jak działają zakresy żywotności zmiennych. Podczas tworzenia zmiennej lub tablicy o stałej wielkości, jej czas życia (*lifetime*) jest tak długi jak czas życia zakresu w którym została stworzona.

Dobrze prezentuje to poniższy przykład:

```c
int g = 10; // zmienna `g` jest globalna, będzie istnieć przez cały okres działania programu

void f() {
  int a = 42; // zmienna `a` jest stworzona w funkcji, tak więc będzie żyła podczas jej wykonywania a potem zniknie

  if (a > 10) {
    double d = 1.23; // zmienna `d` istnieje w ifie, więc po jego zakończeniu zniknie z pamięci
    printf("a = %d, d = %f\n", a, d); // OK - zmienna `a` istnieje bo `if` jest zagnieżdżony w funkcji
  }

  printf("d = %f, g = %d\n", d, g); // błąd - zmienna `d` nie istnieje w tym zakresie
}
```

Jest to bardzo ważne, ponieważ oznacza to że **należy pamiętać o tym żeby bufor z którego korzysta DMA istniał przez cały okres transakcji**.
Istnieją dwa miejsca w programie gdzie stworzona zmienna/bufor będzie istnieć przez cały okres jego działania - funkcja `main` oraz przestrzeń globalna. Jeśli nie możemy "zawęzić" zakresu istnienia bufora, ponieważ nie możemy przewidzieć deterministycznie że transakcja DMA zakończy się przed zakończeniem się zakresu w którym został stworzony dany bufor (co zazwyczaj ma miejsce przy transakcjach blokujących, transakcje asynchroniczne są najczęściej niedeterministyczne, chyba że je blokujemy ręcznie), to **należy bufory tworzyć w przestrzeni globalnej**, co da 100% pewność że będą istnieć w każdym momencie transakcji.

Próba zapisu do niealokowanej pamięci to *undefined behaviour* i może powodować trudne do wyśledzenia błędy w działaniu programu, a najczęściej błąd Hard Fault! Dane odczytane z niealokowanej pamięci są często "niedeterministyczne" i używanie ich również może powodować błędy!

Tworzymy sobie jednobajtowy bufor w globalnym zakresie i następujący handler przerwania:

```
// Gdzieś w globalnym zakresie tego samego pliku
uint8_t byteBuffer;

// Obsługa końca transmisji
void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart) {
    HAL_UART_Transmit(huart, &byteBuffer, 1, HAL_MAX_DELAY);
    HAL_GPIO_TogglePin(LD2_GPIO_Port, LD2_Pin);
    HAL_UART_Receive_DMA(huart, &byteBuffer, 1);
}
```

A następnie w funkcji `main` uruchamiamy odbiór za pomocą funkcji `HAL_UART_Receive_DMA`

```
HAL_UART_Receive_DMA(&huart2, &byteBuffer, 1);
```

Działa to dokładnie tak samo jak blokujący echo server, lub echo server na przerwaniach, z tą różnicą że obsługa seriala leży po stronie DMA a nie rdzenia.

Ten przykład jednak prezentuje tylko to, że używanie DMA może być tak samo łatwe jak używanie przerwań - co jest prawdą, ale DMA błyszczy w innych scenariuszach, głównie w takich gdzie trzeba obsłużyć więcej niż jeden bajt danych na raz. W kolejnych rozdziałach kursu będę prezentował obsługę peryferiów w trybie blokującym oraz DMA.

OBSŁUGA UARTA Z UŻYCIEM DMA ORAZ DETEKCJI WOLNEJ LINII

Wykorzystamy kod z poprzedniego rozdziału, ale zmodyfikujemy go trochę w taki sposób, żeby współpracował z DMA w trybie *circular*.

W trybie kołowym, DMA po dotarciu do końca bufora powtarza cały proces odczytu/zapisu zaczynając od początku bufora, do momentu w którym ręcznie go nie zatrzymamy. Przerwania są wysyłane w taki sam sposób jak przy normalnym trybie (w połowie bufora oraz na jego końcu). Ten tryb idealnie się nadaje do ciągłej obsługi peryferiów, czyli na przykład ciągłego odczytu z ADC lub UARTa bez potrzeby ingerowania procesora w restartowanie operacji.

Należy skonfigurować przerwanie *idle line* dla UARTa poprzez uruchomienie go i obsługę w handlerze globalnego przerwania UARTa, tak jak zostało to opisane w poprzedniej części kursu.

```
    /* Initialize all configured peripherals */
    MX_GPIO_Init();
    MX_DMA_Init();
    MX_USART2_UART_Init();
    /* USER CODE BEGIN 2 */
    __HAL_UART_ENABLE_IT(&huart2, UART_IT_IDLE);
    /* USER CODE END 2 */
```

```
void USART2_IRQHandler(void) {
    /* USER CODE BEGIN USART2_IRQn 0 */

    /* USER CODE END USART2_IRQn 0 */
    HAL_UART_IRQHandler(&huart2);
    /* USER CODE BEGIN USART2_IRQn 1 */
    if (__HAL_UART_GET_FLAG(&huart2, UART_FLAG_IDLE) == SET) {
        __HAL_UART_CLEAR_IDLEFLAG(&huart2);
        handle_uart_message(); // Tą funkcję musimy sami napisać
    }

    /* USER CODE END USART2_IRQn 1 */
}
```

Następnie, musimy pomyśleć co należy zrobić w momencie kiedy

- Bufor jest pełny w połowie
- Bufor jest pełny
- Nastąpiło przerwanie idle line

Z racji że DMA będzie pracowało w trybie kołowym, to nasz bufor do którego zostaną zapisane dane stanie się buforem kołowym. Jeśli przyjdzie do niego ilość danych większa od samego bufora, to część z tych danych zostanie nadpisana. Można tego uniknąć na dwa sposoby:

1. Zapewnienie bufora o wielkości większej od największego komunikatu jaki może przyjść
2. Kopiowanie odebranej wiadomości do innego bufora, z poziomu którego zostanie ona obsłużona

Opcja numer jeden zajmie nam mniej pamięci i teoretycznie jest nieco prostsza w implementacji, więc ją zaprezentuję. Na długą metę, w niektórych przypadkach opcja numer dwa może okazać się lepsza - na przykład kiedy chcemy trzymać zawartość wiadomości przez dłuższy okres i mieć kontrolę nad jej nadpisaniem.
W przypadku tego rozwiązania nie musimy też obsługiwać przerwania połowy/pełnego bufora, ponieważ wiadomość nigdy nie przekroczy wielkości bufora. Przy opcji numer 2, byłby to mus ponieważ wiadomość mogła by przepełnić bufor co spowodowało by utratę części danych.

Załóżmy że wiadomości jakie będziemy wysyłać nie będą przekraczać 128 bajtów, więc tworzymy globalny bufor o takiej wielkości.

```
/* Private variables ---------------------------------------------------------*/
/* USER CODE BEGIN PV */
uint8_t messageBuffer[128];
/* USER CODE END PV */
```

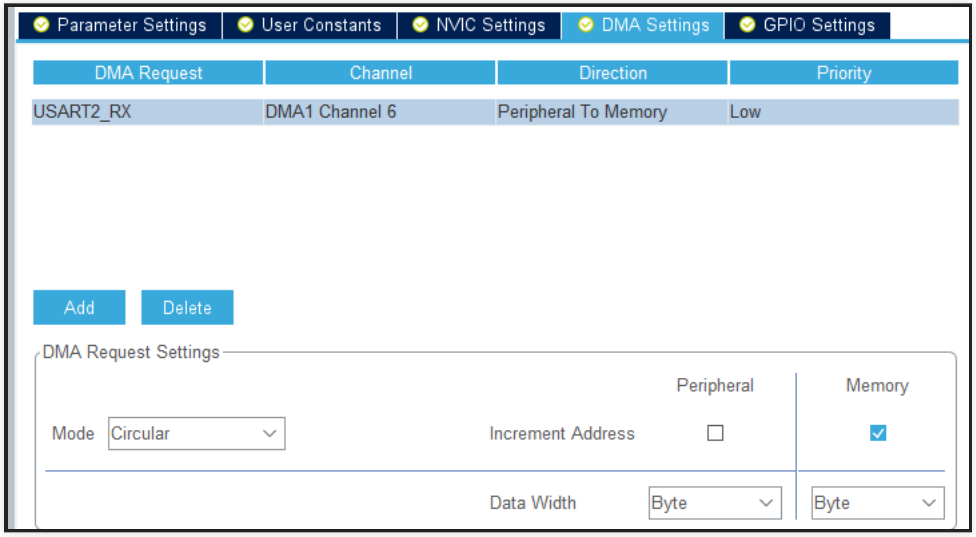
Żeby zmienna globalna była widoczna w innym pliku z kodem, należy ją tam zadeklarować z keywordem `extern`

Teraz, zostało nam jedynie napisać funkcję `handle_uart_message`.
Na start, spróbujmy po prostu wysłać zawartość bufora (sugeruję zrobić to po prostu poprzez `HAL_UART_Transmit`)

Funkcja `handle_uart_message` powinna wyglądać następująco:

```
void handle_uart_message() {
    HAL_UART_Transmit(&huart2, messageBuffer, sizeof messageBuffer, HAL_MAX_DELAY);
}
```

Zostało nam tylko uruchomić DMA w trybie kołowym. W Device Configuration Toolu zmieniamy tryb DMA z *Normal* na *Circular* i uruchamiamy odczyt za pomocą funkcji `HAL_UART_Receive_DMA`




```
HAL_UART_Receive_DMA(&huart2, messageBuffer, sizeof messageBuffer);
```

Jak widzimy, do bufora trafiają wszystkie dane i po jego zapełnieniu nowe dane nadpisują stare dane zaczynając od początku bufora. Teraz, musimy naszą wiadomość z tego bufora "wyciągnąć". Żeby to zrobić, musimy wiedzieć gdzie się zaczyna i gdzie się kończy, oraz obsłużyć sytuację gdy następuje przeskok z końca bufora do jego początku.

Z pomocą może przyjść nam makro `__HAL_DMA_GET_COUNTER` które zwraca ilość bajtów pozostałych do końca bufora.

Dodatkowo, możemy zapisywać offset na którym zakończyła się poprzednia wiadomość, dzięki czemu jesteśmy w stanie obliczyć jej długość. Możemy też, zamiast offsetu, użyć do tego wskaźnika na bufor.

```c
extern uint8_t messageBuffer[128];
uint8_t* messageStartPtr = messageBuffer;
uint8_t const* messageBufferEndPtr = messageBuffer + sizeof messageBuffer;
// [...]
/* USER CODE BEGIN 1 */
void handle_uart_message() {
    // Stwórz wskaźnik na koniec wiadomości
    uint8_t* messageEndPtr = messageBuffer
            + (sizeof messageBuffer) - __HAL_DMA_GET_COUNTER(&hdma_usart2_rx);

    // Sprawdź czy nastąpiło przepełnienie bufora czy nie
    if (messageStartPtr <= messageEndPtr) {
        // Bufor nie wrócił na początek, więc po prostu wyświetl wiadomość
        // Nie użyjemy printfa ponieważ chcemy wyświetlić konkretną ilość bajtów
        HAL_UART_Transmit(&huart2, messageStartPtr,
                (messageEndPtr - messageStartPtr), HAL_MAX_DELAY);
    } else {
        // DMA wróciło na początek bufora
        // Wyświetl najpierw początek wiadomości, a potem jej koniec
        HAL_UART_Transmit(&huart2, messageStartPtr,
                messageBufferEndPtr - messageStartPtr, HAL_MAX_DELAY);
        HAL_UART_Transmit(&huart2, messageBuffer, messageEndPtr - messageBuffer,
        HAL_MAX_DELAY);
    }
    messageStartPtr = messageEndPtr;
}
/* USER CODE END 1 */
```

W ten sposób stworzyliśmy echo server który obsługuje UARTa za pomocą bufora kołowego.

Jest to jeden z najbardziej ergonomicznych sposobów obsługi UARTa. Działa on również w momencie w którym zatrzymamy procesor za pomocą debugera - jeśli UART odbierze komunikat, to po wznowieniu pracy przerwanie *idle line* zostanie obsłużone przez procesor który będzie w stanie odczytać komunikat ponieważ DMA go w tle skopiowało do bufora.

RĘCZNE UŻYWANIE DMA

W przypadku kiedy chcemy wykonać customową operację z użyciem DMA, z pomocą przychodzą dwie funkcje:

- `HAL_DMA_Start` - uruchamia transfer z użyciem DMA w trybie polling
- `HAL_DMA_Start_IT` - uruchamia transfer z użyciem DMA w trybie przerwań

W trybie polling, żeby dowiedzieć się czy transfer się zakończył należy sprawdzić status DMA lub użyć funkcji `HAL_DMA_PollForTransfer` która zablokuje program do momentu zakończenia operacji. Nie ma to zazwyczaj większego sensu, ponieważ mija się to z celem używania DMA, dlatego zazwyczaj wykorzystuje się DMA w trybie przerwań.

Żeby skonfigurować ręcznie transfer DMA, musimy użyć funkcji `HAL_DMA_Init` która przyjmuje adres struktury `DMA_InitTypeDef`, która zawiera informacje na temat konfiguracji kanałów DMA, trybu pracy i adresy funkcji callbacków.

PODSUMOWANIE: KIEDY UŻYWAĆ DMA?

Wtedy, kiedy jesteśmy w stanie i przyniesie nam to widoczne korzyści. DMA pozwala na odciążenie procesora, co z kolei pozwala mu na wykonanie większej ilości zadań jednocześnie, lub zmniejsza pobór mocy jeśli nie musi nic robić podczas pracy DMA.

Nie będziemy zawsze w stanie go użyć, nie tylko przez fakt że istnieją sprzętowe ograniczenia (ilość i ułożenie kanałów, nie wszystkie peryferia wspierają DMA), ale również przez to że czasami nie przyniesie nam to benefitów - na przykład kiedy mamy do czynienia z obsługą małych ilości danych i jednoczesnym wykonywaniem logiki w połączeniu z I/O, ale w większości przypadków odciążenie procesora przez DMA jest zalecanym rozwiązaniem.

DODATKOWE MATERIAŁY

- AN4031 - Używanie DMA w STM32F2, STM32F4 i STM32F7. Jeden z najbardziej kompletnych dokumentów o DMA o
  https://www.st.com/content/ccc/resource/technical/document/application_note/27/46/7c/ea/2d/91/40/a9/DM00