KURS STM32

Wojciech Olech

CZĘŚĆ VI: TIMERY

Timery w mikrokontrolerach mają dwie podstawowe funkcje: stworzenie podstawy czasowej (czyli liczenie czasu), oraz generowanie sygnałów PWM.

Tworzenie podstawy czasowej jest użyteczne kiedy chcemy wykonywać pewną czynność z określonym interwałem, chcemy zmierzyć czas, lub chcemy taktować peryferia/*system operacyjny* (RTOS) z daną częstotliwością.

Sygnał PWM jest bardzo użyteczny do sterowania elementami z określoną mocą - najczęściej stosowany jest do sterowania silnikami elektrycznymi.

RODZAJE TIMERÓW

W mikrokontrolerach STM32 istnieje kilka rodzajów timerów:

- **Prosty timer** (***Basic timer***): zazwyczaj 16-bitowy, posiada jedynie funkcjonalność tworzenia podstawy czasu (co umożliwia taktowanie innych peryferiów lub triggerowanie przerwań z określoną częstotliwością). Nie posiada żadnych pinów I/O.
- **Timer ogólnego przeznaczenia** (***General purpose timer***): 16 lub 32-bitowy timer który posiada funkcjonalność prostego timera oraz posiada wyprowadzenia I/O które umożliwiają mu m. in. generowanie sygnału PWM, pomiar częstotliwości zewnętrznego sygnału, obsługę enkoderów i czujników halla.
- **Zaawansowany timer** (***Advanced timer***): Posiada funkcjonalność timerów ogólnego przeznaczenia ale dodatkowo zaawansowane feature'y, takie jak generowanie trzech komplementarnych sygnałów z wstawianiem czasu przestoju (*dead time*).
- **Timer wysokiej rozdzielczości** (***High resolution timer***): Specjalny timer stosowany w serii STM32F3 który służy stricte do sterowania silnikami.

- **Timer niskiej mocy (*Low-power timer*)**: Timer stworzony do zastosowań low-power, może pracować w prawie każdym trybie pracy/snu procesora, a nawet bez wewnętrznego źródła zegarowego (co pozwala mu pracować na przykład jako licznik impulsów). Tego rodzaju timery mają możliwość wybudzania mikrokontrolera ze snu.
- **Zegar czasu rzeczywistego (*Real-Time Clock*)**: Nie jest to stricte timer, ale znajduje się w tej samej kategorii. Taktowany specjalnym oscylatorem (optymalnie o częstotliwości 32.768kHz) pozwala na dokładny długotrwały pomiar czasu rzeczywistego, bez dryftu czasu który występuje w zegarach taktowanych innymi oscylatorami.

KONFIGURACJA TIMERA - LICZNIK CZASU

Żeby uruchomić timer, należy w Device Configuration Toolu wybrać interesujący nas timer, a następnie go skonfiugurować w interesującym nas trybie. Na start, skonfigurujemy timer tak żeby wywoływał co kilka sekund przerwanie które będzie migało diodą i wysyłało wiadomość po serialu.

Dla naszego zastosowania możemy wybrać dowolny timer. Wybierzmy TIM10 ponieważ jest to prosty timer, nie potrzebujemy niczego lepszego.

Posiada on kilka pól które musimy skonfigurować.

Dwa podstawowe pola jakie nas interesują to

- **Prescaler** - preskaler dzieli nam zegar którym taktujemy timer przez jego wartość. Pozwala on uzyskać dowolną częstotliwość taktowania zegara.
- **Counter Period** - Określa on maksymalną wartość licznika przy której zaczyna on liczyć od zera i przy której wywoła się *update event*.

Żeby poprawnie obliczyć częstotliwość wywołania *update eventu*, czyli przerwania licznika, musimy dodatkowo wiedzieć z jaką częstotliwością jest taktowany nasz timer. Zegary są taktowane za pomocą zegara APB do którego są podłączone, w widoku konfiguracji zegarów są to częstotliwości z pól "APB1 timer clocks" i "APB2 timer clocks". Domyślnie powinny być one taktowane częstotliwością rdzenia, czyli 84MHz dla STM32F401.

Żeby policzyć częstotliwość wywoływania *update eventu*, należy skorzystać z tego wzoru:

$$f_{\mathrm{UpdateEvent}}[\mathrm{Hz}] = \frac{\mathrm{TimerClock}}{(\mathrm{Prescaler}+1)(\mathrm{Period}+1)}$$

Załóżmy że chcemy migać diodą co sekundę. Nasz zegar timera wynosi 84 000 000Hz, musimy więc dobrać odpowiedni preskaler i okres zegara.

Jedna sekunda to 1Hz. Nasz preskaler i okres są wartościami 16-bitowymi, więc maksymalna wartość to 65535.
Żeby otrzymać jakąś *ładną* wartość zegara dla której łatwo będzie ustawić okres, możemy podzielić główny zegar przez 42000, co da nam 2000Hz. Chcąc otrzymać 1Hz, musimy 2000Hz podzielić przez 2000. Te dwie wartości (minus jeden) to nasz preskaler i okres. Finalnie dostajemy wzór

$$f_{\mathrm{UpdateEvent}}[\mathrm{Hz}] = \frac{84000000}{(41999+1)(1999+1)} = 1\mathrm{Hz}$$

OBSŁUGA TIMERA W KODZIE

Po ustawieniu timera i wygenerowaniu kodu możemy przejść do jego obsługi w kodzie.

Przerwanie timera powinno zostac wygenerowane w pliku `stm32f4xx_it.c` Obsługuje ono z użyciem HALa generyczne przerwanie timera, wywołując odpowiedni callback handler. W tym przypadku, interesujący nas callback to `void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef* htim)` który musimy zdefiniować w naszym kodzie.

```
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef* htim) {
    // Sprawdź czy przerwanie przyszło od naszego timera
    if (htim == &htim10) {
      // Zmień stan diody (dla Nucleo makro GPIO będzie inne)
      HAL_GPIO_TogglePin(BOARD_LED_GPIO_Port, BOARD_LED_Pin);
    }
}
```

Żeby uruchomić timer, musimy wywołać funkcję `HAL_TIM_Base_Start_IT`, na przykład po inicjalizacji timera.

```
  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  MX_TIM10_Init();
  /* USER CODE BEGIN 2 */
  HAL_TIM_Base_Start_IT(&htim10);
  /* USER CODE END 2 */
```

Możemy teraz uruchomić wrzucić nasz program na mikrokontroler i zobaczyć efekt - dioda powinna zmieniać swój stan co sekundę.

Dodatkowo, w okienku "Live Expressions" możemy stworzyć nową pozycję z nazwą struktury-uchwytu do naszego timera (`htim10` w tym przypadku). Pozwoli to na podgląd wartości w tej strukturze w czasie rzeczywistym, włącznie z rejestrami - możemy zobaczyć czy nasz timer pracuje i jak szybko zmienia się jego wartość.

ZMIANA OKRESU LICZNIKA

HAL ułatwia inicjalizację, uruchamianie i podstawową obsługę peryferiów ale jest też abstrakcją na wiele rzeczy, o których niekoniecznie możemy wiedzieć, patrząc pobieżnie na to co oferuje. Żeby dostać się do niektórych, bardziej zaawansowanych możliwości peryferiów, trzeba spojrzeć na niskopoziomowe funkcje HALa oraz rejestry.

Timery mają trzy ważne rejestry które nas interesują i które sterują działaniem timera:

- **PSC** - Prescaler Register, w tym rejestrze trzymana jest wartość preskalera jaką ustawiliśmy
- **ARR** - Auto-Reload Register, w tym rejestrze trzymana jest wartość okresu jaką ustawiliśmy
- **CNT** - Counter Register, w tym rejestrze jest trzymana aktualna wartość licznika timera

CNT i ARR są ze sobą ściśle powiązane, bo w momencie w którym CNT osiąga wartość zachowaną w ARR następuje wywołanie przerwania licznika (update event, PeriodElapsedCallback).

Zmieniając ARR i PSC możemy w takim razie zmieniać częstotliwość licznika (oraz wypełnienie sygnału PWM, ale o tym później). Należy jednak pamiętać o jednej ważnej rzeczy - **overflow**. Nie możemy zmniejszać wartości ARR w dowolnym momencie, bo może się okazać że zmniejszyliśmy ją w chwili kiedy CNT jest większy od nowego ARR, co spowoduje overflow i błędne działanie programu. Optymalnym rozwiązaniem jest zmiana ARR przy wywołaniu update eventu, czyli naszego callbacka PeriodElapsedCallback, bo wtedy mamy pewność że CNT == 0

Użyjemy do tego HALowego makra `__HAL_TIM_SET_AUTORELOAD`. **Nie należy zmieniać rejestrów ręcznie dopóki nie mamy 100% pewności że musimy to zrobić, makra są zawsze lepszym rozwiązaniem ponieważ zawsze poprawnie obsłużą wszystkie wymagane rejestry i struktury - `__HAL_TIM_SET_AUTORELOAD` poza zmianą ARR zmienia też pole Period w strukturze timera!**

Możemy na przykład użyć tego makra żeby przyspieszać i spowalniać szybkość migania diody. Przetestujmy ten kod:

```
/* USER CODE BEGIN 4 */
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim) {
    // Sprawdź czy przerwanie przyszło od naszego timera
    if (htim == &htim10) {
        HAL_GPIO_TogglePin(BOARD_LED_GPIO_Port, BOARD_LED_Pin);

        uint16_t arr_value = __HAL_TIM_GET_AUTORELOAD(htim);
        if (arr_value >= 200) {
            __HAL_TIM_SET_AUTORELOAD(htim, arr_value - 100);
        } else if (arr_value >= 100) {
            __HAL_TIM_SET_AUTORELOAD(htim, arr_value - 10);
        } else {
            __HAL_TIM_SET_AUTORELOAD(htim, 2000);
        }
    }
}
/* USER CODE END 4 */
```

Alternatywnie, możemy użyć opcji preloadingu ARR poprzez włączenie w Device Configuratorze "auto-reload preload".

Preloading ARR działa w ten sposób, że wartość jaką zapiszemy do ARR zostanie zapisana do ukrytego rejestru, z którego zostanie skopiowana do ARR dopiero w momencie wystąpienia UEV (update eventu). Pozwala nam to na zapis do ARR w dowolnym momencie, bez potrzeby martwienia się o aktualną wartość countera.

ZATRZYMYWANIE TIMERÓW

Żeby zatrzymać timer, musimy użyć funkcji przeciwnej, do tej którą dany timer uruchomiliśmy

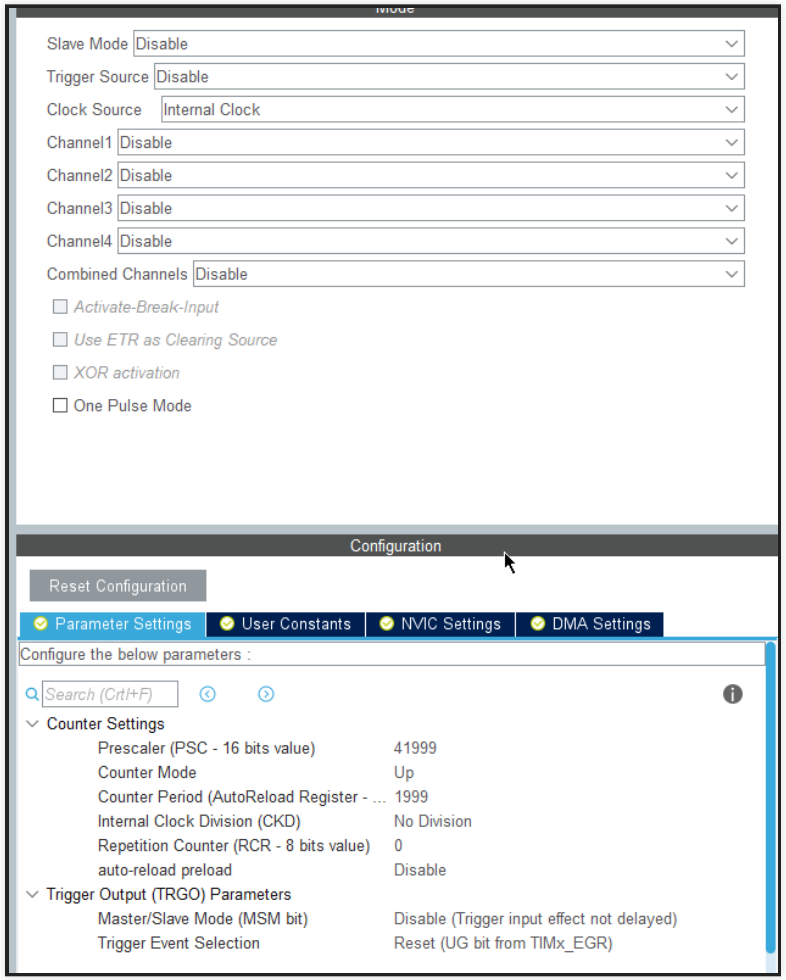
- `HAL_TIM_Base_Stop`
- `HAL_TIM_Base_Stop_IT`
- `HAL_TIM_Base_Stop_DMA`

UŻYWANIE TIMERÓW Z DMA

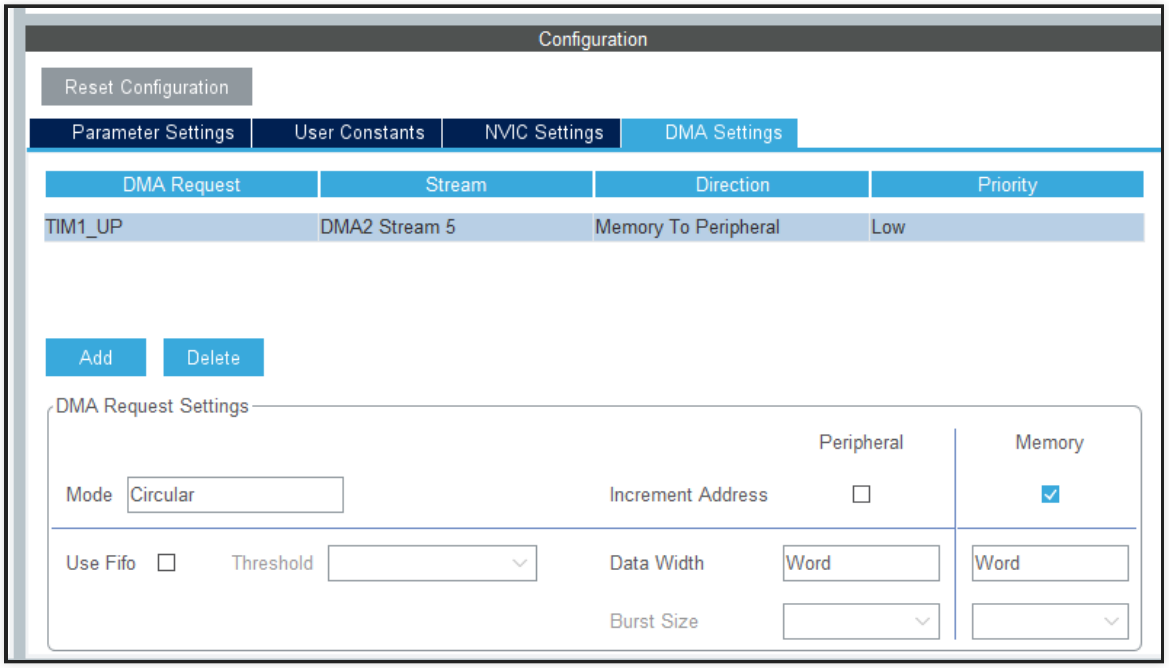
Timery, szczególnie te które pracują z wysokimi częstotliwościami, generują dużo przerwań które procesor musi obsługiwać.
Jeśli timer ma taktować proste czynności, pokroju zmiany stanów na pinach, to można odciążyć procesor poprzez użycie DMA. DMA może taktować kopiowanie pamięci na podstawie UEV zegara.

W tym celu, należy skorzystać z timera ogólnego przeznaczenia, ponieważ proste timery nie są połączone z DMA, oraz funkcji `HAL_DMA_Start` do wystartowania DMA.

W celu skonfigurowania timera ogólnego przeznaczenia, wracamy do Device Configuration Toola i wybieramy jeden z timerów od TIM1 do TIM5. Uruchamiamy go w trybie wewnętrznego zegara (Clock Source -> Internal Clock) i ustawiamy preskaler oraz okres licznika na taki, jak w poprzednim przykładzie, ponieważ ten timer powinien być taktowany tym samym zegarem co poprzedni.


Slave Mode Disable
Trigger Source Disable
Clock Source Internal Clock
Channel1 Disable
Channel2 Disable
Channel3 Disable
Channel4 Disable
Combined Channels Disable
Activate-Break-Input
Use ETR as Clearing Source
XOR activation
One Pulse Mode
Configuration
Reset Configuration
Parameter Settings
User Constants
NVIC Settings
DMA Settings
Configure the below parameters :
Search (Crtl+F)
Counter Settings
Prescaler (PSC - 16 bits value) 41999
Counter Mode Up
Counter Period (AutoReload Register - ... 1999
Internal Clock Division (CKD) No Division
Repetition Counter (RCR - 8 bits value) 0
auto-reload preload Disable
Trigger Output (TRGO) Parameters
Master/Slave Mode (MSM bit) Disable (Trigger input effect not delayed)
Trigger Event Selection Reset (UG bit from TIMx_EGR)

Następnie przechodzimy do zakładki *DMA Settings* i konfigurujemy linię DMA. Chcemy mieć skonfigurowane zapytanie `TIMx_UP` które będzie działać w kierunku *Memory to Peripheral*, w trybie kołowym, gdzie szerokość danych to *Word*.



Taka konfiguracja jest nam potrzebna, ponieważ

- `TIMx_UP` to request który wywołuje się w momencie UEV timera.
- Chcemy kopiować zawartość pamięci do rejestrów GPIO, więc potrzebujemy trybu *Memory to Peripheral*
- Chcemy żeby transfer dział się w nieskończoność, stąd tryb kołowy
- Wielkość rejestru GPIO w którym znajduje się stan pinów wynosi 32 bity, czyli jedno słowo. Tak więc DMA musi być dopasowane do przesyłania danych w słowach.

Finalnie, możemy przejść do napisania kodu który będzie obsługiwać port GPIO używając DMA taktowanego timerem.

GPIO trzyma stany pinów w rejestrze `ODR` (Output Data Register), tak więc DMA będzie musiało pisać do tego rejestru.
Potrzebujemy tablicy w której będziemy trzymali stany rejestru, zakładając że używamy diody podłączonej do pinu 13, możemy wykorzystać taką tablicę:

```
uint32_t pinStates[2] = { 0x00, GPIO_PIN_13 };
```

Następnie, możemy uruchomić timer oraz DMA i powiązać DMA z timerem:

```
HAL_TIM_Base_Start(&htim1);
// HAL_DMA_Start wymaga adresów w postaci int32_t, nie wskaźników - casty są w porządku
HAL_DMA_Start(&hdma_tim1_up, (uint32_t) pinStates, (uint32_t) &GPIOC->ODR, 2);
__HAL_TIM_ENABLE_DMA(&htim1, TIM_DMA_UPDATE); // Włącz timer z powiązanym do niego kanałem DMA, który będzie triggerowany przy UEV
```

Ten program praktycznie nie korzysta z rdzenia procesora. Timer i DMA działają w trybie pollingu, więc nie generują żadnych przerwań, których nie potrzebujemu ponieważ peryferia się obsługują, ze względu na to że timer wystarczy uruchomić, a DMA pracuje w trybie cyklicznym.

Można zauważyć, że ten kod generuje falę prostokątną na pinie IO. Możemy też podać dłuższą tablicę ze stanami pinów, co pozwoli nam na cykliczne generowanie stanów o różnych długościach na różnych pinach. W związku z tym, poprzez zwiększenie częstotliwości zegara, możemy wygenerować **do 32 sygnałów PWM** z różnymi okresami i taką samą częstotliwością za pomocą **jednego** timera bez ingerencji rdzenia procesora - ilość kanałów jest ograniczana tym, ile pinów jest fizycznie dostępnych dla danego portu GPIO.

Jest to dobry sposób jeśli chcemy generować dużą ilość sygnałów PWM, ale jeśli nie potrzebujemy aż takiej ilości to lepiej skorzystać z możliwości generowania sygnałów PWM wbudowanych w timery.

Jeden timer ogólnego przeznaczenia może generować do 4 sygnałów PWM, z różnymi okresami ale taką samą częstotliwością.

Zanim jednak przejdziemy do obsługi PWM, wyjaśnię kilka funkcji jakie mogą spełniać timery ogólnego przeznaczenia

FUNKCJE TIMERÓW OGÓLNEGO PRZEZNACZENIA

SLAVE MODE

Timery w STM32 są ze sobą wewnętrznie połączone, co pozwala łączyć je w hierarche slave-master. Jeden master może taktować kilka slave'ów. Slave może być również taktowany z zewnętrznego źródła zegarowego, oraz uruchamiany na podstawie sygnału zewnętrznego

- Reset mode - rosnące zbocze wybranego wejścia triggerującego restartuje licznik i generuje aktualizację rejestrów
- Gated mode - licznik pracuje kiedy wejście triggerujące jest w stanie wysokim. Licznik nie jest resetowany po zatrzymaniu licznika, więc można go użyć do określenia okresu sygnału.
- Trigger mode - licznik startuje w momencie wykrycia zbocza rosnącego na wejściu triggerującym
- External clock mode - licznik jest taktowany zewnętrznym zegarem

Trigger source pozwala wybrać źródło które będzie triggerowało licznik. Źródła **ITRx** to wewnętrzne połączenia z innych zegarów zawartych w mikrokontrolerze.

WYBÓR ZEGARA

Istnieją dwa tryby taktowania licznika

- Z zegara wewnętrznego - licznik taktowany jest z szyny APB do której jest podłączony
- Z zegara zewnętrznego - tutaj dostępne są dwa pod-tryby
  - External Clock Mode 1 - konfigurowalny jako tryb *slave* pozwala na taktowanie licznika zegarem wewnętrznym lub zewnętrznym. Możemy w tym trybie ustawić na jakim zboczu ma nastąpić zmiana wartości *countera*, oraz filtr triggera który ustala ilość wykrytych zboczy pod rząd wymaganą do zmiany *countera* w liczniku.
  - External Clock Mode 2 - wybierany w *clock source*, pozwala taktować licznik za pomocą zewnętrznego zegara z preskalerem i filtrem cyfrowym.

TRYBY KANAŁÓW

W typowym liczniku ogólnego przeznaczenia, każdy kanał może mieć następujące tryby:

- Input capture - w tym trybie, licznik zapisze swój stan do odpowiedniego rejestru i wygeneruje przerwanie w momencie w którym wykryje zmianę na pinie GPIO. Odczytując wartość licznika można określić czas między dwoma zboczami. Można też użyć tego trybu do obsługi przycisków z debouncingiem w postaci odpowiedniej częstotliwości licznika i filtra cyfrowego.
- Output compare - w tym trybie, licznik wywołuje przerwanie w momencie kiedy osiągnie ustawioną wartość. Może on również zmienić stan na pinie wyjściowym (na wysoki, niski, lub go zanegować, albo wymusić stan bez względu na wartość licznika).
- PWM generation - w tym trybie, licznik pracuje jako generator sygnału PWM

Dodatkowo, licznik można skonfigurować w trybie One Pulse Mode - pozwala on na generowanie sygnału o określonej długości po określonym delayu w momencie wykrycia eventu (OPM można włączyć w trybie Input Capture lub Output Compare)

COMBINED CHANNELS

Liczniki ogólnego przeznaczenia posiadają też specjalne funkcjonalności które zbierają dane z kilku kanałów i na ich podstawie modyfikują wartość licznika lub generują sygnał na jego wyjściu.

- Encoder mode - w trym trybie, licznik jest działa jako licznik impulsów z enkodera kwadraturowego. Wartość licznika zmienia się odpowiednio na podstawie sygnału z enkodera.
- Combined PWM mode - w tym trybie, licznik może generować sygnał PWM na podstawie wartości operacji AND lub OR na stanach dwóch kanałów tego samego licznika, skonfigurowanych w trybie output compare.
- Hall Sensor Mode - w tym trybie, licznik umożliwia zliczanie danych z trzech czujników Halla (zazwyczaj z sygnałami przesuniętymi o 120 stopni), poprzez trzy kanały połączone bramką XOR.

DODATKOWE USTAWIENIA TIMERÓW OGÓLNEGO PRZEZNACZENIA

W ustawieniach licznika, oprócz ustawień które były dostępne w prostych licznikach, mamy również kilka dodatkowych opcji

- Counter mode - liczniki proste mogą liczyć tylko "do góry", natomiast te ogólnego przeznaczenia mogą liczyć "do góry", "w dół" oraz w trybie "center aligned". Dobrze przedstawia to poniższa grafika:

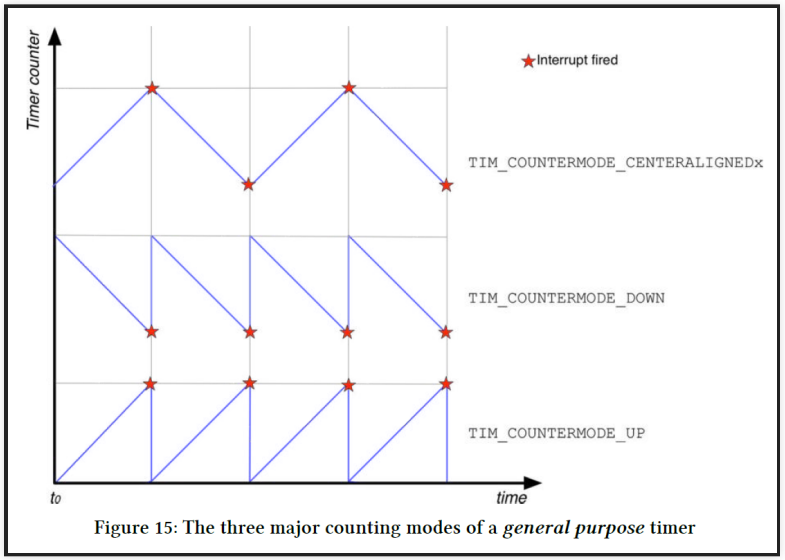


Figure 15: The three major counting modes of a *general purpose* timer

Źródło grafiki: Mastering STM32

- Repetition counter - ta wartość określa ile razy licznik ma osiągnąć ustaloną wartość, zanim zostanie wywołane przerwanie UEV, zastosowanie tej opcji zmienia równanie na częstotliwość występowania przerwań licznika w następujący sposób:

$$f_{\mathrm{UpdateEvent}}[\mathrm{Hz}] = \frac{\mathrm{TimerClock}}{(\mathrm{Prescaler}+1)(\mathrm{Period}+1)(\mathrm{RepetitionCounter}+1)}$$

PRAKTYCZNE PRZEDSTAWIENIE DODATKOWYCH FUNKCJI TIMERA

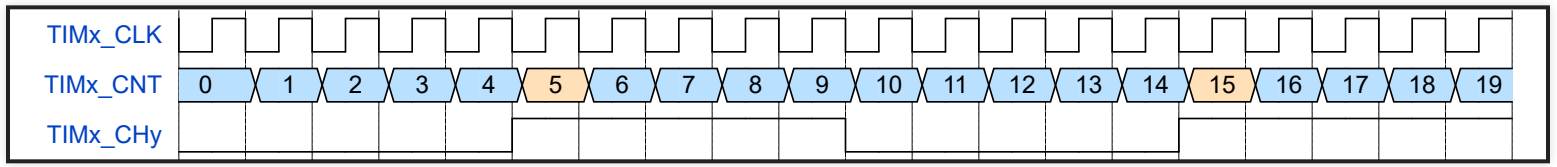
Zaczniemy od trybu **Input capture**. Napiszemy przykładowy program który będzie liczył czas naciśnięcia przycisku - niestety na płytce Nucleo-F401RE nie mamy żadnego kanału timera dostępnego na pinie wbudowanego przycisku, więc będziemy go symulować z użyciem kabelka połączeniowego, lub - jeśli istnieje taka możliwość - zewnętrznego przycisku.

Skonfigurujmy pin **PA8** (który odpowiada pinowi **D7** na Nucleo) jako `TIM1_CH1` poprzez ustawienie kanału pierwszego `TIM1` w trybie *Input Capture direct mode*. Następnie, należy skonfigurować rezystor podciągający na pinie `PA8`

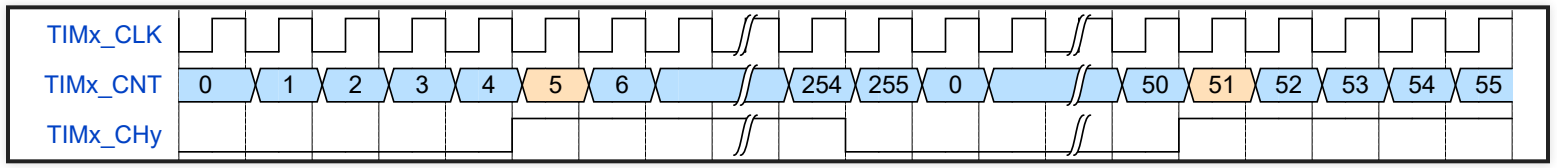
- Jeśli kabelek/przycisk podłączymy po drugiej stronie do GND, to należy skonfigurować rezystor w trybie pull-up i "kliknięcie przycisku" wygeneruje zbocze opadające na pinie
- Jeśli kabelek/przycisk podłączymy po drugiej stronie do 3.3V, to należy skonfigurować rezystor w trybie pull-down i "kliknięcie przycisku" wygeneruje zbocze narastające na pinie

Następnie, należy skonfigurować odpowiednio preskaler i ARR.
W tym trybie, musimy pamiętać o jednej, ważnej zasadzie - **okres między kolejnymi UEV licznika to maksymalny okres między wykrytymi zboczami jaki licznik może poprawnie wykryć, jeśli chcemy obliczyć ich częstotliwość**. Bierze się to ze względu na to że potrzebujemy dwóch pomiarów żeby zmierzyć okres między nimi, a jeśli drugi pomiar nastąpi po dłuższym okresie niż jeden pełny okres licznika, to nie będziemy w stanie obliczyć poprawnie czasu jego trwania.

Przykładowy timing diagram poprawnie skonfigurowanego licznika, gdzie okres badanego sygnału jest mniejszy od okresu między przepełnieniami:



Przykładowy timing diagram źle skonfigurowanego licznika, gdzie okres badanego sygnału jest większy od okresu między przepełnieniami licznika:



Możemy jeszcze skonfigurować na jakim zboczu ma być wywoływane przerwanie w *Polarity Selection*. Jeśli chcemy wykrywać czas między kliknięciami, interesuje nas zbocze narastające lub opadające, w zależności od podłączenia przycisku (*Rising/Falling Edge*). Jeśli interesuje nas czas przez który przycisk był wciśnięty (lub puszczony), to wybieramy *Both Edges*, co spowoduje wywołanie przerwania przy każdej zmianie stanu.

Prescaler Division Ratio pozwoli podzielić zegar licznika przez dany dzielnik, *Input Filter* pozwala skonfigurować filtr cyfrowy, którego wartość określa minimalny czas trwania sygnału jaki zostanie wykryty (ergo, działa trochę jak kolejny dzielnik częstotliwości).

W tym trybie najlepszą opcją jest skonfigurowanie ARR na maksymalną możliwą wartość (`0xFFFF` dla 16-bit ARR) i sterowanie preskalerem i pozostałymi dzielnikami w celu uzyskania odpowiedniej częstotliwości.

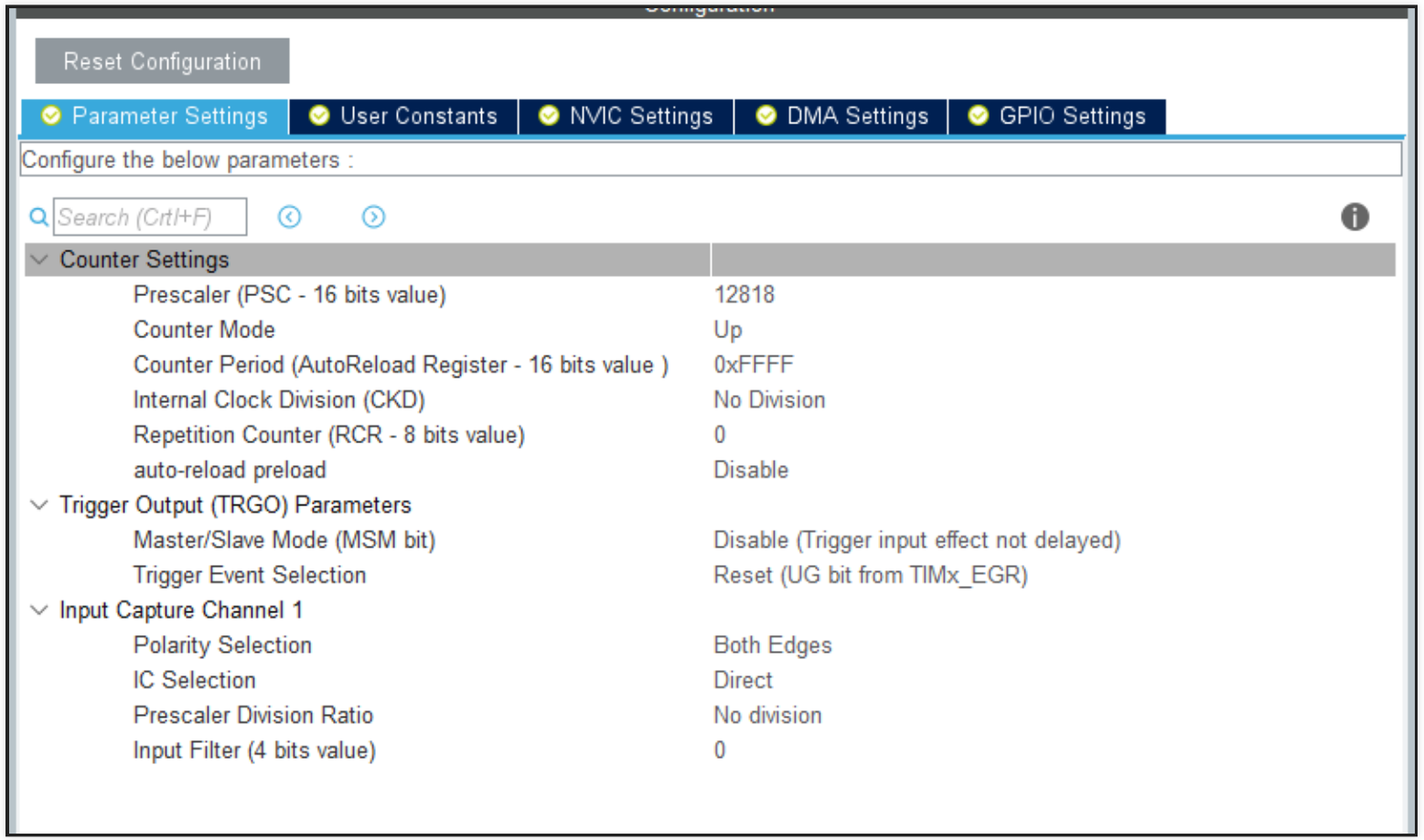
Załóżmy że maksymalny okres jaki będziemy chcieli zmierzyć wynosi 10 sekund. Przez te 10 sekund, licznik musi przeliczyć od 0 do 2^{16}. W takim przypadku, interesuje nas częstotliwość taktowania licznika równa

$$f_{\text{tim}} = \frac{2^{16}}{10} = 6553.6\text{Hz}$$

Znając częstotliwość końcową, możemy obliczyć preskaler

$$\text{Prescaler} = \frac{f_{\text{clk}}}{f_{\text{tim}}} = \frac{84000000}{6553.6} \approx 12818$$

Nie jest to ładna, okrągła wartość, ale błąd będzie niezauważalny. Wartość preskalera zmieści się w jego rejestrze (16-bit), więc nie potrzebujemy dodatkowych dzielników. W przypadku kiedy wartość preskalera obliczona w ten sposób jest zbyt duża, należy wykorzystać dodatkowe dzielniki - *internal clock division*, *repetition counter* i *prescaler division ratio*


Reset Configuration
Parameter Settings
User Constants
NVIC Settings
DMA Settings
GPIO Settings
Configure the below parameters :
Search (Crtl+F)
Counter Settings
Prescaler (PSC - 16 bits value) 12818
Counter Mode Up
Counter Period (AutoReload Register - 16 bits value) 0xFFFF
Internal Clock Division (CKD) No Division
Repetition Counter (RCR - 8 bits value) 0
auto-reload preload Disable
Trigger Output (TRGO) Parameters
Master/Slave Mode (MSM bit) Disable (Trigger input effect not delayed)
Trigger Event Selection Reset (UG bit from TIMx_EGR)
Input Capture Channel 1
Polarity Selection Both Edges
IC Selection Direct
Prescaler Division Ratio No division
Input Filter (4 bits value) 0

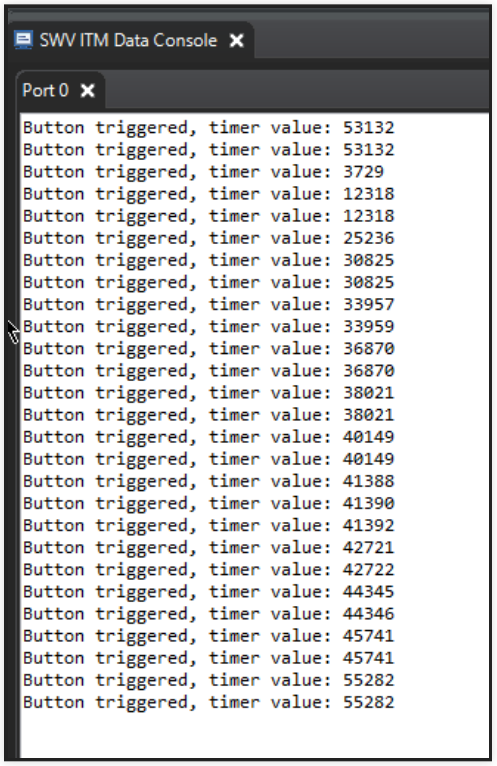
Na koniec, musimy pamiętać o tym, żeby zaznaczyć w ustawieniach NVIC timera generowanie kodu obsługującego przerwanie *capture compare* i możemy przejść do pisania kodu.

W momencie, w którym timer wykryje skonfigurowane zbocze, wygeneruje przerwanie na podstawie którego HAL wywoła funkcję `void HAL_TIM_IC_CaptureCallback(TIM_HandleTypeDef* htim)` - musimy więc w jej ciele zawrzeć kod który będzie przeliczał i zapisywał czas kliknięcia (lub między kliknięciami).

Na początek, sprawdźmy czy przerwanie działa i wykonuje się poprawnie. Za pomocą funkcji `HAL_TIM_ReadCapturedValue` możemy sprawdzić przy jakiej wartości licznika został wciśnięty przycisk - timer automatycznie kopiuje wartość rejestru `CNT` do odpowiedniego rejestru `CCR` po wykryciu zbocza.

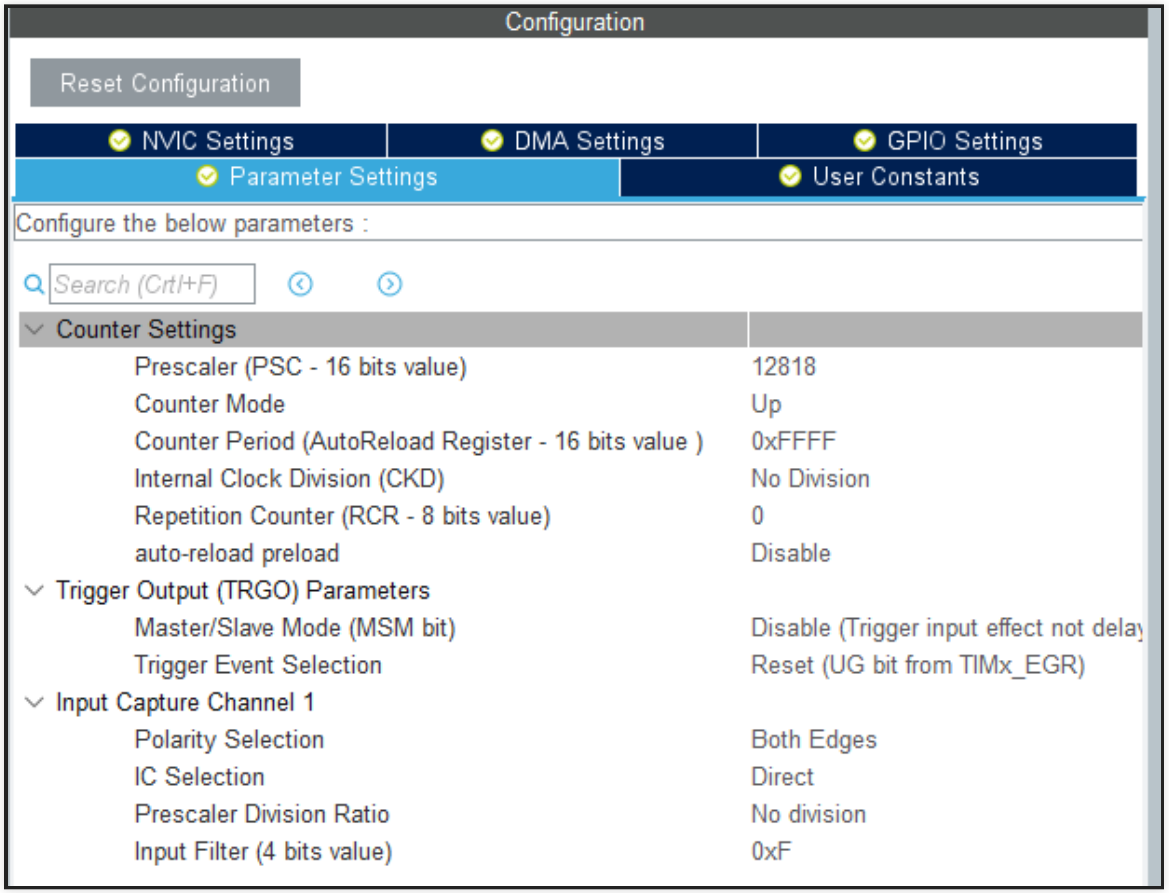
```
void HAL_TIM_IC_CaptureCallback(TIM_HandleTypeDef *htim) {
    if (htim == &htim1) {
        HAL_GPIO_TogglePin(LD2_GPIO_Port, LD2_Pin);
        printf("Button triggered, timer value: %ld\n", HAL_TIM_ReadCapturedValue(htim, TIM_CHANNEL_1));
    }
}
```

Następnie wystarczy uruchomić timer za pomocą funkcji `HAL_TIM_IC_Start_IT(&htim1, TIM_CHANNEL_1);` i uruchomić program, żeby zobaczyć czy działa poprawnie.



Jeśli używamy kabelka do symulowania przycisku, to można zauważyć że bardzo często występuje tutaj problem w postaci powtarzającego się wykrycia zbocza. To problem opisywany jako *drgania styków*, i powstaje przez mechaniczne "rozłączanie i łączenie się" elementów przewodzących prąd co powoduje powstawanie szumów, których zbocza są wykrywane jako zmiany stanów.

Żeby wyeliminować efekt drgania styków, należy zastosować *debouncing*. Możemy w tym celu użyć cyfrowego filtra dostępnego na wejściu timera. Ustawmy wartość tego filtra w Device Configuration Toolu na `0xF` (czyli maksimum, co spowoduje ze zbocze zostanie wykryte dopiero wtedy, kiedy wejście utrzyma swój stan przez 16 cykli zegara) i przetestujmy działanie naszego programu ponownie.


Configuration
Reset Configuration
NVIC Settings
DMA Settings
GPIO Settings
Parameter Settings
User Constants
Configure the below parameters :
Search (Crtl+F)
Counter Settings
Prescaler (PSC - 16 bits value) 12818
Counter Mode Up
Counter Period (AutoReload Register - 16 bits value) 0xFFFF
Internal Clock Division (CKD) No Division
Repetition Counter (RCR - 8 bits value) 0
auto-reload preload Disable
Trigger Output (TRGO) Parameters
Master/Slave Mode (MSM bit) Disable (Trigger input effect not delay
Trigger Event Selection Reset (UG bit from TIMx_EGR)
Input Capture Channel 1
Polarity Selection Both Edges
IC Selection Direct
Prescaler Division Ratio No division
Input Filter (4 bits value) 0xF

Jeśli nadal występuje efekt drgania styków, to istnieją dwie inne możliwości walki z nim

1. Dalsze spowalnianie zegara, co spowoduje że filtr cyfrowy będzie miał więcej czasu na reakcję. Zmniejszy nam to jednak wynikową dokładność pomiaru czasu
2. Filtrowanie drgań w kodzie, poprzez odrzucanie wartości które różnią się od poprzednio zmierzonych wartości o mniej niż ustaloną *deltę*

Ja zastosuję tutaj opcję numer dwa, jako dodatkowy krok przy obliczaniu czasu wciśnięcia przycisku.

Wiedząc że licznik pracuje z częstotliwością ~6553.6Hz, jeden cykl zegara wykonuje się przez ~$153\mu S$. Przyjmijmy, że chcemy żeby nasz debouncing trwał $1mS$, musimy więc ignorować $\frac{1mS}{153\mu S} \approx 7$ cykli po wykryciu zbocza.

Możemy to również liczyć w kodzie - za pomocą funkcji `HAL_RCC_GetPCLKxFreq` możemy zdobyć częstotliwość bazową zegara, preskaler znajduje się w rejestrze `PSC`. *Należy mieć na uwadze to, że musimy pobrać częstotliwość szyny PCLK do której jest wpięty interesujący nas zegar, oraz - jeśli chcemy być na 100% bezpieczni przed zmianami w drzewie zegarowym - sprawdzić dzielniki/mnożniki na danej szynie. Poniższy kod to uproszczenie.*

```
/* USER CODE BEGIN 0 */
double getTimerTickPeriod() {
    // Z block diagramu (datasheet) wiemy że TIM1 jest podłączony do APB2
    // Jeśli PCLK != timer clock, należy sprawdzić dzielniki/mnożniki w rejestrach
    // Nie ma makra do odczytu preskalera, więc czytamy go ręcznie
    double timerFrequency = (double) HAL_RCC_GetPCLK2Freq() / (double) (htim1.Instance->PSC + 1);
    return 1. / timerFrequency;
}

uint32_t getDebouncingDelta(uint32_t debouncingMillis) {
    double deboucingPeriod = (double) debouncingMillis / 1000.;
    return (uint32_t) ceil(deboucingPeriod / getTimerTickPeriod());
}
/* USER CODE END 0 */
```

Zapisujemy deltę debouncingu, okres timera i ostatnią odczytaną wartość w zmiennych globalnych. Deltę debouncingu najlepiej obliczyć i zapisać w funkcji `main`, bo nie musimy jej liczyć więcej niż raz podczas działania programu. Mając te zmienne i dane, możemy przystąpić do napisania kodu przerwania w którym będziemy obliczać czas naciśnięcia.

Możemy od razu wystartować licznik w trybie przerwań za pomocą funkcji `HAL_TIM_IC_Start_IT`.

```
/* USER CODE BEGIN PV */
uint32_t lastTriggerValue = 0; // ostatnia zapisana wartość licznika
uint32_t const debouncingTimeMs = 1; // czas debouncingu w milisekundach
uint32_t debouncingDelta = 0; // ilość cykli jakie należy zignorować
double timerTickPeriod = 0;
/* USER CODE END PV */
// [...]
int main() {
    // [...]
    /* USER CODE BEGIN 2 */
    timerTickPeriod = getTimerTickPeriod();
    debouncingDelta = getDebouncingDelta(debouncingTimeMs);
    HAL_TIM_IC_Start_IT(&htim1, TIM_CHANNEL_1);
    /* USER CODE END 2 */
    // [...]
}
```

Handler przerwania może wyglądać następująco:

```
void HAL_TIM_IC_CaptureCallback(TIM_HandleTypeDef *htim) {
    if (htim == &htim1) {
        uint32_t triggerValue = HAL_TIM_ReadCapturedValue(htim, TIM_CHANNEL_1);
        uint32_t triggerDelta = 0; // różnica między czasami zboczy
        double triggerTime = 0.; // okres między czasami zboczy w sekundach

        // 2 przypadki - nastąpił lub nie nastąpił overflow licznika
        if (triggerValue >= lastTriggerValue) {
            triggerDelta = triggerValue - lastTriggerValue;
        } else {
            // Obliczamy okres poprzez dodanie różnicy do overflowu i wartości po overflowie
            triggerDelta = (__HAL_TIM_GET_AUTORELOAD(htim) - lastTriggerValue)
                    + triggerValue;
        }

        // debouncing
        if (triggerDelta >= debouncingDelta) {
            triggerTime = triggerDelta * timerTickPeriod;
            printf("Czas od ostatniej zmiany stanu: %lf sekund\n", triggerTime);
            lastTriggerValue = triggerValue;
        }
    }
}
```

Ten przykład pokazuje jednak czas między zmianami stanów - żeby sprawdzić ile czasu upłynęło od wciśnięcia przycisku, trzeba dodać mechanizm sprawdzający z którym zboczem mamy do czynienia. To zostawiam jednak do implementacji czytelnikowi, ponieważ jest kilka sposobów na rozwiązanie tego problemu.

Jak widzimy, napisaliśmy też całkiem dużą funkcję przerwania. W praktyce, powinniśmy optymalnie unikać takich konstrukcji i lepiej jest zrobić to na dwa inne sposoby:

1. W funkcji przerwania zapisywać jedynie wartość licznika i powiadomić o tym główną pętlę programu lub jeden z wątków RTOSa
2. Użyć DMA

Opcja numer jeden jest dobra, jeśli musimy powiązać jakąś logikę z momentem wywołania przerwania. Natomiast DMA można użyć kiedy chcemy mieć po prostu ciągły zapis wartości licznika bezpośrednio do pamięci, w momencie wykrycia zbocza.

INPUT CAPTURE W TRYBIE DMA

Funkcja `HAL_TIM_IC_Start_DMA` pozwala na uruchomienie timera w trybie *Input Capture* z DMA. Funkcja ta przyjmuje cztery argumenty:

1. Uchwyt do timera
2. Kanał timera który został skonfigurowany w trybie IC
3. Wskaźnik na bufor z danymi
4. Wielkość bufora

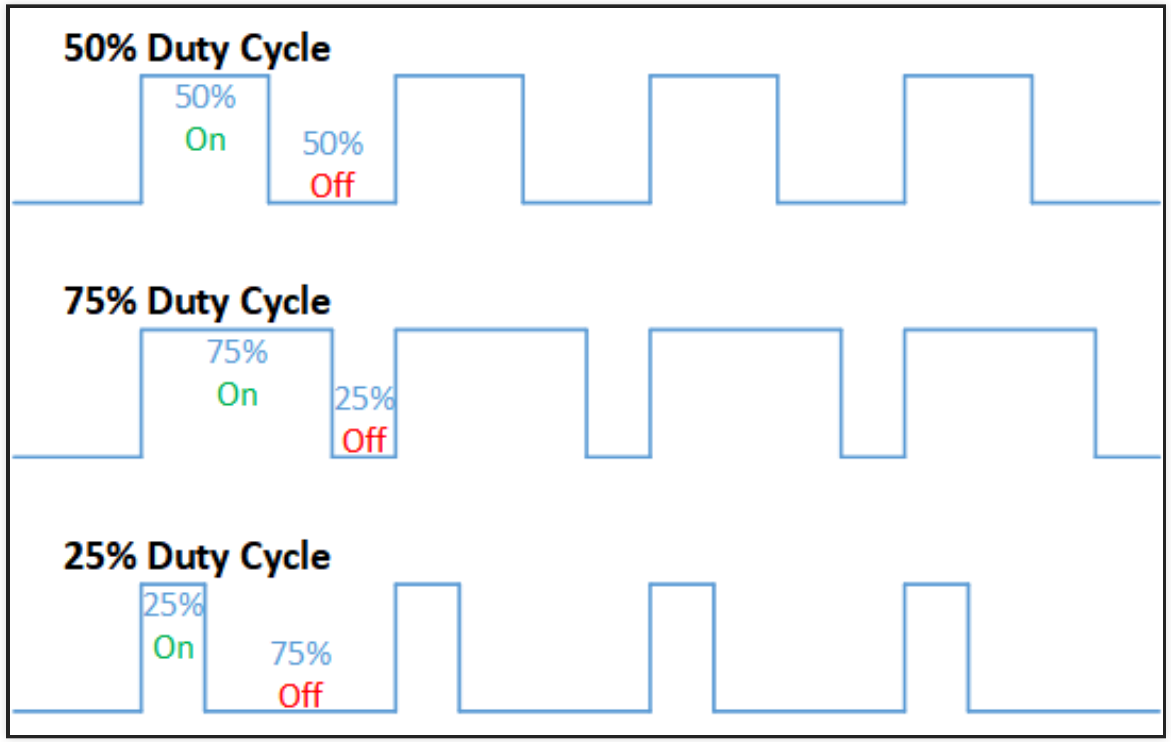
Po uruchomieniu timera w trybie IC z DMA, DMA automatycznie będzie po wykryciu zbocza zapisywać wartości timera do podanego bufora. Przed użyciem należy pamiętać o uprzednim skonfigurowaniu DMA, albo poprzez Device Configuration Toola, albo ręcznie.

TRYB OUTPUT COMPARE

W trybie Output Compare, kanał licznika generuje UEV za każdym razem, kiedy jego wartość osiąga wartość określoną jako *Pulse*. Dodatkowo, jest w stanie zmieniać stan pinu wyjściowego do którego jest podłączony, co pozwala na przykład na generowanie sygnału prostokątnego, lub kilku sygnałów prostokątnych przesuniętych w fazie między sobą na każdym z kanałów licznika.

TRYB PWM

W trybie PWM licznik jest w stanie generować sygnał PWM. Sygnał PWM to fala prostokątna o zmiennym wypełnieniu (tzw. *duty cycle*).



Sterowanie wypełnieniem sygnału PWM pozwala na sterowanie **średnią wartością napięcia sygnału**. Przykładowo, mając PMW o amplitudzie 3.3V i wypełnieniu 50%, średnia wartość napięcia sygnału to $3.3\mathrm{V} \cdot 50\% = 1.65\mathrm{V}$. W przypadku wypełnienia równego 20% - mamy $3.3\mathrm{V} \cdot 20\% = 660\mathrm{mV}$, i tak dalej.

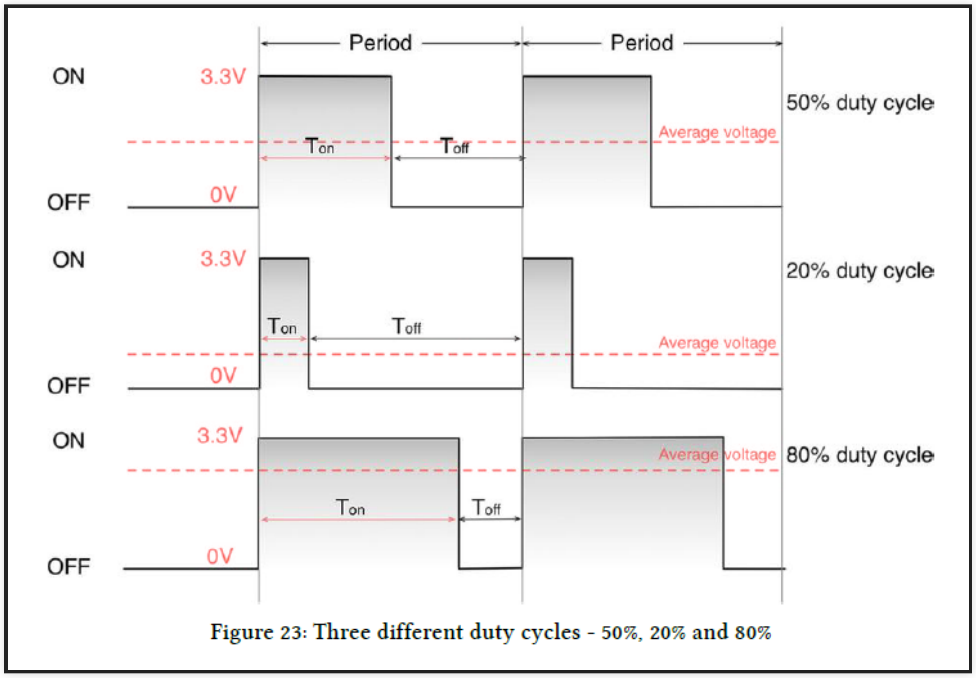


Figure 23: Three different duty cycles - 50%, 20% and 80%

Źródło: Mastering STM32

Sygnał PWM jest często używany do

- Sterowania diodami LED - zmienianie jasności
- Sterowania silnikami DC - zmienianie podawanej mocy
- Sterowania serwomotorami - zmiana kąta/szybkości obrotowej
- Regulowania napięcia wyjściowego w przetwornicach step-up

Jak również w wielu innych zastosowaniach.

KONFIGURACJA TRYBU PWM

W przypadku płytki Nucleo-F401RE, pin do którego jest podłączona dioda ma alternatywną funkcjonalność bycia pierwszym kanałem timera 2. Skonfigurujemy więc ten kanał w trybie PWM żeby sterować jasnością diody.

Uruchamiany go w trybie *PWM Generation CH1*. TIM2 jest 32-bitowy, więc pozwala na uzyskanie wysokiej rozdzielczości sygnału PWM. Częstotliwość timera w tym przypadku nie jest znacząca, wystarczy że będzie na tyle duża żeby nie było widać pojedynczych cykli, dlatego sugeruję co najmniej 100Hz.

Finalną rozdzielczość PWMa określa wartość ARR (Auto-Reload Register). Wypełnienie jest określane stosunkiem wartości *Pulse* do ARR. Sugeruję ustawić ARR na 999 dla naszego przykładu.

PWM posiada dwa tryby:

- W trybie 1, wyjście jest w stanie wysokim tak długo, jak wartość licznika jest mniejsza od wartości *pulse* PWMa, a po przekroczeniu jej przechodzi w stan niski.
- Tryb 2 działa na odwrót - wyjście jest w stanie niskim tak długo, jak wartość licznika jest mniejsza od wartości *pulse*.

W naszym przypadku, tryb PWMa nie ma znaczenia, należy tylko pamiętać że w trybie 1 zmniejszanie wartości *pulse* będzie zmniejszać *duty cycle* i jasność diody, a w trybie drugim na odwrót.

Opcja *Output compare preload* pozwala na zmianę wartości *Pulse* w dowolnym momencie. Tryb PWM jest w rzeczywistości specjalnym przypadkiem trybu output compare, więc wykorzystuje te same rejestry.

Po skonfigurowaniu PWMa możemy przejść do kodu. Żeby zmienić wartość wypełnienia, używamy makra `__HAL_TIM_SET_COMPARE`. Żeby dostać maksymalną wartość jaką możemy ustawić (100% wypełnienia), używamy makra `__HAL_TIM_GET_AUTORELOAD`. Do odczytania aktualnego *duty cycle* możemy użyć `__HAL_TIM_GET_COMPARE`, lub - bezpieczniejszej wersji - `HAL_TIM_ReadCapturedValue`.

```c
/* USER CODE BEGIN 2 */
HAL_TIM_PWM_Start(&htim2, TIM_CHANNEL_1);
uint16_t dutyCycle = HAL_TIM_ReadCapturedValue(&htim2, TIM_CHANNEL_1);
/* USER CODE END 2 */

/* Infinite loop */
/* USER CODE BEGIN WHILE */
while (1) {
    /* USER CODE END WHILE */
    /* USER CODE BEGIN 3 */
    while (dutyCycle > 0) {
        __HAL_TIM_SET_COMPARE(&htim2, TIM_CHANNEL_1, --dutyCycle);
        HAL_Delay(1);
    }
    while (dutyCycle < __HAL_TIM_GET_AUTORELOAD(&htim2)) {
        __HAL_TIM_SET_COMPARE(&htim2, TIM_CHANNEL_1, ++dutyCycle);
        HAL_Delay(1);
    }
}
/* USER CODE END 3 */
```

Jeśli ARR i *Pulse* został ustawiony zgodnie z zaleceniami, to dioda powinna płynnie pulsować, gasnąc i rozjaśniając się w ciągu 4 sekund. Na pierwszy rzut oka, jeśli rozdzielczość PWMa to 1000 kroków, a każdy krok zmienia się co 1 milisekundę, to w teorii jeden cykl rozjaśniania i jeden cykl gaśnięcia powinien trwać sekundę, czyli całe "mignięcie" dwie. W praktyce, zegar wewnętrzny STMa nie jest wybitnie stabilny, przez co 1ms delay nie trwa w rzeczywistości 1ms, tylko trochę dłużej. Problem znika po użyciu dłuższego delaya (2ms lub więcej).

Alternatywnie, można taktować zmiany PWMa za pomocą drugiego timera, lub drugiego kanału tego samego timera, co zostawiam do wykonania jako bonusowe ćwiczenie.

DODATKOWE MATERIAŁY:

- STM32 cross-series timer overview: https://www.st.com/content/ccc/resource/technical/document/application_r
- Funkcjonalność timerów w STM32L4: https://www.st.com/content/ccc/resource/training/technical/product_training/c4/1b/56/83/3a/a1/47/64/STM32L
- General-purpose timer cookbook: https://www.st.com/content/ccc/resource/technical/document/application_nc