2 Obsługa prostych czujników i urządzeń wykonawczych mikroprocesorowego systemu automatyki przy wykorzystaniu systemu przerwań.

2.1 Cel

Celem ćwiczenia jest zapoznanie studentów z metodami obsługi najprostszych czujników i urządzeń wykonawczych z poziomu mikrokontrolera wykorzystując do tego mechanizm przerwań. Sterowanie elementami wykonawczymi odbywać się będzie przy użyciu fali PWM oraz „włączania" i „wyłączania" elementów, natomiast pomiary będą dokonywane poprzez pomiar napięcia na pinach mikrokontrolera. Uwaga skupiona będzie jednak na wykorzystaniu mechanizmu przerwań do „jednoczesnego", regularnego wykonywania zadań oraz na zastąpieniu mechanizmu aktywnego oczekiwania (odpytywania) na rzecz obsługi przerwań.

2.2 Wykorzystane mechanizmy

Przerwania: Mechanizm przerwań, jak sama nazwa wskazuje, pozwala na natychmiastowe przerwanie pracy mikrokontrolera w celu obsługi zdarzenia, które wymaga niezwłocznej reakcji. Wstrzymane może być zarówno wykonywanie głównej pętli programu (tj. funkcja `main`) jak też i samych funkcji obsługujących przerwania. O tym, które z przerwań spowoduje wstrzymanie wykonania kodu na rzecz swojej funkcji obsługi przerwania, które trafi do kolejki obsługiwanych, a które nie zostanie obsłużone wcale decyduje kontroler przerwań NVIC (*Nested Vectored Interrupt Controller*). W dalszej części pojawiać się będzie pojęcie funkcji lub programu obsługi przerwań (*Interrupt Service Routine*), tj. funkcji, która wywoływana jest w wyniku zgłoszenia przerwania, jest to odpowiedź mikrokontrolera na zdarzenie zewnętrzne. Warto mieć na uwadze, że przeciwieństwem przerwań jest odpytywanie. Przykładem odpytywania jest poniższy kod:

```
while(ADC_GetFlagStatus(ADC1, ADC_FLAG_EOC) == RESET);   // czekaj na koniec pomiaru
```

gdzie mikrokontroler nieustannie odpytuje przetwornik, czy zakończył pomiar. Czas, który poświęcany (wręcz marnowany) jest na oczekiwanie na odpowiedź, można by wykorzystać znacznie lepiej, np. wykonując dalsze operacje, które nie wymagają informacji otrzymanych z przetwornika. Efekt ten osiągniemy właśnie dzięki przerwaniom.

Tabela 63 z pliku RM0008 zawiera rozpisaną tablicę wektorów przerwań – podane są: pozycja w tablicy, priorytet, możliwość zmiany priorytetu, akronim, opis oraz adres programu obsługi tego przerwania. Kilka z nich jest wyjątkowo interesujących z punktu widzenia później wykonywanego ćwiczena: SysTick, EXTI0, ADC1_2, EXTI9_5 oraz TIM2, TIM3, TIM4. Z tabeli tej można odczytać pod jakimi adresami w pamięci znajdują się poszczególne programy obsługi przerwań, a więc pod jakimi adresami powinny znaleźć się funkcje, które mają zostać wywołane w momencie nastąpienia odpowiedniego zdarzenia zewnętrznego. Implementacja funkcji służącej do obsługi przerwania będzie realizowana poprzez implementację funkcji o odpowiedniej deklaracji (nazwa wraz z argumentami i typem zwracanym), ponieważ w trakcie inicjalizacji pamięci mikrokontrolera przypisane zostały im już odpowiednie adresy w pamięci.

Bezpośrednio przed rozpoczęciem obsługi przerwania, procesor chroni środowisko przerwanego programu, wysyłając zawartości odpowiednich rejestrów na stos – dzięki temu po zakończeniu wykonania kodu odpowiedzialnego za obsługę przerwania może powrócić do przerwanych zadań. Stąd wynika, że o ile nie zostaną zmodyfikowane zawartości wspomnianych rejestrów, mikrokontroler będzie kontynuował pracę

wcześniej przerwanego programu od miejsca, w którym nastąpiło przerwanie. Świadomość tego jest niezmiernie ważna szczególnie w sytuacji, gdy zarówno w funkcji obsługującej przerwanie jak i w pętli głównej operujemy na tych samych zmiennych w pamięci. Dla przykładu, jeśli w trakcie wykonywania następującej pętli głównej:

```
unsigned char x = 10;
while(1){
    if (x > 0) {
        // (1)
        --x;
    } else {
        break;
    }
}
```

nie nastąpi przerwanie, to po odpowiednio dużej liczbie iteracji, zmienna `x` będzie równa 0. Jeśli natomiast nastąpi przerwanie, które np. wyzeruje zmienną `x`, wtedy (w zależności od momentu w którym przerwanie nastąpi) pętla główna wykona mniejszą lub większą liczbę iteracji niż gdyby przerwanie się nie pojawiło. Jeśli przerwanie nastąpiło przed sprawdzeniem warunku, lub po dekrementacji zmiennej `x` – liczba iteracji zmniejszy się lub pozostanie taka sama, a więc wykonane zostaną następujące operacje:

- sprawdzenie warunku `if (x > 0){ ...`, załóżmy, że `x` = 5, a więc warunek spełniony,
- dekrementacja zmiennej `--x;`, po którym zmienna `x` ma wartość 5-1, a więc `x` = 4,

!! tutaj następuje przerwanie, a więc wejście do funkcji obsługi przerwania:

1. wyzerowanie zmiennej `x`,
2. zakończenie funkcji obsługi przerwania,

- sprawdzenie warunku `if (x > 0){ ...`, który nie jest spełniony, bo `x` = 0 – wykonanie pętli jest przerywane (`break;`).

W przeciwnym razie wykonane zostaną następujące operacje:

- sprawdzenie warunku `if (x > 0){ ...`, załóżmy, że `x` = 5, a więc warunek spełniony,

!! tutaj następuje przerwanie, a więc wejście do funkcji obsługi przerwania:

1. wyzerowanie zmiennej `x`,
2. zakończenie funkcji obsługi przerwania,

- dekrementacja zmiennej `--x;`, po którym zmienna `x` ma wartość 0-1, a więc ze względu na użyty typ `unsigned char`, będzie to wartość 255,
- sprawdzenie warunku `if (x > 0){ ...`, który jest spełniony, bo `x` = 255 – pętla jest kontynuowana.

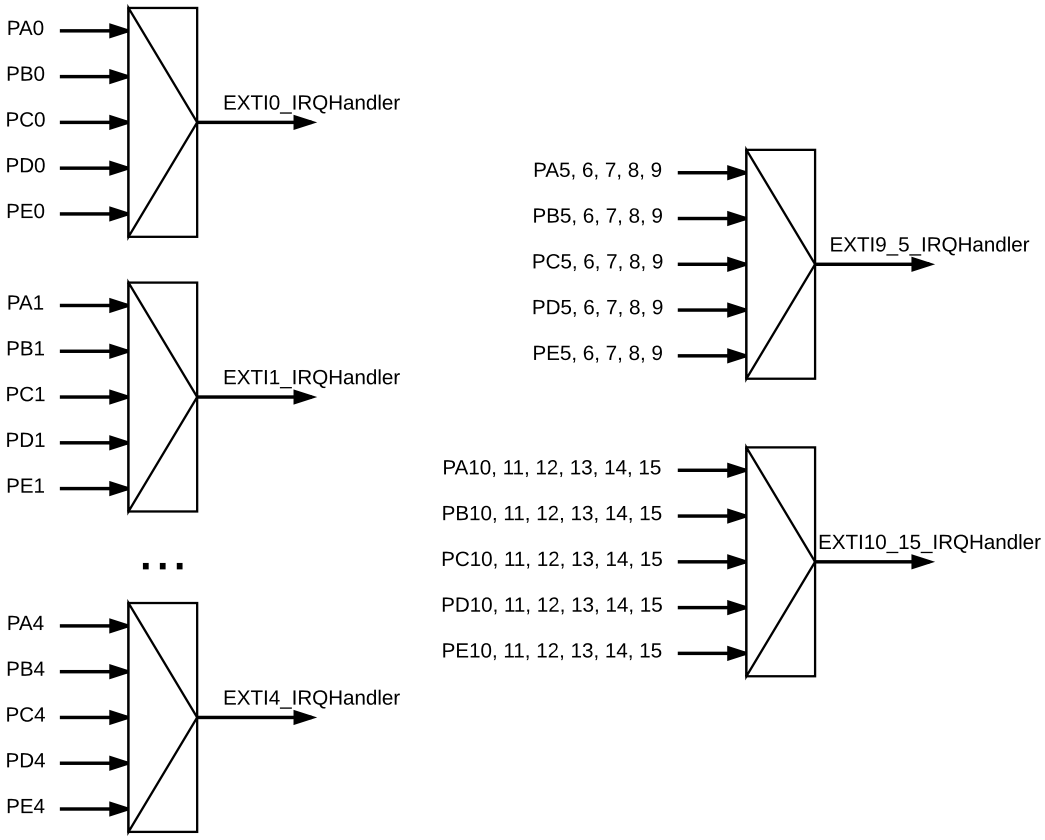
Jak widać działanie takiego programu jest zależne od momentu, w którym nastąpiło przerwanie. Aby zapewnić sobie, że zmienna w trakcie wykonania pewnego bloku programu nie zostanie zmodyfikowana przez wystąpienie przerwania, można posłużyć się mechanizmem monitorów, semaforów lub nawet na ten czas wyłaczyć przerwania, które mogą nam przeszkodzić.

Poza tym, że przerwania powodują zatrzymanie wykonania głównej pętli programu, mogą także powodować zatrzymanie wykonania funkcji obsługującej inne przerwanie. Z tego powodu zostały wprowadzone priorytety przerwań – im niższy priorytet przerwania tym jest ono ważniejsze. Warto tutaj zwrócić uwagę na wspomnianą tabelę 63 z pliku RM0008, gdzie widać, że najważniejszym przerwaniem jest przerwanie związane z resetowaniem mikrokontorlera, a niedaleko za nim znajduje się przerwanie związane z błędami. Co więcej – priorytetu tych przerwać nie można zmienić – zawsze będą ważniejsze od przerwań o dodatniej wartości priorytetu.

Przerwania mogą być obsługiwane natychmiast lub zaraz po obsłudze ważniejszych przerwań. W mikrokontrolerach rodziny STM32 decyzja zapada na podstawie numeru przerwania, a dokładniej poszczególnych jego bitów. Numer przerwania tworzą w rzeczywistości dwa pola, których długość można zmieniać: priorytet wywłaszczenia oraz pod-priorytet w obrębie priorytetu wywłaszczenia. Oba pola w sumie zapisane są na 4 bitach tworząc razem priorytet przerwania. Możliwości podziału priorytetu na wspomiane dwa pola jest 5: 0+4, 1+3, 2+2, 3+1, 4+0. Ostatni podział pozwala na wyłączenie całkowicie pod-priorytetów, co prowadzi do wywłaszczania wyłącznie na podstawie wartości priorytetu (jeśli wyższy priorytet, to następuje wywłaszczenie), który z resztą zajmowany jest w całości przez pole priorytet wywłaszczenia. Pierwszy podział powoduje, że wszystkie przerwania mają jednakowy priorytet wywłaszczenia, a co za tym idzie, w przypadku występowania wielu przerwań jednocześnie, wykonywane są kolejno, od tego z najniższą wartością pod-priorytetu, do tego z największą jego wartością. Tryb mieszany, czyli np. 2+2, pozwala na wyznaczenie przerwań, które muszą zostać obsłużone natychmiastowo (o mniejszym niż inne priorytecie wywłaszczenia), oraz takie, które mogą zostać obsłużone po innych przerwaniach o tym samym priorytecie (ustawiając odpowiednio pod-priorytet).

Odpowiednie i rozważne ustawianie wartości priorytetów jest kluczowe w każdym projekcie, który wykorzystuje mechanizm przerwań. W przeciwnym razie mogą nastąpić bardzo nieprzyjemnie sytuacje takie jak zagłodzenie lub zakleszczenie. Jednym z przypadków, gdzie następuje zakleszczenie jest złe ustawienie priorytetu przerwania SysTick, który ma za zadanie odmierzanie czasu opóźnienia. Jeśli wystąpi przerwanie, które ma wyższy priorytet, a jednocześnie wywołuje funkcję opóźnienia, następuje natychmiastowe wstrzymanie działania programu. Wynika to z faktu, że ponieważ opóźnienie bazuje na przerwaniu SysTick, które ma niższy priorytet niż obecnie obsługiwane, to nie ma możliwości, aby wywłaszczyło ono przerwanie o wyższym priorytecie. Z drugiej strony obsługiwane przerwanie oczekuje na zakończenie funkcji opóźnienia, co powoduje, że program zatrzymuje wykonywanie oczekując w nieskończoność w pętli.

Przerwania zewnętrzne: Układ EXTI (*External interrupt / event controller*) obsługuje zewnętrzne źródła przerwań – może on zgłosić przerwanie lub zdarzenie. Zdarzenia w przeciwieństwie do przerwań nie muszą być obsługiwane poprzez wywołanie funkcji obsługi – mogą one bezpośrednio wywoływać pewną reakcję, np. wybudzić procesor, rozpocząć przetwarzanie sygnału analogowego na cyfrowy, itp. Co więcej niektóre układy mogą generować wiele zdarzeń w ramach jednego przerwania. Przykładem takiego układu jest przetwornik analogowo cyfrowy. Może on generować jedno przerwanie, którego funkcja obsługi musi zawierać kod sprawdzający jakie zdarzenie je wywołało – tych jest kilka: *End of conversion*, *End of injec-*



Rysunek 23: Schemat przypisywania linii do zgłaszanego przerwania

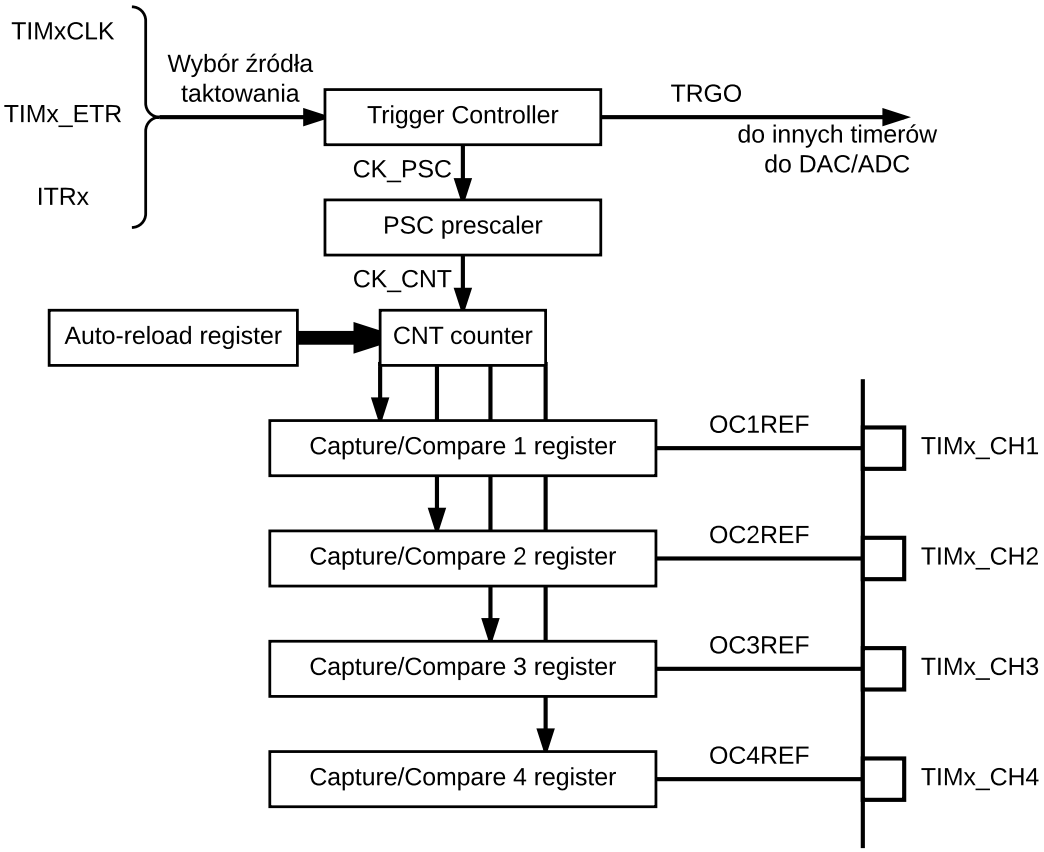
tion, *Analog watchdog event*. Obsługa zewnętrznych źródeł przerwań obejmuje między innymi wykrywanie zboczy sygnału wejściowego. Mogą być wykrywane zarówno zbocza narastające, opadające jak i jednocześnie oba wymienione. Aby wybrane zbocze generowało przerwanie, należy przypisać odpowiednią linię do zgłaszanego przerwania – przedstawia to rys. 23. Każda z linii od 0 do 4 zgłasza osobne przerwanie, są one jednak wspólne dla wszystkich portów, np. `PA0`, `PB0`, `PC0`, `PD0`, `PE0`, `PF0` zgłaszają jedno przerwanie `EXTI0_IRQ`. Linie od 5 do 9 zgłaszają wspólne przerwanie `EXTI9_5_IRQ` (wspólne dla wszystkich portów). Analogicznie linie od 10 do 15 zgłaszają przerwanie `EXTI10_15_IRQ`.

Timery: Timery, są to liczniki, których sygnałem wejściowym jest sygnał zegarowy. W mikrokontrolerach rodziny STM32 nie ma mowy o licznikach, lecz używa się właśnie pojęcia timer. W szczególnym przypadku timery służą do zliczania zboczy sygnału, który nie jest zegarowym (a więc działają jak zwykłe liczniki), lecz ponieważ ich głównym zastosowaniem jest odmierzanie czasu – także i w tym opracowaniu będzie używane pojęcie timer.

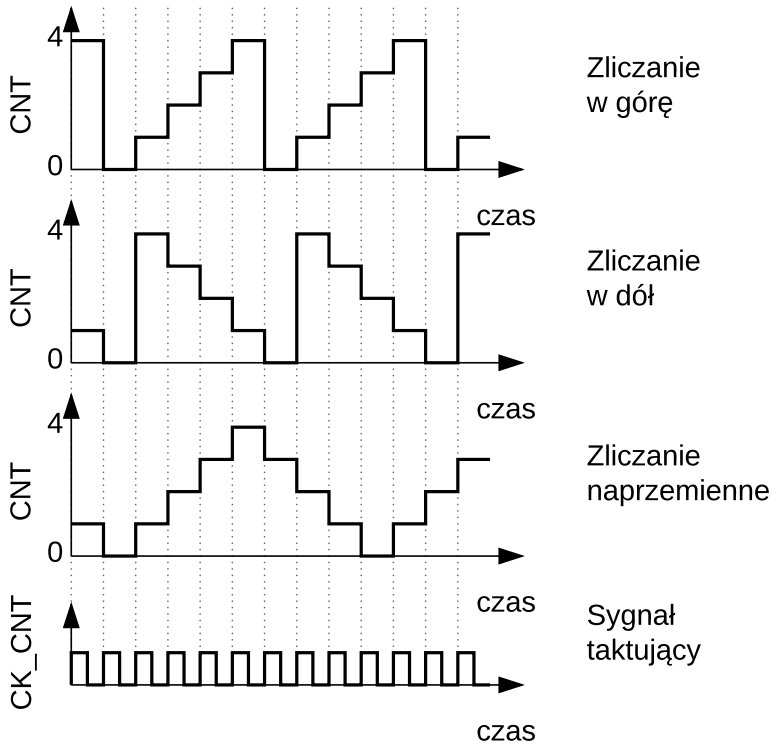
W mikrokontrolerach rodziny STM32 znaleźć można wiele rodzajów timerów. Począwszy od bardzo prostych (mających wyłącznie dwa sygnały zegarowe do wyboru), po niezmiernie skomplikowane (zazwyczaj TIM1 i TIM8, które oznaczane są jako *Advanced-control timers* i poświęcony jest im najczęściej osobny rozdział w dokumentacji).

W tym ćwiczeniu w centrum uwagi będą dwa typy timerów: *Cortex System Timer* (nazywany często *SysTick*) oraz *General-purpose timer*. Pierwszy z nich należy do najprostszych w obsłudze (co wynika z niewielkich jego możliwości) timerów – zlicza on w dół takty sygnału zegarowego HCLK lub sygnału HCLK/8 (tj. sygnał zegarowy HCLK spowolniony ośmiokrotnie). Timer ten zlicza od zadanej 24-bitowej wartości do zera, po czym przeładowuje swój licznik ponownie zadaną wcześniej wartością. Osiągnięcie zera przez ten timer powoduje wygenerowanie przerwania. Warto zwrócić uwagę, że timer ten doskonale się nadaje do wykorzystania w systemach czasu rzeczywistego, gdzie konieczny jest przydział kwantów czasu poszczególnym zadaniom – takie kwanty czasu mogą być wyznaczane właśnie przez ten bardzo prosty, lecz jakże przydatny timer.

Timery z kategorii *General-purpose timer* (TIM2, TIM3, TIM4) mają znacznie więcej możliwości niż wspomniany *Cortex System Timer*. Schemat działania timerów ogólnego przeznaczenia w trybie odmierzania czasu (na tym trybie skupiać się będzie dalszy opis) widoczny jest na Rys. 24. Na schemacie widać, że źródłem taktowania timera może być sygnał TIMxCLK (patrz Rys. 18), zewnętrzny sygnał TIMx_ETR lub wyjście innego licznika ITRx. Wybrany sygnał trafia na wejście prescalera (*PSC prescaler*), gdzie może zostać podzielony przez dowolną 16-bitową wartość. Poszczególne takty takiego sygnału dopiero zliczane są przez licznik (*CNT counter*). Timer może pracować w kilku trybach: zliczania w górę, zliczania w dół, zliczania naprzemiennie w górę i w dół. Rejestr *Auto-reload* zawiera 16-bitową wartość, od której odlicza lub do której zlicza licznik (zależnie od trybu zliczania). Tryby zliczania pokazane zostały na Rys. 25. Przedstawiona została na nim zawartość licznika CNT w zależności od wybranego trybu zliczania, przy założeniu, że rejestr *Auto-reload* ma wartość 4. W trybie zliczania w górę licznik po osiągnięciu wartości 4, zeruje zawartość licznika i kontynuuje zliczanie. W trybie zliczania w dół zawartość licznika jest inicjalizowana wartością 4 za każdym razem jak licznik osiągnie 0. W przypadku zliczania naprzemiennego, gdy zawartość licznika osiągnie 0 (przy zliczaniu w dół), licznik zaczyna zliczać w górę. W przypadku gdy licznik osiągnie wartość 4 (przy zliczaniu w górę), następuje rozpoczęcie zliczania w dół. Ważną cechą tych timerów jest, że gdy do licznika wpisywane jest zero (w przypadku zliczania w górę) lub zawartość rejestru



Rysunek 24: Schemat działania timera ogólnego przeznaczenia w trybie odmierzania czasu



Rysunek 25: Zawartość licznika CNT w zależności od wybranego trybu zliczania

Auto-reload (w przypadku zliczania w dół) generowane jest zdarzenie *Update Event*. W przypadku zliczania naprzemiennego nie jest dokonywane wpisywanie wartości do licznika – można jednak skonfigurować timer tak, aby generował zdarzenie *Update Event* za każdym razem gdy nastąpi zdarzenie przepełnienia *Overflow* lub niedomiaru *Undeflow*.

Każdy z timerów ogólnego przeznaczenia wyposażony jest w 4 kanały. Każdy z kanałów może służyć do przechwytywania zawartości licznika lub do porównywania z zawartością licznika. W tym ćwiczeniu uwaga została skupiona na tej drugiej funkcji, dzięki temu będzie można okresowo wykonywać pewne operacje oraz generować falę PWM (tak jak zostało to wykonane w poprzednim ćwiczeniu).

Zawartość licznika jest porównywana w każdym takcie ze wszystkimi rejestrami *Capture/Compare* (CCR) – w zależności od trybu mogą zostać wykonane różne operacje na wyjściu kanału OCxREF. Dostępne tryby to:

- *Timing* – wyjście OCxREF nie zmienia wartości,
- *Active* – OCxREF ustawiane w stan wysoki gdy zawartości rejestrów CNT i CCR są równe
- *Inactive* – OCxREF ustawiane w stan niski gdy zawartości rejestrów CNT i CCR są równe
- *Toggle* – gdy CNT i CCR są równe, OCxREF jest ustawiane na stan przeciwny
- PWM1 – *Pulse Width Modulation*(tryb 1)
- PWM2 – *Pulse Width Modulation*(tryb 2)

Prostą operację, jaką jest przełączenie bitu rejestru wyjściowego (TIMx_CHy), można wykonać więc na wiele sposobów: odpowiednią implementację obsługi przerwania (tryb *Timing*), wykorzystanie trybu *Toggle* lub jednego z trybów PWM. W poniższym ćwiczeniu będzie wykorzystany głównie tryb *Timing* – pozwoli to na późniejsze wykorzystanie wiedzy o implementacji obsługi przerwań pod kątem bardziej zaawansowanych operacji niż proste przełączanie stanu diody. Same diody należy tutaj traktować bardziej jako prymitywne narzędzie do diagnostyki (na zasadzie „jeśli mryga to zazwyczaj znaczy, że działa").

Tryb PWM został wyjaśniony w poprzednim ćwiczeniu, jednak nie została omówiona różnica między trybem PWM1 a PWM2. Jest ona jednak wyjątkowo prosta – sygnał powstały w trybie PWM2 jest negacją sygnału powstałego w trybie PWM1.

2.3 Przebieg laboratorium

Ćwiczenie obejmuje przełączanie stanu diod z odpowiednimi wymaganiami na momenty włączenia i wyłączenia. Dodatkowo, dla uproszczenia (tj. ograniczenia zgłębiania dokumentacji mikrokontrolera), odpowiednie diody mają wyznaczone timery, którymi będą sterowane. Wynikiem pracy w trakcie ćwiczenia powinno być:

- Przełączanie diody:
  - LED1 z częstotliwością 0,5 Hz z wypełnieniem 50 % (tj. przez 1 s świeci, przez 1 s nie świeci) z wykorzystaniem funkcji `Delay` i timera SysTick,

  - LED2 z częstotliwością 1 Hz z wypełnieniem 20 % (tj. przez 0,2 s świeci, przez 0,8 s nie świeci) z wykorzystaniem timera TIM4,
  - LED3 z częstotliwością 1 Hz z wypełnieniem 70 % (tj. przez 0,7 s świeci, przez 0,3 s nie świeci) z wykorzystaniem timera TIM4,
  - LED4 w wyniku wykrycia zbocza opadającego na przycisku SW0 (PA0),
  - LED5 w wyniku wykrycia zbocza opadającego na przycisku SW0 (PA0) z zaimplementowanym algorytmem niwelacji drgań styków,
  - LED6 wraz z zakończeniem przez przetwornik ADC konwersji sygnału wejściowego.

- Wyzwalanie przetwarzania sygnału wejściowego przez ADC1 (kanał 16 – wewnętrzny termometr mikrokontrolera) z okresem 1 s,
- Okresowe odświeżanie wyświetlacza LCD (tj. zmiana wyświetlanej treści na aktualną) – okres dobrany dowolnie (np. co 5 s),
- Obsługa klawiatury numerycznej – co zostanie wciśnięte na klawiaturze powinno zostać wypisane na wyświetlaczu (wystarczy jeden znak do wyświetlania ostatniego wciśniętego klawisza).

Dla skrócenia listingów wprowadzona została funkcja do obsługi LEDów. Plik `main.h` uzupełniony zostanie następującym kodem:

```
#include "stm32f10x.h" // definicja typu uint16_t i stalych GPIO_Pin_X

#define LED1 GPIO_Pin_8
#define LED2 GPIO_Pin_9
#define LED3 GPIO_Pin_10
#define LED4 GPIO_Pin_11
#define LED5 GPIO_Pin_12
#define LED6 GPIO_Pin_13
#define LED7 GPIO_Pin_14
#define LED8 GPIO_Pin_15
#define LEDALL (LED1|LED2|LED3|LED4|LED5|LED6|LED7|LED8)
enum LED_ACTION { LED_ON, LED_OFF, LED_TOGGLE };

void LED(uint16_t led, enum LED_ACTION act);
```

natomiast do pliku `main.c` dodana zostanie definicja funkcji obsługi LED (należy oczywiście pamiętać o uzupełnieniu nagłówka):

```
void LED(uint16_t led, enum LED_ACTION act) {
    switch(act){
        case LED_ON: GPIO_SetBits(GPIOB, led); break;
        case LED_OFF: GPIO_ResetBits(GPIOB, led); break;
        case LED_TOGGLE: GPIO_WriteBit(GPIOB, led,
            (GPIO_ReadOutputDataBit(GPIOB, led) == Bit_SET?Bit_RESET:Bit_SET));
    }
}
```

Opóźnienie: Realizacja opóźnienia przy użyciu timera SysTick jest wyjątkowo użyteczna a jednocześnie niewymagająca dużego nakładu implementacyjnego. Koncepcja tego typu rozwiązania jest następująca: timer nieustannie odmierza pewien kwant czasu (dla ustalenia uwagi niech to będzie 1 ms), za każdym razem dekrementując zawartość pewnego licznika (zrealizowanego jako zwykła zmienna w pamięci mikrokontrolera). W momencie kiedy licznik ten osiąga zero – odmierzanie czasu oczekiwania kończy się. Powoduje to, że potrzebujemy kilku elementów: licznika (zmiennej), funkcji dekrementującej licznik wywoływanej ze stałą częstotliwością, funkcji ustawiającej i testującej zawartość licznika.

Najpierw zajmijmy się implementacją odmierzania kwantu czasu, a więc 1 ms. W tym celu należy skonfigurować timer SysTick, a wykonuje się to przy użyciu funkcji:

```
SysTick_Config(Ticks);
```

gdzie `Ticks` oznacza liczbę taktów sygnału wejściowego, po których odliczeniu (timer SysTick zlicza w dół) następuje wyzwolenie przerwania oraz reset licznika timera SysTick. Drugą przydatną funkcją jest:

```
SysTick_CLKSourceConfig(SysTick_CLKSource_X);
```

gdzie `SysTick_CLKSource_X` definiuje sygnał wejściowy dla timera SysTick. Do wyboru są dwie opcje: sygnał HCLK – `SysTick_CLKSource_HCLK` lub sygnał HCLK/8 – `SysTick_CLKSource_HCLK_Div8`. Wybór odpowiedniego sygnału taktującego jest bardzo ważny, gdyż licznik timera SysTick jest 24-bitowy, więc przy sygnale wejściowym HCLK przepełniać się on będzie z częstotliwością od HCLK/2^{24} do HCLK, czyli dla HCLK=72 MHz jest to zakres od około 4,29 Hz (okres około 0,23 s) do 72 MHz (okres około 13,89 ns). W przypadku jednak sygnału wejściowego HCLK/8 zakres dostępnych częstotliwości wynosi od HCLK/2^{27} do HCLK/8, czyli dla HCLK=72 MHz, od około 0,54 Hz (okres około 1,86 s) do 9 MHz (okres około 111,11 ns). Tak więc aby zliczać pojedyncze sekundy należy koniecznie użyć sygnału wejściowego HCLK/8. W tym ćwiczeniu proponowane jest zliczanie kwantów 1 ms, a więc wybór zegara wejściowego nie jest krytyczny (na obu można zrealizować to zadanie) – dla przykładu zostanie użyty sygnał HCLK/8. Tak więc **konfiguracja timera SysTick**, który zgłasza przerwanie co 1 ms wygląda następująco:

```
SysTick_Config(9000); // (72MHz/8) / 9000 = 1KHz (1/1KHz = 1ms)
SysTick_CLKSourceConfig(SysTick_CLKSource_HCLK_Div8);
```

Obsługa przerwania generowanego przez SysTick sprowadza się do implementacji funkcji:

```
void SysTick_Handler(void);
```

znajdującej się w pliku `stm32f10x_it.c`.

Deklaracje i definicje funkcji związanych z opóźnieniem warto pisać w osobnych plikach, o nazwach odpowiednio np. `delay.h` oraz `delay.c`. W pliku nagłówkowym należy stworzyć zmienną będącą licznikiem milisekund o przykładowej nazwie `msc` typu `static unsigned int`. Słowo kluczowe `static` w tym przypadku służy do ograniczenia widoczności zmiennej `msc` do pojedynczej jednostki kompilacji (w uproszczeniu, jednostką kompilacji jest pojedynczy plik z nagłówkami). Oznacza to, że kompilator nie będzie zgłaszał błędów dotyczących wielokrotnej deklaracji tej samej zmiennej. Oczywiście zmienną tą należy zainicjalizować zerem.

W pliku źródłowym `delay.c` należy zdefiniować potrzebne funkcje: funkcja do dekrementacji licznika (o ile jest większy od 0) oraz funkcja do zmiany wartości licznika i testowania czy jest on większy od 0. **Definicje tych funkcji pozostawia się czytelnikowi do uzupełnienia**:

```
void DelayTick(void){
    // dekrementacja licznika (o ile jest wiekszy od 0)
}
```

```
void Delay(unsigned int ms){
    // zmiana wartosci licznika
    // testowanie czy jest on wiekszy od 0
}
```

Funkcja `DelayTick` powinna być wywoływana co 1 ms, a więc **należy ją dodać do ciała obsługi przerwania timera SysTick**, natomiast funkcja `Delay` będzie od tej pory wykorzystywana do wprowadzania opóźnień poprzez jej wywołanie w postaci `Delay(time)`, gdzie `time` jest to liczba milisekund jakie chcemy odczekać. Oczywiście, o ile to już nie zostało zrobione, **należy pozbyć się poprzedniej – programowej – implementacji opóźnienia** lub co najmniej zmienić jej nazwę.

Na koniec warto zauważyć, że domyślnie SysTick nie ma zbyt wysokiego priorytetu – zaleca się ustawienie jego priorytetu na dość wysokim poziomie (tj. należy obniżyć jego wartość). Rozsądną z punktu widzenia tego ćwiczenia jest wartość 0. **Zmianę priorytetu przerwania generowanego przez timer SysTick** realizuje się przy użyciu

```
NVIC_SetPriority(SysTick_IRQn, 0);
```

uprzednio konfigurując timer SysTick. Kolejność wynika z zawartości definicji funkcji `SysTick_Config`.

Jak widać obsługa timera SysTick jest wyjątkowo prosta, lecz doskonała do wielu zadań związanych z odliczaniem stałych kwantów czasu – stąd jego wielka użyteczność w kontekście systemów operacyjnych.

Ustawienie podziału priorytetów przerwań: Przed rozpoczęciem pracy nad przerwaniami warto **ustalić w jaki sposób wartość priorytetu przerwania ma być dzielona** na priorytet wywłaszczenia i podpriorytet. Służy do tego funkcja:

```
NVIC_PriorityGroupConfig(NVIC_PriorityGroup_X);
```

gdzie podział zmienia się w zależności od wartości `NVIC_PriorityGroup_X` , a dokładniej od wartości `X` zgodnie z poniższym:

- 0 – 0 bitów priorytetu wywłaszczenia, 4 bity podpriorytetu,
- 1 – 1 bit priorytetu wywłaszczenia, 3 bity podpriorytetu,
- 2 – 2 bity priorytetu wywłaszczenia, 2 bity podpriorytetu,
- 3 – 3 bity priorytetu wywłaszczenia, 1 bit podpriorytetu,
- 4 – 4 bity priorytetu wywłaszczenia, 0 bitów podpriorytetu.

Odmierzanie czasu przy użyciu timera ogólnego przeznaczenia: Poza timerem SysTick, do odmierzania stałych odcinków czasu można wykorzystać także zwykłe timery ogólnego przeznaczenia. Metod realizacji tego zadania jest wiele – jedna z nich wymaga następującej konfiguracji timera (odmierzanie odcinków czasowych o długości 1 s):

- prescaler = 7200,
- zawartość rejestru *auto-reload* = 10000,
- tryb zliczania w górę,
- włączona obsługa przerwania zgłaszanego przy zerowaniu licznika timera.

Tak uruchomiony timer będzie zliczał 10000 taktów wejściowego sygnału zegarowego o częstotliwości HCLK/7200, czyli 10 kHz. Oznacza to, że licznik będzie się przepełniał równo co sekundę, co będzie powodowało wyzerowanie licznika timera, a za razem zgłoszenie stosownego przerwania. Funkcja, która służy do obsługi tego przerwania będzie więc wywoływana dokładnie co sekundę.

Dodatkowe skonfigurowanie odpowiednich kanałów tego timera może być przydatne do odmierzania również odcinków czasu o długości 1 s, lecz tym razem np. przesuniętych w czasie o 0,2 s względem wcześniej skonfigurowanego timera. Aby tego dokonać należy skonfigurować kanał tego timera z następującymi właściwościami:

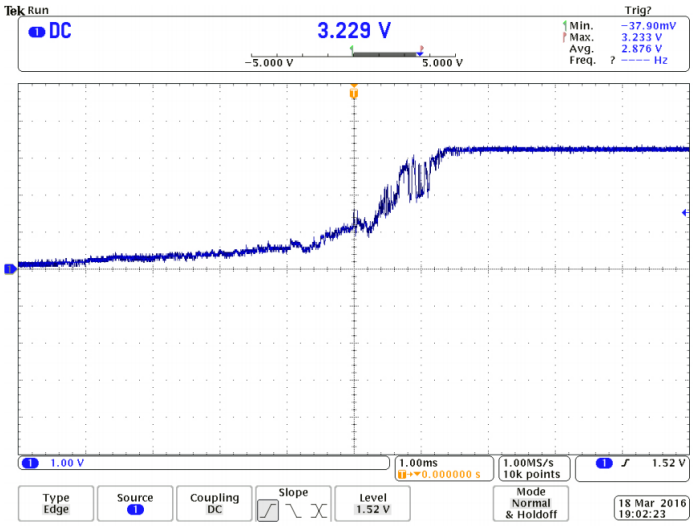
- niezmienna wartość rejestru wyjściowego kanału OCxREF,
- włączona obsługa przerwania zgłaszanego gdy zawartość licznika timera jest równa 2000.

Pozwoli to na uzyskanie wspomnianego przesunięcia wywołania okresowego przerwania kanału względem przerwania związanego z samym timerem. Poniżej znajduje się stosowna **implementacja powyższego odmierzania czasu** w kodzie wykorzystującym standardową bibliotekę peryferiali:

```
TIM_TimeBaseInitTypeDef TIM_TimeBaseStructure;
TIM_OCInitTypeDef TIM_OCInitStructure;

TIM_TimeBaseStructure.TIM_Prescaler = 7200-1;                // 72MHz/7200=10kHz
TIM_TimeBaseStructure.TIM_Period = 10000;                    // 10kHz/10000=1Hz (1s)
TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up; // zliczanie w gore
TIM_TimeBaseStructure.TIM_RepetitionCounter = 0;             // brak powtorzen
TIM_TimeBaseInit(TIM4, &TIM_TimeBaseStructure);              // inicjalizacja TIM4
TIM_ITConfig ( TIM4, TIM_IT_CC2 | TIM_IT_Update, ENABLE );  // wlaczenie przerwan
TIM_Cmd(TIM4, ENABLE);                                       // aktywacja timera TIM4

// konfiguracja kanalu 2 timera
TIM_OCInitStructure.TIM_OCMode = TIM_OCMode_Timing;          // brak zmian OCxREF
TIM_OCInitStructure.TIM_Pulse = 2000;                        // wartosc do porownania
TIM_OCInitStructure.TIM_OutputState = TIM_OutputState_Enable; // wlaczenie kanalu
TIM_OC2Init(TIM4, &TIM_OCInitStructure);                     // inicjalizacja CC2
```

Konfiguracja przerwania związanego z timerem następuje poprzez wykorzystanie modułu NVIC:

```
NVIC_InitTypeDef NVIC_InitStructure;

NVIC_ClearPendingIRQ(TIM4_IRQn);                    // wyczyszczenie bitu przerwania
NVIC_EnableIRQ(TIM4_IRQn);                          // wlaczenie obslugi przerwania
NVIC_InitStructure.NVIC_IRQChannel = TIM4_IRQn;     // nazwa przerwania
NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 2; // priorytet wywlaszczania
NVIC_InitStructure.NVIC_IRQChannelSubPriority = 1;  // podpriorytet
```



Rysunek 26: Oscylogram zjawiska drgań styków (wciśnięcie przełącznika)

```
    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;      // wlaczenie
    NVIC_Init(&NVIC_InitStructure);                      // inicjalizacja struktury
```

Po wprowadzeniu powyższych konfiguracji można **napisać funkcję obsługującą przerwanie**:

```
void TIM4_IRQHandler(void){
    if(TIM_GetITStatus(TIM4,TIM_IT_CC2) != RESET){
        LED(LED2,LED_TOGGLE);
        TIM_ClearITPendingBit(TIM4, TIM_IT_CC2);
    } else if(TIM_GetITStatus(TIM4,TIM_IT_Update) != RESET){
        LED(LED3,LED_TOGGLE);
        TIM_ClearITPendingBit(TIM4, TIM_IT_Update);
    }
}
```

Warto zauważyć, ze jedna funkcja obsługuje oba przerwania, dlatego na początku funkcji ważne jest aby **sprawdzić, które przerwanie zostało zgłoszone**. W przypadku gdy jest to przerwanie wynikające z przepełnienia licznika timera – przełączana jest dioda LED3. Jeśli jest to przerwanie wynikające z nastąpienia równości między zawartością licznika timera a rejestru kanału 2 – przełączana jest dioda LED2. Powoduje to, że obie diody będa świecić z takim samym okresem, lecz będzie między nimi przesunięcie w fazie o 0,2 s. **Na koniec każdego programu do obsługi przerwań koniecznie trzeba wyczyścić bit świadczący o oczekiwaniu na obsługę tego przerwania.** Wykonuje się to (w przypadku timerów) przy użyciu funkcji `TIM_ClearITPendingBit`. W przypadku niewykonania tej operacji przerwanie to będzie nieustannie fałszywie zgłaszane jako nieobsłużone.

Niwelowanie drgań styków: Zjawisko drgań styków jest powszechnie znanym problemem związanym głównie z przełącznikami mechanicznymi i enkoderami. Szczegóły powstawania tego zjawiska nie będą omawiane – warto jednak mieć świadomość jakie są jego skutki. Na Rys. 26 widoczny jest przykład pomiaru wartości napięcia na przełączniku w momencie jego wciśnięcia. Zamiast zaobserwować pojedynczą zmianę wartości napięcia, widać wyraźnie wiele krótkich impulsów. Właśnie te krótkie piki powodują, że prosty odczyt stanu przełącznika może być wyjątkowo uciążliwy w implementacji. Dwie podstawowe metody odczytu stanu przełącznika to nieustanne odpytywanie lub wykorzystanie przerwań. Nieustanne odpytywanie implementuje się poprzez odczytywanie w nieskończonej pętli stanu przycisku:

```
while(1){
    // ...
    stan_przelacznika = GPIO_ReadInputDataBit(GPIOx, GPIO_Pin_y);
    // ...
}
```

Powyższy kod jest często uważany za błędny lub co najmniej niewłaściwy. Wynika to z zasady jego działania – zamiast reagować wyłącznie na zmiany poziomu napięcia na odpowiedniej linii wybranego portu, nieustannie testowany jest jego stan. Dodatkowym problemem jest czas odpytywania – jeśli mamy mało do zrobienia to będzie on krótki, lecz jeśli nagle postanowimy wykonać jakąś dłuższą, konieczną operację jesteśmy pozbawieni możliwości monitorowania stanu przełącznika.

Drugim podejściem jest wykorzystanie przerwań. Jest ono znacznie wygodniejsze od poprzedniego, ponieważ bez względu na obciążenie zadaniami wciąż jesteśmy w stanie zareagować na zmianę stanu przełącznika. W przypadku odpytywania jeśli wykonywana operacja jest czasochłonna, to dopiero po jej zakończeniu można na nowo sprawdzić stan przełącznika. Przekłada się to bezpośrednio na wygodę w obsłudze mikrokontrolera przez użytkownika końcowego. W przypadku nieustannego odpytywania przełącznika, użytkownik musi trzymać go tak długo wciśniętym, aż mikrokontroler zdąży go odpytać o obecny stan. Sytuacja ta przypomina „zawieszenie się" systemu operacyjnego – mikrokontroler nie reaguje na interakcję. W sytuacji wykorzystania mechanizmu przerwań, mikrokontroler otrzymuje natychmiastową informację o zmianie stanu przełącznika, co może skutkować anulowaniem obecnie wykonywanego zadania lub choćby prośbą o oczekiwanie na zakończenie działania. Jest to niemal konieczne w przypadku interakcji użytkownik-mikrokontroler, aby ten pierwszy miał świadomość, że mikrokontroler nie przestał działać, a jest po prostu bardzo zajęty realizacją niezmiernie ważnych zadań.

Wykorzystanie przerwań prowadzi jednak do innego problemu – reakcje na zmiany stanu przełącznika są niemal natychmiastowe. A więc widoczny na Rys. 26 stan przełacznika jest widziany często właśnie z taką dokładnością – każda jego zmiana jest rejestrowana i zgłaszana za pomocą mechanizmu przerwań. Zamiast więc uzyskać jedną pewną informację o wciśnięciu przełącznika, otrzymujemy ich wiele o różnym poziomie zaufania.

Tak jak w wielu innych aspektach programowania mikrokontrolerów, tak i tutaj sposobów na radzenie sobie z drganiem styków jest mnóstwo. Jednym z najpopularniejszych jest sprzętowa realizacja filtru dolnoprzepustowego (niwelującego sygnał o wysokiej częstotliwości). Tutaj poruszony jednak zostanie mechanizm programowy, tj. opóźnienie odczytu. Zasada działania jest następująca:

1. jeśli zmieniony został stan przełącznika, np. z wysokiego na niski – zgłoś przerwanie,
2. jeśli przerwanie zgłoszone, to odczekaj chwilę, aby zniknęły drgania,

3. odczytaj ustalony stan przełącznika.

Pomysł ten opiera się na założeniu, że istnieje pewien maksymalny czas występowania drgań styków – czasem taką informację można znaleźć w dokumentacji przełączników. W pozostałych przypadkach warto rozważyć czas od 20 do 50 ms – jest to na tyle krótki czas, aby opóźnienie nie było zauważalne, a na tyle długi, żeby skuteczność takiego mechanizmu niwelacji drgań styków była satysfakcjonująca.

Aby zrealizować powyższą koncepcję należy **skonfigurować przerwanie zewnętrzne** (wykrycie zbocza opadającego na wybranej linii) oraz **timer odpowiadający za realizację opóźnienia**. Oczywiście należy to **poprzedzić dodaniem odpowiednich plików** standardowej biblioteki peryferiali do projektu oraz stosowną konfiguracją pinu, do którego podpięty jest rozważany przełącznik. Konfiguracja przerwania wygląda następująco:

```
GPIO_EXTILineConfig(GPIO_PortSourceGPIOA, GPIO_PinSource0); // PA0 -> EXTI0_IRQn
EXTI_InitStructure.EXTI_Line = EXTI_Line0;                  // linia : 0
EXTI_InitStructure.EXTI_Mode = EXTI_Mode_Interrupt;         // tryb  : przerwanie
EXTI_InitStructure.EXTI_Trigger = EXTI_Trigger_Falling;     // zbocze: opadajace
EXTI_InitStructure.EXTI_LineCmd = ENABLE;                   // aktywowanie konfig.
EXTI_Init(&EXTI_InitStructure);                             // inicjalizacja

NVIC_ClearPendingIRQ(EXTI0_IRQn);                           // czyszcz. bitu przerw.
NVIC_EnableIRQ(EXTI0_IRQn);                                 // wlaczenie przerwania
NVIC_InitStructure.NVIC_IRQChannel = EXTI0_IRQn;            // przerwanie EXTI0_IRQn
NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;   // prior. wywlaszczania
NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;          // podpriorytet
NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;             // aktywowanie konfig.
NVIC_Init(&NVIC_InitStructure);                             // inicjalizacja
```

Konfiguracja timera jest analogiczna jak w przypadku okresowego wyzwalania przerwania, lecz tym razem sam **timer zostaje skonfigurowany jako wyłączony** (wraz z przerwaniami przez niego generowanymi):

```
TIM_TimeBaseStructure.TIM_Prescaler = 7200-1;               // 72MHz/7200=10kHz
TIM_TimeBaseStructure.TIM_Period = 350;                     // 10kHz/350~=29Hz(35ms)
TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up; // zliczanie w gore
TIM_TimeBaseStructure.TIM_RepetitionCounter = 0;            // brak powtorzen
TIM_TimeBaseInit(TIM3, &TIM_TimeBaseStructure);             // inicjalizacja TIM3
TIM_ITConfig ( TIM3, TIM_IT_Update, DISABLE );              // wylaczenie przerwan
TIM_Cmd(TIM3, DISABLE);                                     // wylaczenie timera

NVIC_ClearPendingIRQ(TIM3_IRQn);                            // czyszcz. bitu przerw.
NVIC_EnableIRQ(TIM3_IRQn);                                  // wlaczenie przerwania
NVIC_InitStructure.NVIC_IRQChannel = TIM3_IRQn;             // nazwa przerwania
NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 1;   // prior. wywlaszczania
NVIC_InitStructure.NVIC_IRQChannelSubPriority = 1;          // podpriorytet
NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;             // aktywowanie konfig.
NVIC_Init(&NVIC_InitStructure);                             // inicjalizacja
```

Taka konfiguracja pozwala nam na wstrzymanie uruchomienia timera do momentu kiedy będzie on potrzebny. A potrzebny będzie w momencie gdy zgłoszone zostanie przerwanie wynikające z wykrycia zbocza opadającego na linii podłączonej do przełącznika. Z tego powodu należy następująco **zaimplementować obsługę tego przerwania**:

```
void EXTI0_IRQHandler(void){
    if(EXTI_GetITStatus(EXTI_Line0) != RESET){      // sprawdzenie przyczyny
        EXTI_ClearITPendingBit(EXTI_Line0);         // wyczyszczenie bitu przerwania

        TIM_SetCounter(TIM3, 0);                    // reset licznika timera
        TIM_ITConfig(TIM3, TIM_IT_Update, ENABLE ); // aktywacja przerwania
        TIM_Cmd(TIM3, ENABLE);                      // aktywacja timera TIM3
    }
}
```

Powyższy kod spowoduje, że w momencie wykrycia przerwania na linii 0, wyzerowany i uruchomiony zostanie timer. Gdy timer się przepełni wyzwolone zostanie jego przerwanie, które należy obsłużyć:

```
void TIM3_IRQHandler(void){
    if(TIM_GetITStatus(TIM3,TIM_IT_Update) != RESET){ // sprawdzenie przyczyny
        TIM_ClearITPendingBit(TIM3, TIM_IT_Update);   // wyczyszczenie bitu przerw.
        if(GPIO_ReadInputDataBit(GPIOA, GPIO_Pin_0) == Bit_RESET) // jesli wcisniety
            LED(LED5,LED_TOGGLE);                     // zrob cos (przelacz LED5)
        TIM_ITConfig (TIM3, TIM_IT_Update, DISABLE ); // deaktywacja przerwania
        TIM_Cmd(TIM3, DISABLE);                       // deaktywacja timera TIM3
    }
}
```

Jest to oczywiście jedna z mnóstwa możliwości niwelacji drgań styków. Eliminacja tego zjawiska jest bardzo trudna i nie istnieje niestety metoda, która dobrze by sprawdzała się dla wszelkiego rodzaju przełączników i enkoderów – rozsądek projektanta i dostosowanie do potrzeb jest tutaj kluczową kwestią.

Obsługa klawiatury numerycznej Do zestawu ZL27ARM można bardzo łatwo podpiąć klawiaturę o 4 kolumnach i 4 wierszach (Rys. 27). Wciśnięcie poszczególnych klawiszy na tej klawiaturze powoduje zwarcie linii kolumny oraz wiersza, w których znajduje się dany klawisz. A więc jeśli wciśnięty zostanie klawisz „1”, to linia kolumny pierwszej i linia wiersza pierwszego zostaną ze sobą zwarte, jeśli zostanie wciśnięty klawisz „2” to zwarta zostanie linia wiersza 1 i kolumny 2, itd. Wykrywanie wciśniętego klawisza wymaga drobnych ulepszeń sprzętowych (podłączenia kilku rezystorów, a najlepiej również kondensatorów), które zostały wprowadzone do omawianej klawiatury w postaci osobnej płytki łączącej zestaw uruchomieniowy i klawiaturę. Schemat usprawnionej klawiatury jest przedstawiony na Rys. 28. Na schemacie widoczne są rezystory 10 kΩ podciągające linie ROW1, ROW2, ROW3 i ROW4 do napięcia zasilania, rezystory 2,2 kΩ podciągające linie COL1, COL2, COL3, COL4 do napięcia zasilania oraz pary rezystor-kondensator realizujące filtr dolnoprzepustowy. W każdej z par rezystor ma wartość 220 kΩ, a kondensator 47 µF.

Klawiatura jest w całości podłączona do portu D. Piny od 10 do 13 należy skonfigurować w tybie **otwartego drenu**, natomiast pozostałe (od 6 do 9) w trybie ***floating***. Zasada testowania, który klawisz jest wciśnięty jest następująca: pinom od 10 do 13 przypisana jest logiczna jedynka, co oznacza, że są one zawieszone w powietrzu, jednak dzięki rezystorom podciągającym ustala się na nich napięcie zasilania (3,3 V). Piny od 6 do 9 są również podciągniete do zasilania, co powoduje, że domyślnie występuje na nich właśnie napięcie zasilania. Ponieważ wszystkie piny mają ten sam poziom napięcia, procedura sprawdzenia, który klawisz jest wciśnięty wymaga wprowadzenia dodatkowego sygnału testującego. Kolejno więc należy ustawić zero logiczne na pinie 10 (napięcie na tym pinie jest teraz równe napięciu masy), co powoduje, że

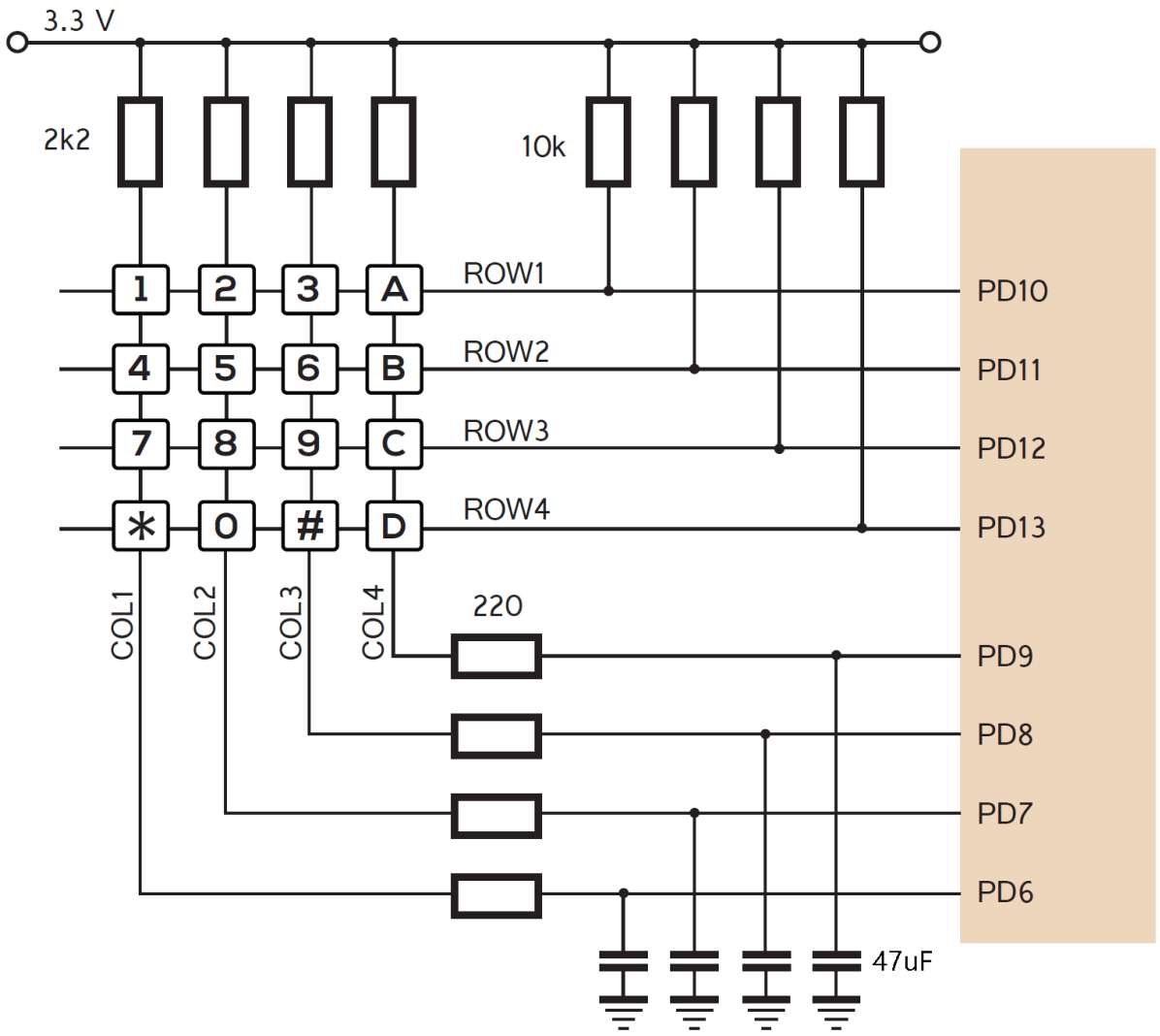
Rysunek 27: Klawiatura 4×4

jeśli któryś z klawiszy w pierwszym rzędzie jest wciśnięty, to na jednym z pinów od 6 do 9 pojawi się logiczne 0. Analogiczny zabieg nalezy przeprowadzić dla pinów 11, 12 i 13 ustawiając logiczną 1 na pozostałych pinach. Po przeskanowaniu wszystkich wierszy można odczytać informację o tym, dla których rzędów, w których kolumnach występowały zera logiczne. Oczywiście podejście to nie gwarantuje wykrycia wszystkich klawiszy, które są wciśnięte – możliwości tej nie daje jednak już sama budowa klawiatury. Odpowiednia kombinacja klawiszy może spowodować błędne wykrycie klawiszy niewciśniętych.

Wybrane do tego zadania tryby linii portu D są kluczowe dla bezpieczeństwa mikrokontrolera. Dla takiej konfiguracji wciśnięcie kilku klawiszy klawiatury jednocześnie nie spowoduje zwarcia – nie można wywołać sytuacji kiedy masa jest zwierana z zasilaniem. Inaczej byłoby gdyby zamiast trybu otwartego drenu zastosować wyjście *push-pull*. Wtedy dla logicznego 0 na pinie np. 10 występowałoby tam napięcie masy, a na pinach 11, 12 i 13 występowałoby napięcie zasilania. Wciśnięcie w tym momencie klawiszy z rzędów pierwszego i któregokolwiek innego spowodowałoby zwarcie masy z zasilaniem, a zarazem prawdopodobnie uszkodzenie płyty uruchomieniowej.

Poniżej znajduje się **kod służący do skanowania i odczytu klawisza** wciśniętego na klawiaturze:

```
char KB2char(void){
    unsigned int GPIO_Pin_row, GPIO_Pin_col, i, j;
    const unsigned char KBkody[16] = {'1','2','3','A',\
                                      '4','5','6','B',\
                                      '7','8','9','C',\
                                      '*','0','#','D'};
    GPIO_SetBits(GPIOD, GPIO_Pin_10|GPIO_Pin_11|GPIO_Pin_12|GPIO_Pin_13);
    GPIO_Pin_row = GPIO_Pin_10;
    for(i=0;i<4;++i){
```



Rysunek 28: Schemat podłączenia klawiatury 4×4 (źródło: *Systemy Mikroprocesorowe w Sterowaniu. Część I: ARM Cortex-M3*)

```
        GPIO_ResetBits(GPIOD, GPIO_Pin_row);
        Delay(5);
        GPIO_Pin_col = GPIO_Pin_6;
        for(j=0;j<4;++j){
            if(GPIO_ReadInputDataBit(GPIOD, GPIO_Pin_col) == 0){
                GPIO_ResetBits(GPIOD, GPIO_Pin_10|GPIO_Pin_11|
                  GPIO_Pin_12|GPIO_Pin_13);
                return KBkody[4*i+j];
            }
            GPIO_Pin_col = GPIO_Pin_col << 1;
        }
        GPIO_SetBits(GPIOD, GPIO_Pin_row);
        GPIO_Pin_row = GPIO_Pin_row << 1;
    }
    GPIO_ResetBits(GPIOD, GPIO_Pin_10|GPIO_Pin_11|GPIO_Pin_12|GPIO_Pin_13);
    return 0;
}
```

Powyższy kod może być wykorzystany w pętli głównej programu do odpytywania (tj. nieustannego skanowania) klawiatury w celu ustalenia, które klawisze są wciśnięte (przydatne do testu podłączenia klawiatury do płyty uruchomieniowej). Podejście to jest jednak niewygodne i mało wydajne – lepiej wykorzystać przerwania.

Wybór pinów, do których zostały podpięte piny kolumn nie był przypadkowy – wykrycie na nich zbocza opadającego powoduje wyzwolenie wspólnego przerwania `EXTI_Line9_5`. Oznacza to, że bez względu na to, który klawisz zostanie wciśnięty, zostanie wywołana ta sama funkcja obsługująca przerwanie, gdzie będzie można wykonać powyższą funkcję do skanowania klawiatury. **Implementacja przerwania oraz jego obsługa jest analogiczna jak w przypadku poprzednich przykładów, nie będzie więc przytaczana.** Warto jednak zauważyć, że ponieważ zastosowane zostały filtry dolnoprzepustowe na wejściach mikrokontrolera można założyć, że zbocze opadające nie jest w żaden sposób zakłócone. Oznacza to, że wykryte zbocze jest jednoznacznie związane z pojedynczym wciśnięciem klawisza na klawiaturze. Nie należy implementować tutaj mechanizmu niwelacji drgań styków.

Okresowe wykonywanie pomiarów: Do tej pory wykonywanie przetwarzania analogowo cyfrowego rozpoczynane było poprzez programowe jego uruchomienie. Jest to mało wydajna metoda, gdyż w trakcie gdy wykonywany jest pomiar można by wykonać inne potrzebne operacje, zamiast oczekiwania na otrzymanie wyniku. Poza tym regularne próbkowanie sygnału mierzonego jest kluczowe z punktu widzenia późniejszego zadania regulacji. Dlatego warto by było aby przetwarzanie rozpoczynało się ze stałą częstotliwością. Dodatkowo w trakcie wykonywania przetwarzania warto zwolnić procesor, aby nie oczekiwać bezsensownie na cyfrową postać pomiaru – zamiast tego niech zgłoszone przerwanie będzie sygnałem, że procedura przetwarzania została zakończona.

Aby regularnie wyzwolić przerwanie, które może zostać potem wykryte przez przetwornik, należy **skonfigurować timer i jego kanał w trybie PWM**:

```
    TIM_TimeBaseInitTypeDef TIM_TimeBaseStructure;
    TIM_OCInitTypeDef TIM_OCInitStructure;

    TIM_TimeBaseStructure.TIM_Prescaler = 7200-1;
```

```
    TIM_TimeBaseStructure.TIM_Period = 5000; // 2Hz -> 0.5s
    TIM_TimeBaseStructure.TIM_RepetitionCounter = 0;
    TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up;
    TIM_TimeBaseInit(TIM2, &TIM_TimeBaseStructure);

    // konfiguracja kanalu timera
    TIM_OCInitStructure.TIM_OCMode = TIM_OCMode_PWM1;
    TIM_OCInitStructure.TIM_Pulse = 1; // minimalne przesuniecie
    TIM_OCInitStructure.TIM_OutputState = TIM_OutputState_Enable;
    TIM_OCInitStructure.TIM_OCPolarity = TIM_OCPolarity_Low;
    TIM_OC2Init(TIM2, &TIM_OCInitStructure);
    TIM_ITConfig ( TIM2, TIM_IT_CC2 , ENABLE );
    TIM_Cmd(TIM2, ENABLE);
```

Warto zwrócić uwagę na wartość rejestru kanału – jest ona możliwie mała, aby nie wprowadzać zbędnego przesunięcia w fazie między przeładowaniem licznika timera, a wyzwoleniem przerwania przez jeden z kanałów tego timera. Dodatkowo, w przeciwieństwie do wcześniejszych przykładów, tym razem **nie implementujemy funkcji obsługującej przerwanie** `TIM_IT_CC2`.

Konfiguracja przetwornika jest bardzo podobna jak poprzednio:

```
void ADC_Config(void) {
    ADC_InitTypeDef  ADC_InitStructure;

    ADC_DeInit(ADC1);                                   // reset ustawien ADC1
    ADC_InitStructure.ADC_Mode = ADC_Mode_Independent;  // niezalezne dzialanie ADC 1 i 2
    ADC_InitStructure.ADC_ScanConvMode = DISABLE;       // pomiar pojedynczego kanalu
    ADC_InitStructure.ADC_ContinuousConvMode = DISABLE;  // pomiar automatyczny
    ADC_InitStructure.ADC_ExternalTrigConv = ADC_ExternalTrigConv_T2_CC2; // T2CC2->ADC
    ADC_InitStructure.ADC_DataAlign = ADC_DataAlign_Right; // pomiar wyrownany do prawej
    ADC_InitStructure.ADC_NbrOfChannel = 1;             // jeden kanal
    ADC_Init(ADC1, &ADC_InitStructure);                 // inicjalizacja ADC1

    ADC_RegularChannelConfig(ADC1, ADC_Channel_16, 1, ADC_SampleTime_41Cycles5); // konf.
    ADC_ExternalTrigConvCmd(ADC1, ENABLE);
    ADC_ITConfig(ADC1, ADC_IT_EOC, ENABLE);

    ADC_Cmd(ADC1, ENABLE);                              // aktywacja ADC1

    ADC_ResetCalibration(ADC1);                         // reset rejestru kalibracji ADC1
    while(ADC_GetResetCalibrationStatus(ADC1));         // oczekiwanie na koniec resetu
    ADC_StartCalibration(ADC1);                         // start kalibracji ADC1
    while(ADC_GetCalibrationStatus(ADC1));              // czekaj na koniec kalibracji

    ADC_TempSensorVrefintCmd(ENABLE);                   // wlaczenie czujnika temperatury
}
```

Do głównych różnic należy **wybór przerwania** `TIM_IT_CC2` **jako sygnału rozpoczynającego przetwarzanie**, **zmiana kanału, który będzie wykorzystany do pomiarów** i co za tym idzie dodatkowa **linijka uruchamiająca czujnik temperatury** znajdujący się w samym mikrokontrolerze. Oczywiście należy również **pamiętać o aktywowaniu i konfiguracji przerwania** wyzwalanego na zakończenie konwersji `ADC_IT_EOC`. **W obsłudze tego przerwania należy**, tak jak zawsze, **sprawdzić**, co wywo-

łało uruchomienie funkcji obsługi przerwania, **wyczyścić** bity oczekujących przerwań i **pobrać** wartość przetworzoną przez przetwornik:

```
void ADC1_2_IRQHandler(void){
    if(ADC_GetITStatus(ADC1, ADC_IT_EOC) != RESET){
        ADC_ClearITPendingBit(ADC1, ADC_IT_EOC);
        sprintf((char*)bufor,"%2d*C",((V25-ADC_GetConversionValue(ADC1))/Avg_Slope+25));
    }
}
```

Wartości `V25` i `Agv_Slope` są zdefiniowane jako:

```
const uint16_t V25 = 1750;      // gdy V25=1.41V dla napięcia odniesienia 3.3V
const uint16_t Avg_Slope = 5;   // gdy Avg_Slope=4.3mV/C dla napięcia odniesienia 3.3V
```

Obie te wartości wynikają z dokumentacji, tak jak sam wzór na przeliczenie wartości odczytanej z przetwornika na faktyczną wartość temperatury (rozdział 11.10 w dokumencie RM0008).

Na koniec należy **pamiętać o konfiguracji NVIC** (`ADC1_2_IRQn`)– kod ten nie różni się niemal niczym od wyżej prezentowanych fragmentów.

Temperatura mikrokontrolera jest przeważnie stała – aby zaobserwować jej zmianę warto zresetować mikrokontroler (tuż po uruchomieniu jest odrobinę chłodniejszy). Dodatkowo temperatura mikrokontrolera jest zależna od częstotliwości taktowania zegarów – spowolnienie zegara HSE powoduje zmniejszenie temperatury. Zarówno informacja o obecnej temperaturze jak i o sposobach na jej obniżenie pozwala na wykorzystanie mikrokontrolerów w wymagającym środowisku, np. małej zamkniętej obudowie telefonu, gdzie głównym źródłem ciepła jest właśnie sam mikrokontroler.

Dodatkowe informacje: Warto pamiętać o kolejności konfiguracji kolejnych peryferiali:

1. włączenie taktowania poszczególnych elementów – `RCC_APBxPeriphClockCmd`,
2. konfiguracja pinów GPIO – wypełnienie struktury `GPIO_InitTypeDef`,
3. konfiguracja docelowej funkcjonalności (timer, przetwornik ADC) – wypełnienie dodatkowych struktur np. `TIM_TimeBaseInitTypeDef` lub `ADC_InitTypeDef`,
4. włączenie generowania przerwań przez poszczególne moduły – np. `TIM_ITConfig` lub `ADC_ITConfig`,
5. konfiguracja przerwań – uzupełnienie struktury `NVIC_InitTypeDef` i wywołanie `NVIC_EnableIRQ`,
6. implementacja obsługi przerwania – wypełnienie funkcji z pliku `stm32f10x_it.c`.