Mikrokontrolery STM32

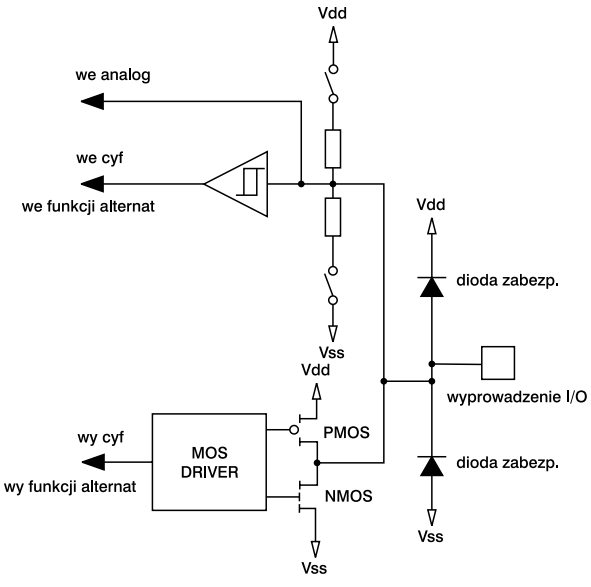
Obsługa portów we/wy, przerwań i timerów z wykorzystaniem funkcji API

Pierwszą umiejętnością, jaką trzeba nabyć, gdy rozpoczyna się pracę z nową rodziną mikrokontrolerów jest zapanowanie nad portami we/wy, przerwaniami i timerami, ponieważ stanowią one pomost łączący MCU ze światem zewnętrznym. W artykule przedstawiamy informacje, jak tego dokonać w przypadku mikrokontrolerów STM32 w oparciu o płytę ewaluacyjną STM3210B – EVAL/A.

Operowanie bezpośrednio na rejestrach jakiegokolwiek 32–bitowego procesora lub mikrokontrolera nie należy do zadań prostych. Mimo, iż samo napisanie (stosunkowo zaawansowanych) aplikacji przy użyciu nazw rejestrów jest możliwe, to wprowadzanie zmian do istniejącego kodu po upływie na przykład kilku miesięcy, dodatkowo przez osobę, która nie jest autorem programu, jest w zasadzie niemożliwe do wykonania w sensownym czasie. Z tego powodu do takich operacji programiści wykorzystują funkcje o mniej lub bardziej kojarzących się nazwach. Jeżeli mamy do czynienia z bardzo skomplikowanym projektem wykorzystującym wiele peryferiów mikrokontrolera, to napisanie stosownych funkcji jest pracochłonne i wymaga dobrej znajomości architektury mikrokontrolera. Firma STMicroelectronics zauważyła ten problem i udostępnia kompletne biblioteki API, które pozwalają w pełni kontrolować MCU. W pewnych przypadkach może oczywiście zajść potrzeba bezpośredniego odwołania się do rejestru, we wszystkich pozostałych funkcje API znacznie skracają czas potrzebny na napisanie i uruchomienie aplikacji. Zasada wykorzystania biblioteki API zostanie przedstawiona przy okazji omówienia obsługi portów we/wy. Kompletne listingi przykładów można znaleźć na stronie *http://paprocki.wemif.net*.

Porty wejścia/wyjścia

Porty we/wy mikrokontrolerów STM32 mogą pełnić do ośmiu funkcji. Oprócz pracy jako alternatywne wejście lub wyjście, określone wyprowadzenie może być skonfigurowane jako: wejście „pływające", *pull–up*, *pull–down*, lub jako wejście analogowe. W konfiguracji wyjścia wyprowadzenie może pracować w konfiguracji z otwartym drenem lub *push–pull*. Uproszczoną budowę takiego uniwersalnego portu przedstawiono na **rys. 1**.



Rys. 1. Uproszczona budowa portu uniwersalnego

Sposób konfigurowania i obsługiwania GPIO (*General Purpose Input Output*) wyjaśnimy może na niezbyt wyrafinowanym przykładzie, jednak dzięki temu będzie on przejrzysty i czytelny.

Układ ma odzwierciedlać stany swoich wejść na wyjściach, innymi słowy diody sygnalizacyjne mają się zapalać w takt zmian poziomu logicznego na wejściach. Jest to zrealizowane przy pomocy joysticka znajdującego się na płycie ewaluacyjnej. Biorąc po uwagę budowę płyty uruchomieniowej, wejścia z podłączonym joystickiem należy skonfigurować jako „pływające" (*input floating*), natomiast wyjścia jako *push–pull*. Praca wyjść w konfiguracji *push–pull* oznacza, że dzięki odpowiedniemu podłączeniu wewnętrznych tranzystorów MOS, ustawienie wyjścia w stan logicznej „1" spowoduje pojawienie się na końcówce układu napięcia zasilania, natomiast ustawienie w programie logicznego „0" będzie skutkowało podaniem na wyprowadzenie układu potencjału masy.

By dobrze wykorzystać możliwości portów we/wy, w pierwszej kolejności należy je odpowiednio do określonego zadania skonfigurować. Najpierw jednak zostanie przedstawiony mechanizm, jaki wykorzystują do pracy funkcje API.

Dla każdego urządzenia, czy jest to GPIO, kontroler przerwań, czy jakikolwiek inny element systemu, są stworzone odrębne typy danych. W przypadku portów we/wy nazywają się one GPIO_TypeDef, zaś do inicjacji jest wykorzystywany typ GPIO_InitTypeDef. Z punktu widzenia programisty największe znaczenie ma typ inicjujący, ponieważ to właśnie zmienną tego typu jawnie tworzymy w pisanym kodzie. Typ GPIO_TypeDef zapewnia dostęp do poszczególnych rejestrów mikrokontrolera i jest wykorzystywany przede wszystkim przez funkcje API, natomiast zmienna typu GPIO_InitTypeDef musi istnieć w każdej aplikacji wykorzystującej porty we/wy, ponieważ jest wykorzystywana do inicjalizowania i konfigurowania portów. Na **list. 1** przedstawiono kluczowy fragment kodu odpowiedzialny za konfigurację portów oraz operacje na nich.

Utworzona na początku zmienna GPIO_InitStruct jest, *de facto*, strukturą. Inicjowanie pinów lub w szczególnym przypadku całego portu odbywa się w ten sposób, że wypełnia się poszczególne pola struktury, a następnie

List. 1. Kluczowy fragment kodu odpowiedzialny za konfigurację portów oraz operacje na nich

```
GPIO_InitTypeDef GPIO_InitStruct;

int main(void)
{
  RCC_Conf();
  NVIC_Conf();

  RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOC, ENABLE);

  GPIO_InitStruct.GPIO_Pin = GPIO_Pin_6 | GPIO_Pin_7 |
                             GPIO_Pin_8 | GPIO_Pin_9;
  GPIO_InitStruct.GPIO_Mode = GPIO_Mode_Out_PP;
  GPIO_InitStruct.GPIO_Speed = GPIO_Speed_50MHz;
  GPIO_Init(GPIOC, &GPIO_InitStruct);

  RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOD, ENABLE);

  GPIO_InitStruct.GPIO_Pin = GPIO_Pin_8 | GPIO_Pin_12 |
                             GPIO_Pin_14;
  GPIO_InitStruct.GPIO_Mode = GPIO_Mode_IN_FLOATING;
  GPIO_InitStruct.GPIO_Speed = GPIO_Speed_50MHz;
  GPIO_Init(GPIOD, &GPIO_InitStruct);

  RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOE, ENABLE);

  GPIO_InitStruct.GPIO_Pin = GPIO_Pin_0 | GPIO_Pin_1;
  GPIO_InitStruct.GPIO_Mode = GPIO_Mode_IN_FLOATING;
  GPIO_InitStruct.GPIO_Speed = GPIO_Speed_50MHz;
  GPIO_Init(GPIOE, &GPIO_InitStruct);

  while (1)
  {
    if(GPIO_ReadInputDataBit(GPIOD, GPIO_Pin_14))
        GPIO_ResetBits(GPIOC, GPIO_Pin_6);
        else
                GPIO_SetBits(GPIOC, GPIO_Pin_6);

    if(GPIO_ReadInputDataBit(GPIOD, GPIO_Pin_8))
        GPIO_ResetBits(GPIOC, GPIO_Pin_9);
        else
                GPIO_SetBits(GPIOC, GPIO_Pin_9);

    if(GPIO_ReadInputDataBit(GPIOE, GPIO_Pin_1))
        GPIO_ResetBits(GPIOC, GPIO_Pin_8);
        else
                GPIO_SetBits(GPIOC, GPIO_Pin_8);

    if(GPIO_ReadInputDataBit(GPIOE, GPIO_Pin_0))
        GPIO_ResetBits(GPIOC, GPIO_Pin_7);
        else
                GPIO_SetBits(GPIOC, GPIO_Pin_7);
  }
}
```

List. 2. Zmiana funkcji pinów Timera 3

```
void GPIO_Conf(void)
{
  GPIO_InitTypeDef GPIO_InitStruct;

  GPIO_PinRemapConfig(GPIO_FullRemap_TIM3, ENABLE);

  GPIO_InitStruct.GPIO_Pin = GPIO_Pin_6 | GPIO_Pin_7 |
                             GPIO_Pin_8 | GPIO_Pin_9;
  GPIO_InitStruct.GPIO_Mode = GPIO_Mode_AF_PP;
  GPIO_InitStruct.GPIO_Speed = GPIO_Speed_50MHz;

  GPIO_Init(GPIOC, &GPIO_InitStruct);
}
```

przekazuje tak przygotowaną zmienną przez referencję do funkcji inicjującej. W przedstawianej sytuacji, w naszym kręgu zainteresowań leżą wszystkie trzy pola struktury GPIO_InitStruct. W pierwszej kolejności ustalamy, które z pinów będą konfigurowane, następnie wybieramy żądany tryb pracy, w tym przypadku będzie to wyjście *push-pull* lub wejście pływające (*input floating*). Następnie ustalamy maksymalną prędkość, z jaką będą mogły pracować wyprowadzenia układu. Tak przygotowaną zmienną należy przekazać poprzez referencję w argumencie do funkcji inicjującej *GPIO_Init*, podając przy tym również, do jakiego portu mają być zastosowane wybrane ustawienia. Odczytywanie stanu wyprowadzenia, dzięki zdefiniowanym przez firmę STMicroelectronics bibliotekom jest bardzo proste, gdyż podajemy jedynie nazwę konkretnego portu oraz pinu – instrukcje tego typu zawiera nieskończona pętla *while(1)* z list. 1.

Producent mikrokontrolerów STM32 zaleca, aby wszystkie nieużywane wyprowadzenia były skonfigurowane jako analogowe wejścia, czego konsekwencją jest mniejsze zużycie energii oraz większa odporność na EMI. O ile w mniejszych jednostkach 8-bitowych miało to mniejsze znaczenie, to należy pamiętać, że tutaj mikrokontroler pracuje z dużo większą częstotliwością, ponadto znaczna część rodziny STM32 posiada relatywnie dużą liczbę wyprowadzeń, zatem takie ich ustawienie ma istotne znaczenie dla optymalizacji pracy systemu.

Tab. 1. Przykład remapowania pinów Timera 3

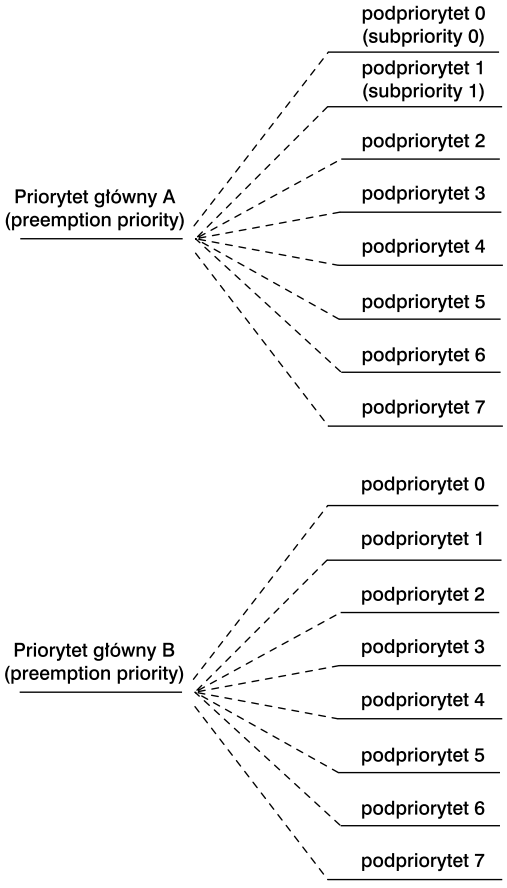
funkcja	domyślnie	częściowy remapping	całkowity remapping
TIM3_CH1	PA6	PB4	PC6
TIM3_CH2	PA7	PB5	PC7
TIM3_CH3	PB0		PC8
TIM3_CH4	PB1		PC9

Funkcje alternatywne i *„remapping"*

W niektórych przypadkach fizyczne rozmieszczenie elementów na docelowej płytce drukowanej nie pozwala na podłączenie urządzenia, bądź elementu zewnętrznego do wyprowadzenia, które jest domyślnie powiązane z interesującą projektanta alternatywną jego funkcją. W takich sytuacjach na ratunek przychodzi możliwość „przemapowania" (*remapping*) GPIO. Jeśli więc przypisanie danej funkcji alternatywnej, np. portu USART nie odpowiada potrzebom projektowanej aplikacji, to można ową funkcję przepisać do innego, bardziej odpowiedniego dla aktualnego zastosowania wyprowadzenia. Takie zabiegi mogą być przeprowadzane tylko dla ściśle określonych wyprowadzeń, co jest szczegółowo podane w nocie katalogowej każdego mikrokontrolera z rodziny STM32. Przykłady możliwości zmiany funkcji różnych pinów dla czterech kanałów Timera 3 są przedstawione w **tab. 1**, natomiast jedno z możliwych rozwiązań programowych przedstawiono na **list. 2**. Do zmiany przypisania domyślnego funkcji alternatywnej służy funkcja *GPIO_PinRemapConfig*. Informacje, które należy przekazać funkcji to określenie interesujących nas peryferiów, podanie czy przemapowanie ma być tylko częściowe, czy całkowite oraz czy włączamy, czy wyłączamy *remapping*. Przedstawiony fragment kodu sprawi, że Timer 3 będzie sterował wyjściami od GPIO_Pin_6 do GPIO_Pin_9. Należy oczywiście pamiętać o włączeniu taktowania dla funkcji alternatywnej i samego portu (tutaj będzie to port GPIOC) w bloku konfiguracji sygnałów zegarowych i zerowania – funkcja *RCC_Conf*.

Przerwania zewnętrzne

Jak wiadomo, aby system mikroprocesorowy poprawnie radził sobie z przychodzącymi zdarzeniami, czy to ze świata zewnętrznego przez porty we/wy, czy od wewnętrznych peryferiów, w ogromnej większości przypadków muszą być one obsługiwane przez przerwania. Ma to szczególne znaczenia dla zadań krytycznych, w których nie może być mowy o zbyt dużych opóźnieniach w wykonaniu owego zadania, ani tym bardziej o pominięciu zdarzenia. Często nie zauważa się tego problemu, ponieważ wydaje się, że na przykład cykliczne sprawdzanie w pętli stanu danego wejścia jest wystarczające. Niestety takie podejście prędzej czy później powoduje generowanie błędów w pracy urządzenia. Jeżeli mamy do czynienia z projektem hobbistycznym, to nie jest to specjalnie dotkliwe, jednakże



Rys. 2. Relacje pomiędzy przerwaniami

w przypadku rozwiązań komercyjnych nie można już sobie na takie błędy pozwolić.

W mikrokontrolerach z rdzeniem Cortex M3 za obsługę przerwań odpowiada sprzętowy kontroler przerwań (NVIC). Dzięki niemu podprogram, który ma się wykonać po nadejściu przerwania jest wywoływany szybciej, a sama implementacja obsługi przerwań jest łatwiejsza, niż miało to miejsce w przypadku rdzeni ARM7 i ARM9.

Fragment programu, który działa w oparciu o zewnętrzne przerwanie przedstawiono na **list. 3**. Najważniejsza jest właściwa konfiguracja MCU, natomiast program, jaki ma być docelowo wywołany umieszcza się w pliku *stm32f10x_it.c*, który, przy wykorzystaniu pakietu CrossWorks, Keil lub Iar, znajduje się domyślnie w projekcie.

System ustalania priorytetów w mikrokontrolerach wyposażonych w rdzeń Cortex M3 jest rozdzielony na dwie części. Przerwania mają przypisany priorytet główny, tzw. *Preemption prioritet*, ponadto ustalany jest dodatkowy poziom podpriorytetów (*subprioritet*). Relacje pomiędzy nimi przedstawiono na **rys. 2**. Taki podział ma uzasadnienie w działaniu: gdy obsługiwane jest w danej chwili przerwanie i nadejdzie zgłoszenie od innego przerwania, to NVIC porównuje priorytety główne. Jeżeli nowe przerwanie dysponuje wyższym priorytetem, następuje jego wywłaszczenie i ono będzie teraz wykonywane. Wartości podpriorytetów mają jedynie znaczenie w momencie, gdy wystąpią dwa przerwania o takim samym priorytecie

List. 3. Fragment programu wykorzystującego przerwanie zewnętrzne

```
int main(void)
{
  RCC_Conf();

#ifdef  VECT_TAB_RAM
  NVIC_SetVectorTable(NVIC_VectTab_RAM, 0x0);
#else  /* VECT_TAB_FLASH  */
  NVIC_SetVectorTable(NVIC_VectTab_FLASH, 0x0);
#endif

  /* Wybranie grupy priorytetów */
  NVIC_PriorityGroupConfig(NVIC_PriorityGroup_1);

  /* Konfiguracja NVIC i wlaczenie obslugi przerwania */
  NVIC_InitStruct.NVIC_IRQChannel = EXTI9_5_IRQChannel;
  NVIC_InitStruct.NVIC_IRQChannelPreemptionPriority = 0;
  NVIC_InitStruct.NVIC_IRQChannelSubPriority = 0;
  NVIC_InitStruct.NVIC_IRQChannelCmd = ENABLE;
  NVIC_Init(&NVIC_InitStruct);

  GPIO_Conf();

  /* Poinformowanie uC o zrodle przerwania */
  GPIO_EXTILineConfig(GPIO_PortSourceGPIOB, GPIO_PinSource9);

  /* Bedzie generowane przerwanie na zboczu opadajacym na EXTI_Line9 */
  EXTI_InitStruct.EXTI_Line = EXTI_Line9;
  EXTI_InitStruct.EXTI_Mode = EXTI_Mode_Interrupt;
  EXTI_InitStruct.EXTI_Trigger = EXTI_Trigger_Falling;
  EXTI_InitStruct.EXTI_LineCmd = ENABLE;
  EXTI_Init(&EXTI_InitStruct);

  /* Wygenerowanie przerwania EXTI_Line9 programowo */
  EXTI_GenerateSWInterrupt(EXTI_Line9);

  while (1);
}

/******* zawartosc pliku stm32f10x.it.c *******/

/* Funkcja obslugi przerwan zewnetrznych od 9 do 5 */
void EXTI9_5_IRQHandler(void)
{
  if(EXTI_GetITStatus(EXTI_Line9) != RESET)
  {
    /* Przerwanie wywoluje zmiane stanu wyprowadzenia */
    GPIO_WriteBit(GPIOC, GPIO_Pin_6, (BitAction)
                  ((1-GPIO_ReadOutputDataBit(GPIOC, GPIO_Pin_6))));

    EXTI_ClearITPendingBit(EXTI_Line9);
  }
}
```

List. 4. Odmierzanie czasu z wykorzystaniem timera SysTick

```
int main(void)
{
  RCC_Conf();
  GPIO_Conf();
  NVIC_Conf();

  /* SysTick bedzie taktowany z f = 72MHz/8 = 9MHz */
  SysTick_CLKSourceConfig(SysTick_CLKSource_HCLK_Div8);

  /* Przerwanie ma byc co 1ms, f = 9MHz czyli liczy od 9000 */
  SysTick_SetReload(9000);

  /* Odblokowanie przerwania od timera SysTick */
  SysTick_ITConfig(ENABLE);

  /* Wlaczenie timera */
  SysTick_CounterCmd(SysTick_Counter_Enable);

  while (1);
}

/******* zawartosc pliku stm32f10x.it.c *******/

/* Funkcja obslugi przerwania od SysTick timer */
void SysTickHandler(void)
{
  if(TDelay == 500)       //co 500ms
  {
    TDelay = 0;
    /* zmienia stan portu na przeciwny */
    GPIO_Write(GPIOC, (u16)~GPIO_ReadOutputData(GPIOC));
  }
  TDelay++;
}
```

głównym w tym samym czasie. W takiej sytuacji w pierwszej kolejności zostanie obsłużone przerwanie o wyższym podpriorytecie. Programista wybierając tzw. grupy priorytetów (*Priority Group*) może ustalać relację pomiędzy liczbą poziomów priorytetów głównych, a liczbą podpriorytetów w zależności od wymagań danej aplikacji. Takich „zestawów" mikrokontrolery STM32 obsługują pięć (od 0 do 4). Jeżeli w docelowym urządzeniu przewidujemy konieczność

List. 5. Fragment programu odpowiedzialny za odpowiednie skonfigurowanie układu licznikowego

```
int main(void)
{
  RCC_Conf();

  RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM3, ENABLE);
  RCC_APB2PeriphClockCmd(RCC_APB2Periph_AFIO, ENABLE);

  RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOC, ENABLE);

  NVIC_Conf();
  GPIO_Conf();

  // Konfiguracja Timera 3
  TIM_TimeBaseStruct.TIM_Period = 999;
  TIM_TimeBaseStruct.TIM_Prescaler = 0;
  TIM_TimeBaseStruct.TIM_ClockDivision = TIM_CKD_DIV1;
  TIM_TimeBaseStruct.TIM_CounterMode = TIM_CounterMode_Up;
  TIM_TimeBaseInit(TIM3, &TIM_TimeBaseStruct);

  // Konfiguracja kanalu 1
  TIM_OCInitStruct.TIM_OCMode = TIM_OCMode_PWM1;
  TIM_OCInitStruct.TIM_OutputState = TIM_OutputState_Enable;
  TIM_OCInitStruct.TIM_Pulse = 950;
  TIM_OCInitStruct.TIM_OCPolarity = TIM_OCPolarity_High;
  TIM_OC1Init(TIM3, &TIM_OCInitStruct);

  TIM_OC1PreloadConfig(TIM3, TIM_OCPreload_Enable);

  // Konfiguracja kanalu 2
  TIM_OCInitStruct.TIM_OCMode = TIM_OCMode_PWM1;
  TIM_OCInitStruct.TIM_OutputState = TIM_OutputState_Enable;
  TIM_OCInitStruct.TIM_Pulse = 350;
  TIM_OCInitStruct.TIM_OCPolarity = TIM_OCPolarity_High;
  TIM_OC2Init(TIM3, &TIM_OCInitStruct);

  TIM_OC2PreloadConfig(TIM3, TIM_OCPreload_Enable);

  // Konfiguracja kanalu 3
  TIM_OCInitStruct.TIM_OCMode = TIM_OCMode_PWM1;
  TIM_OCInitStruct.TIM_OutputState = TIM_OutputState_Enable;

  TIM_OCInitStruct.TIM_Pulse = 250;
  TIM_OCInitStruct.TIM_OCPolarity = TIM_OCPolarity_High;
  TIM_OC3Init(TIM3, &TIM_OCInitStruct);

  TIM_OC3PreloadConfig(TIM3, TIM_OCPreload_Enable);

  // Konfiguracja kanalu 4
  TIM_OCInitStruct.TIM_OCMode = TIM_OCMode_PWM1;
  TIM_OCInitStruct.TIM_OutputState = TIM_OutputState_Enable;
  TIM_OCInitStruct.TIM_Pulse = 100;
  TIM_OCInitStruct.TIM_OCPolarity = TIM_OCPolarity_High;
  TIM_OC4Init(TIM3, &TIM_OCInitStruct);

  TIM_OC4PreloadConfig(TIM3, TIM_OCPreload_Enable);

  TIM_ARRPreloadConfig(TIM3, ENABLE);

  // Wlaczenie Timera 3
  TIM_Cmd(TIM3, ENABLE);

  while(1);
}
```

zastosowania większej liczby zagnieżdżonych przerwań, to w takim wypadku należy wybrać odpowiednio wysoką grupę priorytetów. Zasada jest taka, że im mniejszy numer grupy priorytetów, tym mniej jest do dyspozycji priorytetów głównych, a więcej podpriorytetów. Jest to przedstawione w **tab. 2**. Wynika z niej, że w omawianym przypadku z list. 3 jest wybrana opcja zawierająca dwa priorytety główne (*preemption priority*), każdy dysponuje ośmioma podpriorytetami (*subpriority*). Sumarycznie otrzymujemy w ten sposób 16 możliwych poziomów priorytetów, czyli tyle, ile obsługują mikrokontrolery z rodziny STM32.

Po wybraniu grupy priorytetów należy odpowiednio ustawić parametry wykorzystywanego przerwania. W tym przykładzie będzie to przerwanie pochodzące od przycisku użytkownika na płycie uruchomieniowej, czyli wyprowadzenie PB9. Ponieważ jest to przerwanie zewnętrzne o numerze 9, to trzeba poinformować NVIC o tym, że będzie ono wykorzystywane. Dokonujemy tego modyfikując pole NVIC_IRQChannel struktury inicjującej. Po tej czynności należy ustalić priorytety dla tego konkretnego przerwania. Zarówno priorytet główny, jak i podpriorytet zostają ustawione na zera, a zatem, to przerwanie ma pierwszeństwo w obsłużeniu. Gdy wszystkie pola struktury są już wypełnione, jest ona przekazywana, podobnie jak miało to miejsce w przypadku konfiguracji GPIO, przez referencję do funkcji inicjującej. W ten sposób NVIC jest przygotowany na przyjęcie przerwania, należy jeszcze skonfigurować samo przerwanie zgodnie z wymaganiami.

Wyprowadzenie PB9 jest przyporządkowane do zewnętrznego przerwania o numerze 9 (EXTI_Line9), stąd informujemy o tym MCU wypełniając pole EXTI_Line. Kolejne dwie linie kodu definiują zachowanie wyprowadzenia. Będzie ono pracować w trybie przerwania oraz będzie reagować na zbocze opadające. W chwili, gdy takie zdarzenie wystąpi, mikrokontroler przystępuje do wykonywania funkcji *EXTI9_5_IRQHandler*.

Tab. 2. Przykładowe przypisanie priorytetów i podpriorytetów

grupa	preempiton priority	subpriority
NVIC_PriorityGroup_0	0 bitów	4 bity
NVIC_PriorityGroup_1	1 bit	3 bity
NVIC_PriorityGroup_2	2 bity	2 bity
NVIC_PriorityGroup_3	3 bity	1 bit
NVIC_PriorityGroup_4	4 bity	0 bitów

SysTick Timer

Zadaniem każdego systemu operacyjnego jest takie zarządzenie uruchomionymi wątkami, aby wszystkie zostały optymalnie obsłużone. Można to zrealizować na kilka sposobów. Jednym z nich jest cykliczne – w pewnych określonych ramach czasowych – przełączanie kontekstu zadań. Implementacje tego typu systemów operacyjnych w mikrokontrolerach z rdzeniem Cortex M3 inżynierowie z firmy ARM znacznie uprościli wbudowując w kontroler przerwań NVIC timer SysTick. Jest to 24–bitowy układ licznikowy, który zliczając w dół odmierza określone, zdefiniowane przez programistę interwały czasowe. Każde przejście licznika przez zero i rozpoczęcie nowego cyklu odliczania powoduje wygenerowanie przerwania, które może zajmować się przełączaniem kontekstów uruchomionych w systemie zadań. Takie podejście ma też ważną zaletę, mianowicie zawieszenie się któregoś z realizowanych zadań nie spowoduje zawieszenia całego systemu, ponieważ zawieszony proces zostanie przerwany na rzecz innych zadań przy najbliższym przepełnieniu timera SysTick.

Wbudowanie oddzielnego układu czasowego w architekturę Cortex pozwala na dużo łatwiejsze przenoszenie aplikacji napisanych na mikrokontrolery różnych producentów wykorzystujących tę platformę. Dzieje się tak dlatego, ponieważ w każdym takim mikrokontrolerze znajduje się taki sam timer SysTick.

Oczywiście SysTick może być wykorzystywany do różnych zadań. Konstruktor ustala wartość, od jakiej timer ma liczyć w dół, tym samym wyznacza, jakie odcinki czasowe będą odmierzane. Widać zatem, że może być on również wykorzystywany do standardowego odmierzania czasu i dla takiego przypadku, pokazanego na **list. 4**, zostanie omówiona konfiguracja oraz uruchomienie timera SysTick. Aby jakikolwiek układ czasowy w ogóle działał, musi być taktowany jakimś sygnałem zegarowym. SysTick może zliczać impulsy z taką częstotliwością, z jaką pracuje rdzeń mikrokontrolera, lub z tą częstotliwością podzieloną przez 8. Ta ostatnia opcja jest zastosowana w omawianym przypadku, mamy więc 9 MHz. Licznik liczy w dół, więc regulacja odcinków czasowych, po których są generowane przerwania obywa się poprzez ustawienie wartości początkowej, czym zajmuje się funkcja

SysTick_SetReload. Do poprawnej pracy timera pozostaje jeszcze odblokowanie jego przerwania i włączenie odliczania. Funkcja obsługująca przerwanie to *SysTickHandler.* Jej cykliczne wywoływania powodują odliczenie żądanego czasu i okresową zmianę stanu portu GPIOC, a tym samym miganie diodami LED.

Timery

Pod względem wyposażenia w układy czasowo–licznikowe, mikrokontrolery STM32 prezentują się dość okazale. W sumie mamy do dyspozycji aż siedem timerów (wliczając SysTick), które mogą pracować w różnych konfiguracjach. Ponieważ możliwość odmierzania czasu została już przedstawiona przy okazji omawiania timera SysTick, to teraz zajmiemy się generacją czterech sygnałów PWM o różnym współczynniku wypełnienia. Do tego celu wykorzystamy Timer 3 wraz z jego czterema kanałami. Pozwoli to na wygenerowanie żądanych sygnałów, z tym, że rzecz jasna będą one miały taki sam okres. Fragment programu odpowiedzialny za odpowiednie skonfigurowanie układu licznikowego przedstawiono na **list. 5**. Oprócz standardowej konfiguracji sygnałów zegarowych, która jest wykonywana przez funkcję *RCC_Conf*, należy włączyć taktowanie timera i GPIO, natomiast funkcja *GPIO_Conf*, której zadaniem jest pełne przemapowanie Timera 3 jest identyczna z przedstawioną wcześniej na list. 2. W następnych krokach definiujemy podstawowe parametry pracy licznika. Będzie on pracował jako licznik w górę z częstotliwością taktowania 36 MHz (nie dzielimy sygnału zegarowego i nie wykorzystujemy preskalera). Wartość, do jakiej będzie liczył timer to 999, po czym nastąpi przepełnienie i proces liczenia rozpocznie się od nowa. Przy takich ustawieniach częstotliwość generowanego przebiegu PWM wyniesie: f=36 [MHz]/(999+1)=36 [kHz]. Kolejnym krokiem jest konfiguracja poszczególnych kanałów timera. Informujemy MCU o tym, że poszczególne kanały będą pracować w trybie PWM, długość impulsu będzie wynosiła odpowiednio (licząc od kanału pierwszego): 950, 350, 250, 100 taktów licznika. Aktywnym stanem jest stan wysoki, co oznacza, że np. dla kanału drugiego stan wysoki będzie panował na wyjściu PC7 przez 350 taktów, a pozostałe 650 to stan niski, zatem współczynnik wypełnienia wyniesie w tym przypadku 35%. Po zaprogramowaniu mikrokontrolera i uruchomieniu układu, diody od LD1 do LD4 będą świecić z różną intensywnością.

Na zakończenie

Z przedstawionych pobieżnie przykładów widać, jak wielkie możliwości drzemią we współczesnych mikrokontrolerach 32–bitowych. Zarówno projektanci rdzenia Cortex M3, jak i producent układów STM32, firma STMicroelectronics dołożyli wszelkich starań, aby czas potrzebny na napisanie i uruchomienie aplikacji był możliwie krótki. Dzięki temu otrzymujemy do dyspozycji świetnie wyposażone mikrokontrolery, które poza tym można stosunkowo łatwo programować. Ponadto układy z rdzeniami Cortex znajdują się w ofercie coraz większej liczby producentów, zatem należy się spodziewać dalszych spadków ich cen i wzrostu możliwości. Wyścig trwa...

Krzysztof Paprocki
paprocki.krzysztof@gmail.com