32 bity jak najprościej (2)

Pierwsze kroki z modułem STM32F0DISCOVERY

Samouczek jest dedykowany szczególnie tym projektantom, którzy stają przed perspektywą zmiany mikrokontrolera z 8-bitowego na nowszy i tańszy 32-bitowy. Wszystkie programy są napisane w języku C. W przykładach nie użyto bibliotek służących do obsługi peryferiali, dzięki czemu kod programów jest krótki i czytelny, a zajętość pamięci znacznie mniejsza, niż w typowych programach demonstracyjnych, udostępnianych przez producentów mikrokontrolerów. Zwrócono również szczególną uwagę na poprawność prezentowanych rozwiązań i sposób zapisu programu ułatwiający optymalizację kodu i wychwytywanie błędów przez kompilator. Przedstawione programy zostały napisane w taki sposób, że nie generują one żadnych ostrzeżeń kompilatora.

Odmierzanie czasu przy użyciu timera systemowego SysTick

Kolejny program będzie, tak jak poprzedni, migał diodami, ale tym razem użyjemy prawidłowej techniki odmierzania czasu przy użyciu timera zgłaszającego przerwania ze stałą, zadaną przez programistę częstotliwością.

Gotowy program znajduje się w projekcie blink1.

Zaprogramowanie timera SysTick

Timer *SysTick* jest częścią samego procesora Cortex-M. Zwykle służy on do zgłaszania przerwań ze stałą częstotliwością. W naszym przykładzie zaprogramujemy go tak, aby zgłaszał przerwania z częstotliwością 100 Hz. Zadanie obliczenia wartości, którą należy w tym celu załadować do rejestr okresu timera, pozostawimy kompilatorowi, zapisując stosowne wyrażenie w instrukcji ładowania rejestru sterującego timera. Zdefiniujemy jako symbol preprocesora częstotliwość procesora *SYSCLK_FREQ*. Jest ona równa częstotliwości wewnętrznego oscylatora *HSI_VALUE* zdefiniowanej w pliku nagłówkowym definicji zasobów mikrokontrolera. Zdefiniujemy również częstotliwość zgłaszania przerwań przez timer *SYSTICK_FREQ* podając jej wartość w Hz oraz okres zmiany stanu diod, wyrażając go w okresach timera *SysTick*, czyli w dziesiątkach milisekund. Programowanie portów wygląda tak samo, jak w poprzednim przykładzie. Następną czynnością jest zaprogramowanie timera *SysTick*. W tym celu kolejno:

- ustawiamy w rejestrze *LOAD* maksymalną wartość licznika timera (o 1 mniejszą od długości okresu),
- zerujemy rejestr licznika czasu *VAL*,
- wybieramy jako źródło zegara timera zegar procesora, włączamy przerwanie timera i uruchamiamy timer zapisując odpowiednią stałą do rejestru *CTRL*.

Struktura programu

Zmianą stanu diod zajmie się procedura obsług przerwania timera. Rola programu głównego kończy się na zainicjowaniu portu i timera, dlatego po wykonaniu tych czynności włączymy w procesorze automatyczne usypianie po powrocie z obsługi przerwania (zapis rejestru SCR procesora), a następnie uśpimy procesor w oczekiwaniu na przerwanie (pseudofunkcja *__WFI()*). Na tym zakończy się wykonanie programu głównego. Procesor będzie budził się na czas obsługi przerwania, a po zakończeniu obsługi będzie usypiał. Nie jest potrzebna żadna „pętla główna”, nawet pusta. Taki sposób programowania jest typowy dla większości prostych aplikacji, które nie wymagają użycia systemu operacyjnego i zapewnia oszczędność energii.

Przerwanie timera *SysTick* nie wymaga oddzielnego włączenia w sterowniku przerwań *NVIC*, gdyż timer ten nie jest traktowany jak moduł peryferyjny.

Musimy napisać procedurę obsługi przerwania timera. Jej nazwa jest zdefiniowana w module startowym. Procedura deklaruje zmienną statyczną blink_timer, służącą do odliczania czasu pomiędzy zmianami stanu diod. Jest ona dekrementowana przy każdym przerwaniu, a po jej wyzerowaniu następuje załadowanie długości okresu i zmiana stanu diod.

Dlaczego nie zaprogramowaliśmy timera na docelową częstotliwość migotania diod? Oczywiście można by było tak zrobić, ale zazwyczaj w programach, służących do czegoś więcej niż demonstracja działania mikrokontrolera, potrzebujemy odmierzania dużo krótszych odcinków czasu – typowo od 1 do 10 ms. Do sygnalizacji stanu urządzenia użyjemy tego samego timera, który byłby użyty do generowania bazy czasu urządzenia, zmieniając stan diod co pewną liczbę przerwań. Zademonstrowane w programie zamieszczonym na **listingu 1** rozwiązanie jest często spotykane w realnym oprogramowaniu.

Drobne usprawnienia

Kolejny projekt, *blink2*, nie różni się funkcjonalnie od poprzedniego. Wprowadzono w nim jednak dwie modyfikacje:

- Procedura *SystemInit* włącza bufor pobierania instrukcji, co powoduje wzrost wydajności procesora.
- Sekwencję instrukcji podstawienia służącą do inicjowania peryferiali zastępujemy prostą funkcją

Listing 1. Program demonstracyjny – blink 1 (migotanie diody LED)

```
/*
  STM32F0DISCOVERY
  SysTick-based blinker
  gbm, 12'2012
*/

#include „stm32f0xx.h"
//============================================================================
// defs for STM32F05x chips
#define GPIO_MODER_OUT     1
//============================================================================
// defs for STM32F0DISCOVERY board
#define LED_PORT           GPIOC
#define BLUE_LED_BIT       8
#define GREEN_LED_BIT      9
//============================================================================
// Timings
#define SYSCLK_FREQ                HSI_VALUE
#define SYSTICK_FREQ       100
#define BLINK_PERIOD       50       // * 10 ms
//============================================================================
void SystemInit(void)
{
}
//============================================================================
int main(void)
{
  // port setup
  RCC->AHBENR = RCC_AHBENR_GPIOCEN;        // GPIOC
  LED_PORT->MODER = GPIO_MODER_OUT << (GREEN_LED_BIT << 1)
                  | GPIO_MODER_OUT << (BLUE_LED_BIT << 1);         // LED outputs
  LED_PORT->ODR = 1 << GREEN_LED_BIT;      // green initially ON
  //SysTick setup
  SysTick->LOAD = SYSCLK_FREQ / SYSTICK_FREQ - 1;
  SysTick->VAL = 0;
  SysTick->CTRL = SysTick_CTRL_CLKSOURCE_Msk | SysTick_CTRL_TICKINT_Msk
                | SysTick_CTRL_ENABLE_Msk;
  SCB->SCR = SCB_SCR_SLEEPONEXIT_Msk;      // sleep while not in handler
  __WFI();       // go to sleep
}
//============================================================================
void SysTick_Handler(void)
{
  static uint8_t blink_timer = BLINK_PERIOD;
  if (-- blink_timer == 0)
  {
    blink_timer = BLINK_PERIOD;
    // toggle both LEDs
    LED_PORT->ODR ^= 1 << GREEN_LED_BIT | 1 << BLUE_LED_BIT;
  }
}
```

Listing 2. Zmodyfikowany program demonstracyjny – blink 2

```
/*
  STM32F0DISCOVERY tutorial
  SysTick-based blinker with table-driven init
  gbm, 12'2012
*/

#include „stm32f0xx.h"
//============================================================================
// defs for STM32F05x chips
#define GPIO_MODER_OUT     1
//============================================================================
// defs for STM32F0DISCOVERY board
#define LED_PORT    GPIOC
#define BLUE_LED_BIT  8
#define GREEN_LED_BIT  9
//============================================================================
// Timings
#define SYSCLK_FREQ                HSI_VALUE
#define SYSTICK_FREQ       100      // 100 Hz -> 10 ms
#define BLINK_PERIOD       50       // * 10 ms
//============================================================================
struct init_entry_ {
  volatile uint32_t *loc;
  uint32_t value;
};

static __INLINE void writeregs(const struct init_entry_ *p)
{
  for (; p->loc; p ++) *p->loc = p->value;
}
//============================================================================
void SystemInit(void)
{
  FLASH->ACR = FLASH_ACR_PRFTBE;  // enable prefetch
}
//============================================================================
static const struct init_entry_ init_table[] =
{
  // port setup
  {&RCC->AHBENR, RCC_AHBENR_GPIOCEN},  // GPIOC
  {   &LED_PORT->MODER, GPIO_MODER_OUT << (GREEN_LED_BIT << 1)
      | GPIO_MODER_OUT << (BLUE_LED_BIT << 1)
  },     // set LED pins as outputs
  {(__IO uint32_t *)&LED_PORT->ODR, 1 << GREEN_LED_BIT},  // green initially ON
  //SysTick setup
  {&SysTick->LOAD, SYSCLK_FREQ / SYSTICK_FREQ - 1},
```

Listing 2. c.d.

```
  {&SysTick->VAL, 0},
  {   &SysTick->CTRL, SysTick_CTRL_CLKSOURCE_Msk | SysTick_CTRL_TICKINT_Msk
      | SysTick_CTRL_ENABLE_Msk
  },
  {&SCB->SCR, SCB_SCR_SLEEPONEXIT_Msk},    // sleep while not in handler
  {0, 0}
};
//============================================================================
int main(void)
{
  writeregs(init_table);
  __WFI();        // go to sleep
}
//============================================================================
void SysTick_Handler(void)
{
  static uint8_t blink_timer = BLINK_PERIOD;
  if (-- blink_timer == 0)
  {
    blink_timer = BLINK_PERIOD;
    // toggle both LEDs
    LED_PORT->ODR ^= 1 << GREEN_LED_BIT | 1 << BLUE_LED_BIT;
  }
}
```

Listing 3. Program demonstracyjny – zmiana stanu świecenia diod za pomocą przycisku

```
/*
  STM32F0DISCOVERY tutorial
  SysTick-based LED toggle by button press
  gbm, 12'2012
*/

#include „stm32f0xx.h"
//============================================================================
// defs for STM32F05x chips
#define GPIO_MODER_OUT     1
//============================================================================
// defs for STM32F0DISCOVERY board
#define LED_PORT           GPIOC
#define BLUE_LED_BIT       8
#define GREEN_LED_BIT      9
#define BUTTON_PORT                GPIOA
#define BUTTON_BIT         0
//============================================================================
// Timings
#define SYSCLK_FREQ                HSI_VALUE
#define SYSTICK_FREQ       100     // 100 Hz -> 10 ms
//============================================================================
struct init_entry_ {
  volatile uint32_t *loc;
  uint32_t value;
};

static __INLINE void writeregs(const struct init_entry_ *p)
{
  for (; p->loc; p ++) *p->loc = p->value;
}
//============================================================================
void SystemInit(void)
{
  FLASH->ACR = FLASH_ACR_PRFTBE;  // enable prefetch
}
//============================================================================
static const struct init_entry_ init_table[] =
{
  // port setup
  {&RCC->AHBENR, RCC_AHBENR_GPIOCEN |  RCC_AHBENR_GPIOAEN}, // GPIOC, GPIOA
  {   &LED_PORT->MODER, GPIO_MODER_OUT << (GREEN_LED_BIT << 1)
      | GPIO_MODER_OUT << (BLUE_LED_BIT << 1)
  },     // set LED pins as outputs
  {(__IO uint32_t *)&LED_PORT->ODR, 1 << GREEN_LED_BIT},    // set green LED on
  // SysTick setup
  {&SysTick->LOAD, SYSCLK_FREQ / SYSTICK_FREQ - 1},
  {&SysTick->VAL, 0},
  {   &SysTick->CTRL, SysTick_CTRL_CLKSOURCE_Msk | SysTick_CTRL_TICKINT_Msk
      | SysTick_CTRL_ENABLE_Msk
  },
  // sleep
  {&SCB->SCR, SCB_SCR_SLEEPONEXIT_Msk},    // sleep while not in handler
  {0, 0}
};

//============================================================================
int main(void)
{
  writeregs(init_table);
  __WFI();        // go to sleep
}
//============================================================================
void SysTick_Handler(void)
{
  static uint8_t bstate = 0;       // button history - last 4 samples
  if ((bstate = (bstate << 1 & 0xf) | (BUTTON_PORT->IDR >> BUTTON_BIT & 1)) == 1)
  // button was released, now it is pressed - change LED state
    LED_PORT->ODR ^= 1 << GREEN_LED_BIT | 1 << BLUE_LED_BIT;
}
```

kopiującą dane pod wskazane adresy oraz tablicą zawierającą adresy rejestrów sterujących i wartości, które mają być do nich zapisane. Takie rozwiązanie sprawdza się zwłaszcza w bardziej złożonych programach, gdzie należy zainicjować kilkadziesiąt rejestrów sterujących – podnosi ono czytelność kodu i redukuje błędy. W tak zmodyfikowanym programie warto zwrócić uwagę na następujące szczegóły:

– W programie konsekwentnie ograniczono widoczność danych i procedury *writeregs*, deklarując je z atrybutem *static*, który w tym przypadku oznacza, że będą to obiekty prywatne dla modułu, w którym zostały zdefiniowane. Umożliwia to kompilatorowi skuteczniejszą optymalizację i zwiększa prawdopodobieństwo wykrycia przez kompilator błędów w kodzie.
– Tablica *init_table* została zadeklarowana z atrybutem *const*, dzięki czemu nie zajmuje ona miejsca w pamięci RAM.
– Procedura writeregs została zadeklarowana z atrybutem *inline* (użyto tu makrodefinicji *__INLINE*, gdyż kompilator środowiska Keil nie reaguje na standardowe słowo kluczowe). Jest to sugestia dla kompilatora, że może on zastąpić wywołanie procedury wstawieniem jej ciała w miejsce, z którego jest wywoływana.
– W tablicy inicjowania użyto rzutowania typu wskaźnikowego (*__IO uint32_t* *); wynika to z nietypowej definicji wielu rejestrów sterujących w pliku nagłówkowym dla STM32F0 – są one zdefiniowane jako 16-bitowe.

Zmodyfikowany program zamieszczono na **listingu 2**.

Sterowanie świeceniem diod przy użyciu przycisku

Ponieważ na płytce DISCOVERY jest umieszczony przycisk (koloru niebieskiego), którego stan może być testowany przez mikrokontroler, wypada użyć go w kolejnym projekcie. Tym razem diody nie będą zaświecały się i gasły samoczynnie, lecz będą one sterowane przez naciśnięcie przycisku. Każde naciśnięcie spowoduje zmianę stanu świecenia obu diod.

Ten projekt jest dobrym pretekstem do zaprezentowania techniki programowego ignorowania drgań styków przycisku. W celu uniknięcia reakcji na drgania styków, stan przycisku będzie testowany ze stałą częstotliwością (100 razy na sekundę), a reakcja na naciśnięcie nastąpi, gdy po trzech kolejnych okresach zwolnienia przycisku zostanie zarejestrowane jego naciśnięcie. Dzięki temu program nie będzie reagował na naprzemienne odczyty naciśnięcia i zwolnienia, które mogą mieć miejsce w przypadku drgań styków. Taka technika obsługi przycisku może być stosowana, gdy nie występują zmiany stanów wejścia przycisku spowodowane czynnikami innymi niż samym drżeniem styków, np. zakłóceniami elektrycznymi na liniach łączących przyciski z wejściami mikrokontrolera.

Program zaprezentowany na **listingu 3** różni się od poprzedniego sekwencją inicjującą oraz procedurą obsługi przerwania timera *SysTick*. Sekwencję inicjującą uzupełniono o włączenie portu A, do którego podłączony jest przycisk. W procedurze obsługi przerwania SysTick zdefiniowano zmienną statyczną bstate, przechowującą cztery ostatnio zarejestrowane stany przycisku. Zmienna ta zachowuje się jak 4-bitowy rejestr przesuwający. Na początku obsługi przerwania jest ona przesuwana o jeden bit w lewo, a na najmniej znaczący bit wsuwana jest informacja o nowym stanie przycisku. Stany wejść portu GPIO są dostępne dla oprogramowania w rejestrze wejściowym portu GPIO - IDR. Ponieważ przycisk został podłączony na płytce STM32F0DISCOVERY dość nietypowo – pomiędzy dodatnim biegunem zasilania i wejściem, przy naciśnięciu przycisku wejście portu przyjmie stan 1. Procedura obsługi przerwania zmieni stan diod, gdy zmienna bstate osiągnie wartość 1 (binarnie - 0001), co odpowiada trzem okresom zwolnienia i następującym po nim naciśnięciu przycisku.

Program demonstracyjny pokazano na listingu 3.

Przełączanie trybu migotania diody przy użyciu przycisku

Kolejny projekt, BtnBlink, jest niemal dokładnym funkcjonalnym odpowiednikiem przykładowego programu dostarczanego wraz z płytką STM32F0DISCOVERY, dzięki rezygnacji z użycia bibliotek do obsługi peryferiali jest jednak od niego krótszy i przejrzystszy w zapisie i zajmuje mniej miejsca w pamięci. Nowy program stworzymy na bazie poprzedniego, uzupełniając go o obsługę zmiany trybu świecenia diod.

Procedura obsługi przerwania timera SysTick zostanie rozbudowana o reakcję na naciśnięcie przycisku. Każde naciśnięcie przycisku ma powodować zaświecenie niebieskiej diody na określony czas (1 sekunda) oraz zmieniać tryb sterowania diody zielonej. Trzy tryby odpowiadają kolejno: wolnemu migotaniu, szybkiemu migotaniu i wygaszeniu. Program nie reaguje na naciśnięcie przycisku podczas świecenia diody niebieskiej.

W procedurze obsługi przerwania zadeklarujemy następujące zmienne statyczne (czyli takie, których wartości są zachowywane pomiędzy wywołaniami procedury):

– *blink_mode* – określa tryb sterowania zielonej diody,
– *blink_timer* – do odmierzania okresu świecenia zielonej diody,
– *blue_led_timer* – do odmierzania czasu świecenia diody niebieskiej,
– *green_on_time* – przechowuje czas zaświecenia diody zielonej,
– *bstate* – przechowuje stan przycisku rejestrowany po każdym przerwaniu timera.

Tablica stałych *blink_period[]* zawiera okresy świecenia zielonej diody dla poszczególnych trybów pracy. Na końcu każdego okresu świecenia następuje ustawienie czasu kolejnego okresu świecenia i czasu zapalenia diody równego połowie okresu świecenia, zaokrąglonej w dół. Dla trybu wygaszenia (okres równy 1) czas świecenia jest równy 0.

Do zaświecania i gaszenia diod użyto rejestrów *BSRR* i *BRR* modułu GPIO. Umożliwiają one zmianę stanu dowolnych bitów portu z zachowaniem stanu pozostałych bitów, bez wcześniejszego odczytu stanu portu i konieczności wykonywania operacji logicznych Zapis jedynki do dolnej połowy rejestru *eeeee* powoduje ustawienie wyjścia w stan 1, a zapis jedynki do BRR – ustawienie wyjścia w stan 0. Ustawienie wyjścia w stan 0 jest również możliwe poprzez zapis jedynki do górnej połowy rejestru BSRR.

Naciśnięcie nie jest wykrywane w czasie świecenia niebieskiej diody, czyli w ciągu sekundy od poprzednie-

Listing 4. Program demonstracyjny – przełączanie trybu migotania diody przy użyciu przycisku

```
/*
    STM32F0DISCOVERY tutorial
    SysTick-based blinker with table-driven init and blink mode switching
    gbm, 12'2012
*/

#include „stm32f0xx.h"
//=========================================================================
// defs for STM32F0DISCOVERY board
#define LED_PORT            GPIOC
#define BLUE_LED_BIT        8
#define GREEN_LED_BIT       9
#define BUTTON_PORT                 GPIOA
#define BUTTON_BIT          0
#define GPIO_MODER_OUT      1
#define SYSCLK_FREQ         HSI_VALUE

#define SYSTICK_FREQ        100     // 100 Hz -> 10 ms
#define BLINK_PERIOD        50      // * 10 ms
#define BLUE_ACK_TIME       100
//=========================================================================
struct init_entry_ {
    volatile uint32_t *loc;
    uint32_t value;
};

static __INLINE void writeregs(const struct init_entry_ *p)
{
    for (; p->loc; p ++)
        *p->loc = p->value;
}

//=========================================================================
void SystemInit(void)
{
    FLASH->ACR = FLASH_ACR_PRFTBE; // enable prefetch
}
//=========================================================================
static const struct init_entry_ init_table[] =
{
    // port setup
    {&RCC->AHBENR, RCC_AHBENR_GPIOCEN | RCC_AHBENR_GPIOAEN},         // GPIOC, GPIOA
    {   &LED_PORT->MODER, GPIO_MODER_OUT << (GREEN_LED_BIT << 1)
        | GPIO_MODER_OUT << (BLUE_LED_BIT << 1)
    },   // set LED pins as outputs
    //SysTick setup
    {&SysTick->LOAD, SYSCLK_FREQ / SYSTICK_FREQ - 1},
    {&SysTick->VAL, 0},
    {   &SysTick->CTRL, SysTick_CTRL_CLKSOURCE_Msk | SysTick_CTRL_TICKINT_Msk
        | SysTick_CTRL_ENABLE_Msk
    },
    {&SCB->SCR, SCB_SCR_SLEEPONEXIT_Msk},  // sleep while not in handler
    {0, 0}
};
//=========================================================================
int main(void)
{
    writeregs(init_table);
    __WFI();     // go to sleep
}
//=========================================================================
void SysTick_Handler(void)
{
    static enum {BM_SLOW, BM_FAST, BM_OFF} blink_mode = BM_SLOW;
    static uint8_t blink_timer = BLINK_PERIOD;
    static uint8_t blue_led_timer = 0, green_on_time = 0;
    static uint8_t bstate = 0;
    static const uint8_t blink_period[] = {100, 50, 1};

    bstate = (bstate << 1 & 0xf) | (BUTTON_PORT->IDR >> BUTTON_BIT & 1);

    if (blue_led_timer)
    {
        if (-- blue_led_timer == 0)
            LED_PORT->BRR = 1 << BLUE_LED_BIT;      // blue OFF
    }
    else if (bstate == 1)
    {
        // button was released, now it is pressed - change blink mode
        if (++ blink_mode > BM_OFF)
            blink_mode = BM_SLOW;
        blue_led_timer = BLUE_ACK_TIME;
        LED_PORT->BSRR = 1 << BLUE_LED_BIT;         // blue ON
    }

    if (blue_led_timer && -- blue_led_timer == 0)
        LED_PORT->BRR = 1 << BLUE_LED_BIT; // blue OFF

    if (-- blink_timer == 0)
    {
        blink_timer = blink_period[blink_mode];
        green_on_time = blink_timer >> 1;
        LED_PORT->BRR = 1 << GREEN_LED_BIT;         // green OFF
    }
    else if (blink_timer == green_on_time)
        LED_PORT->BSRR = 1 << GREEN_LED_BIT;        // green ON
}
```

go naciśnięcia przycisku – wtedy, gdy zmienna blue_led_timer ma wartość różną od zera. Program demonstrujący opisane funkcjonalności zamieszczono na **listingu 4**.

Sterowanie LED przebiegiem PWM

W kolejnym projekcie, PWMblink1, zamiast włączać i wyłączać diody będziemy zmieniać ich jasność poprzez sterowanie wypełnieniem impulsów powodujących świecenie diod. Użyjemy do tego celu timera TIM3, ponieważ może on sterować sprzętowo liniami portów, do których są dołączone diody. Nie będzie w tym projekcie potrzebny timer SysTick – do zgłaszania periodycznych przerwań użyjemy timera TIM3.

Timer jako generator PWM

Mikrokontrolery STM32F05x są wyposażone w kilka modułów timerów. Poszczególne timery są do siebie podobne, chociaż ich bardziej zaawansowane własności są nieco zróżnicowane. Każdy timer jest wyposażony w preskaler, umożliwiający podział częstotliwości przebiegu zegarowego przez dowolną wartość 16-bitową. Każdy timer ma również rejestr długości okresu ARR. Timery, które mogą być użyte do generowania przebiegów PWM, są wyposażone w rejestry wartości wypełnienia dla każdego kanału – CCRx. Współczynnik wypełnienia jest określony przez iloraz wartości wypełnienia i długości okresu timera. Rejestry sterujące timera umożliwiają włączenie generowania przebiegów PWM, określenie ich polaryzacji oraz sposobu synchronizacji modyfikacji wypełnienia z okresem timera.

Timer zlicza od zera do wartości zadanej w rejestrze ARR, po czym następuje wyzerowanie timera i dalsze zliczanie. Osiągnięcie przez licznik timera wartości równej wartości rejestru CCRx powoduje zmianę stanu wyjścia PWM.

Określenie parametrów czasowych

Przygotowanie projektu zaczynamy od określenia wartości parametrów czasowych. W celu uniknięcia zauważalnego migotania diod częstotliwość sterujących ich świeceniem przebiegów PWM będzie równa 400 Hz. Diody będą miały 80 stopni jasności, a więc okres timera będzie równy 80 cyklom zegara wejściowego. Do określenia częstotliwości zegara timera służy preskaler. Stopień podziału preskalera zostanie wyliczony przez kompilator na podstawie wyrażenia podanego w programie. Wartości wpisywane do rejestrów preskalera i okresu timera są o jeden mniejsze od stopnia podziału i długości okresu.

Programowanie peryferiali

Użycie timera do generowania przebiegu PWM wymaga zaprogramowania linii portów sterujących diodami jako wyjść timera. W celu zaprogramowania timera do generowania dwóch przebiegów PWM oraz zgłaszania przerwania na końcu okresu należy:

- Ustawić linie portów używanych do sterowania diod jako wyjścia timera *TIM3* (funkcja *AF*) poprzez zapis do rejestru *MODER* portu *GPIOC*.
- Włączyć timer poprzez ustawienie bitu w rejestrze *APB1ENR*.
- Ustawić wartość preskalera w rejestrze *PSC*.
- Ustawić okres timera w rejestrze *ARR*.
- Ustawić początkowe wartości wypełnień w rejestrach *CCRx*.
- Włączyć tryb PWM z buforowaniem rejestrów *CCRx* poprzez zapis rejestru *CCMR2*.
- Włączyć sterowanie wyjść PWM przez timer poprzez zapis do rejestru *CCER*.
- Włączyć zgłaszanie przerwań na końcu okresu timera – rejestr *DIER*.
- Włączyć automatyczne ładowanie okresu i uruchomić timer – rejestr *CR1*.
- Włączyć przerwanie timera w sterowniku przerwań – rejestr ISER[0].

Obsługa przerwania timera

Przerwanie timera TIM3 jest zgłaszane z częstotliwością 400 Hz. Procedura obsługi przerwania musi skasować zgłoszenie przerwania poprzez zapis bitu o wartości 0 na odpowiednią pozycję rejestru *SR* timera, zawierającego znaczniki przerwań; w przeciwnym przypadku procedura obsługi przerwania timera byłaby wywoływana ciągle.

Ponieważ pozostałe czynności będą wykonywane z częstotliwością 100 Hz, użyjemy zmiennej *tdiv* zliczającej przerwania i instrukcji warunkowej z blokiem warunkowym wykonywanym co cztery przerwania – gdy dwa najmniej znaczące bity *tdiv* będą miały wartość 0.

Obsługa przerwania jest podobna do obsługi z poprzedniego przykładu, tym razem jednak nie będziemy zaświecać i gasić diod poprzez ustawienie stanu linii portu – zamiast tego będziemy modyfikować ich jasność poprzez ustawienie wypełnienia w rejestrach *CCRx*. W naszym przykładzie diody nie będą całkowicie gaszone, zmieniana będzie tylko ich jasność pomiędzy minimalną i maksymalną. Służą do tego stałe *LED_MAX* i *LED_DIM*.

Odpowiedni program demonstracyjny zamieszczono na **listingu 5**.

Płynna zmiana jasności diod

Kolejny projekt będzie zmodyfikowaną wersją poprzedniego. Tym razem zmiana jasności diod nie będzie następowała natychmiast, lecz płynnie. Przejście od jasności minimalnej do maksymalnej zajmie ok. 1/5 sekundy. Osiągniemy to poprzez wprowadzenie zmiennych przechowujących docelowe wartości wypełnień i powolną modyfikację wartości rejestrów wypełnień w przerwaniu timera poprzez ich inkrementację lub dekrementację, aż do uzyskania wartości zadanej. Gotowy projekt nosi nazwę *PWMblink2*.

Oprogramowanie ma taką samą strukturę, jak poprzednio. Różni się ono od poprzedniej wersji projektu w następujących szczegółach:

- Wprowadzono zmienne *blue_target* i *green_target* przechowujące zadane docelowe wartości wypełnienia dla obu diod.
- Ustawienie docelowej wartości wypełnienia następuje poprzez zapis zmiennej (a nie, jak poprzednio, przez zapis rejestru *CCRx*).
- W każdym przerwaniu timera następuje porównanie rejestrów wypełnień ze zmiennymi wypełnień zadanych, a w przypadku stwierdzenia różnicy – inkrementacja lub dekrementacja rejestru wypełnienia. Służą do tego dwie instrukcje warunkowe umieszczone na końcu procedury obsługi przerwania TIM3.

Listing 5. Program demonstracyjny – sterowanie LED przebiegiem PWM

```
/*
    STM32F0DISCOVERY tutorial
    TIM3 PWM blinker with table-driven init and blink mode switching
    gbm, 12'2012
*/

#include „stm32f0xx.h"
//=========================================================================
// defs for STM32F05x chips
#define GPIO_MODER_OUT     1
#define GPIO_MODER_AF      2
#define TIM_CCMR2_OC3M_PWM1         0x0060   // OC3M[2:0] - PWM mode 1
#define TIM_CCMR2_OC4M_PWM1         0x6000   // OC3M[2:0] - PWM mode 1
typedef __IO uint32_t * __IO32p;
//=========================================================================
// defs for STM32F0DISCOVERY board
#define LED_PORT           GPIOC
#define BLUE_LED_BIT       8
#define GREEN_LED_BIT      9
#define BUTTON_PORT                GPIOA
#define BUTTON_BIT         0
#define BLUE_LED_PWM       TIM3->CCR3
#define GREEN_LED_PWM      TIM3->CCR4
//=========================================================================
// Timings
#define SYSCLK_FREQ        HSI_VALUE
#define BLINK_PERIOD       50       // * 10 ms
// PWM constants
#define PWM_FREQ  400      // Hz
#define PWM_STEPS 80
#define   PWM_CLK          SYSCLK_FREQ
#define   PWM_PRE          (PWM_CLK / PWM_FREQ / PWM_STEPS)
#define LED_MAX    (PWM_STEPS - 1)
#define LED_DIM    1
#define LED_OFF    0
//=========================================================================
struct init_entry_ {
    volatile uint32_t *loc;
    uint32_t value;
};

static __INLINE void writeregs(const struct init_entry_ *p)
{
    for (; p->loc; p ++) *p->loc = p->value;
}
//=========================================================================
void SystemInit(void)
{
    FLASH->ACR = FLASH_ACR_PRFTBE;// enable prefetch
}
//=========================================================================
static const struct init_entry_ init_table[] =
{
    // port setup
    {&RCC->AHBENR, RCC_AHBENR_GPIOCEN |  RCC_AHBENR_GPIOAEN},        // GPIOC, GPIOA
    {&LED_PORT->MODER, GPIO_MODER_AF << (GREEN_LED_BIT << 1)
     | GPIO_MODER_AF << (BLUE_LED_BIT << 1)},       // set LED pins as AF
    {&RCC->AHBENR, RCC_AHBENR_GPIOAEN},    // GPIOA
    // PWM timer setup - TIM3
    {&RCC->APB1ENR, RCC_APB1ENR_TIM3EN},   // TIM3
    {(__IO32p)&TIM3->PSC, PWM_PRE - 1},
    {(__IO32p)&TIM3->ARR, PWM_STEPS - 1},
    // blue - CH3, green - CH4
    {(__IO32p)&BLUE_LED_PWM, LED_DIM},
    {(__IO32p)&GREEN_LED_PWM, LED_DIM},
    {(__IO32p)&TIM3->CCMR2, TIM_CCMR2_OC4M_PWM1 | TIM_CCMR2_OC4PE
        | TIM_CCMR2_OC3M_PWM1 | TIM_CCMR2_OC3PE},   // PWM mode 1, buffered preload
    {(__IO32p)&TIM3->CCER, TIM_CCER_CC4E | TIM_CCER_CC3E},  // enable CH3, 4 output
    {(__IO32p)&TIM3->DIER, TIM_DIER_UIE},  // enable update interrupt
        {(__IO32p)&TIM3->CR1, TIM_CR1_ARPE | TIM_CR1_CEN}, // auto reload, enable
        // interrupts and sleep
        {&NVIC->ISER[0], 1 << TIM3_IRQn}, // enable interrupt
        {&SCB->SCR, SCB_SCR_SLEEPONEXIT_Msk},      // sleep while not in handler
        {0, 0}
};
//=========================================================================
int main(void)
{
    writeregs(init_table);
    __WFI();      // go to sleep
}
//=========================================================================
void TIM3_IRQHandler(void)
{
    static uint8_t blink_timer = BLINK_PERIOD;
    static enum {BM_SLOW, BM_FAST, BM_OFF} blink_mode = BM_SLOW;
    static uint8_t bstate = 0;
    static uint8_t blue_led_timer = 0, on_time = 0;
    static const uint8_t blink_periods[] = {100, 50, 1};
    static uint8_t tdiv;
    TIM3->SR = ~TIM_SR_UIF;        // clear interrupt flag
    if ((++ tdiv & 3) == 0)
    {
            // 100 Hz
        bstate = (bstate << 1 & 0xf) | (BUTTON_PORT->IDR >> BUTTON_BIT & 1);
        if (blue_led_timer)
        {
            if (-- blue_led_timer == 0)
                BLUE_LED_PWM = LED_DIM;
        }
        if (bstate == 1)
```

Listing 5. c.d.

```
        {
            // button was released, now it is pressed - change blink mode
            if (++ blink_mode > BM_OFF) blink_mode = BM_SLOW;
            blue_led_timer = 100;
            BLUE_LED_PWM = LED_MAX;
        }
        if (-- blink_timer == 0)
        {
            blink_timer = blink_periods[blink_mode];
            on_time = blink_timer >> 1;
            GREEN_LED_PWM = LED_DIM;
        }
        else if (blink_timer == on_time) GREEN_LED_PWM = LED_MAX;
    }
}
```

Przetwornik analogowo-cyfrowy

Mikrokontrolery STM32F051 jest wyposażony w 12-bitowy przetwornik analogowo-cyfrowy, umożliwiający pomiar napięć zewnętrznych, a także wewnętrznego wzorca napięcia odniesienia oraz napięcia generowanego przez wbudowany w mikrokontroler czujnik temperatury. W celu zademonstrowania działania przetwornika użyjemy czujnika temperatury. Oprogramowanie będzie sterowało diodami LED na podstawie odczytu temperatury. Gdy temperatura będzie rosła – zaświeci się dioda zielona. Podczas spadku temperatury będzie świeciła dioda niebieska. W ten sposób użytkownik będzie miał możliwość sterowania świeceniem diod poprzez dotknięcie obudowy mikrokontrolera palcem. Skorzystamy z poprzedniego projektu, zachowując sterowanie PWM z powolnym rozjaśnianiem i ściemnianiem. Gotowy projekt nosi nazwę ADC-ts.

Algorytm

Ponieważ odczyty przetwornika analogowo-cyfrowego zazwyczaj są niestabilne, program powinien zapewniać odpowiednią ich filtrację (uśrednianie). Wyniki pomiarów będą odczytywane ze stałą częstotliwością w przerwaniu timera. Obliczane będą dwie uśrednione wartości odczy-

Listing 6. Program demonstracyjny – obsługa przetwornika A/D

```
/*
    STM32F0DISCOVERY tutorial
    TIM3 PWM LED control by ADC temperature sensor
    gbm, 12'2012
*/

#include „stm32f0xx.h"
//=========================================================================
// defs for STM32F05x chips
#define GPIO_MODER_OUT     1
#define GPIO_MODER_AF      2
#define TIM_CCMR2_OC3M_PWM1         0x0060   // OC3M[2:0] - PWM mode 1
#define TIM_CCMR2_OC4M_PWM1         0x6000   // OC4M[2:0] - PWM mode 1
#define ADC_SMPR_71_5      6
typedef __IO uint32_t * __IO32p;
//=========================================================================
// defs for STM32F0DISCOVERY board
#define LED_PORT           GPIOC
#define BLUE_LED_BIT       8
#define GREEN_LED_BIT      9
#define BUTTON_PORT                GPIOA
#define BUTTON_BIT         0
#define BLUE_LED_PWM       TIM3->CCR3
#define GREEN_LED_PWM      TIM3->CCR4
//=========================================================================
// Timings
#define SYSCLK_FREQ        HSI_VALUE
// PWM constants
#define PWM_FREQ  400      // Hz
#define PWM_STEPS 50
#define   PWM_CLK          SYSCLK_FREQ
#define   PWM_PRE          (PWM_CLK / PWM_FREQ / PWM_STEPS)
#define BLUE_LED_PWM       TIM3->CCR3
#define GREEN_LED_PWM      TIM3->CCR4
#define LED_MAX    (PWM_STEPS - 1)
#define LED_DIM    1
// temperature sensor parms
#define LONG_AVG_SHIFT     4
#define SHORT_AVG_SHIFT    8
#define T_DELTA    2
//=========================================================================
struct init_entry_ {
    volatile uint32_t *loc;
    uint32_t value;
};

static __INLINE void writeregs(const struct init_entry_ *p)
{
    for (; p->loc; p ++)
        *p->loc = p->value;
}
//=========================================================================
void SystemInit(void)
{
    FLASH->ACR = FLASH_ACR_PRFTBE; // enable prefetch
}
//=========================================================================
static const struct init_entry_ init_table[] =
{
    // port setup
    {&RCC->AHBENR, RCC_AHBENR_GPIOCEN |  RCC_AHBENR_GPIOAEN},        // GPIOC, GPIOA
    {   &LED_PORT->MODER, GPIO_MODER_AF << (GREEN_LED_BIT << 1)
        | GPIO_MODER_AF << (BLUE_LED_BIT << 1)
```

Listing 6. c.d.

```
    },   // set LED pins as AF
    {&RCC->AHBENR, RCC_AHBENR_GPIOAEN},    // GPIOA
    // ADC setup
    {&RCC->APB2ENR, RCC_APB2ENR_ADC1EN},   // ADC
    {&ADC1->CHSELR, ADC_CHSELR_CHSEL16},   // ADC
    {&ADC1->SMPR, ADC_SMPR_71_5}, // ADC
    {&ADC->CCR, ADC_CCR_TSEN | ADC_CCR_VREFEN},    // ADC
    {&ADC1->CFGR1, ADC_CFGR1_WAIT | ADC_CFGR1_CONT},        // ADC
    // ADC cal
    {&ADC1->CR, ADC_CR_ADCAL},     // ADC
    // PWM timer setup - TIM3
    {&RCC->APB1ENR, 1 << 1},       // TIM3
    {(__IO32p)&TIM3->PSC, PWM_PRE - 1},
    {(__IO32p)&TIM3->ARR, PWM_STEPS - 1},
    // blue - CH3, green - CH4
    {   (__IO32p)&TIM3->CCMR2, TIM_CCMR2_OC4M_PWM1 | TIM_CCMR2_OC4PE
        | TIM_CCMR2_OC3M_PWM1 | TIM_CCMR2_OC3PE
    },   // PWM mode 1, buffered preload
    {(__IO32p)&TIM3->CCER, TIM_CCER_CC4E | TIM_CCER_CC3E},  // enable CH3, 4 output
    {(__IO32p)&TIM3->DIER, TIM_DIER_UIE},  // enable update interrupt
    {(__IO32p)&TIM3->CR1, TIM_CR1_ARPE | TIM_CR1_CEN},      // auto reload buffer, enable
    // interrupts and sleep
    {&NVIC->ISER[0], 1 << TIM3_IRQn},      // enable interrupt
    {&SCB->SCR, SCB_SCR_SLEEPONEXIT_Msk},  // sleep while not in handler
    {0, 0}
};

//=========================================================================
int main(void)
{
    writeregs(init_table);
    __WFI();     // go to sleep
}

//=========================================================================
void TIM3_IRQHandler(void)
{
    static uint8_t tdiv = 0;
    static uint8_t blue_target = LED_DIM, green_target = LED_DIM;
    static uint32_t long_avg, short_avg;
    uint32_t val;

    TIM3->SR = ~TIM_SR_UIF;        // clear interrupt flag

    if ((++ tdiv & 3) == 0)
    {
        // 100 Hz
        if (ADC1->ISR & ADC_ISR_EOC)
        {
            val = ADC1->DR;        // 0 before first conversion

            if (short_avg == 0)
            {
                // initial measure - set
                short_avg = val << SHORT_AVG_SHIFT;
                long_avg = val << LONG_AVG_SHIFT;
            }
            else
            {
                // low-pass filters
                short_avg = short_avg + val - (short_avg >> SHORT_AVG_SHIFT);
                long_avg = long_avg + val - (long_avg >> LONG_AVG_SHIFT);
                if ((short_avg >> SHORT_AVG_SHIFT) == (long_avg >> LONG_AVG_SHIFT))
                {
                    // stable
                    blue_target = LED_DIM;
                    green_target = LED_DIM;
                }
                else if ((short_avg >> SHORT_AVG_SHIFT) > (long_avg >> LONG_AVG_SHIFT) + T_DELTA)
                {
                    // warming up
                    blue_target = LED_DIM;
                    green_target = LED_MAX;
                }
                else if ((short_avg >> SHORT_AVG_SHIFT) + T_DELTA < (long_avg >> LONG_AVG_SHIFT))
                {
                    // cooling
                    blue_target = LED_MAX;
                    green_target = LED_DIM;
                }
            }
        }
        else if (ADC1->ISR & ADC_ISR_ADRDY)
        {
            // ready for conversion
            ADC1->CR = ADC_CR_ADSTART | ADC_CR_ADEN;        // start cont. conversion
        }
        else if ((ADC1->CR & (ADC_CR_ADCAL | ADC_CR_ADEN)) == 0)
        {
            // calibrated but not enabled yet - enable
            ADC1->CR = ADC_CR_ADEN;
        }
    }

    if ((val = BLUE_LED_PWM) != blue_target)
        BLUE_LED_PWM = val < blue_target ? val + 1 : val - 1;

    if ((val = GREEN_LED_PWM) != green_target)
        GREEN_LED_PWM = val < green_target ? val + 1 : val - 1;
}
```

tów – krótko- i długoterminowa. Sterowanie jasnością diod będzie bazowało na różnicy tych średnich. Zaświecenie diody nastąpi, gdy średnia krótkoterminowa będzie różniła się od średniej długoterminowej nie mniej, niż o wartość progową zadaną poprzez stałą *T_DELTA*.

Na uwagę zasługuje zastosowany algorytm filtracji wyników pomiarów, niewymagający przechowywania wyników ostatnich pomiarów. Program przechowuje dwie sumy pomiarów – krótko- i długoterminową. Przy każdym pomiarze od każdej sumy odejmowana jest wartość średnia, obliczona przez podzielenie sumy przez stałą i dodawana jest wartość bieżącego pomiaru. Ponieważ liczba sumowanych wartości próbek jest potęgą liczby 2, dzielenie jest realizowane przez przesunięcie bitowe. Liczby uśrednianych próbek są zdefiniowane przez stałe *SHORT_AVG_SHIFT* i *LONG_AVG_SHIFT* – są to logarytmy binarne z liczb uśrednianych wyników pomiarów.

Przy pierwszym pomiarze następuje zainicjowanie obu wartości uśrednionych wynikiem pomiaru.

Zaprogramowanie przetwornika

Przetwornik analogowo-cyfrowy zaimplementowany w kładach STM32F05x jest dość złożony, dlatego warto poświęcić mu nieco więcej uwagi. Producent zaleca przeprowadzenie procedury automatycznej kalibracji przetwornika przy starcie mikrokontrolera. Zarówno kalibracja, jak i przygotowanie przetwornika do pracy zajmują pewien czas (dziesiątki mikrosekund). Aby uniknąć programowego oczekiwania na zakończenie tych czynności warto odpowiednio zaprojektować obsługę przetwornika w przerwaniu timera.

Początkowe programowanie przetwornika składa się z sześciu kolejnych czynności:

– Włączenia przetwornika – zapis do rejestru *AHB2ENR*.
– Wyboru wejścia pomiarowego – czujnika temperatury (kanał 16) w rejestrze *CHSELR*.
– Określenia okresu próbkowania – rejestr *SMPR*.
– Włączenia źródła napięcia odniesienia i czujnika temperatury – rejestr *CCR*.
– Wyboru trybu pracy przetwornika – pomiary ciągłe z automatycznym wyzwalaniem kolejnego pomiaru po odczycie wyniku przetwarzania – rejestr *CFGR1*.
– Zainicjowania procedury kalibracji – rejestr *CR*.

Po wykonaniu tych czynności następuje rozpoczęcie kalibracji.

Obsługa przetwornika w przerwaniu timera

Po zainicjowaniu kalibracji przetwornik nie jest jeszcze gotowy do pracy. Uruchomienie przetwornika nastąpi w procedurze obsługi przerwania timera. Podobnie, jak w poprzednim projekcie, procedura ta będzie wywoływana z częstotliwością 400 Hz, jednak większość czynności będzie wykonywana z częstotliwością 100 Hz. W trakcie działania programu przetwornik ADC będzie kolejno znajdował się w jednym z trzech stanów:

– kalibracji,
– przygotowania do pracy,
– pomiarów.

Po uruchomieniu pomiarów przetwornik pozostanie w stanie pomiarów. Bieżący stan przetwornika jest określany przez oprogramowanie na podstawie wartości rejestrów sterujących. W stanie pomiarów w rejestrze *ISR* jest ustawiony bit *EOC*. W stanie przygotowania do pracy bit EOC jest wyzerowany, ale ustawiony jest bit *ADRDY*. W stanie kalibracji oba wymienione bity rejestru *ISR* są wyzerowane, a zakończenie kalibracji może być rozpoznane na podstawie zerowych wartości bitów *ADCAL* i *ADEN* w rejestrze *CR*. Sekwencja instrukcji *if-then-else* zapewnia przejście od kalibracji poprzez włączenie przetwornika do stanu pomiaru.

Po wykryciu zakończenia kalibracji przetwornik zostanie włączony poprzez ustawienie bitu *ADEN* w rejestrze *CR*. Instrukcja ta zostanie wykonana tylko jeden raz – w ostatnim bloku *else if*.

Bezpośrednio po włączeniu przetwornik nie jest jeszcze gotowy do pomiarów. Ponieważ pomiary nie zostały rozpoczęte, bit *EOC* w rejestrze *ISR* będzie miał wartość 0, co spowoduje pominięcie pierwszego bloku *if* i przejście do kolejnego *else if*, w którym następuje sprawdzenie gotowości przetwornika do rozpoczęcia pomiarów i rozpoczęcie pomiarów poprzez ustawienie bitu *ADSTART* w rejestrze *CR*. Instrukcja ta wykona się tylko jeden raz – przy kolejnych wykonaniach będzie już spełniony warunek pierwszego bloku *if*. W bloku tym jest odczytywany wynik pomiaru, a następnie, w zależności od tego, czy jest to pierwszy pomiar czy kolejny, następuje zainicjowanie albo aktualizacja wartości uśrednionych i wysterowanie LED na ich podstawie. Odczyt wyniku spowoduje wyzwolenie kolejnego pomiaru, którego wynik będzie pobrany w następnym przerwaniu timera.

Sterowanie świeceniem diod

Kolejna instrukcja if w obsłudze przerwania timera służy do zaświecania i gaszenia diod na podstawie odczytów temperatury. Decyzja o stanie diod jest podejmowana na podstawie porównania dwóch średnich – krótko- i długoterminowej. Wartości obu średnich są wyznaczone poprzez przesunięcie w prawo wartości odpowiednich sum, co jest równoważne ich podzieleniu przez liczbę filtrowanych wyników pomiarów

Jeżeli obie średnie pomiarów są równe, diody są przyciemniane. Jeśli średnia krótkoterminowa jest większa o wartość progową lub więcej od średniej długoterminowej, jest rozjaśniana dioda zielona, sygnalizująca wzrost temperatury. Jeśli średnia krótkoterminowa jest mniejsza od średniej długoterminowej o wartość progową lub więcej, rozjaśniana jest dioda niebieska, sygnalizująca spadek temperatury.

Grzegorz Mazur
gbm@ii.pw.edu.pl