grafika pochodzi z http://7art-screensavers.com

Opóźnienia w STM32 (1)

Precyzyjne odmierzanie czasu w połączeniu z trybami oszczędzania energii

Częstym problemem, który występuje przy opracowywaniu programów na mikrokontrolery, jest wprowadzenie opóźnienia. W artykule przedstawiono sposób precyzyjnego odmierzania czasu opóźnienia w połączeniu z użyciem trybów oszczędzania energii w mikrokontrolerach STM32 z rdzeniem Cortex-M3. Do odmierzania czasu wykorzystano układy licznikowe. Na czas opóźnienia mikrokontroler jest usypiany w jednym z trybów oszczędzania energii. Mikrokontroler może być wybudzony przed upływem zadanego czasu, jeśli zajdzie potrzeba obsłużenia jakiegoś przerwania, a po jego obsłużeniu powraca do stanu oszczędzania energii, aby dokończyć „drzemkę”.

Jedną z najprostszych funkcji opóźniających, jaką da się napisać w języku C, wygląda następująco:

```
void
Delay(volatile unsigned count)
{
  while(count--);
}
```

W wielu zastosowaniach jest to rozwiązanie zadowalające. Jednak funkcja ta ma kilka poważnych wad. Realizowane opóźnienie jest nieprecyzyjne. Zależnie od ustawionej dla kompilatora opcji optymalizacji i opóźnień wprowadzanych przez pamięć programu (*flash latency*), jedna iteracja pętli może trwać od kilku do kilkunastu taktów zegara. Jeśli w trakcie wykonywania tej funkcji zostanie zgłoszone przerwanie, to czas opóźnienia wydłuży się o czas jego obsługi. Ponadto mikrokontroler przez cały czas wykonywania pętli pobiera energię, która jest marnowana. Chyba jedyną zaletą tej funkcji jest prostota zapisu.

Można pokusić się o implementację bardziej precyzyjnych funkcji opóźniających. Przykładowy program został

List. 1. Sygnatury funkcji opóźniających umieszczone w pliku sleep.h

```
#ifndef _SLEEP_H
#define _SLEEP_H 1

void USleep(unsigned);
void MSleep(unsigned);
void Sleep(unsigned);
void Standby(unsigned);
int SleepConfigure(unsigned);

#endif
```

napisany w języku C na mikrokontroler STM32F107 i można go uruchomić na „motylu", czyli zestawie uruchomieniowym STM32Butterfly, co zostanie zaprezentowane w drugiej części artykułu. Program bardzo łatwo daje się przenieść na inne modele z rodziny STM32, gdyż korzysta ze Standard Peripheral Library i biblioteki CMSIS (*Cortex Microcontroller Software Interface Standard*), które są dostarczane przez STMicroelectronics [1]. Do precyzyjnego odmierzania krótkich czasów, od kilku do ok. 65535 mikrosekund lub od 1 do 32767 milisekund, zostały wykorzystane dwa spośród liczników TIM2, TIM3, TIM4 i TIM5, taktowane zegarem systemowym przez odpowiednio dobrane dzielniki wstępne (*prescaler*). Do realizacji długich opóźnień, wyrażanych w sekundach, zastosowano zegar czasu rzeczywistego (*real time clock*), taktowany sygnałem z generatora z kwarcem zegarkowym 32768 Hz, tykający z częstotliwością 1 Hz. Podczas czekania na zakończenie opóźnienia mikrokontroler jest przełączany w stan oszczędzania energii. Gdy zostaje zgłoszone przerwanie, mikrokontroler wraca na czas jego obsługi do trybu aktywnego. Po zakończeniu obsługi przerwania ponownie jest wprowadzany w stan oszczędzania energii. Czas obsługi przerwania nie wydłuża opóźnienia.

Tryby oszczędzania energii

Mikrokontrolery STM32 wyposażono w kilka trybów oszczędzania energii. W trybie uśpienia (*sleep mode*) wstrzymywane jest taktowanie rdzenia i pamięci, wszystkie układy peryferyjne pozostają taktowane. W trybach głębokiego uśpienia (*deep sleep mode*) wstrzymywane jest taktowanie rdzenia, pamięci i większości peryferii. Są dwa tryby głębokiego uśpienia: tryb zatrzymania (*stop mode*) i tryb czuwania (*standby*).

W trybie zatrzymania wyłączone są wszystkie sygnały taktujące w domenie 1,8 V. Wstrzymane są wewnętrzny oscylator RC o częstotliwości nominalnej 8 MHz (HSI – *high speed internal*), zewnętrzny oscylator kwarcowy (HSE – *high speed external*) i wszystkie synchroniczne pętle fazowe (PLL – *phase locked loop*). Wewnętrzny regulator napięcia 1,8 V może pozostać w stanie normalnej pracy albo

List. 2. Implementacja funkcji opóźniających umieszczona w pliku sleep.c

```
#include „check.h"
#include „clock.h"
#include „sleep.h"
#include <misc.h>
#include <stm32f10x_exti.h>
#include <stm32f10x_pwr.h>
#include <stm32f10x_rcc.h>
#include <stm32f10x_rtc.h>
#include <stm32f10x_tim.h>

/* Licznik dla opóźnień mikrosekundowych,
   zwiększany z częstotliwością 1 MHz */
#define US_TIM                  TIM2
#define US_TIM_IRQn             TIM2_IRQn
#define US_RCC_APB1Periph_TIM RCC_APB1Periph_TIM2
#define US_TIM_IRQHandler     TIM2_IRQHandler

/* Licznik dla opóźnień milisekundowych,
   zwiększany z częstotliwością 2 kHz */
#define MS_TIM                  TIM3
#define MS_TIM_IRQn             TIM3_IRQn
#define MS_RCC_APB1Periph_TIM RCC_APB1Periph_TIM3
#define MS_TIM_IRQHandler     TIM3_IRQHandler

static unsigned usCalibration = 0;

void USleep(unsigned count) {
  if (count > usCalibration)
    count -= usCalibration;
  else if (count > 0)
    count = 1; /* minimalna możliwa wartość */
  else
    return; /* count == 0 */

  US_TIM->CNT = 0;
  US_TIM->CCR1 = count;
  US_TIM->CR1 |= TIM_CR1_CEN;
  do
    __WFE();
  while (US_TIM->CR1 & TIM_CR1_CEN);
}

void US_TIM_IRQHandler(void) {
  if (TIM_GetITStatus(US_TIM, TIM_IT_CC1)) {
    TIM_ClearITPendingBit(US_TIM, TIM_IT_CC1);
    US_TIM->CR1 &= ~TIM_CR1_CEN;
    __SEV();
  }
}

void MSleep(unsigned count) {
  if (count > 0) {
    MS_TIM->CNT = 0;
    MS_TIM->CCR1 = count << 1; /* tyka 2 kHz */
    MS_TIM->CR1 |= TIM_CR1_CEN;
    do
      __WFE();
    while (MS_TIM->CR1 & TIM_CR1_CEN);
  }
}

void MS_TIM_IRQHandler(void) {
  if (TIM_GetITStatus(MS_TIM, TIM_IT_CC1)) {
    TIM_ClearITPendingBit(MS_TIM, TIM_IT_CC1);
    MS_TIM->CR1 &= ~TIM_CR1_CEN;
    __SEV();
  }
}

/* #define PWR_REGULATOR_MODE PWR_Regulator_ON */
#define PWR_REGULATOR_MODE PWR_Regulator_LowPower

static unsigned hclkMHz;

void Sleep(unsigned count) {
  if (count > 0) {
    /* Alarm zgłaszany jest w ostatnim cyklu oscylatora RTC
       odpowiadającego sekundzie, w której alarm należy
       zgłosić, więc count <= czas uśpienia < count + 1. */
    RTC_WaitForLastTask();
    RTC_SetAlarm(RTC_GetCounter() + count);
    RTC_WaitForLastTask();

    do
      PWR_EnterSTOPMode(PWR_REGULATOR_MODE, PWR_STOPEntry_WFE);
    while (!RTC_GetFlagStatus(RTC_IT_ALR));

    RTC_WaitForLastTask();
    RTC_ClearITPendingBit(RTC_IT_ALR);

    /* Po obudzeniu trzeba przywrócić pracę oscylatorów. */
    ClockConfigure(hclkMHz);
  }
}

void Standby(unsigned time) {
  PWR_WakeUpPinCmd(ENABLE);
  RTC_WaitForLastTask();
  RTC_SetAlarm(RTC_GetCounter() + time);
  RTC_WaitForLastTask();
  PWR_EnterSTANDBYMode();
}
```

może być przełączony w stan niskiego poboru energii (*low power mode*). W trybie zatrzymania zawartość rejestrów i pamięci pozostaje niezmieniona. Wybudzenie z tego trybu może nastąpić w wyniku zgłoszenia jednego z 16 przerwań zewnętrznych (EXTI – *external interrupt*), przerwania wygenerowanego przez programowalny detektor napięcia zasilania (PVD – *programmable voltage detector*), na skutek wyzwolenia alarmu RTC lub za pomocą sygnału budzenia (*wake-up*) wygenerowanego przez interfejs USB lub Ethernet.

W trybie czuwania wewnętrzny regulator napięcia 1,8 V jest wyłączony. Oscylatory HSI, HSE i wszystkie PLL są również wyłączone. Po wejściu do tego trybu zawartość pamięci i większości rejestrów zostaje utracona. Zachowywana jest jedynie zawartość rejestrów zapasowych (*backup registers*). Wybudzenie z trybu czuwania może nastąpić za pomocą wyprowadzenia $\overline{\text{RESET}}$, zerowania wywołanego przez niezależnego nadzorcę (IWDG – *independent watchdog*), narastającym zboczem sygnału na wyprowadzeniu WKUP-PA0 lub alarm RTC. Tryb czuwania różni się od pozostałych jeszcze tym, że po wybudzeniu z niego wykonywanie programu rozpoczyna się od początku. W pozostałych trybach wykonywanie programu jest wznawiane od miejsca, w którym nastąpiło uśnięcie. Program wykorzystujący tryb czuwania musi sprawdzić przyczynę rozpoczęcia wykonania, czy jest to wynik resetu, włączenia zasilania czy wybudzenia.

We wszystkich trybach oszczędzania energii mogą pozostawać włączone:

- wewnętrzny oscylator RC o częstotliwości nominalnej 40 kHz (LSI – *low speed internal*), zwykle taktujący układ nadzorcy IWDG;
- zewnętrzny oscylator kwarcowy 32768 Hz (LSE – *low speed external*), taktujący zegar czasu rzeczywistego (RTC – *real time clock*).

Według noty katalogowej producenta [2], wybudzanie z trybu uśpienia trwa typowo 1,8 μs, z trybu zatrzymania z włączonym regulatorem napięcia – typowo 3,6 μs, z trybu zatrzymania z regulatorem napięcia w stanie niskiego poboru energii – typowo 5,4 μs, a z trybu czuwania – typowo 50 μs. Jeśli korzystamy z oscylatora kwarcowego HSE, to przy wybudzaniu z głębokiego uśpienia (tryby zatrzymania i czuwania) do powyższych czasów należy jeszcze doliczyć, wynoszący typowo 2 ms, czas startu tego oscylatora (wyżej podane czasy obejmują jedynie start wewnętrznego oscylatora RC HSI) oraz czasy synchronizacji PLL, wynoszące maksymalnie 350 μs dla każdej pętli. Pętle pracują zwykle kaskadowo, więc czasy ich synchronizacji sumują się.

List. 2. c.d.

```
int SleepConfigure(unsigned freqMHz) {
  TIM_TimeBaseInitTypeDef TIM_TimeBaseStruct;
  TIM_OCInitTypeDef       TIM_OCInitStruct;
  NVIC_InitTypeDef        NVIC_InitStruct;
  EXTI_InitTypeDef        EXTI_InitStruct;

  hclkMHz = freqMHz;

  /* Kalibracja 1 us + 120 cykle zegara */
  usCalibration = 1U + (120U / freqMHz);

  RCC_APB1PeriphClockCmd(US_RCC_APB1Periph_TIM |
                         MS_RCC_APB1Periph_TIM |
                         RCC_APB1Periph_PWR |
                         RCC_APB1Periph_BKP,
                         ENABLE);

  TIM_TimeBaseStructInit(&TIM_TimeBaseStruct);
  TIM_TimeBaseStruct.TIM_CounterMode = TIM_CounterMode_Up;

  TIM_OCStructInit(&TIM_OCInitStruct);
  TIM_OCInitStruct.TIM_OCMode = TIM_OCMode_Timing;

  NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2);
  NVIC_InitStruct.NVIC_IRQChannelPreemptionPriority = 1;
  NVIC_InitStruct.NVIC_IRQChannelSubPriority = 0;
  NVIC_InitStruct.NVIC_IRQChannelCmd = ENABLE;

  /* Konfiguruj licznik mikrosekundowy na 1 MHz. */
  TIM_TimeBaseStruct.TIM_Prescaler = freqMHz - 1;
  TIM_TimeBaseInit(US_TIM, &TIM_TimeBaseStruct);
  TIM_OC1Init(US_TIM, &TIM_OCInitStruct);
  TIM_OC1PreloadConfig(US_TIM, TIM_OCPreload_Disable);
  NVIC_InitStruct.NVIC_IRQChannel = US_TIM_IRQn;
  NVIC_Init(&NVIC_InitStruct);
  TIM_ClearITPendingBit(US_TIM, TIM_IT_CC1);
  TIM_ITConfig(US_TIM, TIM_IT_CC1, ENABLE);

  /* Konfiguruj licznik milisekundowy na 2 kHz. */
  TIM_TimeBaseStruct.TIM_Prescaler = freqMHz * 500 - 1;
  TIM_TimeBaseInit(MS_TIM, &TIM_TimeBaseStruct);
  TIM_OC1Init(MS_TIM, &TIM_OCInitStruct);
  TIM_OC1PreloadConfig(MS_TIM, TIM_OCPreload_Disable);
  NVIC_InitStruct.NVIC_IRQChannel = MS_TIM_IRQn;
  NVIC_Init(&NVIC_InitStruct);
  TIM_ClearITPendingBit(MS_TIM, TIM_IT_CC1);
  TIM_ITConfig(MS_TIM, TIM_IT_CC1, ENABLE);

  /* Konfiguruj licznik sekundowy. */
  PWR_BackupAccessCmd(ENABLE);
  if (PWR_GetFlagStatus(PWR_FLAG_SB) != RESET) {
    /* Mikrokontroler obudził się ze stanu czuwania. Nie ma
       potrzeby ponownego konfigurowania RTC - konfiguracja
       jest zachowywana po wybudzeniu ze stanu czuwania. */
    PWR_ClearFlag(PWR_FLAG_SB);
    RTC_WaitForSynchro();
  }
  else {
    /* Mikrokontroler został zresetowany, konfiguruj RTC, aby
       był taktowany kwarcem 32768 Hz i tykał z okresem 1 s. */
    RCC_LSEConfig(RCC_LSE_ON);
    active_check(RCC_GetFlagStatus(RCC_FLAG_LSERDY), 10000000);
    RCC_RTCCLKConfig(RCC_RTCCLKSource_LSE);
    RCC_RTCCLKCmd(ENABLE);
    RTC_WaitForSynchro();
    RTC_SetPrescaler(32767);
    RTC_WaitForLastTask();
  }
  /* Konfiguruj linię 17 EXTI (alarm RTC), aby narastające
     zbocze ustawiało rejestr zdarzenia */
  EXTI_StructInit(&EXTI_InitStruct);
  EXTI_InitStruct.EXTI_Line = EXTI_Line17;
  EXTI_InitStruct.EXTI_Trigger = EXTI_Trigger_Rising;
  EXTI_InitStruct.EXTI_LineCmd = ENABLE;
  EXTI_InitStruct.EXTI_Mode = EXTI_Mode_Event;
  EXTI_Init(&EXTI_InitStruct);
  RTC_WaitForLastTask();
  RTC_ClearITPendingBit(RTC_IT_ALR);

  return 0;
}
```

Do aktywowania trybu oszczędzania energii w mikrokontrolerach z rdzeniem Cortex-M3 służą instrukcje asemblerowe WFI (*wait for interrupt*) i WFE (*wait for event*). Dla ścisłości należy wspomnieć, że mikrokontroler w tryb uśpienia można też przełączyć ustawiając bit SLEEPONEXIT w rejestrze SCB_SCR (*system control block – system control register*). Bit ten można też ustawić za pomocą funkcji bibliotecznej NVIC_SystemLPConfig. Mikrokontroler zaśnie wtedy po zakończeniu obsługi najbliższego przerwania, a następnie każdorazowo, gdy zostanie zgłoszone przerwanie obudzi się, aby go obsłużyć i zaśnie po zakończeniu jego obsługi. Jednak w omawianym przykładzie nie korzystamy z funkcjonalności sleep-on-exit.

Po wywołaniu instrukcji WFI mikrokontroler natychmiast zasypia. Budzi się, gdy wystąpi wyjątek (*exception*), który ma ustawiony dostatecznie wysoki priorytet, aby wywołać procedurę obsługi. Wyjątkami są wszystkie przerwania zgłaszane

przez peryferie oraz przerwania sprzętowe i programowe generowane przez sam rdzeń. Po obudzeniu mikrokontroler wykonuje procedurę obsługi zgłoszonego wyjątku (przerwania), a następnie kontynuuje wykonywanie kodu znajdującego się bezpośrednio za instrukcją WFI, która spowodowała uśpienie. Ten mechanizm można spróbować wykorzystać do realizacji opóźnień. Algorytm mógłby wyglądać następująco:

```
zeruj TIMER;
konfiguruj TIMER, aby po osiągnięciu zadanej wartości zgłosił przerwanie i zatrzymał się;
wystartuj TIMER;
do
  WFI;
while (TIMER nie osiągnął zadanej wartości);
```

Najpierw zerujemy licznik TIMER. Konfigurujemy go, aby zgłosił przerwanie po osiągnięciu zadanej wartości opóźnienia. Startujemy licznik, a następnie zasypiamy. Po wybudzeniu sprawdzamy czy przyczyną obudzenia było przerwanie zgłoszone przez TIMER. Jeśli źródłem przerwania nie był TIMER, ponownie zasypiamy. Ten algorytm wydaje się prawidłowy, jednak zawiera bardzo subtelny błąd. Jeśli najpierw zostanie zgłoszony wyjątek (przerwanie) niepochodzący od licznika TIMER, warunek pętli będzie prawdziwy i zostanie ponownie wywołana instrukcja WFI. Między przeczytaniem odpowiedniego rejestru statusu licznika a ponownym zaśnięciem mikrokontroler wykona kilka instrukcji maszynowych: sprawdzenie warunku, skok warunkowy. Jeśli w tym czasie TIMER zgłosi przerwanie, to zostanie ono obsłużone przed wywołaniem instrukcji WFI i jej wywołanie uśpi mikrokontroler, aż pojawi się kolejny wyjątek. Opóźnienie wydłuży się nieprzewidywalnie.

Prawidłowym rozwiązaniem jest użycie instrukcji WFE. Instrukcja ta współpracuje z jednobitowym rejestrem zdarzenia (*one-bit event register*). Rejestr ten jest testowany po wywołaniu tej instrukcji. Jeśli rejestr zdarzenia jest wyzerowany, mikrokontroler zasypia. Jeśli rejestr zdarzenia jest ustawiony, mikrokontroler zeruje go i kontynuuje wykonywanie programu – nie zasypia. Jeśli mikrokontroler zasnął na instrukcji WFE i następnie został ustawiony rejestr zdarzenia, mikrokontroler budzi się, a rejestr zdarzenia zostaje wyzerowany. Rejestru zdarzenia nie można odczytywać programowo. Rejestr ten jest zerowany tylko przez instrukcję WFE i może być ustawiany sprzętowo. Z pewnymi ograniczeniami, o których będzie napisane dalej, źródłami, które ustawiają ten rejestr mogą być te same źródła, które wyzwalają wyjątki (przerwania). Co będzie dla nas bardzo ważne, rejestr zdarzenia może być też ustawiony programowo instrukcją SEV (*send event*). Poprawny algorytm jest następujący:

```
zeruj TIMER;
konfiguruj TIMER, aby po osiągnięciu zadanej wartości ustawił rejestr zdarzenia i zatrzymał się;
wystartuj TIMER;
do
  WFE;
while (TIMER nie osiągnął zadanej wartości);
```

Z poziomu języka C instrukcje WFE, WFI i SEV można wywoływać za pomocą, dostępnych w bibliotece CMSIS, makr:

```
void __WFE(void);
void __WFI(void);
void __SEV(void);
```

Żeby wprowadzić mikrokontroler w tryb głębokiego uśpienia, trzeba przed wywołaniem instrukcji WFE lub WFI ustawić bit SLEEPDEEP w rejestrze SCB_SCR. Dodatkowo dla trybu zatrzymania (*stop mode*) należy wyzerować bit PDDS (*power down deep sleep*) w rejestrze PWR_CR (*power control register*) i skonfigurować w tym rejestrze bit LPDS (*low power deep sleep*) zależnie od wybranego sposobu pracy regulatora 1,8 V. Wyzerowanie tego bitu oznacza pozostawienie regulatora włączonego, a jego ustawienie – wprowadzenie regulatora w stan niskiego poboru energii. Prostszym sposobem jest wywołanie funkcji bibliotecznej:

```
void
PWR_EnterSTOPMode(
  uint32_t PWR_Regulator,
  uint8_t  PWR_STOPEntry
);
```

Parametr PWR_Regulator specyfikuje sposób pracy regulatora i może przyjmować wartość PWR_Regulator_ON lub PWR_Regulator_Low. Natomiast parametr PWR_STOPEntry specyfikuje instrukcję wywoływaną w celu zaśnięcia i może przyjmować wartość PWR_STOPEntry_WFI lub PWR_STOPEntry_WFE.

Aby wprowadzić mikrokontroler w tryb czuwania (*standby*), należy przed wywołaniem instrukcji WFE lub WFI, oprócz ustawienia bitu SLEEPDEEP, wyzerować bit WUF (*wake-up flag*) w rejestrze PWR_CR przez ustawienie bitu CWUF (*clear wake-up flag*) oraz ustawić bit PDDS w tym rejestrze. Jednak prostszym sposobem jest wywołanie funkcji bibliotecznej:

```
void
PWR_EnterSTANDBYMode(void);
```

Ta funkcja wywołuje instrukcję WFI. W przypadku przechodzenia w tryb czuwania nie ma różnicy między działaniem instrukcji WFE a WFI.

Funkcje opóźniające

Plik nagłówkowy definiujący sygnatury funkcji opóźniających umieszczony jest na **listingu 1**. Funkcja USleep dopuszcza wartości parametru wejściowego z zakresu od 0 do 65535 i realizuje opóźnienie o zadaną wartość mikrosekund. Funkcja MSleep dopuszcza wartości parametru wejściowego z zakresu od 0 do 32767 i realizuje opóźnienie o zadaną wartość milisekund. Funkcja Sleep dopuszcza wartości parametru wejściowego z zakresu od 0 do 2^{32}–1 i realizuje opóźnienie o zadaną wartość sekund. Funkcja Standby wprowadza mikrokontroler w tryb czuwania na zadaną liczbę sekund i dopuszcza wartości parametru wejściowego z zakresu od 0 do 2^{32}–1. Funkcja SleepConfigure konfiguruje moduł realizujący opóźnienia i powinna być wywołana zanim zostanie wywołana któraś z funkcji opóźniających. Jej argumentem jest częstotliwość taktowania układów licznikowych TIMx w MHz. W omawianym przykładzie konfigurujemy ją tak, aby była równa częstotliwości taktowania rdzenia, co zostanie omówione w drugiej części artykułu.

Implementacja powyższych funkcji została pokazana na **listingu 2**. Najpierw definiujemy, że licznik TIM2 jest używany do opóźnień mikrosekundowych, a licznik TIM3 – do opóźnień milisekundowych. Dzięki temu, gdyby któryś z tych liczników był zajęty przez inną funkcję, łatwo można go zamienić na licznik TIM4 lub TIM5. Zawartość TIM2 jest zwiększana z częstotliwością 1 MHz, aby łatwo było odliczać okresy 1 µs, natomiast TIM3 jest zwiększana z częstotliwością 2 kHz. Taki wybór spowodowany jest tym, że przyjęliśmy założenie o taktowaniu liczników tym samym zegarem co rdzenia, a przy maksymalnej częstotliwości taktowania rdzenia 72 MHz i największej możliwej wartości dzielnika wstępnego 65536 nie uda się uzyskać sygnału 1 kHz.

Funkcja USleep korzysta z trybu uśpienia, a jak zostało napisane wyżej, wybudzanie z tego trybu może trwać ok. 2 µs. Ten narzut jest niezależny od częstotliwości taktowania rdzenia. Dodatkowy narzut wprowadza konieczność konfigurowania licznika. Możemy starać się go zminimalizować pomijając wywołania funkcji biblio-

List. 3. Pomocnicze makro umieszczone w pliku check.h

```
#ifndef _CHECK_H
#define _CHECK_H 1

#define active_check(cond, limit) { \
  int i;                            \
  for (i = (limit); !(cond); --i)   \
    if (i <= 0)                     \
      return -1;                    \
}

#endif
```

tecznych i odwołując się bezpośrednio do rejestrów konfiguracyjnych licznika. Ten narzut można oszacować w taktach zegara i zależy on od częstotliwości taktowania rdzenia. Jeśli chcemy odmierzać opóźnienia bardzo precyzyjnie, możemy spróbować eksperymentalnie dobrać wartość zmiennej usCalibration i ustawić ją na początku funkcji SleepConfigure w zależności od częstotliwości taktowania rdzenia. Wymienione narzuty powodują, że w praktyce za pomocą funkcji USleep nie uda się osiągnąć opóźnienia krótszego niż ok. 3 µs.

Funkcja USleep realizuje kolejno:

- kalibruje wartość opóźnienia;
- zeruje licznik – rejestr CNT (*counter*);
- konfiguruje wartość count, po której osiągnięciu przez licznik ma nastąpić ustawienie rejestru zdarzenia – rejestr CCR1 (*capture-compare register 1*);
- uaktywnia licznik – ustawia bit TIM_CR1_CEN (*counter enable*) w rejestrze CR1 (*control register 1*);
- usypia mikrokontroler za pomocą instrukcji WFE;
- po obudzeniu sprawdza czy licznik zatrzymał się, tzn. czy wyzerowany jest bit TIM_CR1_CEN – jeśli nie jest, ponownie usypia.

Funkcja USleep współpracuje z procedurą US_TIM_IRQHandler obsługującą przerwanie licznika mikrosekundowego. Zgodnie z opisem, dla instrukcji WFE nie ma potrzeby implementowania obsługi przerwania. Wystarczyłoby, gdyby rejestr zdarzenia (*event register*) został ustawiony po zajściu zdarzenia zgodności zawartości licznika z wartością count. Niestety, jeśli zdarzenie zgodności zaszłoby między sprawdzeniem warunku zakończenia pętli do while a wywołaniem instrukcji WFE, rejestr zdarzenia nie zostałby ustawiony – mielibyśmy dokładnie taki sam problem, jak omówiony wyżej dla instrukcji WFI. Problem ten jest znany i opisany w [3] jako błąd „563915: Event Register is not set by interrupts and debug" oraz w [4] w podrozdziale „Cortex-M3 event register is not set by interrupts and debug". Dlatego w procedurze obsługi przerwania wołamy makro __SEV, aby mieć pewność, że rejestr zdarzenia, na który czeka instrukcja WFE, zostanie ustawiony. Wcześniej zatrzymujemy licznik przez wyzerowanie bitu TIM_CR1_CEN.

Funkcja MSleep jest bardzo podobna do funkcji USleep. Różnice są następujące. Nie kalibrujemy czasu uśpienia, gdyż przy opóźnieniach milisekundowych narzut związany z usypianiem i budzeniem jest pomijalny. Sprawdzamy jedynie czy funkcja nie została wywołana z zerową wartością czasu opóźnienia. Ponadto licznik milisekundowy tyka z częstotliwością 2 kHz, więc trzeba wartość count pomnożyć przez dwa. Implementujemy też odpowiednią procedurę obsługi przerwania MS_TIM_IRQHandler dla tego licznika.

Funkcja Sleep jest koncepcyjnie podobna do dwóch wyżej omówionych funkcji opóźniających. Różnice wynikają głównie z innych interfejsów programistycznych dla liczników TIMx i RTC. Zapis do rejestrów konfiguracyjnych RTC trwa co najmniej trzy takty zegarowe LSE (32768 Hz) i jest wykonywany niejako w tle wykonywania programu. Zatem przed każdym zapisem do rejestru konfiguracyjnego RTC musimy upewnić się czy zakończył się poprzedni zapis. Służy do tego funkcja RTC_WaitForLastTask. Do odmierzania czasu wykorzystujemy funkcje alarmu RTC.

Funkcja Sleep realizuje kolejno:

- odczytuje bieżącą zawartość licznika RTC (wywołanie RTC_GetCounter) i ustawia wartość, przy której ma nastąpić obudzenie (wywołanie RTC_SetAlarm);
- zasypia w pętli, wywołując funkcję PWR_EnterSTOPMode, która z kolei wywołuje instrukcję WFE stała PWR_REGULATOR_MODE definiuje sposób pracy regulatora 1,8 V;
- opuszcza pętlę, jeśli obudzenie jest wynikiem zgłoszenia alarmu RTC, czyli gdy ustawiony jest znacznik zdarzenia alarmu RTC_IT_ALR (wywołanie RTC_GetFlagStatus);
- po opuszczeniu pętli kasuje znacznik zdarzenia alarmu (wywołanie RTC_ClearITPendingBit).

Po obudzeniu z trybu zatrzymania mikrokontroler jest taktowany sygnałem z wewnętrznego oscylatora RC (HSI, 8 MHz). Podczas wykonywania funkcji Sleep mikrokontroler obsługuje przerwania będąc taktowany sygnałem zegarowym HSI. Przed zakończeniem tej funkcji trzeba przywrócić pracę oscylatora kwarcowego HSE oraz pętli PLL za pomocą funkcji ClockConfigure, która będzie omówiona w drugiej części artykułu. W przypadku funkcji Sleep nie musimy implementować procedury obł-ugi przerwania, gdyż alarm RTC poprawnie ustawia rejestr zdarzenia.

Funkcja Standby odczytuje bieżącą wartość licznika RTC za pomocą funkcji RTC_GetCounter. Ustawia wartość licznika RTC, przy której nastąpi wybudzenie – funkcja RTC_SetAlarm. Mikrokontroler zostaje przełączony w tryb czuwania – funkcja PWR_EnterSTANDBYMode. Mikrokontroler może zostać wybudzony z trybu czuwania przed upływem zadanego czasu opóźnienia, przez narastające zbocze sygnału na wyprowadzeniu WKUP-PA0. Funkcja PWR_WakeUpPinCmd uaktywnia tę opcję.

Ponieważ RTC nie jest resetowany w trybach głębokiego uśpienia (zatrzymania i czuwania), sposób jego konfigurowania jest uzależniony od ustawienia znacznika PWR_FLAG_SB testowanego za pomocą funkcji PWR_GetFlagStatus. Jeśli znacznik PWR_FLAG_SB jest ustawiony, to nastąpiło wybudzenie – RTC pracuje i nie trzeba go rekonfigurować. Jeśli znacznik ten jest wyzerowany, to mikrokontroler został zresetowany lub zostało włączone zasilanie i trzeba skonfigurować RTC. Aby funkcja SleepConfigure nie zawiesiła się w przypadku uszkodzenia oscylatora zegarkowego, korzystamy z pomocniczego makra, przedstawionego na **listingu 3**. Makro to ogranicza czas oczekiwania na start oscylatora. Jeśli oscylator nie zadziała, funkcja SleepConfigure zwróci wartość ujemną sygnalizującą błąd. Przy poprawnym zakończeniu funkcja zwraca zero.

Marcin Peczarski

Literatura

[1] ARM-based 32-bit MCU STM32F10xxx standard peripheral library, wersja 3.2.0, www.st.com/stonline/products/support/micro/files/stm32f10x_stdperiph_lib.zip

[2] STM32F105xx, STM32F107xx, Connectivity line, ARM-based 32-bit MCU with 64/256 KB Flash, USB OTG, Ethernet, 10 timers, 2 CANs, 2 ADCs, 14 communication interfaces, http://www.st.com/stonline/products/literature/ds/15274.pdf

[3] ARM Core Cortex-M3/Cortex-M3 with ETM (AT420/AT425) Errata Notice, http://infocenter.arm.com/help/topic/com.arm.doc.eat0420c/Cortex-M3-Errata-r2p0-v2.pdf

[4] STM32F105xx and STM32F107xx Errata sheet, STM32F105xx and STM32F107xx revision Z connectivity line device limitations, www.st.com/stonline/products/literature/es/15866.pdf