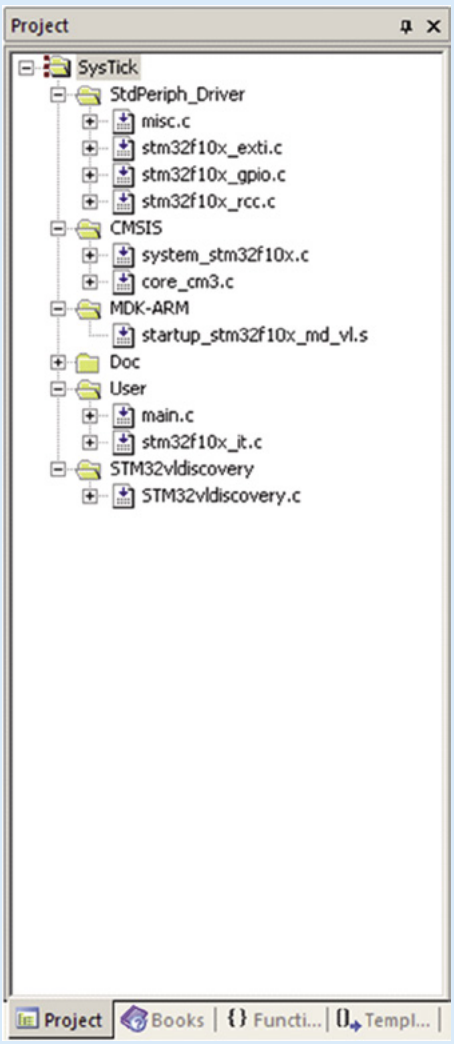
STM32 dla użytkowników 8-bitowców (2)

32-bitowe mikrokontrolery są postrzegane przez konstruktorów jako elementy do bardziej wymagających zadań. Czasy się jednak zmieniają, co widać już nawet w sklepach „za rogiem”.

Przykładowe procedury STM32VLDISCOVERY firmware package (AN3268) można pobrać ze strony producenta www.st.com/stm32. Po rozpakowaniu trzeba otworzyć gotowy projekt SysTick dla uVison4 z katalogu *an3268/stm32vldiscovery/Project/examples/SysTick/MDK_ARM*. Strukturę plików i katalogów projektu pokazano na **rysunku 1**. Jest ona dość mocno rozbudowana jak na funkcje, która ma wykonywać. Przypomnijmy: konfigurujemy zegar systemowy, linie portów, licznik SysTick i obsługujemy przerwanie od jego przepełnienia. Można sobie zadać pytanie: po co tyle plików, katalogów skoro dla mikrokontrolera 8-bitowego podobny projekt można napisać w kilkunastu - kilkudziesięciu linijkach w jednym pliku? Jest to naturalna konsekwencja wyboru gotowych bibliotek i pewnego stylu programowania. Po nabyciu wprawy takie podejście jest o wiele bardziej efektywne niż pisanie za każdym razem wszystkiego od nowa. Trzeba też pamiętać, że mikrokontrolery STM32 są jednak bardziej rozbudowane niż 8-bitowe jednostki i przez to wymagają inicjalizacji większej liczby peryferiów. Decydując się na ewentualna zamianę trzeba to uwzględnić.



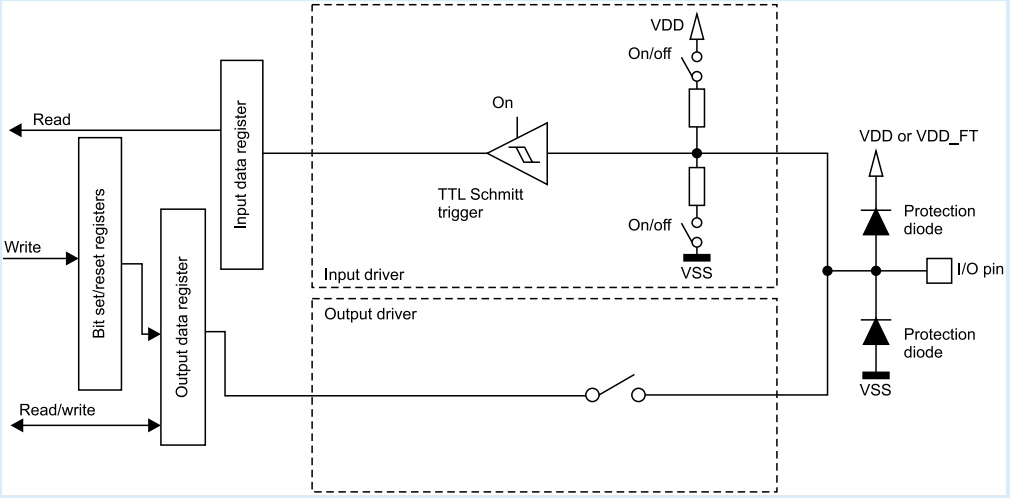
Rysunek 1. Struktura katalogu projekt SysTick

Jak już wiemy, jedną z początkowych czynności jest skonfigurowanie modułu taktowania. Głównym plikiem projektu jest main.c i tam rozpoczniemy szukanie konfiguracji zegara. Jednak na początku funkcji void main(void) umieszczona jest informacja, że konfigurację zegara umieszczono w procedurze SystemInit() wywoływanej przez procedury startowe startup. Procedury startowe są napisane w asemblerze i mogą być automatycznie dodawane do projektu przez IDE uVision4 w trakcie jego tworzenia.

SystemInit() umieszczono w pliku system_stm32f10x.c. Po jej odnalezieniu okazuje się, że wywołuje kolejna funkcję SetSysClock(), którą pokazano na **listingu 1**. Jej działanie jest powiązane z definicjami pokazanymi na **listingu 2**. Uwaga: ustalenie źródła i częstotliwości sygnału zegarowego taktującego rdzeń jest bardzo istotne dla poprawnego działania programu. Przy pierwszym kontakcie z mikrokontrolerem sposób ustalenia jak taktować i z jaka częstotliwością uznałem za jedna z najważniejszych czynności. Tutaj mamy możliwość wyboru 2 predefiniowanych częstotliwości : częstotliwości zegara HSE (generatora kwarcowego), lub częstotliwości 24 MHz. W tym drugim wypadku jest to częstotliwość 8 MHz pomnożona przez 3 w układzie PLL. W pliku system_stm32f10x.c. korzystamy z definicji:

```
#if defined (STM32F10X_LD_VL) || (defined
STM32F10X_MD_VL)
 //#define SYSCLK_FREQ_HSE HSE_Value
 #define SYSCLK_FREQ_24MHz 24000000
#else
```

i powinniśmy taktować rdzeń z częstotliwością 24 MHz, czyli z maksymalną, na którą pozwala hardware w rodzinie STM32F100. Dla porządku pokażę na **listingu 3** jak wygląda firmowa procedura ustawiania zegara systemowego dla 24 MHz. Wystarczy spojrzeć na listing aby stwierdzić, że wybór bibliotek firmowych zamiast samodzielnej konfiguracji zegara był właściwy. Zaoszczędziło to wiele pracy i w sumie stało się tak łatwe, jak konfigurowanie taktowania w popularnych mikrokontrolerach 8–bitowych. Musimy pamiętać, że po opanowaniu podstawowych czynności związanych z konfigurowaniem zegara warto zapoznać się z mechanizmami zabezpieczeń przed zakłóceniami sygnału zegara Clock Security System. Możemy zobaczyć na list. 3, że już w trakcie konfigurowania zegara wykonywane jest sprawdzanie poprawności generowania sygnału przez HSE. Clock Security System potrafi zdiagnozować problemy z generatorem zewnętrznym, wysłać sygnał przerwania informujący o problemie i przełączyć się na wewnętrzny generator HSI o częstotliwości 8 MHz. Podobne rozwiązania są stosowane w mikrokontrolerach 8 bitowych na przykład w rodzinie PIC18F i są bardzo cenione w aplikacjach obsługujących krytyczne algorytmy sterowania,



Rysunek 2. Schemat linii portu skonfigurowanej jako wejście

gdzie niestabilność taktowania wiąże się z poważnymi problemami.

Po zapewnieniu poprawnego taktowania popatrzmy jak wyglądają operacje na liniach portów. Moduł portów jest standardowo umieszczany we wszystkich mikrokontrolerach. Obsługa linii wejściowo – wyjściowych jest zwykle bardzo łatwa. Najczęściej do sterownia potrzebne są 2 rejestry. Jeden to rejestr portów. Zapisanie go powoduje wysłanie stanu na linię. Kiedy chcemy odczytać stan linii , to odczytujemy ten rejestr. Drugi rejestr określa kierunek przesyłania danych, czyli czy linia jest wyjściowa, czy wejściowa. Jeżeli mikrokontroler ma wbudowany przetwornik DAC to konieczny staje się trzeci rejestr określający czy wejście ma być cyfrowe, czy analogowe.

W mikrokontrolerze 8051 do obsługi portów zdefiniowany jest tylko jeden rejestr, bo nie ma potrzeby ustalania kierunku przepływu danych. Kiedy chcemy odczytać stan linii to najpierw wystawiamy na niej poziom wysoki i to już wystarczy. Bywa też, że do linii zdefiniowanych jako wyjściowe jest jeden rejestr, a po zmianie kierunku na wejściową inny rejestr. Moduł portów jest taktowany z taką samą częstotliwością jak rdzeń. Określa to maksymalną częstotliwość sygnałów generowanych i odczytywanych z linii. Linie portów są tez współdzielone z innymi modułami peryferyjnymi mikrokontrolera. Najczęściej po włączeniu zasilania wyprowadzenia układu są przypisywane do linii portów, a układy peryferyjne są wyłączone. Jednak nie zawsze tak jest. W mikrokontrolerach PIC część wyprowadzeń domyślnie po włączeniu zasilania jest wejściami analogowymi przetwornika A/C i jeżeli nie chcemy ich używać w tej roli to trzeba je przekonfigurować. Pomimo prostoty sterowania przy używaniu linii portów trzeba pamiętać o pewnych niuansach właściwych konstrukcji danej rodziny mikrokontrolerów. Na przykład w mikrokontrolerach PIC przy wystawianiu szybkich zmiennych przebiegów na porty wymaga się by stan był stabilny co najmniej przez 2-3 cykle maszynowe. Trzeba też zachować ostrożność przy mieszanych kierunkach wejścia i wyjścia na jednym porcie i wykonywaniu operacji czytaj- modyfikuj-zapisz. Doświadczony konstruktor powinien znać te ograniczenia i umieć sobie z nimi radzić.

My najpierw postaramy się wykonać prostą operację wystawienia stanu na linię portów po to, aby na przykład zaświecić i zgasić diodę LED. Jeżeli to się nam uda, to będziemy mieć pewność, że mikrokontroler działa i jest prawidłowo taktowany.

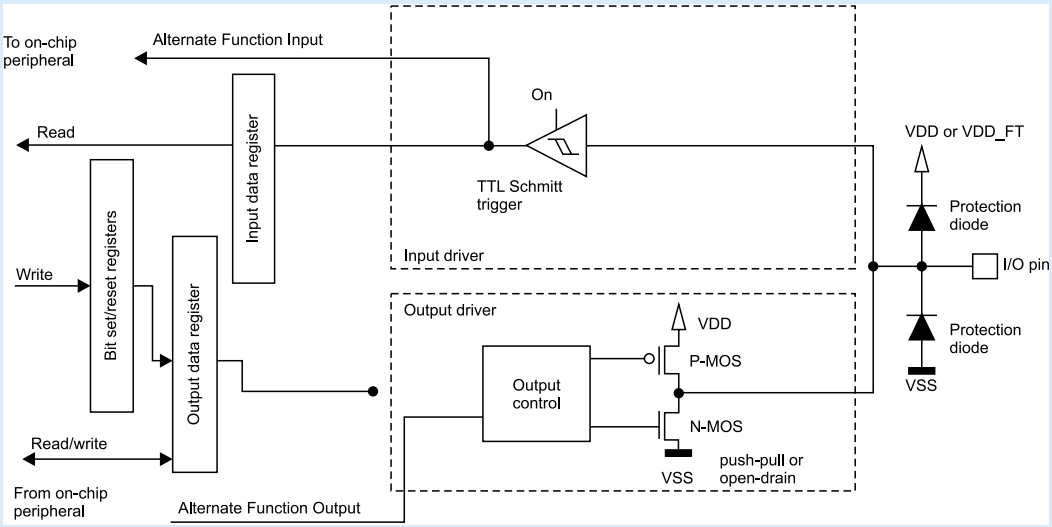
W pierwszej kolejności trzeba zobaczyć jak są zbudowane porty w STM32F100 i co trzeba zrobić by je skonfigurować. Na **rysunku 2** pokazano schematycznie struktura linii portu zaprogramowanej jako wejście. Wejście można zaprogramować jako:

- Wejścia wysokiej impedancji (input floating).
- Wejścia podciągnięte przez wewnętrzny rezystor do plusa zasilania (input pull-up).
- Wejścia podciągnięte przez wewnętrzny rezystor do masy (input pull-down).
- Wejście analogowe przetwornika analogowo cyfrowego.
- Na **rysunku 3** pokazano schematycznie struktura linii portu zaprogramowanej jako wyjście.
- Wyjście typu otwarty dren (output open-drain).
- Klasyczne wyjście dwustanowe push-pull.
- Konfiguracja odbywa się standardowo za pomocą rejestrów opisanych w dokumencie *Reference manual RM001*. My tak jak poprzednio spróbujemy skorzystać z funkcji bibliotecznych. Inicjalizacja portów jest umieszczona w pliku STM32vldiscovery.c. W module STM32 Value Line Discovery umieszczone są 2 diody LED: LD3 (zielona) dołączona do linii portów PC9i LD4 (niebieska) połączona do linii PC8. Definicje tych linii są umieszczone w pliku STM32vldiscovery.h (**listing 4**).

Przed właściwym skonfigurowaniem modułu portów trzeba mu zapewnić sygnał zegarowy. W mikrokontrolerach STM32 każde urządzenie peryferyjne wymaga jawnego włączenia sygnału zegarowego. Bloki peryferyjne są połączone do 2 magistral APB1 i APB2. Magistra APB1 może pracować z maksymalną częstotliwością 36MHz, a APB2 72MHz. Ponieważ nasz rdzeń pracuje z częstotliwością maksymalną 24 MHz, to magistrale APB1 i APB2 również mogą pracować z częstotliwością 24 MHz. Układy peryferyjne są dołączone do magistral na stałe i programista musi wiedzieć, że wszystkie porty są połączone z APB2. Stąd definicje

```
 #define LED3_GPIO_CLK    RCC_APB2Periph_
GPIOC
 #define LED4_GPIO_CLK    RCC_APB2Periph_
GPIOC
```

na list. 4. Konfigurację taktowania wykonuje funkcja RCC_APB2PeriphClockCmd .



Rysunek 3. Schemat linii portu skonfigurowanej jako wyjście

Listing 1. Funkcja konfiguracji zegara

```
static void SetSysClock(void)
{
#ifdef SYSCLK_FREQ_HSE
  SetSysClockToHSE();
#elif defined SYSCLK_FREQ_24MHz
  SetSysClockTo24();
#elif defined SYSCLK_FREQ_36MHz
  SetSysClockTo36();
#elif defined SYSCLK_FREQ_48MHz
  SetSysClockTo48();
#elif defined SYSCLK_FREQ_56MHz
  SetSysClockTo56();
#elif defined SYSCLK_FREQ_72MHz
  SetSysClockTo72();
#endif

/* If none of the define above is enabled, the HSI is used as System
clock source (default after reset) */
}
```

Listing 2. Definicje zegara

```
* Clock Definitions
#ifdef SYSCLK_FREQ_HSE
  uint32_t SystemCoreClock   = SYSCLK_FREQ_HSE;  /*!< System Clock
Frequency (Core Clock) */
#elif defined SYSCLK_FREQ_24MHz
  uint32_t SystemCoreClock   = SYSCLK_FREQ_24MHz;  /*!< System Clock
Frequency (Core Clock) */
#elif defined SYSCLK_FREQ_36MHz
 uint32_t SystemCoreClock   = SYSCLK_FREQ_36MHz;  /*!< System Clock
Frequency (Core Clock) */
#elif defined SYSCLK_FREQ_48MHz
  uint32_t SystemCoreClock   = SYSCLK_FREQ_48MHz;  /*!< System Clock
Frequency (Core Clock) */
#elif defined SYSCLK_FREQ_56MHz
  uint32_t SystemCoreClock   = SYSCLK_FREQ_56MHz;  /*!< System Clock
Frequency (Core Clock) */
#elif defined SYSCLK_FREQ_72MHz
  uint32_t SystemCoreClock   = SYSCLK_FREQ_72MHz;  /*!< System Clock
Frequency (Core Clock) */
#else /*!< HSI Selected as System Clock source */
  uint32_t SystemCoreClock   = HSI_Value;  /*!< System Clock
Frequency (Core Clock) */
#endif
```

Właściwa konfiguracja, czyli określenie kierunku przepływu danych wejście - wyjście, oraz rodzaj wejścia (cyfrowe float, z podciąganiem , analogowe) i wyjścia (push-pull, open drain) odbywa się przez zapisanie struktury konfigurującej GPIO_InitStructure. Dla początkujących może wyglądać to groźnie, ale jest to prosta i przejrzysta operacja. Po zapisaniu struktury trzeba wywołać funkcję GPIO_Init z parametrami określającymi definiowane linie i operatorem adresu struktury. Na listingu 5 jest pokazana procedura inicjalizacji 2 naszych linii. Linie są skonfigurowane jako wyjścia push –pull . Parametr GPIO_Speed_50MHz określa maksymalną częstotliwość zmian stanów linii portów. W tym przypadku jest to 50 MHz. Oczywiście przy taktowaniu rdzenia z częstotliwością 24 MHz nie będzie możliwa zmiana z częstotliwością 50 MHz, ale oznacza to, że porty mogą pracować z najwyższą możliwą prędkością. Opcjonalnie można tą prędkość ograniczyć do 2 MHz, lub 10 MHz. W każdym z tych przypadków przy taktowaniu 24 MHz maksymalna prędkość może być ograniczana.

Podobnie inicjalizuje się linie wejściowe cyfrowe i analogowe. Operacje na liniach portów można wykonywać używając funkcji bibliotecznych GPIO_SetBits i GPIOResetBits. Ich argumentami są nazwa portu i numer linii w porcie. Obie funkcje są wprowadzone do bibliotek dla porządku, ale nie są zbyt skomplikowane, bo składają się w zasadzie z jednej linijki i można je pominąć. W naszym analizowanym projekcie zdefiniowano w pliku STM32vldiscovery.c dwie własne funkcje void STM32vldiscovery_LEDOn i void STM32vldiscovery_LEDOff (**listing 6**).

Inicjalizowanie i używanie linii portów jest trochę bardziej skomplikowane niż w mikrokontrolerach 8-bitowych. Wynika to z większej liczby rejestrów konfiguracyjnych i bardziej rozbudowanego modułu portów. Jednak z zamian dostajemy możliwość włączenia taktowania tylko tych portów (i innych układów peryferyjnych), które są używane. Daje to możliwość ograniczania poboru energii. Jednak w tym momencie nie jest to istotne. Ważne, że mamy zainicjowany port i możemy wykonywać operacje zmiany stanu linii wyjściowych. To oczywiście tylko początek nawet jeżeli chodzi o moduł portów. Bardziej dociekliwi mogą poznać konfiguracje wejść, funkcje blokady konfiguracji, czy remapowanie funkcji alternatywnych.

Najprostszy program zadający diodę LED może wyglądać tak:

```
STM32vldiscovery_LEDOn(LED3);
while (1);
```

Przykładowy program nazwany przez producenta SysTick pozwala na skonfigurowanie zegara odliczającego kalibrowane opóźnienia. Tutaj funkcja doliczania opóźnienia pozwoli na napisanie procedury migania diodą LED.

SysTick , czyli Cortex System Timer jest 24- bitowym licznikiem zliczającym impulsy o częstotliwości taktowania rdzenia (HCLK). Licznik można tez tak zaprogramować by ta częstotliwość była podzielona przez 8.

Listing 3. Ustawianie częstotliwości taktowania 24 MHz

```
static void SetSysClockTo24(void)
{
  __IO uint32_t StartUpCounter = 0, HSEStatus = 0;
/* SYSCLK, HCLK, PCLK2 and PCLK1 configuration */
/* Enable HSE */
  RCC->CR |= ((uint32_t)RCC_CR_HSEON);

/* Wait till HSE is ready and if Time out is reached exit */
  do
  {
    HSEStatus = RCC->CR & RCC_CR_HSERDY;
    StartUpCounter++;
  } while((HSEStatus == 0) && (StartUpCounter != HSEStartUp_TimeOut));

  if ((RCC->CR & RCC_CR_HSERDY) != RESET)
  {
    HSEStatus = (uint32_t)0x01;
  }
  else
  {
    HSEStatus = (uint32_t)0x00;
  }

  if (HSEStatus == (uint32_t)0x01)
  {
#if !defined STM32F10X_LD_VL && !defined STM32F10X_MD_VL
 /* Enable Prefetch Buffer */
    FLASH->ACR |= FLASH_ACR_PRFTBE;
 /* Flash 0 wait state */
    FLASH->ACR &= (uint32_t)((uint32_t)~FLASH_ACR_LATENCY);
    FLASH->ACR |= (uint32_t)FLASH_ACR_LATENCY_0;
#endif
 /* HCLK = SYSCLK */
    RCC->CFGR |= (uint32_t)RCC_CFGR_HPRE_DIV1;
 /* PCLK2 = HCLK */
    RCC->CFGR |= (uint32_t)RCC_CFGR_PPRE2_DIV1;
 /* PCLK1 = HCLK */
    RCC->CFGR |= (uint32_t)RCC_CFGR_PPRE1_DIV1;
#ifdef STM32F10X_CL
 /* Configure PLLs */
 /* PLL configuration: PLLCLK = PREDIV1 * 6 = 24 MHz */
    RCC->CFGR &= (uint32_t)~(RCC_CFGR_PLLXTPRE | RCC_CFGR_PLLSRC | RCC_CFGR_PLLMULL);
    RCC->CFGR |= (uint32_t)(RCC_CFGR_PLLXTPRE_PREDIV1 | RCC_CFGR_PLLSRC_PREDIV1 |
    RCC_CFGR_PLLMULL6);
 /* PLL2 configuration: PLL2CLK = (HSE / 5) * 8 = 40 MHz */
 /* PREDIV1 configuration: PREDIV1CLK = PLL2 / 10 = 4 MHz */
 RCC->CFGR2 &= (uint32_t)~(RCC_CFGR2_PREDIV2 | RCC_CFGR2_PLL2MUL |
    RCC_CFGR2_PREDIV1 | RCC_CFGR2_PREDIV1SRC);
 RCC->CFGR2 |= (uint32_t)(RCC_CFGR2_PREDIV2_DIV5 | RCC_CFGR2_PLL2MUL8 |
    RCC_CFGR2_PREDIV1SRC_PLL2 | RCC_CFGR2_PREDIV1_DIV10);
 /* Enable PLL2 */
    RCC->CR |= RCC_CR_PLL2ON;
 /* Wait till PLL2 is ready */
    while((RCC->CR & RCC_CR_PLL2RDY) == 0)
    {
    }
#elif defined (STM32F10X_LD_VL) || defined (STM32F10X_MD_VL)
 /* PLL configuration: = (HSE / 2) * 6 = 24 MHz */
    RCC->CFGR &= (uint32_t)((uint32_t)~(RCC_CFGR_PLLSRC | RCC_CFGR_PLLXTPRE | RCC_CFGR_PLLMULL));
    RCC->CFGR |= (uint32_t)(RCC_CFGR_PLLSRC_PREDIV1 | RCC_CFGR_PLLXTPRE_PREDIV1_Div2 | RCC_CFGR_PLLMULL6);
#else
 /* PLL configuration: = (HSE / 2) * 6 = 24 MHz */
    RCC->CFGR &= (uint32_t)((uint32_t)~(RCC_CFGR_PLLSRC | RCC_CFGR_PLLXTPRE | RCC_CFGR_PLLMULL));
    RCC->CFGR |= (uint32_t)(RCC_CFGR_PLLSRC_HSE | RCC_CFGR_PLLXTPRE_HSE_Div2 | RCC_CFGR_PLLMULL6);
#endif /* STM32F10X_CL */
 /* Enable PLL */
    RCC->CR |= RCC_CR_PLLON;
 /* Wait till PLL is ready */
    while((RCC->CR & RCC_CR_PLLRDY) == 0)
    {
    }
 /* Select PLL as system clock source */
    RCC->CFGR &= (uint32_t)((uint32_t)~(RCC_CFGR_SW));
    RCC->CFGR |= (uint32_t)RCC_CFGR_SW_PLL;
 /* Wait till PLL is used as system clock source */
    while ((RCC->CFGR & (uint32_t)RCC_CFGR_SWS) != (uint32_t)0x08)
    {
    }
  }
  else
  { /* If HSE fails to start-up, the application will have wrong clock
  configuration. User can add here some code to deal with this error */
  }
}
```

Zliczanie odbywa się w dół (dekrementacja). Po wyzerowaniu licznik jest automatycznie zapisywany wartością zapisaną na 24 bitach w rejestrze LOAD (STK_LOAD) i zliczanie odbywa się od początku. W pliku core_cm3.h jest umieszczona funkcja konfigurująca licznika uint32_t SysTick_Config(uint32_t ticks) pokazana na **listingu 7**. Funkcja oprócz sprawdzenia czy wartość wpisywana do licznika jest prawidłowa (nie większa niż pojemność licznika), zapisuje rejestr LOAD i konfiguruje przerwanie zgłaszane w momencie wyzerowania.

Załóżmy, że rdzeń jest taktowany z częstotliwością 27MHz, a SysTick ma zgłaszać przerwanie co 1 ms. W zmiennej System CoreClock jest zapisana wartość 27000000. Jeżeli tą wartość podzielimy przez 1000, to otrzymamy 27000. Zapisanie wartości 27000 do rejestru LOAD spowoduje, że licznik będzie zgłaszał przerwanie co 1 ms. Wywołanie funkcji konfiguracji licznika SysTick pokazano na **listingu 8**.

Odblokowanie przerwania wymaga umieszczenia w programie jego obsługi. Procedury obsługi przerwań są

Listing 4. Definicje linii sterujących diody LED

```
#define LED3_PIN         GPIO_Pin_9
#define LED3_GPIO_PORT   GPIOC
#define LED3_GPIO_CLK    RCC_APB2Periph_GPIOC
#define LED4_PIN         GPIO_Pin_8
#define LED4_GPIO_PORT   GPIOC
#define LED4_GPIO_CLK    RCC_APB2Periph_GPIOC
```

Listing 5. Inicjalizacja linii portów

```
GPIO_TypeDef* GPIO_PORT[LEDn] = {LED3_GPIO_PORT, LED4_GPIO_PORT};
const uint16_t GPIO_PIN[LEDn] = {LED3_PIN, LED4_PIN};
const uint32_t GPIO_CLK[LEDn] = {LED3_GPIO_CLK, LED4_GPIO_CLK};
void STM32vldiscovery_LEDInit(Led_TypeDef Led)
{
  GPIO_InitTypeDef GPIO_InitStructure;
 /* Enable the GPIO_LED Clock */
  RCC_APB2PeriphClockCmd(GPIO_CLK[Led], ENABLE);
 /* Configure the GPIO_LED pin */
  GPIO_InitStructure.GPIO_Pin = GPIO_PIN[Led];
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
  GPIO_Init(GPIO_PORT[Led], &GPIO_InitStructure);
}
```

Listing 6. Funkcje sterowania liniami połączonymi z diodami LED

```
void STM32vldiscovery_LEDOn(Led_TypeDef Led)
{
  GPIO_PORT[Led]->BSRR = GPIO_PIN[Led];
}

void STM32vldiscovery_LEDOff(Led_TypeDef Led)
{
  GPIO_PORT[Led]->BRR = GPIO_PIN[Led];
}
```

Listing 7. Funkcja konfiguracji licznika SysTick

```
/**
 * @brief Initialize and start the SysTick counter and
its interrupt.
 * @param ticks number of ticks between two interrupts
 * @return 1 = failed, 0 = successful
 * Initialise the system tick timer and its interrupt
and start the
 * system tick timer / counter in free running mode to
generate
 * periodical interrupts.
 */
static __INLINE uint32_t SysTick_Config(uint32_t ticks)
{
/* Reload value impossible */
  if (ticks > SysTick_LOAD_RELOAD_Msk) return (1);
/* set reload register */
  SysTick->LOAD = (ticks & SysTick_LOAD_RELOAD_Msk) -
1;
/* set Priority for Cortex-M0 System Interrupts */
  NVIC_SetPriority (SysTick_IRQn, (1<<__NVIC_PRIO_BITS)
- 1);
/* Load the SysTick Counter Value */
  SysTick->VAL = 0;
  SysTick->CTRL = SysTick_CTRL_CLKSOURCE_Msk |
  SysTick_CTRL_TICKINT_Msk |
/* Enable SysTick IRQ and SysTick Timer */
  SysTick_CTRL_ENABLE_Msk;
/* Function successful */
  return (0);
}
```

Listing 8. Konfigurowanie licznika SysTick

```
/* Setup SysTick Timer for 1 msec interrupts */
  if (SysTick_Config(SystemCoreClock / 1000))
  {
 /* Capture error */
    while (1);
    }
    {
```

umieszczone w pliku stm32f10x_it.c. Dal naszego timera jest to funkcja SysTick_Handler(). Każde przerwanie powoduje wywołanie prostej funkcji TimingDelay_Decrement(), która dekrementuje 32-bitową zmienną TimingDelay do momentu jej wyzerowania.

Nasz stosunkowo prosty program jest bardzo dobrym miejscem startowym do pisania bardziej zaawansowanych aplikacji. Mamy zdefiniowany moduł portów i mechanizm odliczania czasu z rozdzielczością 1 ms.

Podsumowanie

Producenci mikrokontrolerów 32-bitowych oczekują, że z powodu niskiej ceny i dobrego wyposażenia mogą one konkurować z jednostkami 8 bitowymi. Nie jestem analitykiem rynku, nie śledzę tendencji w sprzedaży mikrokontrolerów i nie spekuluję czy się tak stanie. Z technicznego punktu widzenia stosowanie 32-bitowców zamiast 8-bitowców nie jest problemem. Jednak chyba trudno się spodziewać, że 8-bitowce zostaną całkowicie wyparte przez silniejsze jednostki. Być może w bardziej wymagających aplikacjach tak już się dzieje. Na pewno w konstrukcjach, które potencjalnie mogą wymagać w przyszłości rozszerzenia funkcji bezpieczniej jest stosować szybsze i lepiej wyposażone jednostki. Z tych powodów warto przeanalizować czy nie lepiej zastosować tani STM32 zamiast „wypasionej" ATmegi lub PIC18.

Tomasz Jabłoński, EP