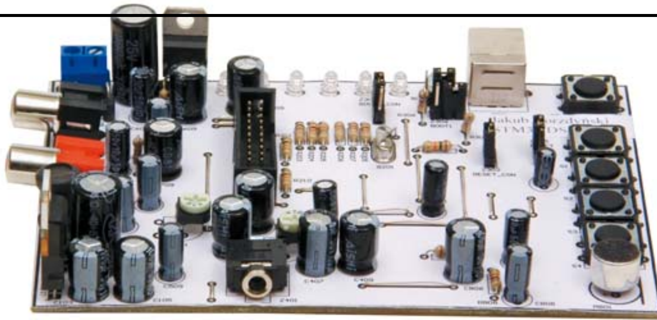


STM32 DSP KIT część 2

Drugi projekt – przyciski i diody LED

Drugi projekt ma za zadania pokazać, w jaki sposób obsłużyć przyciski oraz zmienić częstotliwość taktowania mikrokontrolera na 72MHz. Mikroswitchami będziemy chcieli wybierać jedną z dwóch animacji LED i zmieniać jej szybkość.

Na początek należy utworzyć nowy projekt w sposób opisany poprzednio. Tym razem potrzebna będzie także biblioteka do pracy z pamięcią flash, dlatego należy dołączyć stosowny plik nagłówkowy (*stm32f10x_flash.h*), usuwając komentarz w pliku *stm32f10x_conf.h* oraz dodać do projektu plik *stm32f10x_flash.c*. Większość plików można przekopiować z poprzedniego przykładu, z tym że teraz utworzony został dla porządku dodatkowy katalog *lib*, w którym umieszczone zostaną wszystkie pliki pochodzące z biblioteki STM32. Jedyną różnicą będzie zmiana pierwszej linii w pliku *main.c* na:

#include „lib/stm32f10x.h”.

Przygotowanie dwóch różnych efektów generowanych za pomocą diod LED nie powinno sprawić problemów. Wystarczy jedynie skonfigurować porty, do których dołączone są pozostałe diody LED. Warto zauważyć, że najpierw wypełniana jest struktura dla portu B, a później dla portu A (lub odwrotnie, kolejność nie ma znaczenia). Można wykorzystać tę samą strukturę i nie tworzyć niepotrzebnie jeszcze jednej. W tym przypadku najpierw wypełnia się ją dla portu B, inicjuje się port B, potem wypełnia dla portu A i inicjuje port A. Poszczególne wyprowadzenia można konfigurować „hurtem”, korzystając z operatora sumy logicznej „|”. Podobnie hurtem można dołączyć sygnał zegarowy do obu portów jedną instrukcją.

W jaki sposób zainicjować porty do pracy jako wejścia? W zasadzie tak samo jak do pracy w roli wyjścia, trzeba jedynie użyć innego parametru dla pola *GPIO_Mode*, wybierając jedną z opcji:

* *GPIO_Mode_IN_FLOATING* – wejście bez podciągania,

* *GPIO_Mode_IPD* – wejście z podciąganiem do masy,

* *GPIO_Mode_IPU* – wejście z podciąganiem do plusa zasilania.

W tym przypadku trzeba wybrać ostatnią opcję, gdyż płytka nie ma własnych rezystorów podciągających (tryb *GPIO_Mode_IN_FLOATING* odpada), a po naciśnięciu jest podawana masa, więc po puszczeniu musi być plus zasilania. Zostaje więc *GPIO_Mode_IPU*. Fragment kodu, w postaci funkcji, inicjujący porty diod i przycisków pokazano na **listingu 2**. Po wywołaniu tej funkcji porty sterujące diodami zostaną ustawione jako wyjścia, natomiast porty, do których podłączono przyciski – jako wejścia z podciąganiem do plusa zasilania.

Przygotowanie dwóch funkcji animujących diody LED w różny sposób nie powinno sprawić większych problemów. Można to zadanie zrealizować na bardzo wiele sposobów, na **listingu 3** przedstawiono dwie przykładowe realizacje. Każda z nich pobiera argument, na podstawie którego włączana jest odpowiednia kombinacja diod LED. Warto zauważyć, że również diody LED mogą być włączane „hurtem”, podobnie jak miało to miejsce wcześniej, czyli z wykorzystaniem operatora sumy logicznej.

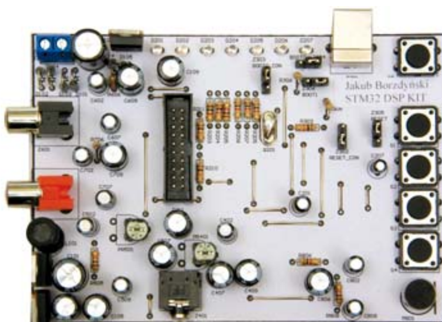
Sprawdzenie stanu portu jest proste i wymaga wywołania funkcji *GPIO_ReadInputDataBit*, która zwraca stan logiczny obecny na wyprowadzeniu. Argumenty tej funkcji są w zasadzie takie same jak funkcji ustawiającej czy kasującej bity: najpierw port, potem stała określająca wyprowadzenie. Chcąc sprawdzić stan portu, do którego podłączono S1, należy użyć instrukcji:

!GPIO_ReadInputDataBit(GPIOB, GPIO_Pin_0)

Zwracam uwagę na obecność wykrzyknika (negacja zwracanej przez funkcję wartości). Na porcie normalnie występuje jedynka logiczna, więc po puszczeniu przycisku zawsze będzie zwracana wartość true. Oznacza to, że każdy z warunków *if* byłby spełniany przez cały czas, gdy przyciski są puszczone. Powinno być dokładnie odwrotnie, stąd negacja.

Przykładowy program obrazujący wykorzystanie przycisków przedstawiono na **listingu 4**. Naciskając S1 oraz S2, można zmieniać szybkość animacji, natomiast przyciskami S3 oraz S4 wybierać animację.

Ostatnim problemem, jaki zostanie poruszony w ramach tego przykładu, będzie taktowanie mikrokontrolera. Część Czytelników być może nie jest świadoma faktu, że do tej pory mikrokontroler był taktowany przez wewnętrzny obwód RC, a nie dołączony rezonator kwarcowy. W przypadku układów STM32 sytuacja z taktowaniem jest odmienna od tej znanej np. z rodziny AVR. Po każdym

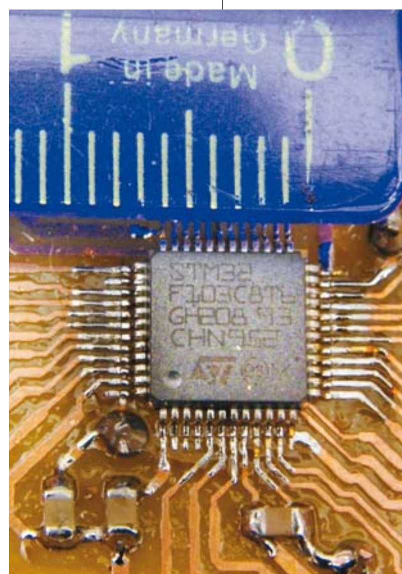


Listing 2

```
void gpioInit(){
  //struktura inicjujaca
  GPIO_InitTypeDef GPIO_InitStructure;
  //dolacz sygnal zegarowy do modulu portu A oraz B
  RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA | RCC_APB2Periph_GPIOB, ENABLE);
  //konfiguracja diod LED - port B
  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_5 | GPIO_Pin_6 | GPIO_Pin_7 | GPIO_Pin_8 | GPIO_Pin_9;
  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_2MHz;
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
  GPIO_Init(GPIOB, &GPIO_InitStructure);
  //konfiguracja diod  LED - PORT A
  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_11 | GPIO_Pin_12 ;
  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_2MHz;
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
  GPIO_Init(GPIOA, &GPIO_InitStructure);
  //konfiguracja przyciskow
  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0 | GPIO_Pin_1 | GPIO_Pin_10 | GPIO_Pin_11;
  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPU ;
  GPIO_Init(GPIOB, &GPIO_InitStructure);
}
```

```
//efekt LED - "plywajaca dioda"
void ledFun1(__IO uint32_t par){
  //wylacz wszystkie diody
  GPIO_ResetBits(GPIOB, GPIO_Pin_5 | GPIO_Pin_6 | GPIO_Pin_7
                      | GPIO_Pin_8 | GPIO_Pin_9 );
  GPIO_ResetBits(GPIOA, GPIO_Pin_11 | GPIO_Pin_12);
  //pokaz wybrana "klatke animacji"
  switch(par){
    case 0: GPIO_SetBits(GPIOA, GPIO_Pin_11 ); break ;
    case 1: GPIO_SetBits(GPIOA, GPIO_Pin_12 ); break ;
    case 2: GPIO_SetBits(GPIOB, GPIO_Pin_5 ); break ;
    case 3: GPIO_SetBits(GPIOB, GPIO_Pin_6 ); break ;
    case 4: GPIO_SetBits(GPIOB, GPIO_Pin_7 ); break ;
    case 5: GPIO_SetBits(GPIOB, GPIO_Pin_8 ); break ;
    case 6: GPIO_SetBits(GPIOB, GPIO_Pin_9 ); break ;
    case 7: GPIO_SetBits(GPIOB, GPIO_Pin_8 ); break ;
    case 8: GPIO_SetBits(GPIOB, GPIO_Pin_7 ); break ;
    case 9: GPIO_SetBits(GPIOB, GPIO_Pin_6 ); break ;
    case 10: GPIO_SetBits(GPIOB, GPIO_Pin_5 ); break ;
    case 11: GPIO_SetBits(GPIOA, GPIO_Pin_12 ); break ;
  }
}
//efekt LED - "fala"
void ledFun2(__IO uint32_t par){
  //wylacz wszystkie diody
  GPIO_ResetBits(GPIOB, GPIO_Pin_5 | GPIO_Pin_6 | GPIO_Pin_7
                      | GPIO_Pin_8 | GPIO_Pin_9 );
  GPIO_ResetBits(GPIOA, GPIO_Pin_11 | GPIO_Pin_12);
  //pokaz wybrana "klatke animacji"
  switch(par){
    case 0:
      GPIO_SetBits(GPIOB, GPIO_Pin_6 );
      break ;
    case 1:
      GPIO_SetBits(GPIOB, GPIO_Pin_6 | GPIO_Pin_5 | GPIO_Pin_7);
      break ;
    case 2:
      GPIO_SetBits(GPIOB, GPIO_Pin_6 | GPIO_Pin_5 | GPIO_Pin_7 | GPIO_Pin_8);
      GPIO_SetBits(GPIOA, GPIO_Pin_12 );
      break ;
    case 3:
     GPIO_SetBits(GPIOB, GPIO_Pin_6 | GPIO_Pin_5 | GPIO_Pin_7 | GPIO_Pin_8 | GPIO_Pin_9);
      GPIO_SetBits(GPIOA, GPIO_Pin_12 | GPIO_Pin_11 );
      break ;
    case 4:
      GPIO_SetBits(GPIOB, GPIO_Pin_5 | GPIO_Pin_7 | GPIO_Pin_8 | GPIO_Pin_9);
      GPIO_SetBits(GPIOA, GPIO_Pin_12 | GPIO_Pin_11 );
      break ;
    case 5:
      GPIO_SetBits(GPIOB, GPIO_Pin_8 | GPIO_Pin_9);
      GPIO_SetBits(GPIOA, GPIO_Pin_12 | GPIO_Pin_11 );
      break ;
    case 6:
      GPIO_SetBits(GPIOB, GPIO_Pin_9);
      GPIO_SetBits(GPIOA, GPIO_Pin_11 );
      break ;
  }
}
```

Listing 3



torem w sposób programowy. Podobnie, tzn. za pomocą odpowiednich instrukcji, następuje uaktywnienie pętli PLL, której zadaniem jest zwielokrotnienie częstotliwości zegara. Oznacza to, że PLL jest w stanie zamienić dołączone 8MHz na 72MHz (albo i więcej, niemniej należy się wtedy liczyć z niestabilną pracą mikrokontrolera).

Gotowa procedura wymuszająca taktowanie rdzenia sygnałem o maksymalnej częstotliwości pokazana została na **listingu 5**. Funkcja ta została opracowana na podstawie przykładowego projektu dołączonego do biblioteki STM32. Należy wywołać jedynie tę funkcję wewnątrz *main*, aby procesor pracował znacznie szybciej. Po skompilowaniu i załadowaniu kodu można się przekonać, że animacja jest znacznie szybsza. Poszczególne funkcje zostały krótko skomentowane na listingu. Można do tego dodać, że pamięć flash wymaga opóźnienia, gdyż taktowanie jej zegarem 72MHz pochodzącym bezpo-

```
#ifndef stm32dspLib_h //1
#define stm32dspLib_h //2
//dolacz biblioteke
#include "lib/stm32f10x.h"
//inicjuj porty I/O
void gpioInit() ;
//konfiguracja ukladu zegarowego
void configRCC() ;
//konfiguracja SPI dla DAC (SPI2)
void initDACs() ;
//zapis do lewego kanalu
void writeDAC_L(__IO uint16_t val);
//zapis do prawego kanalu
void writeDAC_R(__IO uint16_t val);
#endif //3
```

Listing 6

Listing 4

```
int main(){
  //zmienne
  __IO uint32_t i ;
  __IO uint32_t number = 0 ;
  __IO uint32_t speed = 0x5FFF ;
  __IO uint32_t mode = 0 ;
  //inicjacja
  gpioInit() ;
  configRCC() ;
  //petla glowna
  while(1){
    //realizacja opoznienia
    if(i++>=speed){
     //wywolanie funkcji zaleznie od stanu zmiennej mode
      if(mode==0){
        ledFun1(number);
      }else{
        ledFun2(number);
      }
      //odliczanie od poczatku
      i = 0 ;;
      //liczenie kolejnych klatek animacji
      if(++number==12){number=0;}
    }
    //obsluga klawiatury
    if(!GPIO_ReadInputDataBit(GPIOB, GPIO_Pin_0)){
      speed = 0x1FFFF ;
    }
    if(!GPIO_ReadInputDataBit(GPIOB, GPIO_Pin_1)){
      speed = 0x5FFF ;
    }
    if(!GPIO_ReadInputDataBit(GPIOB, GPIO_Pin_10)){
      mode = 0 ;
    }
    if(!GPIO_ReadInputDataBit(GPIOB, GPIO_Pin_11)){
      mode = 1 ;
    }
  }
}
```

resecie układ będzie pracował z wewnętrznym oscylatorem i należy przełączyć go na pracę z zewnętrznym rezona-

Listing 5

```
//konfiguracja ukladu zegarowego
void configRCC(){
  //informacja o powodzeniu/niepowodzeniu
  ErrorStatus status ;
  //przywrocenie stanu poczatkowego
  RCC_DeInit() ;
  //przelaczenia na taktowania z zewnetrznego rezonatora
  RCC_HSEConfig(RCC_HSE_ON);
  //oczekiwania na rozpoczecie pracy przez rezonator
  status = RCC_WaitForHSEStartUp() ;
  //czy rezonator pracuje poprawnie ?
  if(status == SUCCESS ) {
    //aktywacja bufora pamieci Flah
    FLASH_PrefetchBufferCmd(FLASH_PrefetchBuffer_Enable) ;
    //opoznienie dostepu do pamieci (2 cykle)
    FLASH_SetLatency(FLASH_Latency_2) ;
    //zegar rdzenia
    RCC_HCLKConfig(RCC_SYSCLK_Div1) ;
    //taktowanie magistrali APB2
    RCC_PCLK2Config(RCC_HCLK_Div1) ;
    //taktowanie magistrali APB1
    RCC_PCLK1Config(RCC_HCLK_Div2);
    //mnoznik czestotliwosci -> 8MHz * 9 = 72MHz
    RCC_PLLConfig(RCC_PLLSource_HSE_Div1, RCC_PLLMul_9);
    //aktywacja petli PLL
    RCC_PLLCmd(ENABLE);
    //oczekiwanie na gotowosc PLL
    while(RCC_GetFlagStatus(RCC_FLAG_PLLRDY) == RESET){;}
   //wybranie sygnalu pochodzacego z PLL jako sygnalu systemowego
    RCC_SYSCLKConfig(RCC_SYSCLKSource_PLLCLK);
    //ocekiwanie na przelaczenie taktowania na PLL
    while(RCC_GetSYSCLKSource() != 0x08) {;}
  }
}
```

średnio z pętli PLL uniemożliwi jej pracę i tym samym program nie będzie się wykonywał. Magistrale APB1 oraz APB2 są taktowane, odpowiednio, sygnałem 36MHz oraz 72MHz (36MHz powstaje z podziału 72MHz przez 2, o czym świadczy parametr RCC_HCLK_Div2). Magistrala APB1 może pracować z maksymalną częstotliwością 36MHz, stąd konieczność podziału sygnału 72MHz przez dwa. W zasadzie pokazany na **listingu 5** kod można traktować jako stałą procedurę i nie wnikać więcej w jej działanie. Po prostu, aby wymusić działanie mikrokontrolera z maksymalną częstotliwością, należy wykonać podane polecenia.

Pełny kod źródłowy dla tego przykładu można znaleźć w Elportalu.

Obsługa przetwornika DAC

W tym przykładzie zostanie przedstawiona metoda obsługi przetwornika MCP4921. Pierwszą rzeczą będzie ustawienie portów I/O do obsługi przycisków i diod LED oraz „podkręcenie" rdzenia do 72MHz. Czynność tę będziemy wykonywać jeszcze parokrotnie, więc można uprościć sobie życie, tworząc własną bibliotekę, w której znajdą się procedury inicjacji poszczególnych modułów. Umieszczone zostaną tam także funkcje obsługujące peryferia.

Na chwilę obecną biblioteka musi mieć następujące funkcje:

- configRCC() – wymuszającą taktowanie rdzenia zegarem 72MHz,
- gpioInit() – inicjującą porty I/O,
- initDACs() – inicjującą interfejs przetworników,
- writeDAC_L() oraz writeDAC_R – zapisującą do przetworników DAC kanału lewego oraz prawego.

Pierwsze dwie funkcje wystarczy przekopiować z poprzedniego przykładu, a utworzenie dwóch pozostałych będzie przedmiotem niniejszego paragrafu. Warto rozpocząć od stworzenia pliku nagłówkowego, powiedzmy że będzie to *stm32dspLib.h*, w którym zawarte będą deklaracje wyżej wymienionych funkcji. Zawartość tego pliku przedstawiono na **listingu 6** i umieszczono w katalogu *LIB*, aby później można było „za jednym zamachem" skopiować wszystkie potrzebne pliki do kolejnych projektów. Komentarza mogą wymagać dyrektywy preprocesora. Pierwsza z nich, oznaczona cyfrą jeden, odpowiada za sprawdzenie, czy zdefiniowano stałą *stm32dspLib_h*. Jeżeli taka stała nie została zdefiniowana, następuje wykonanie kodu pomiędzy dyrektywą pierwszą i trzecią, tzn. wykonuje się kod pomiędzy instrukcjami:

```
#ifndef stm32dspLib_h //1
//...
#endif //3
```

Druga dyrektywa deklaruje stałą *stm32dspLib_h*. Przy następnym dodaniu tej biblioteki do tego samego projektu warunek pierwszy nie będzie już spełniony i kod pomiędzy dyrektywami pierwszą i trzecią nie zostanie wykonany. Oznacza to w praktyce, że nieważne, ile razy biblioteka zostanie dołączona, jej właściwy kod zostanie przetworzony tylko RAZ. W tym przypadku jest to może trochę na wyrost, ale warto pamiętać o wstawianiu tych trzech instrukcji do każdego pliku. W bardziej złożonych programach może to być pomocne lub wręcz niezbędne. Wykonywanie dwa razy tego samego kodu nie jest dobrym rozwiązaniem. Czasami może prowadzić do błędów kompilacji, jeżeli deklarowane są jakieś zmienne. Stała w dyrektywie pierwszej i drugiej musi być taka sama, ale może mieć dowolną nazwę, jednak wygodniej jest, gdy jest ona powiązana z nazwą pliku – istnieje mniejsze prawdopodobieństwo, że zostanie użyta ponownie i uniemożliwi kompilację kodu.

```
//konfiguracja SPI dla DAC (SPI2)
void initDACs(){
  //struktura inicjujaca
  SPI_InitTypeDef spiInitStructure  ;
  GPIO_InitTypeDef GPIO_InitStructure;
  //dolaczenie sygnalu taktujacego
  RCC_APB1PeriphClockCmd(RCC_APB1Periph_SPI2 , ENABLE);
  RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA | RCC_APB2Periph_GPIOB, ENABLE);
  //ustawianie parametrów struktury inicjujacej
  spiInitStructure.SPI_BaudRatePrescaler = SPI_BaudRatePrescaler_2;
  spiInitStructure.SPI_CPHA = SPI_CPHA_1Edge ;
  spiInitStructure.SPI_CPOL = SPI_CPOL_Low ;
  spiInitStructure.SPI_DataSize  = SPI_DataSize_16b  ;
  spiInitStructure.SPI_Direction = SPI_Direction_1Line_Tx  ;
  spiInitStructure.SPI_FirstBit  = SPI_FirstBit_MSB ;
  spiInitStructure.SPI_Mode = SPI_Mode_Master ;
  spiInitStructure.SPI_NSS  = SPI_NSS_Soft ;
  //zapisanie struktury
  SPI_Init(SPI2, &spiInitStructure) ;
  SPI_Cmd (SPI2, ENABLE) ;
  //konfiguracja - PORT A
  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_8;
  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
  GPIO_Init(GPIOA, &GPIO_InitStructure);
  //konfiguracja - port B
  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_13 | GPIO_Pin_15;
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
  GPIO_Init(GPIOB, &GPIO_InitStructure);
  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_12 ;
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
  GPIO_Init(GPIOB, &GPIO_InitStructure);
}
```

Listing 7

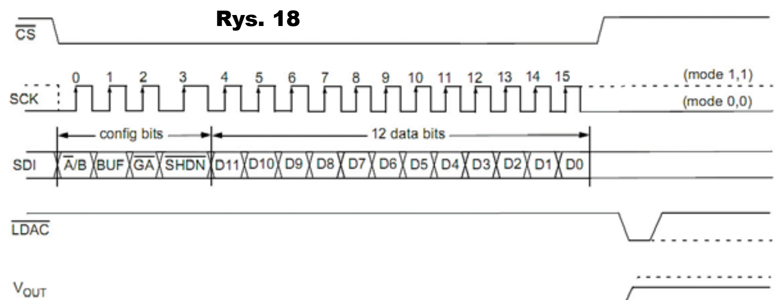
Oczywiście należy utworzyć plik *stm32dspLib.c*, w którym zostaną umieszczone właściwe definicje funkcji, czyli konkretny kod, który ma zostać skompilowany.

Po stworzeniu nowego projektu i własnej, prostej biblioteki nadeszła pora na napisanie procedur obsługi DAC. Jak wspomniano przed chwilą, potrzebne będą dwie funkcje: konfigurująca oraz zapisująca dane. Najwygodniej będzie najpierw zająć się konfiguracją. Po pierwsze, należy odpowiedzieć sobie na pytanie, jaki interfejs komunikacyjny obsługuje DAC. Patrząc na przebiegi czasowe na **rysunku 18** (pochodzące z noty katalogowej MCP4921) nietrudno zauważyć, że wykorzystany w tym miejscu będzie port SPI. Transmisja jest jednokierunkowa (zapis do układu scalonego), gdyż MCP921 nie ma portu wyjściowego. Najważniejsze wnioski, jakie można wysunąć na podstawie rysunku 18, to:

- bity danych są zatrzaskiwane zboczem narastającym sygnału zegarowego,
- dane przesyłane są w postaci paczek o rozmiarze 16 bitów,
- w czasie transmisji sygnał CS musi mieć poziom niski,
- transmitowany jest najpierw najstarszy bit,
- SPI może pracować w trybie (1,1) lub (0,0), tu wybrany zostanie ten ostatni tryb, który oznacza, że w czasie nieaktywności stan linii SCK będzie niski.

Konfiguracja SPI w zasadzie sprowadza się do zadeklarowania i wypełnienia struktury, którą biblioteka STM32 przewidziała do tego celu. Tą strukturą jest *SPI_InitTypeDef*, w ramach której znajdują się następujące parametry:

- *SPI_BaudRatePrescaler* – preskaler określający, przez ile podzielić częstotliwość magistrali; częstotliwość APB1 do której dołączony jest SPI2, wynosi 36MHz, więc podział przez 2 zapewni taktowanie na poziomie 18MHz (MCP4921 akceptuje do 20MHz),
- *SPI_CPHA* – określa aktywne zbocze, tzn. czy bity mają być zatrzaskiwane zboczem opadającym, czy narastającym (w tym przypadku ma to być zbocze narastające),
- *SPI_CPOL* – określa poziom na wyjściu SCK, kiedy nie jest prowadzona transmisja, zgodnie z wcześniejszymi założeniami ma to być zero logiczne,



Rys. 18

- *SPI_CRCPolynomial* – gdyby transmisja przez SPI była zabezpieczona sumą kontrolną CRC, to można w tym miejscu podać wykorzystywany wielomian, transmisja do DAC nie jest zabezpieczona, więc ten parametr zostanie pominięty i będzie miał wartość domyślną,
- *SPI_DataSize* – rozmiar ramki, jak wspomniano wcześniej, będzie to 16 bitów (do wyboru jest jeszcze ramka 8-bitowa),
- *SPI_Direction* – za pomocą tego parametru można określić czy transmisja ma być dwukierunkowa, czy jednokierunkowa i w którą stronę, w naszym przypadku będziemy chcieli jedynie zapisywać informacje (wysyłać),
- *SPI_FirstBit* – określa, czy najpierw ma być przesłany najstarszy czy najmłodszy bit, rysunek 16 nie pozostawia wątpliwości – najpierw przesyłany jest bit MSB,
- *SPI_Mode* – określa, czy port SPI ma pracować w trybie master czy slave, w tym przypadku będzie to master, gdyż to mikrokontroler inicjuje transfery i jest układem nadrzędnym,
- *SPI_NSS* – pozwala określić, czy końcówka NSS ma być sterowana sprzętowo, czy programowo, niestety jeden interfejs SPI obsługuje dwa przetworniki, więc pozostaje obsługa programowa.

Oczywiście konfiguracja SPI jest możliwa dopiero po odkomentowaniu biblioteki *stm32f10x_spi.h* w pliku *stm32f10x_conf.h*. Należy również dodać *stm32f10x_spi.c* do projektu (warto uprzednio skopiować pliki *stm32f10x_spi.c* oraz *stm32f10x_spi.h* folderu *lib*).

Ostatnią rzeczą, którą należy ustawić, są porty I/O. Porty I/O współdzielone z magistralą SPI muszą być ustawione jako *GPIO_Mode_AF_PP* (AF – funkcja alternatywna, PP – włączone podciąganie). Porty sterujące liniami CS będą kontrolowane programowo, więc ustawione zostały jako zwyczajne porty wyjściowe. Ostateczny kod funkcji konfigurującej SPI został przedstawiony na **listingu 7**.

Po skonfigurowaniu interfejsu SPI pozostaje napisanie funkcji, które umożliwią zapis do przetworników wyjściowych. Jak wspomniano wcześniej, będą to funkcje *writeDAC_L()* oraz *writeDAC_R()*. Zasadniczo oba przetworniki dzielą tę samą magistralę, więc funkcje te będą się różniły jedynie portem, na który wystawiany jest stan niski podczas transmisji. Kod obsługi prawego kanału przedstawiono na **listingu 8**. To co rzuca się w oczy, to manipulacje bitowe na zmiennej *val*, które mają na celu ustawienie bitów kontrolnych przetwornika DAC (wyłączyć wzmocnienie, wybrać kanał wyjściowy, buforowanie, etc.). Na pozostałych 12 bitach przesyłana jest wartość napięcia. Instrukcja *while* testuje wartość zwracaną przez funkcję *SPI_I2S_GetFlagStatus*, aby ustalić, czy można zapisać nowe dane, czy też coś jest jeszcze wysyłane. Po ustawieniu sygnału CS w stan niski (aktywacja magistrali w wybranym przetworniku) następuje wysłanie słowa 16-bitowego zawierającego konfigurację oraz wartość za pomocą instrukcji *SPI_I2S_SendData*. Jako argument podawany jest interfejs SPI (SP1, SPI2 bądź SPI3 – tu będzie to SPI2). Pętla for oczekuje na zakończenie wysyłania, aby zmienić stan wyprowadzenia CS na wysoki i zakończyć transmisję.

```
//zapis do lewego kanalu
void writeDAC_R(__IO uint16_t val){
   __IO uint32_t speed ;
   //przetworzenie wartosci
   val &= 0x0FFF ;
   val |= 0x7000 ;
   while (SPI_I2S_GetFlagStatus(SPI2,¬
             SPI_I2S_FLAG_TXE) == RESET);
   GPIO_ResetBits(GPIOB, GPIO_Pin_12);
   SPI_I2S_SendData(SPI2, val) ;
   for(speed=0 ; speed<4 ; speed++){;}
   GPIO_SetBits(GPIOB, GPIO_Pin_12);
}
```
Listing 8

W tym momencie można już zapisywać sample do przetwornika DAC. Byłoby niepowetowaną stratą, gdyby nie sprawdzić, jak pracują przetworniki. Proponuję napisanie oprogramowania, które zamieni płytkę w miniorganki. Nie jest to może najbardziej spektakularny przykład, niemniej powinien być prosty do zrozumienia i interesujący. Przykład gotowego oprogramowania bazującego na opracowanych uprzednio funkcjach przedstawiono na **listingu 9**. Pierwsza pętla *for* jest przeznaczona do wygenerowania próbek przebiegu sinusoidalnego. Jak można zauważyć, wykorzystywane są tutaj operacje zmiennoprzecinkowe, które charakteryzują się dużą złożonością obliczeniową. Z tego względu przygotowana została tablica próbek, z których sample te będą po prostu pobierane. Znacznie przyspieszy to czas wykonania programu i przełoży się na możliwość pracy z większymi częstotliwościami. W jaki sposób regulować częstotliwość dźwięku? Wystarczy pobierać próbki z tablicy nie po kolei. Pobierając co czwartą próbkę, otrzymamy dwa razy wyższą częstotliwość niżby miało to miejsce przy pobieraniu co drugiej próbki.

```
int main(){
  //tablica probek
  __IO int16_t samples[8000] ;
  //zmienne
  __IO uint32_t step1=0, step2=5, step3=10, step4=15 ;
  __IO uint32_t i ;
  __IO double arg = 0.0;
  __IO double step = 0.0;
  int32_t dacValue = 0 ;
  //inicjacja
  gpioInit() ;
  configRCC() ;
  initDACs() ;
  //wypelnij tablice probek
  for(i=0 ; i<8000 ; i++){
    arg = 400.0 * sin(M_TWOPI*i/8000.0) ;
    samples[i] = arg ;
    step += 1.0 ;
  }
  //petla glowna
  while(1){
    //tworzenie sygnalu wyjsciowego
   dacValue = 2000 ; //przesuniecie, aby uniknac ujemnych liczb
    if(!GPIO_ReadInputDataBit(GPIOB, GPIO_Pin_0)){
      dacValue += samples[step1] ;
      step1+=8 ;
    }
    if(!GPIO_ReadInputDataBit(GPIOB, GPIO_Pin_1)){
      dacValue += samples[step2] ;
      step2+=14;
    }
    if(!GPIO_ReadInputDataBit(GPIOB, GPIO_Pin_10)){
      dacValue += samples[step3] ;
      step3+=19;
    }
    if(!GPIO_ReadInputDataBit(GPIOB, GPIO_Pin_11)){
      dacValue += samples[step4] ;
      step4+=29;
    }
    //pilnowanie zakresow
    step1 = step1 % 8000 ;
    step2 = step2 % 8000 ;
    step3 = step3 % 8000 ;
    step4 = step4 % 8000 ;
    //zapis  do przetwornika
    writeDAC_L(dacValue) ;
    writeDAC_R(dacValue) ;
  }
}
```
Listing 9

Naciśnięcie jednego z przycisków wymusza pobranie próbki z tablicy z jedną z czterech prędkości wyznaczanych przez zmienne *stepX*, modyfikując te wartości, np. ze *step1+=8* na *step1+=20,* można uzyskać inną, wyższą częstotliwość.

Naciśnięcie kilku przycisków jednocześnie spowoduje, że próbki będą pobierane kilka razy, ale z różnym odstępem (skokiem fazy) i tym samym będą miały różną częstotliwość. Mieszanie sygnałów o różnej częstotliwości osiąga się przez zwyczajne ich sumowanie. Warto, a nawet trzeba, pamiętać, że tablica próbek ma ograniczoną długość. W związku z tym nie można pobierać próbek spoza zakresu. Stąd instrukcje reszty z dzielenia (modulo %), które „pilnują" tego zakresu, „obcinając" nadmierne wartości.

Na końcu pozostaje przesłać wygenerowaną próbkę dźwięku do przetwornika za pomocą napisanych poprzednio funkcji.

Zachęcam do eksperymentów. Zepsucie urządzenia przez źle napisane oprogramowanie jest niemożliwe. Jedyne, co się ryzykuje to nieprzyjemne dźwięki w słuchawkach, dlatego warto na początku „przykręcić" potencjometry lub trzymać słuchawki w pewnej odległości od uszu.

Jakub Borzdyński
jakub.borzdynski@elportal.pl