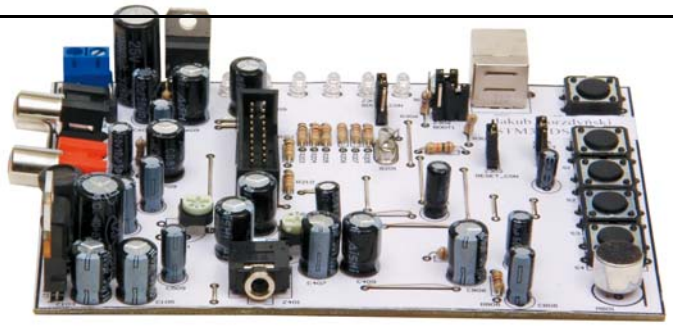


STM32 DSP KIT część 1

Nazwa urządzenia w pełni oddaje czym ono jest i w jakim celu został stworzony. Kluczowym elementem jest mikrokontroler z rodziny STM32, wyposażony w rdzeń ARM, konkretnie Cortex-M3. Celem artykułu jest pokazanie, jak postawić pierwsze kroki w świecie 32-bitowych mikrokontrolerów z rdzeniem ARM firmy STMicroelectronics. Do rozpoczęcia zabawy potrzebna jest jedynie płytka omówiona w dalszej części artykułu, przewód USB, zwykły zasilacz 9V AC lub DC (najlepiej około 0,3A), oczywiście komputer, dostęp do Internetu (trzeba poprać oprogramowanie, ale można to zrobić u kolegi), słuchawki i w zasadzie jest to wszystko. Co istotne, do programowania nie potrzeba programatora! Wystarczy fabrycznie wbudowany bootloader.

Litery DSP w nazwie projektu to angielski skrót od Digital Signal Processing, czyli cyfrowe przetwarzanie sygnałów (CPS). Obejmuje ona zagadnienia takie jak zamiana sygnału z postaci analogowej na cyfrową i odwrotnie, kompresja, przetwarzanie, filtrację i wiele więcej. Jest to bardzo interesująca dziedzina wiedzy obecna w wielu aspektach życia, czego najlepszym przykładem może być telefonia komórkowa oraz przenośne odtwarzacze multimedialne. Moją intencją było stworzenie płytki testowej, która pozwoli zapoznać się z zagadnieniami takimi jak:

* obsługa podstawowych peryferii STM 32 (ADC, SPI, UART, porty IO, timery, etc.),
* próbkowanie/rekonstrukcja dźwięku,
* nagrywanie dźwięku,
* DFT/FFT,
* filtracja cyfrowa,
* prosty generator DDS,
* efekty specjalne (echo, pogłos, vibrato, metalizacja dźwięku, etc.),
* wobuloskop (pozwalający badać np. charakterystyki filtrów),
* syntezator dźwięków,
* equalizer,
* maskowania czasowego i częstotliwościowego dźwięku (wstęp do modelu psychoakustycznego),
* szum kwantyzacji, aliasing,
* spektrogramy,
* syntezator mowy,
* kodowanie DPCM/ADPCM,
* odtwarzanie rewersyjne,
* zmiana częstotliwości dźwięku,
* synchronizator półkul mózgowych.

Uwaga! Układ stanowi zestaw edukacyjny, więc nie ma parametrów urządzeń komercyjnych. Zastosowane zostały tanie i dostępne przetworniki 12-bitowe. Oprogramowanie zaprezentowane w dalszej części artykułu zostało w całości napisane w języku C. Znajomość C/C++ przynajmniej w podstawowym stopniu jest tu niezbędna. Czytelników nieznających C lub C++ zachęcam do zapoznania się z kursem C, jaki był publikowany w EdW, przejrzenia zasobów Internetu lub kupienia książki związanej z tym zagadnieniem. Ze swojej strony mogę polecić książkę „Symfonia C++", J. Grębosza, z której sam się uczyłem.

Sprzęt

Schemat urządzenia został przedstawiony na **rysunku 1**. „Dziwna" trzycyfrowa numeracja elementów pozwala wyróżnić trzy główne bloki i ułatwia projektowanie PCB.

Pierwszym blokiem urządzenia jest zasilacz z elementami o numerach zaczynających się od 1. Rozdział na część analogową i cyfrową ma na celu zmniejszenie zakłóceń i poprawę jakości przetwarzanego dźwięku. Urządzenie może być zasilane napięciem stałym bądź przemiennym.

Drugi blok to 32-bitowy mikrokontroler, wyposażony w nowoczesny rdzeń Cortex-M3 z grupy ARM. Wbudowany bootloader umożliwia programowanie pamięci bez konieczności posiadania programatora. Użyty układ oferuje 64KB pamięci programu oraz 20KB pamięci RAM. Ma on m.in. dwa układy SPI, trzy układy UART oraz dwa 12-bitowe przetworniki ADC, obsługujące 10 kanałów.

Czytelnicy, którzy zdecydują się na zakup dedykowanych narzędzi (np. JTAG-a, ale nie jest to wymagane) mogą wykorzystać złącze Z201. Na interfejs użytkownika składa się siedem hiperjasnych diod LED. Można zmniejszyć rezystancję oporników R201...R207, jednak prąd pobierany z portów nie powinien być większy niż 8mA. Do interakcji z użytkownikiem przewidziane zostały 4 przyciski pozwalające, dzięki wewnętrznemu podciągnięciu, wymuszać stan niski.

Mikrokontroler jest taktowany kwarcem o częstotliwości „tylko" 8MHz, ale wbudowany mnożnik częstotliwości (pętla PLL) pozwoli uzyskać maksymalnie 72MHz.

Układ scalony U202 jest pamięcią typu Flash, przeznaczoną do przechowywania próbek dźwięku, co ma umożliwić stworzenie prostego odtwarzacza plików WAV oraz nagrywanie/odtwarzanie dźwięku.

Układ jest wyposażony w port USB z konwerterem USB<->RS232 FT232, co umożliwia korzystanie z bootloadera i pozwala wyświetlać informacje na ekranie komputera. Na płytce znajduje się pięć zworek odpowiedzialnych za bootowanie mikrokontrolera.

W urządzeniu występuje pięć kanałów analogowych: dwa wejścia liniowe (WE_L oraz WE_R), wejście mikrofonowe oraz dwa kanały wyjściowe z przetwornikami DAC. W każdym występuje prosty filtr antyaliasingowy ze wzmacniaczem NE5532. Zadaniem tego filtru jest usuwanie częstotliwości wyższych niż połowa częstotliwość próbkowania (wynika to z twierdzenia Nyquista).

Kanały wyjściowe nie mają wzmacniacza mocy i słuchawki są podłączone bezpośrednio do wyjść wzmacniaczy operacyjnych, co pozwoliło uprościć płytkę drukowaną przy zachowaniu przyzwoitej głośności. Oczywiście bezpośrednio do płytki nie można podłączyć głośnika. Do regulacji głośności można wykorzystać potencjometry PR401 oraz PR501.

Wszystkie wzmacniacze operacyjne są zasilane napięciem 5V, co pozwala uzyskać wartość międzyszczytową sygnału zmiennego około 3V (3Vpp). Przetwornik procesora pracuje w zakresie napięć od 0V do 3,3V, dlatego w każdym kanale wejściowym występuje dzielnik rezystorowy (np. elementy C709, R705, R706).

Środowisko programistyczne

Wszystkie przykłady zostały skompilowane w środowisku Ride 7 firmy Raisonance. Zostało ono wybrane głównie ze względu na brak ograniczenia kodu, przyjazny interfejs, współpracę z bibliotekami dostarczonymi przez producenta układów STM32 oraz brak opłat licencyjnych. Niestety, jak to zazwyczaj

bywa, jest pewien „haczyk”, mianowicie Ride 7 współpracuje TYLKO z debuggerem/programatorem Rlink, który można kupić na terenie Polski za około 400pln. Po zakupie podstawowej wersji Rlinka w tej cenie istnieje możliwość debugowania programów zajmujących do 32KB, aby obsłużyć większy kod, należy rozszerzyć wersję standardową do profesjonalnej, co kosztuje znacznie więcej.

Jak wspomniałem na początku, do programowania zostanie wykorzystany bootloader, więc od Ride 7 wymagać będziemy jedynie wygenerowania pliku HEX zawierającego skompilowany kod.

Czytelnicy, którzy chcą za wszelką cenę mieć debugger, mogą zakupić klon ST-Linka – ZL30PRG w przystępniejszej cenie. Jest on obsługiwany przez środowiska takie jak Keil czy Atollic TrueSTUDIO. Niemniej należy się wtedy liczyć z ograniczeniami w funkcjonowaniu IDE. Oba te środowiska również obsługują biblioteki dla STM32, więc nie powinno być problemów ze zmianą narzędzia.

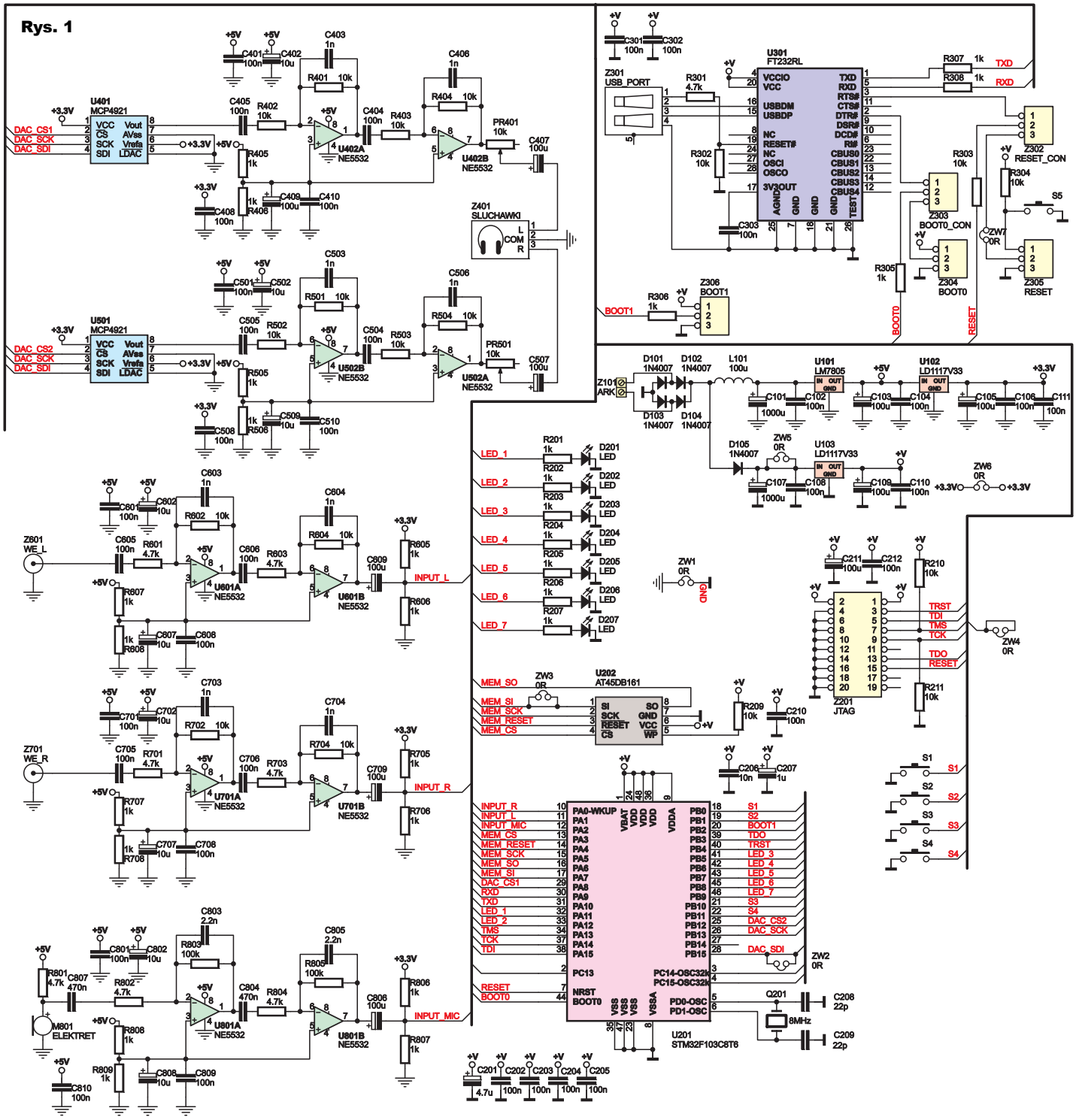
Przed przystąpieniem do pracy należy pobrać z Internetu trzy rzeczy. Pierwszą są bezpłatne sterowniki dla układu FT232 ze strony **http://www.ftdichip.com/** Na stronie głównej należy kliknąć link *Drivers*, a następnie *VCP* i wybrać odpowiednie dla swojego systemu operacyjnego. Warto kliknąć *setup executable,* nawet kosztem trochę starszej wersji.

Po zainstalowaniu sterowników należy przejść na stronę Raisonance (**http://raisonance.com/**), pobrać Ride 7 oraz Rkit-ARM i zainstalować w takiej kolejności. Gdy instalator zapyta, czy zainstalować sterowniki, warto się zgodzić.

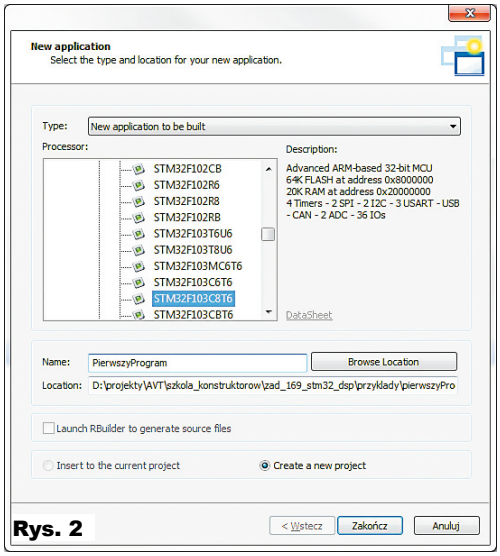
Pierwszy projekt

Tworzenie nowego projektu nie jest szczególnie skomplikowane. Wystarczy otworzyć środowisko Ride 7, wybierając je z menu start i następnie wybrać pozycję *New Project* z menu *Project*. Spowoduje to otwarcie okna dialogowego. W polu *Processor* należy rozwinąć grupę *STM32x* i odnaleźć STM32F103C8T6, wybrać folder do przechowywania plików i kliknąć *Zakończ*. Warto zmienić *Name*, aby projekt miał bardziej przystępną nazwę niż *Application0*. Poprawnie wypełnione okienko pokazano na **rysunku 2**.

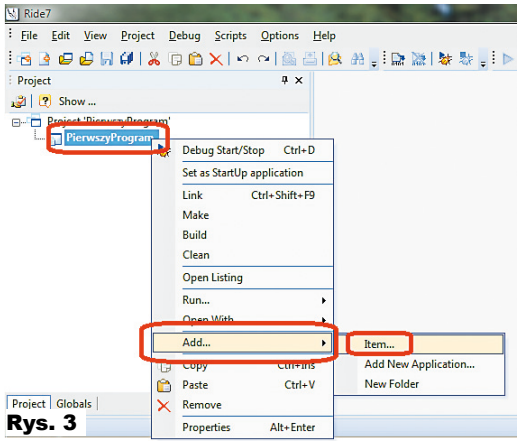
Rys. 1



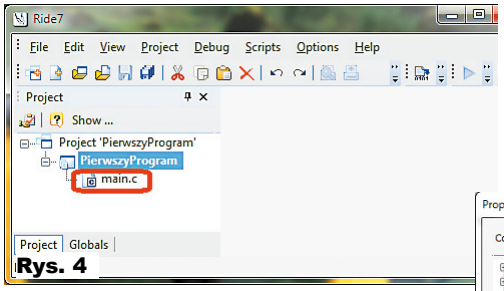
Na początku należy utworzyć główny plik, w którym zawarta będzie funkcja *main*. Najprościej to zrobić, klikając na nazwie projektu prawym przyciskiem myszy i wybierając *Add*, a następnie Item (**rysunek 3**). W otwartym oknie dialogowym należy przejść do folderu z projektem, w pole *Nazwa pliku* wpisać *main.c* i nacisnąć *otwórz*. Oczywiście pliku takiego nie ma w katalogu, ale zostanie w tym momencie utworzony i dodany do projektu – **rysunek 4**. Podwójne kliknięcie na nim otworzy w edytorze ten plik. Jest on pusty, więc na początku warto go wypełnić podstawowymi instrukcjami – **rysunek 5**. W ten oto sposób powstał najprostszy możliwy program. Można go skompilować przyciskiem F9 lub wybierając *Project->Make Project*. Czytelnicy, którzy zajrzeli do folderu z projektem, przekonają się, że nie powstał oczekiwany plik HEX. Jest on generowany dopiero po „zbudowaniu" projektu – *Project->Build Project* (lub ALT + F9).



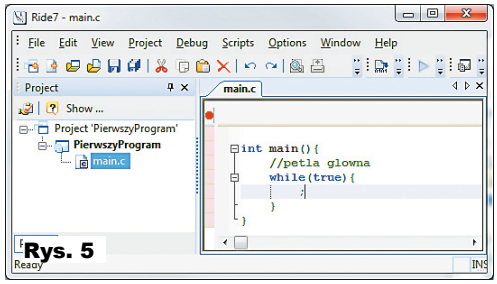
Rys. 2



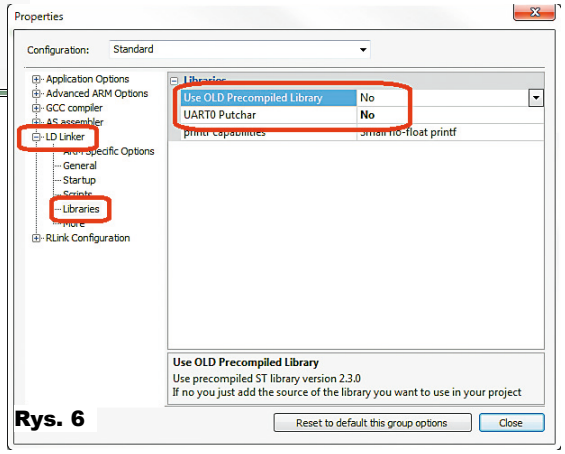
Rys. 3



Rys. 4



Rys. 5



Rys. 6

Zanim napiszemy sensowny program, parę zdań komentarza. Po pierwsze – biblioteki. Układ STM32 jest nowoczesnym, rozbudowanym procesorem z rdzeniem 32-bitowym, przez co niełatwa jest obsługa peryferii i trudno określić, co i gdzie wpisać. Analiza dokumentacji jest uciążliwa, więc firma STMicroelectronics przygotowała biblioteki, upraszczające korzystanie z peryferii. Pozwalają one łatwo skonfigurować układy peryferyjne, bez konieczności wgłębiania się w znaczenie każdego bitu. Ceną jest nieznaczne zwiększenie rozmiaru kodu wynikowego wskutek dołączenia bibliotek. Z reguły dołącza się tylko te biblioteki, które są aktualnie potrzebne.

Drugą rzeczą, o której warto wiedzieć, jest to, że Ride 7 (przynajmniej wersja, która została wykorzystana do napisani przykładowych programów) jest dostarczana z dwiema wersjami bibliotek, nowszą i starszą. Nas interesować będzie tylko ta nowsza, więc należy wyłączyć używanie starszej, aby nie dopuścić do konfliktu wersji (inaczej kod może się nie skompilować). Dlatego przed rozpoczęciem pisania programów trzeba wybrać z menu *Project* pozycję *Properties* i zmienić dwie opcje z YES na NO, tak jak pokazano to na **rysunku 6**.

Ostatnią rzeczą, o której należy wspomnieć, jest pomoc związana z biblioteką, która znajduje się w pliku:
stm32f10x_stdperiph_lib_um.chm

Plik ten jest zlokalizowany w folderze:
(...)\Ride\lib\ARM\STM32Lib-v312

Korzystanie z bibliotek rozpoczyna się od dołączenia pliku *stm32f10x.h* do projektu poprzez dodanie linijki:

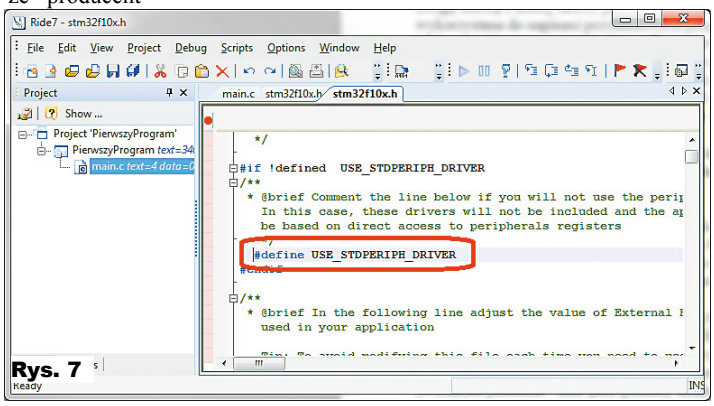
#include „stm32f10x.h"

na początku pliku main.c. Próba kompilacji skończy się błędem z informacją o konflikcie wersji, który omija się kopiując plik *stm32f10x.h* do folderu z projektem (tam gdzie jest main.c). Po skopiowaniu pliku kompilacja ponownie kończy się błędem, gdyż brakuje plików: *core_cm3.h* oraz *system_stm32f10x.h*. Również te dwa pliki należy skopiować do folderu z projektem z katalogu:

(..)\Ride\lib\ARM\STM32Lib-v312\Libraries\CMSIS\Core\CM3.

Tym razem kompilacja zakończony się bez błędu.

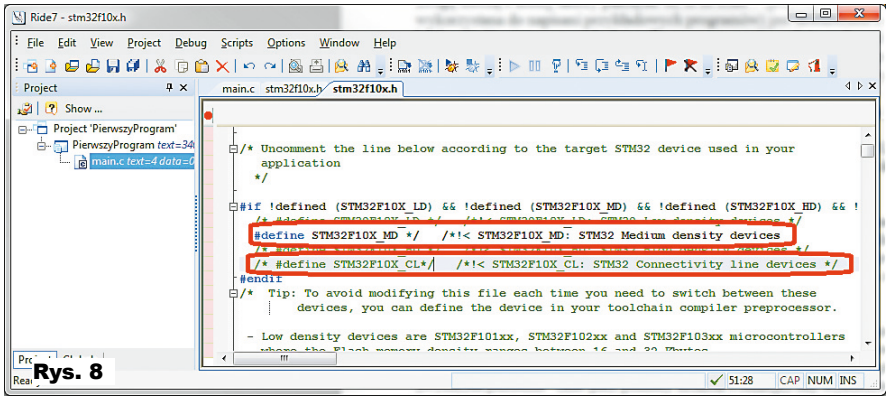
Należy też wspomnieć, że producent dostarcza dwa rodzaje bibliotek: umożliwiające bezpośrednie działanie na rejestrach oraz pozwalające konfigurować peryferia na „wyższym poziomie" (tzn. za pomocą struktur i funkcji). My wykorzystamy tę drugą wersję, która domyślnie jest wyłączona. W związku z tym należy otworzyć skopiowany uprzednio plik *stm32f10x.h* i zmodyfikować go. Modyfikacja ta sprowadza się jedynie do usunięcia komentarza w 71 linii – pokazuje to **rysunek 7**. Przy okazji w tym samym pliku należy zmienić grupę mikrokontrolera z *Connectivity line devices* na *Medium density devices*. Sprowadza się to do zakomentowania linii 51 i odkomentowania linii 49 – **rysunek 8**.



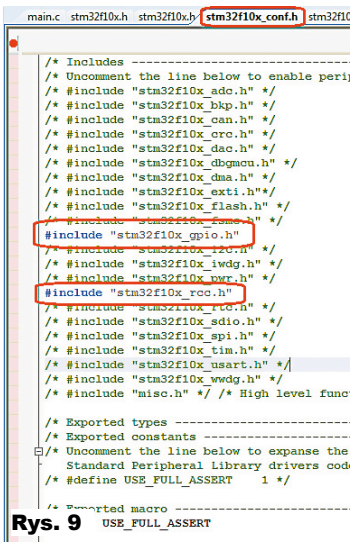
Rys. 7

Nadal nie uda się skompilować projektu, gdyż brakuje pliku *stm32f10x_conf.h*. Jest to plik, w którym wyszczególniono wszystkie pliki nagłówkowe dołączane do projektu i przeznaczone do obsługi modułów peryferyjnych. Usuwając bądź dodając komentarze można zdecydować, które moduły będą używane. Plik ten można znaleźć np. w folderze:

(...)\Ride\lib\ARM\STM32Lib-v312\Project\Template



Rys. 8



Rys. 9

i należy go przekopiować do katalogu z projektem. Niezbędna będzie edycja tego pliku, polegająca na dołączeniu plików odpowiedzialnych za obsługę portów I/O oraz systemu taktowania (RCC). Widok poprawnie zmodyfikowanego pliku pokazano na **rysunku 9**. W celu uniknięcia konfliktu wersji, należy przekopiować odkomentowane pliki – *stm32f10x_gpio.h* oraz *stm32f10x_rcc.h* z folderu:

(…)\Ride\lib\ARM\STM32Lib-v312\Libraries\STM32F10x_StdPeriph_Driver\inc do folderu z projektem.

Teraz można przystąpić do pisania oprogramowania. Pierwszy program tradycyjnie będzie migał jedną z diod LED. Pierwszym etapem jest zazwyczaj skonfigurowanie portu I/O. W tym wypadku też tak będzie. Należy jednak zapamiętać ważny szczegół: każdy moduł, taki jak port I/O, przetwornik ADC czy interfejs SPI, ma odłączony sygnał zegarowy po resecie, co oznacza, że w ogóle nie pracuje. Takie podejście pozwala zredukować zużycie energii poprzez wyłączenie niewykorzystywanych modułów. Sprawia to jednak, że wyłączonych modułów nie można skonfigurować i dlatego konfigurację zawsze trzeba rozpocząć od podłączenia sygnału zegarowego. W przeciwnym razie moduł nie zostanie skonfigurowany i nie będzie pracował zgodnie z oczekiwaniami. Za dołączanie sygnału zegarowego odpowiedzialne są funkcje *RCC_APB1PeriphClockCmd* oraz *RCC_APB2PeriphClockCmd* przyjmujące dwa argumenty: stałą, określającą moduł, o który nam chodzi, oraz polecenie *ENABLE* bądź *DISABLE*, włączające bądź wyłączające taktowanie. Chcąc dołączyć sygnał zegarowy do portu B, należy użyć następującej instrukcji:

RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOB, ENABLE);.

Wybór pomiędzy *RCC_APB1PeriphClockCmd* a *RCC_APB2PeriphClockCmd* jest uzależniony od tego, do której magistrali podpięty jest układ peryferyjny. Na **rysunku 10** widać, że port B jest podłączony do magistrali APB2 (rysunek pochodzi z noty katalogowej układu STM32).

Kolejnym zadaniem jest skonfigurowanie portu I/O. Odbywa się to poprzez deklarację struktury *GPIO_InitTypeDef* i przypisanie jej składnikom określonych wartości. Struktura przypomina klasę, w której wszystkie składniki są publiczne. Innymi słowy jest to pojemnik, w którym przechowywana jest dowolna ilość zmiennych. Do wybranej zmiennej odnosi się, podając jej nazwę poprzedzoną kropką i nazwą struktury. Dzięki takiemu podejściu można przekazywać szybko duże ilości zmiennych w argumencie funkcji.

Struktura konfigurująca port ma trzy pola, o których warto wspomnieć:

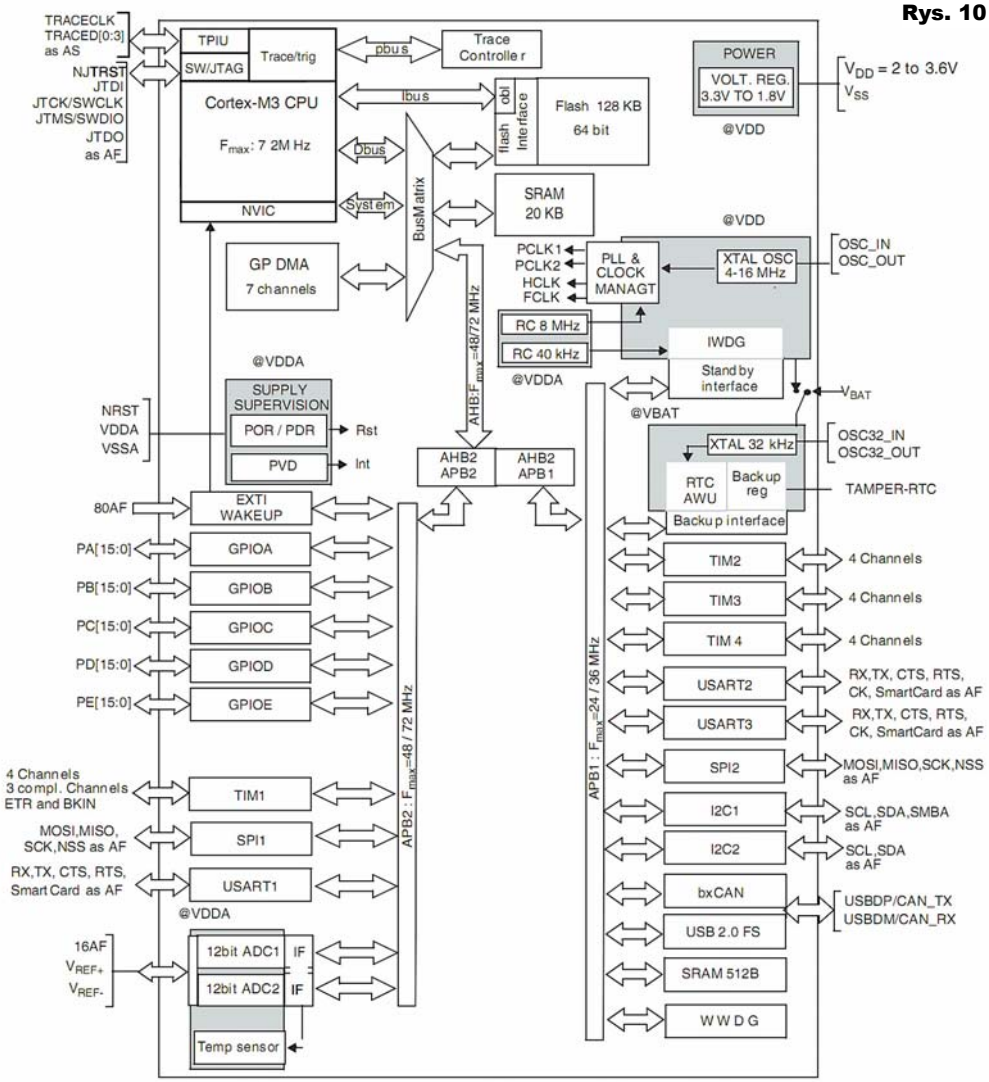
* *GPIO_InitStructure.GPIO_Pin* – określa piny, które mają zostać skonfigurowane.
* *GPIO_InitStructure.GPIO_Speed* – określa prędkość pracy portu (im niższa częstotliwość, tym mniej zaburzeń elektromagnetycznych jest generowanych).
* *GPIO_InitStructure.GPIO_Mode* – tryb pracy portu (wejście, wyjście, wejście z podciąganiem, etc.).

Załóżmy, że sterować będziemy diodą dołączoną do portu PB8, port będzie pracował z najniższą częstotliwością i będzie ustawiony jako wyjściowy. W takim przypadku poszczególne elementy struktury należy ustawić następująco:

GPIO_InitStructure.GPIO_Pin = GPIO_Pin_8 ;

GPIO_InitStructure.GPIO_Speed = GPIO_Speed_2MHz;

Rys. 10



GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;.

Mając wypełnioną strukturę, należy przekazać ją do funkcji *GPIO_Init,* odpowiedzialnej za skonfigurowanie portu zgodnie z parametrami umieszczonymi w strukturze. Funkcja ta przyjmuje dwa parametry: pierwszym jest nazwa konfigurowanego portu, a drugim ta właśnie struktura poprzedzona ampersandem (&).

Pozostało odpowiedzieć na pytanie, jak sterować końcówkami portu. Służą do tego funkcje *GPIO_SetBits* oraz *GPIO_ResetBit*, które odpowiednio ustawiają lub zerują wybrane bity. Każda z tych dwóch funkcji przyjmuje dwa argumenty: pierwszym jest nazwa portu (tu będzie to GPIOB) a drugim identyfikator żądanego wyprowadzenia (GPIO_Pin_X, gdzie X to numer portu).

Ostateczny program migającej diody pokazano na **listingu 1**. Zawarto tu dodatkowo pętle *for* służące do opóźnienia działania programu, aby miganie LED było dostrzegalne.

Kompilacja programu zakończy się błędem (ostatni raz!), gdyż należy do projektu dołączyć biblioteki. W tym celu należy skopiować do folderu z projektem pliki *.c odpowiadające użytym plikom nagłówkowym (tu będzie to *stm32f10x_gpio.c* oraz *stm32f10x_rcc.c*). Pliki te należy dodać do projektu podobnie jak plik *main.c,* z tym że tym razem trzeba wskazać rzeczywiście istniejące pliki. Inną, prostszą metodą jest po prostu przeciągnięcie ich do zakładki *Project*. Wygląd Ride 7 z poprawnie dodanymi plikami i właściwym kodem źródłowym przedstawiono na **rysunku 11**. Naciśnięcie ALT+F9 skompiluje program i utworzy plik HEX.

Programowanie mikrokontrolera

Po kompilacji projektu powstaje plik HEX, którym należy zaprogramować mikrokontroler. Po pierwsze, należy pobrać z Internetu program, który współpracuje z bootloaderem obecnym w układzie STM32. Program ten nosi nazwę Flash Loader Demonstrator (dla wygody dalej będzie oznaczany skrótem FLD). Bezpośredni link do pliku jest następujący: http://www.st.com/internet/com/SOFTWARE_RESOURCES/SW_COMPONENT/SW_DEMO/um0462.zip. Ten link jest aktualny w momencie przygotowywania niniejszego artykułu, ale w międzyczasie może stać się nieaktualny. W takie sytuacji warto wpisać w Google: *flash loader demonstrator stm32* lub na stronie **www.st.com** odnaleźć mikrokontroler użyty w urządzeniu i poszukać wśród plików z nim związanych. Po pobraniu oprogramowania należy je zainstalować, co przebiega w typowy sposób.

Na płytce znajduje się pięć goldpinów, na które należy nałożyć zworki. Ich położenie decyduje o tym, co będzie się działo z układem. Najbardziej typowe scenariusze przedstawiono w tabelce na **rysunku 12**. Pin pierwszy można rozpoznać po kwadratowym padzie lub oddzieleniu go od pozostałych pinów kreską na warstwie opisowej. Na PCB umieszczono również numerację. Zła konfiguracja zworek nie uszkodzi w żaden sposób mikrokontrolera, co najwyżej nie pozwoli go zaprogramować do czasu poprawnego ich nałożenia.

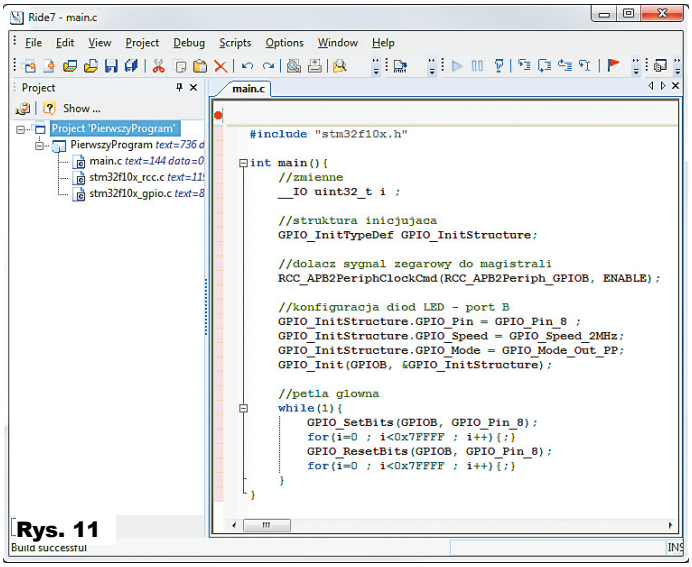
Programowanie w zasadzie wymaga nałożenia zworek zgodnie z pozycją „programowanie" tabeli, zresetowanie mikrokontrolera przyciskiem s5 i uruchomienia FLD. W otwartym okienku (**rysunek 13**) należy wskazać port szeregowy, do którego podłączony jest mikrokontroler. Tu wyjaśnienie: moduł jest podłączany do komputera za pomocą USB, jednakże obecność układu FT232 i zainstalowanych sterowników sprawia, że jest on widoczny jako COMx, gdzie x to numer portu, do którego podłączono układ. W przypadku nieznajomości portu pozostaje poeksperymentowanie i wybieranie kolejnych portów COM z listy. W praktyce zapewne będzie to ostatni port na liście, gdyż system Windows zazwyczaj numeruje dołączane urządzenia kolejnymi numerami, stąd ostatnio dołączony układ będzie miał najwyższy numer.

Poprawnie nałożone zworki w połączeniu z poprawnie wybranym portem spowodują, że po kliknięciu przycisku Next w FLD przejdziemy do kolejnego okienka, zamiast zobaczyć komunikat o błędzie.

Klikając trzy razy *Next*, dochodzi się do okienka pokazanego na **rysunku 14**. Należy w nim wskazać plik HEX, który ma być wgrany do pamięci mikrokontrolera, klikając na przycisk z trzema kropkami. Warto zaznaczyć także opcje *Jump to user program* (po zapisaniu pamięci programu nastąpi automatyczne uruchomienie programu – bardzo wygodne rozwiązanie) oraz *verify after download* (po programowaniu nastąpi weryfikacja). Należy również wymusić kasowanie pamięci przed programowaniem (Global Erase). Po wybraniu pliku i zaznaczeniu niezbędnych opcji pozostaje kliknięcie *Next*, co rozpocznie programowanie.

```
#include "stm32f10x.h"
int main(){
  //zmienne
  __IO uint32_t i ;
  //struktura inicjujaca
  GPIO_InitTypeDef GPIO_InitStructure;
  //dolacz sygnal zegarowy do magistrali
  RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOB, ENABLE);
  //konfiguracja diod LED - port B
  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_8 ;
  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_2MHz;
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
  GPIO_Init(GPIOB, &GPIO_InitStructure);
  //petla glowna
  while(1){
    GPIO_SetBits(GPIOB, GPIO_Pin_8);
    for(i=0 ; i<0x7FFFF ; i++){;}
    GPIO_ResetBits(GPIOB, GPIO_Pin_8);
    for(i=0 ; i<0x7FFFF ; i++){;}
  }
}
```

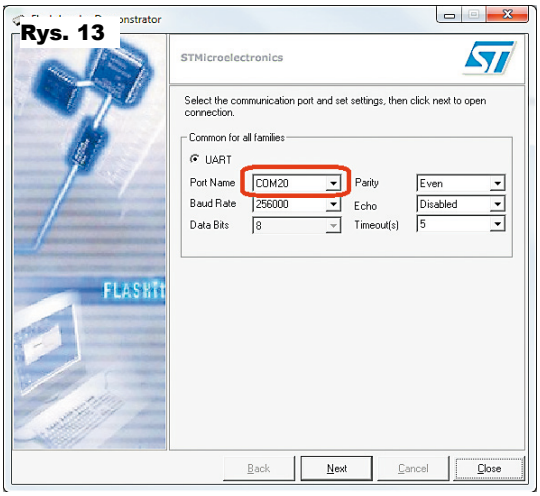
Listing 1



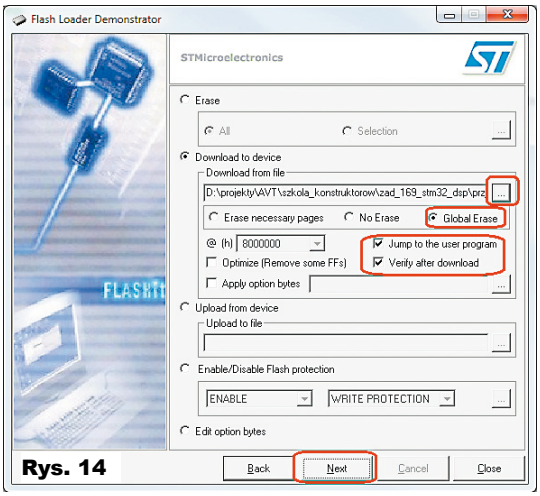
Rys. 11

Rys. 12	BOOT0	BOOT1	RESET	BOOT0_CON	RESET_CON
Programowanie	1-2	2-3	1-2	2-3	2-3
Uruchomienie programu	2-3	2-3	1-2	2-3	2-3
Tryb automatyczny	x	2-3	x	1-2	1-2

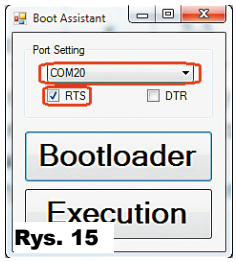
1-2 – zworka nałożona na pierwszy i drugi pin
2-3 – zworka nałożona na drugi i trzeci pin
x – nie ma znaczenia
UWAGA! W trybie automatycznym przycisk s5 (reset) nie działa



Rys. 13



Rys. 14



Rys. 15

Po chwili zostanie ono zakończone, a nowy program uruchomiony. Dioda LED zacznie migać.

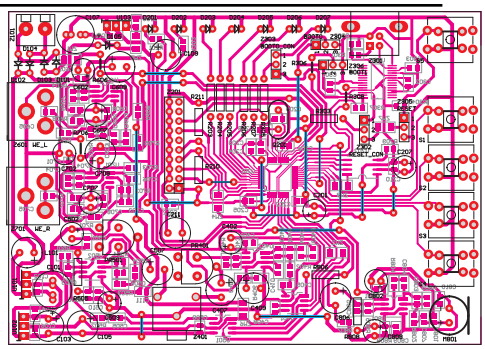
I na koniec jeszcze dwie uwagi. Po pierwsze, warto zauważyć, że zworki mogą być cały czas w pozycji „programowanie". Program zostanie uruchomiony po zakończeniu programowania, a po naciśnięciu przycisku Reset mikrokontroler uruchomi bootloader. Dzięki temu praca jest wygodniejsza: wgrywamy nowy program, obserwujemy jego działania, a w celu przeprogramowania naciskamy reset, programujemy, itd. Oczywiście przy takim podejściu nie można zresetować mikrokontrolera, gdyż naciśnięcie s5 załaduje bootloader, a nie program użytkownika.

Druga rzecz to ostatnia pozycja tabeli – tryb automatyczny. Jest to tryb dla leniwych, które lubią zlutować układ, podłączyć co trzeba, odłożyć płytkę na skraj biurka i kontrolować jego pracę za pomocą komputera. W związku z tym w materiałach dostępnych na Elportalu udostępniono mały program *Boot Assistant* napisany w Visual Studio (wymaga .NET Framework). Jest to prosta aplikacja kontrolująca linie RTS oraz DTR obecne w układzie FT232. Mogą one zostać podłączone do wyprowadzeń Reset oraz BOOT0 (zworka BOOT1 zawsze musi być w pozycji 2-3, chyba że ktoś zamierza uruchamiać program z RAM-u). Tym samym można z poziomu komputera kontrolować tryb pracy STM32. Po uruchomieniu aplikacji *Boot Assistant* ukazuje się okno, jak na **rysunku 15**. Odpowiednim przyciskiem można uruchomić bootloader lub zapisany program w pamięci Flash. Każde naciśnięcie *Execution* powoduje reset mikrokontrolera i wykonanie zapisanego w nim programu. Należy jednak zaznaczyć, że linia RTS musi być zanegowana, co sprowadza się do odznaczenia pola RTS. Potem należy wybrać port (ten sam co w FLD) i można korzystać z aplikacji. Dodam, że oba programy mogą być uruchomione jednocześnie, gdyż oba po zakończeniu pracy zwalniają port.

Montaż i uruchomienie

Układ został zmontowany na jednostronnej płytce drukowanej, której wzór przedstawiono na **rysunku 16**. Najtrudniejszym elementem w montażu będzie zapewne mikrokontroler oraz układ FT232, gdyż mają one gęsty raster. Montaż najlepiej rozpocząć od lekkiego pocynowania jednego, dowolnego padu i ułożenia układu scalonego za pomocą pęsety. Trzymając układ tak, aby pasował do pozostałych padów, należy lutownicą podgrzać pocynowany pad, aby przytwierdzić układ. Po precyzyjnym ułożeniu można pocynować kolejny pad i należy się upewnić, że mikrokontroler się nie przesunął. Kiedy wszystko jest w porządku, należy przyłożyć cynę do pozostałych wyprowadzeń i podgrzać lutownicą, nie przejmując się zwarciami powstałymi pomiędzy pinami. Kiedy wszystkie wyprowadzenia zostaną przylutowane, to plecionką należy odessać nadmiar cyny i patrząc pod światło upewnić się, że nie ma zwarć. Jest to metoda bardzo prosta, ale skuteczna i niewymagająca żadnych zaawansowanych narzędzi. Układ FT232 lutuje się podobnie. Pozostałe układy scalone mają rozstaw wyprowadzeń na tyle duży, że można przyłożyć cynę punktowo, podgrzać i lutować wyprowadzenia po kolei.

Przerażenie może budzić również płytka drukowana, która ma w niektórych miejscach bardzo wąskie ścieżki. Niemniej prototyp został zrobiony na płytce domowej roboty, którą wykonano według instrukcji podanych w artykule z EdW 6/2009. Osoby mniej doświadczone mogą kupić płytkę drukowaną w sklepie AVT z wlutowanymi dwoma układami – procesorem i FT232. Płytka jest zaprojektowana dość ciasno, więc należy zwrócić uwagę, aby stabilizatory U101 oraz U102 nie stykały się po wlutowaniu w płytkę. Montaż należy zacząć bezwzględnie od zworek, gdyż jedna z nich wypadnie pod złączem Z201. W celu


Rys. 16 Skala 50%

ograniczenia zaburzeń emitowanych do otoczenia warto ekranować rezonator kwarcowy. Sprowadza się to do podłączenia jego metalowej obudowy do masy. W tym celu obok rezonatora znajduje się pojedynczy, wolny otwór w który można włożyć kawałek drutu i doprowadzić w ten sposób masę do obudowy.

Płytka została zaprojektowana w taki sposób, że można wlutować przyciski w obudowach 6x6mm lub 12x12mm – do wyboru. Na modelu widocznym na fotografii tytułowej wlutowane zostały przyciski w większych obudowach.

Koszmarny zapewne będzie montaż elementów SMD, takich jak rezystory i kondensatory, ze względu na duże upakowanie i trudności z określeniem, których elementów dotyczą poszczególne sygnatury. Warto wykorzystać „ściągę" z **rysunku 17** (można ją ściągnąć z Elportalu), powinno to rozwiać wszelkie wątpliwości.

W dwóch następnych częściach artykułu zostaną podane dalsze szczegóły i przykłady wykorzystania.

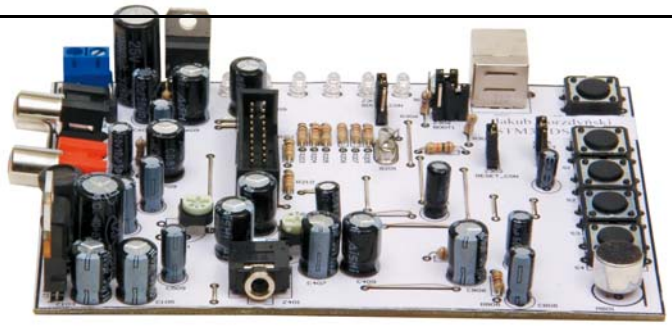
Jakub Borzdyński
jakub.borzdynski@elportal.pl

Od Redakcji. ***Zgodnie z prośbą Redaktora Naczelnego ze wstępniaka na str. 5, napiszcie (pg@elportal.pl), czy chcielibyście rozszerzyć ten artykuł w minicykl, pomagający postawić pierwsze kroki w świecie mikrokontrolerów 32-bitowych.***

Wykaz elementów

R201,R202, R307,R405, R406,R506,R605,R607, R608, R705,R707,R708,R807,R809	1kΩ 0805
R203,R204,-R207,R305,R505,R806,R808	1kΩ
R209,R302,R304,R401-R404,R501-R504,R602,R604, R702,R704	10kΩ 0805
R210,R211, R303	10kΩ
R301,R601,R603,R701,R703,R801,R802,R804	4,7kΩ 0805
R306,R308,R606,R706	1kΩ
PR401,PR501	10kΩ PR
C101,C107	1000µF
C102,C104,C106,C108,C110,C111,C202-C205,C210,C212, C301,C302,C303,C401,C404,C405,C408,C410,C501,C504, C505,C508,C510, C601,C605,C606,C608,C701,C705,C706, C708,C801,C809,C810	100nF 0805
C103,C105,C109,C211,C407,C409,C507,C609,C709, C806	100µF
C201	4,7µF
C206	10nF 0805
C207	1µF
C208,C209	22pF 0805
C402,C502,C509,C602,C607,C702,C707,C802,C808	10µF
C403,C406,C503,C506,C603,C604,C703,C704	1nF 0805
C803,C805	2,2nF 0805
C804,C807	470nF 0805
D101-D105	1N4007
D201-D207	LED
U101	LM7805
U102,U103	LD1117V33
U201	STM32F103C8T6
U202	AT45DB161
U301	FT232RL
U401,U501	MCP4921
U402,U502,U601,U701,U801	NE5532
L101	100µH
M801	elektret
Q201	8MHz
S1-S5	uSwitch
Z101	ARK2
Z201	JTAG
Z301	USB CON
Z302,Z305	mikroswich
Z401	gniazdo słuchawkowe do druku
Z601,Z701	gniazdo CINCH
ZW1-ZW7	0Ω 1206

Komplet podzespołów z płytką jest dostępny w sieci handlowej AVT jako kit szkolny AVT-2975.



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.

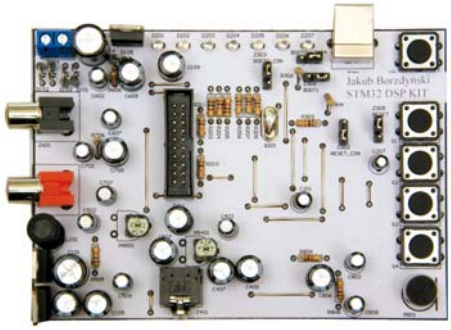
Sprawdzenie stanu portu jest proste i wymaga wywołania funkcji *GPIO_ReadInputDataBit*, która zwraca stan logiczny obecny na wyprowadzeniu. Argumenty tej funkcji są w zasadzie takie same jak funkcji ustawiającej czy kasującej bity: najpierw port, potem stała określająca wyprowadzenie. Chcąc sprawdzić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

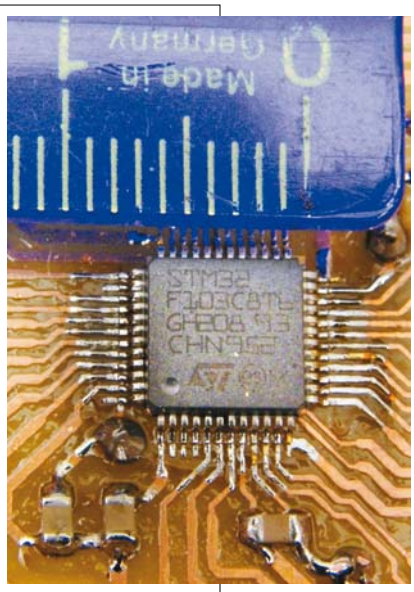


Listing 2

```
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

```
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 ;
    }
  }
}
```

Listing 4

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

```
//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) {;}
  }
}
```

Listing 5

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.

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

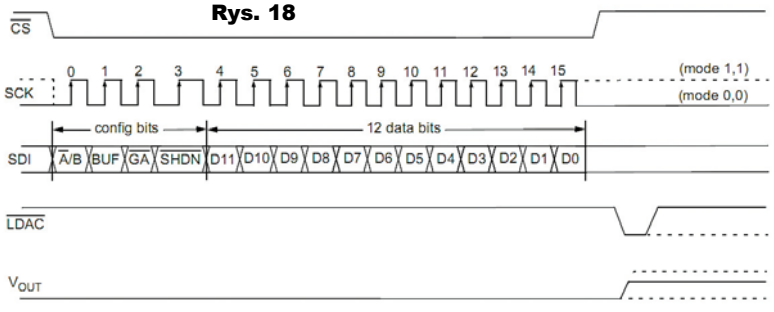
Po stworzeniu nowego projektu i własnej, prostej biblioteki nadeszła pora na napisanie procedur obsługi DAC. Jak wspomniano przed chwilą, potrzebne będą dwie funkcje: konfigurująca oraz zapisująca dane. Najwygodniej będzie najpierw zająć się konfiguracją. Po pierwsze, należy odpowiedzieć sobie na pytanie, 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 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

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

```
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

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.

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