32 bity jak najprościej (1)

Pierwsze kroki z modułem STM32F0DISCOVERY

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

Środowisko Keil MDK-ARM

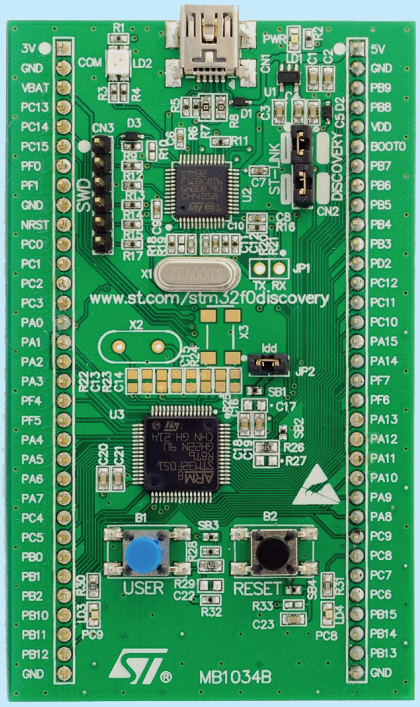
Przykłady programów zostały przygotowane przy użyciu środowiska MDK-ARM firmy Keil dla systemu Windows, którego darmową wersję ewaluacyjną z ograniczeniem długości kodu do 32 kB można pobrać z portalu www.keil.com. Spośród wielu dostępnych środowisk programowania dla mikrokontrolerów z rdzeniami ARM, MDK-ARM wyróżnia się łatwością przygotowania do pracy i tworzenia pierwszych projektów. Jest to więc to dobry wybór dla stawiających pierwsze kroki w programowaniu mikrokontrolerów z rdzeniami Cortex.

Przed rozpoczęciem działań z STM32F0 należy pobrać i zainstalować aktualną wersję środowiska. Prezentowane przykłady zostały utworzone przy użyciu wersji 4.70 i 4.60.

Wszystkie przykładowe projekty opisane w tekście znajdują się w pliku F0tutorial.zip. Po rozpakowaniu pliku do nowego foldera projekty można otworzyć przy użyciu środowiska Keil indywidualnie (klikając na pliku projektu z rozszerzeniem .uvproj) lub poprzez plik przestrzeni roboczej f0tutorial.umpw. Warto jednak stworzyć je od nowa samodzielnie, nabierając w ten sposób wprawy w posługiwaniu się środowiskiem.

Przygotowanie do pracy z STM32F0DISCOVERY

Po zainstalowaniu środowiska w komputerze, przed pierwszym podłączeniem modułu Discovery, należy w folderze, w którym zostało zainstalowane środowisko (domyślnie C:\Keil) odnaleźć folder *ARM\STLink\USBdriver* i uruchomić zawarty w nim instalator *ST Link_V2_USBdriver.exe*. Następnie należy dołączyć moduł Discovery i korzystając z dialogu instalacji drivera w systemie, przerwać poszukiwanie drivera w sieci. System Windows powinien odnaleźć świeżo zainstalowany driver i skonfigurować go do pracy z płytką Discovery. Uwaga: Próby instalacji drivera STLink w inny sposób często kończą się niepowodzeniem.



Po rozpoznaniu modułu Discovery (co zostaje potwierdzone ciągłym świeceniem większej z czerwonych diod umieszczonych na płytce w okolicy złącza USB) należy zaktualizować wbudowane oprogramowanie STLink, uruchamiając z foldera ARM\STLink program ST LinkUpgrade.exe.

Tworzenie projektu

Na początek stworzymy projekt z programem, który nic nie robi oraz przyjrzymy się procesowi tworzenia projektu i budowie modułu startowego

- Wybieramy z menu opcję Project – new mVision Project.
- Wybieramy lokalizację projektu. Warto założyć folder grupujący wszystkie projekty, a w nim dopiero fol-

Listing 1. Pierwszy, niemal pusty program dla STM32F0DISCOVERY

```
/*
  STM32F0DISCOVERY
  empty program project
   gbm, 12'2012
*/

#include „stm32f0xx.h"

//=======================================================
void SystemInit(void)
{
}
//=======================================================
int main(void)
{
}
```

der projektu. Pierwszemu projektowi nadamy nazwę *Empty*.

- Po wprowadzeniu nazwy projektu wybieramy producenta i typ mikrokontrolera – w naszym przypadku STMicroelectronics STM32F051R8.
- Akceptujemy sugestię wygenerowania przez środowisko pliku startowego i włączenia go do projektu.

W ten sposób stworzyliśmy pierwszy, niekompletny jeszcze projekt oprogramowania. Został wygenerowany moduł startowy zawierający deklaracje procedur obsługi wyjątków oraz procedurę inicjującą działanie mikrokontrolera. Środowisko ustawiło również ścieżki dla plików nagłówkowych, dzięki czemu będziemy mogli włączać je do naszych modułów pisanych w języku C.

Rozwijamy foldery w panelu widoku plików z lewej strony okna i otwieramy moduł startowy. W środowisku Keil jest on napisany w języku asemblera ARM, pomimo że nie ma przeciwwskazań, aby był on napisany w C. W module możemy zauważyć deklaracje nazw procedur obsługi wyjątków oraz słabe definicje tych procedur. Dzięki temu mamy zapewnioną domyślną obsługę niespodziewanych wyjątków w postaci pętli nieskończonych, a jednocześnie mamy możliwość stworzenia własnych procedur, które przysłonią procedury domyślne.

Fragment kodu rozpoczynający się etykietą *ResetHandler* jest procedurą inicjującą działanie mikrokontrolera, uruchamianą po zainicjowaniu go, np. przez włączenie zasilania lub podanie sygnału *Reset*. Procedura ta w środowisku Keil składa się jedynie z wywołań dwóch procedur: *SystemInit* i *__main*. Procedura *SystemInit* służy do wczesnego zainicjowania mikrokontrolera; zwykle zawiera ona programowanie generatora przebiegu zegarowego oraz konfigurację parametrów dostępu do pamięci. Procedura *__main* jest zawarta w bibliotece dostarczanej wraz ze środowiskiem. Przygotowuje ona środowisko pracy dla programów napisanych w C, a następnie wywołuje procedurę *main*. Procedury *SystemInit* i *main* muszą zostać dostarczone przez programistę. Można również skorzystać modułu z procedurą *SystemInit* dostarczonego przez producenta mikrokontrolera.

Konfiguracja projektu

Przed rozpoczęciem pisania programu warto ustawić podstawowe parametry konfiguracji projektu. W wypadku prostych projektów możemy wszystkie pliki źródłowe przechowywać w głównym folderze projektu. Nie ma jednak sensu zaśmiecać tego foldera plikami roboczymi, postaciami pośrednimi i innymi plikami tworzonymi podczas kompilacji i konsolidacji, dlatego warto na wstępie zmienić kilka opcji ustawień projektu.

Zaczynamy od zmiany domyślnej nazwy nazwy generowanej wersji projektu z domyślnej Target 1 na nazwę projektu – w naszym wypadku *Empty*. Możemy to zrobić klikając powoli dwukrotnie na nazwie *Target 1* w panelu z lewej strony okna i wpisując nową nazwę. Następnie klikamy prawym przyciskiem myszy na folderze *Empty* i z menu kontekstowego wybieramy pierwszą opcję – *Options for target ‚Empty'*. Mamy tu do wykonania kilka zmian:

- W zakładce *Output* wybieramy *Select Folder for Objects*, w otwartym dialogu tworzymy folder o nazwie *Obj* i wybieramy do składowania plików pośrednich.
- W zakładce *Listing* wybieramy *Select Folder for Listings*, w otwartym dialogu tworzymy folder o nazwie *Lst* i wybieramy do składowania plików raportów.
- W zakładce C/C++ wybieramy poziom optymalizacji kodu 2.

Tworzenie programu

Wybieramy opcję *File* → *New* i piszemy nasz pierwszy, niemal pusty program w języku C (**listing 1**). Program zawiera dyrektywę preprocesora włączającą plik nagłówkowy z definicją zasobów mikrokontrolera, która w pustym programie jest zbędna oraz dwie puste procedury – *SystemInit* i *main*.

Po napisaniu programu zapamiętujemy go w folderze projektu pod nazwą *main.c*, a następnie dodajemy plik programu do projektu korzystając z menu kontekstowego folderu *Source Group 1* – opcja *Add Files to Source Group 1*.

Gotowy projekt przygotowany według powyższego opisu można znaleźć w folderze *Empty* powstałym po rozpakowaniu pliku *F0Tutorial.zip*.

Następnie z paska narzędzi wybieramy *Build*, co spowoduje skompilowanie i konsolidację projektu. Raport z tych czynności możemy zaobserwować w dolnym panelu okna. Po usunięciu przyczyn ewentualnych błędów i ostrzeżeń sprawdzamy rozmiar wygenerowanego obrazu raportowany przez konsolidator w dolnym panelu okna. Nasz pusty program zajmuje po kilkaset bajtów kodu i stałych danych. Stałe (sekcja *RO-data*) – zawiera tablicę adresów procedur obsługi wyjątków zawarta w module startowym oraz dane o pozostałych sekcjach, z których korzysta procedura *__main* podczas inicjowania pamięci RAM. W procesorze Cortex-M0 tablica wyjątków zajmuje 192 bajty. W sekcji stałych są również zapisane początkowe wartości danych statycznych inicjowanych wartościami różnymi od zera, o ile takie dane wstępują w programie. Większość sekcji kodu zajmuje procedura *__main* i inne procedury pochodzące z biblioteki kompilatora. Procedury *ResetHandler*, *SystemInit* i *main* zajmują łącznie zaledwie kilkadziesiąt bajtów.

Dwie pierwsze sekcje (*Code* i *RO-data*) zajmują pamięć Flash mikrokontrolera. Pozostałe dwie sekcje – dane statyczne o niezerowych wartościach początkowych (*RW-data*) oraz pozostałe dane statyczne wraz ze stosem i stertą (sekcja *ZI-data*) zajmują jedynie pamięć RAM.

Sekcja danych statycznych jest pusta, gdyż nasz program nie definiuje żadnych danych. Ostatnia sekcja

oprócz stosu (domyślnie 1024 B) i sterty (domyślnie 512 B) zawiera również obszar roboczy dla biblioteki (96 B). Łączny rozmiar tej sekcji przekracza 1,5 kB, może on jednak zostać zmieniony poprzez konfigurację modułu startowego.

Przykłady dla płytki STM32F0DISCOVERY bez dodatkowych układów

Kolejny program będzie tradycyjnym pierwszym programem dla nowego mikrokontrolera – będzie on naprzemiennie zaświecał i gasił dwie diody świecące dostępne na płytce STM32F0DISCOVERY. Gotowy program znajduje się w projekcie *blink0*.

Tworzenie programu

Tworzymy projekt i plik programu w taki sam sposób, jak w poprzednik przykładzie, pamiętając o zmianie domyślnych nazw oraz konfiguracji opcji projektu. W pliku programu przed funkcjami umieszczamy definicje połączeń LED i przycisku, określające nazwy portów i pozycje linii portów, do których są podłączone te elementy na płytce DISCOVERY. Są to linie 8 i 9 portu GPIOC.

Po zainicjowaniu mikrokontroler pracuje z wewnętrznym generatorem przebiegu zegarowego o częstotliwości 8 MHz. Ponieważ nie ma powodu, by zmieniać źródło przebiegu ani jego częstotliwość na potrzeby pierwszych, prostych programów, funkcja *SystemInit()* może pozostać pusta. Napiszemy natomiast funkcję *main()*, która będzie cyklicznie zaświecała i gasiła diody. Wewnątrz funkcji deklarujemy zmienną, która posłuży do realizacji programowej pętli opóźnienia. Zmienna ma typ *uint32_t* – jest to 32-bitowa zmienna całkowita bez znaku. Użycie typów o jawnie określonych rozmiarach, zdefiniowanych w pliku nagłówkowym *stdint.h* (włączanym pośrednio przez plik *stm32f0xx.h*), podnosi czytelność i przenośność programu, dlatego będziemy ich używali w kolejnych przykładach.

W celu przygotowania portu do pracy wykonujemy kolejno następujące czynności:

- włączamy moduł GPIOC poprzez ustawienie bitu w rejestrze AHBENR,
- ustawiamy linie portu GPIOC odpowiadające LED jako wyjścia,
- zaświecamy jedną z diod.

Po zainicjowaniu portu program wchodzi w pętlę nieskończoną, w której następują dwie akcje:

- opóźnienie zrealizowane jako prosta pętla programowa,
- zmiana stanu obu wyjść sterujących diodami poprzez zanegowanie bitów portu, które im odpowiadają.

W ten sposób co określony czas dioda dotychczas świecąca zgaśnie, a ta, która była wygaszona – zaświeci się. Gotowy tekst programu jest przedstawiony na **listingu 2**.

Programiści znający mikrokontrolery 8-bitowe mogą zauważyć, że zaprezentowany program różni się od analogicznych programów dla układów 8-bitowych jednym istotnym szczegółem – włączeniem portu GPIOC przed jego konfiguracją. To typowa cecha wszystkich peryferiali w mikrokontrolerach z rdzeniami Cortex – przed użyciem modułu należy go uaktywnić. Jest to również najczęstsze źródło błędów w programach pisanych przez początkujących. Jeżeli programista nie włączy modułu, odwołania do niego nie będą miały żadnych skutków albo będą powodowały błędy podczas wykonania programu.

Mikrokontrolery z rdzeniami Cortex mają zwykle wiele modułów peryferyjnych, a każdy z nich jest widoczny dla oprogramowania w postaci wielu rejestrów sterujących. Pliki nagłówkowe dostarczane przez producentów mikrokontrolerów najczęściej definiują moduły peryferyjne jako struktury języka C, a poszczególne rejestry modułów – jako pola tych struktur. Przy odwoływaniu się do rejestrów używamy adresu modułu zdefiniowanego jako stałej typu wskaźnikowego, wskazującej strukturę zawierającą rejestry danego modułu, operatora „->” i nazwy rejestru.

Plik nagłówkowy *stm32f0xx.h* definiuje zarówno nazwy rejestrów jak i nazwy pól (w tym pojedynczych bitów) tych rejestrów. Brakuje w nim jednak definicji symboli odpowiadających możliwym wartościom pól o szerokości powyżej jednego bitu, dlatego potrzebne w programie wartości pól bitowych będziemy definiować jako własne symbole preprocesora. W praktyce warto byłoby stworzyć własny plik nagłówkowy zawierający takie definicje, z którego można będzie korzystać w wielu projektach, jednak w pierwszych projektach będziemy umieszczać te definicje w głównym pliku programu, dla zwiększenia czytelności kodu.

Aby zrozumieć znaczenie poszczególnych instrukcji programu należy zajrzeć do dokumentacji mikrokontrolera. Rejestr AHBENR w module RCC służy do włączania niektórych peryferiali, w tym portów GPIO. Każdy port GPIO zawiera wiele rejestrów sterujących. W naszym

Listing 2. Program naprzemiennie zaświecający diody LED

```
/*
  STM32F0DISCOVERY
  Trivial blinker - very first project
  gbm, 12'2012
*/

#include „stm32f0xx.h”
//================================================================
// defs for STM32F05x chips
#define GPIO_MODER_OUT     1
//================================================================
// defs for STM32F0DISCOVERY board
#define LED_PORT           GPIOC
#define BLUE_LED_BIT       8
#define GREEN_LED_BIT      9
//================================================================
void SystemInit(void)
{
}
//================================================================
int main(void)
{
  uint32_t i;
// port setup
  RCC->AHBENR = RCC_AHBENR_GPIOCEN;        // turn on GPIOC
  LED_PORT->MODER = GPIO_MODER_OUT << (GREEN_LED_BIT << 1)
  | GPIO_MODER_OUT << (BLUE_LED_BIT << 1);  // LED pins as outputs
  LED_PORT->ODR = 1 << GREEN_LED_BIT;  // green initially ON

  for (;;)
  {
    for (i = 0; i < 1000000; i ++);        // quick-and-dirty delay
// toggle both LEDs
    LED_PORT->ODR ^= 1 << GREEN_LED_BIT | 1 << BLUE_LED_BIT;
  }
}
```

przykładzie użyliśmy dwóch z nich: MODER i ODR. Rejestr MODER służy do ustawiania trybu pracy linii portu. Każdej z 16 linii portu odpowiadają dwa kolejne bity rejestru MODER, a ustawienie tych bitów na wartość binarną 01 powoduje zaprogramowanie linii portu jako wyjścia. Nasz program zawiera definicję tej wartości pod nazwą *GPIO_MODER_OUT*. Przy zapisie do rejestru *MODER* wartość ta musi być przesunięta w lewo o liczbę pozycji równą podwojonemu numerowi bitu portu.

Rejestr *ODR* zawiera wartości wyjść. Ponieważ diody są włączone pomiędzy wyjścia mikrokontrolera i masę, ich zaświecenie wymaga wysterowania wyjścia w stan wysoki. Operacja różnicy symetrycznej wykonywana w pętli głównej powoduje zanegowanie stanu obu wyjść sterujących diodami i – w konsekwencji - zmianę stanu obu diod.

Kompilowanie i uruchomienie programu

Podobnie jak poprzednio, do kompilacji programu użyjemy przycisku Build. Możemy zauważyć, że w porównaniu z programem pustym nasz program jest dłuższy o ok. 50 bajtów – tyle zajmuje kod napisany przez programistę wraz z użytymi w nim wartościami adresów i stałych.

Po pomyślnym skompilowaniu programu należy przygotować środowisko do uruchomienia programu. Czynność tę należy wykonać przy tworzeniu każdego projektów, gdyż konfiguracja dotycząca programowania pamięci i uruchamiania programu jest zapamiętywana w pliku projektu.

Po dołączeniu płytki STM32F0DISCOVERY i rozpoznaniu jej przez system z menu Flash wybieramy opcję *Configure Flash Tools*. W zakładce *Utilities* wybieramy moduł debugowania *ST-Link Debugger*, a następnie otwieramy dialog konfiguracji, naciskając przycisk *Settings*. W dialogu ustawień zaznaczamy opcje: *Erase Sectors*, *Program*, *Verify* i *Reset and Run*, a następnie dodajemy algorytm programowania STM32F05x Flash. Zamykamy dialog ustawień, przechodzimy do zakładki Debug w dialogu konfiguracji, wybieramy również *ST-Link Debugger* i zaznaczamy opcję *Use*.

Po skonfigurowaniu środowiska do pracy z modułem ST-Link możemy zaprogramować mikrokontroler przy użyciu przycisku LOAD umieszczonego na pasku narzędzi. Zaraz po zaprogramowaniu program rozpocznie działanie – diody zaczną naprzemiennie migać. Częstotliwość migania jest określona przez czas wykonania pętli opóźniającej i wynika m.in. z wybranego w ustawieniach kompilatora poziomu optymalizacji kodu wynikowego. Oczywiście nie jest to poprawny sposób odmierzania czasu – ten problem rozwiążemy w następnym przykładzie.

Klonowanie projektu

Jeżeli nasz kolejny projekt bazuje na innym, już uruchomionym, zamiast tworzyć projekt od nowa możemy sklonować istniejący projekt, oszczędzając czas spędzony na konfiguracji opcji i tworzeniu plików.

W celu sklonowania istniejącego projektu:

- Zamykamy środowisko Keil MDK-ARM.
- Kopiujemy folder istniejącego projektu.
- Nadajemy nowej kopii foldera projektu nazwę nowego projektu.
- Z nowego foldera projektu usuwamy foldery Obj i Lst oraz wszystkie pliki oprócz plików źródłowych (w naszym przypadku *startup_stm32f0xx.s* i *main.c*) oraz plików z rozszerzeniami .uvproj i .uvopt.
- Zmieniamy nazwy plików projektu z rozszerzeniami .uvproj i .uvopt, nadając im nazwę nowego projektu i pozostawiając ich rozszerzenia.
- Uruchamiamy środowisko klikając na pliku z rozszerzeniem .uvproj.
- W opcjach projektu, w zakładce *Output*, zmieniamy nazwę pliku wynikowego na nazwę nowego projektu.
- Po zamknięciu projektu i środowiska usuwamy z folderów projektu pliki o nazwach odpowiadających nazwie poprzedniego projektu. Po kolejnym otwarciu projektu środowisko stworzy już pliki o nazwach odpowiadających nowemu projektowi.

Kolejne przykładowe projekty możemy tworzyć przez klonowanie wcześniejszych projektów.

Grzegorz Mazur
gbm@ii.pw.edu.pl