Bootloader dla mikrokontrolerów STM32

Aktualizacja oprogramowanie z zastosowaniem karty SD lub przez USB

Nie jest dla nikogo tajemnicą, że mikrokontroler można zaprogramować z użyciem programatora. Współczesne mikrokontrolery mogą być programowane zarówno za pomocą programatora, jak i w systemie. Niżej zaprezentujemy metodę będącą wariantem programowania w systemie, a mianowicie aktualizację oprogramowania przez interfejs USB lub z zastosowaniem karty SD podłączonej poprzez SPI. Atrakcyjna jest zwłaszcza ta druga metoda, ponieważ wymaga tylko maleńkiej karty SD, którą można zabrać ze sobą chociażby do kieszeni.

Większość obecnie produkowanych mikrokontrolerów ma pamięć FLASH, która może być modyfikowana wiele razy. Dodatkowo, producenci wychodząc naprzeciw użytkownikom udostępniają programowe interfejsy pozwalające na dostęp do tychże pamięci z poziomu aplikacji, przez co można je modyfikować bez konieczności użycia programatora. Nawet jeśli programator nie jest drogim urządzeniem, to sama konieczność rozebrania obudowy może być dość uciążliwa. Najczęściej stosowanym współcześnie rozwiązaniem jest programowanie w systemie, gdyż podłączając się np. przez często używany przez nasze urządzenie interfejs dokonujemy szybko i prosto operacji serwisowych.

Programując w ten sposób mikrokontrolery STM32 zauważyłem możliwość zrobienia uniwersalnej aplikacji, która umieszczona obok docelowego programu będzie służyć łatwej wymianie oprogramowania.

Pytaniem jest, jaki interfejs najlepiej nadaje się do tego celu? Najczęściej nowe mikrokontrolery wyposażone są w interfejs UART, SPI, I²C. UART nadawałby się świetnie do komunikacji z komputerem, jednak nowe komputery nie mają już interfejsu RS232, z którym UART łączy się przez układ pośredniczący.

Z drugiej strony wiele urządzeń ma możliwość rejestrowania wyników swojej pracy na karcie SD. Najprostszym sposobem jej podłączenia jest port SPI, który jest wspierany przez część kart. Kolejnym kandydatem, jest interfejs USB służący do połączenia urządzenia z komputerem PC, który wypiera w tej roli UART. Dodatkowo, nowe mikrokontrolery mają często standardowo wbudowany kontroler USB w swoją strukturę.

Z wymienionych wyżej powodów, zdecydowałem się na użycie tych dwóch interfejsów, czyli USB i SD (SPI) jako dwie metody uaktualnienia oprogramowania w procesorze rodziny STM32.

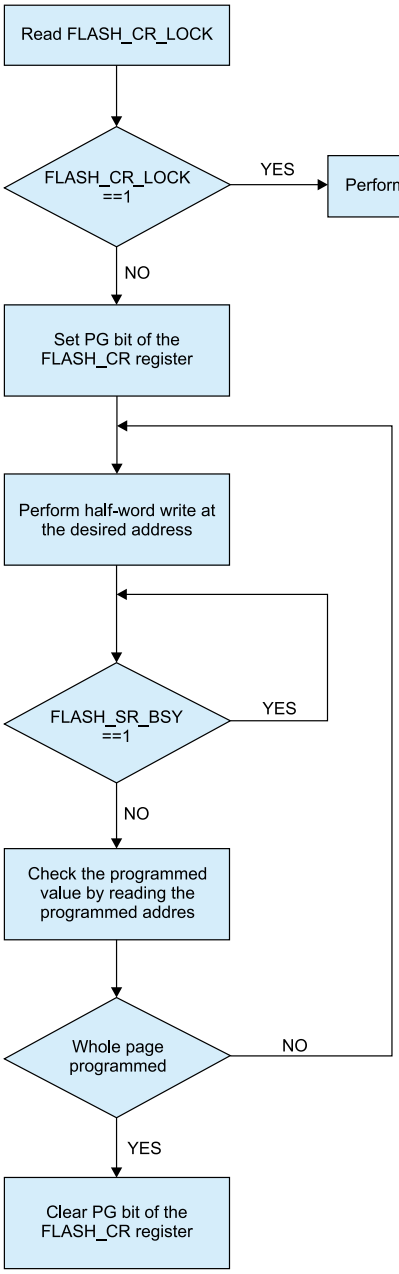
Bootloader

Bootloader jest małym programem umieszczonym obok programu użytkownika. Jego zadaniem jest pobranie przez dedykowany interfejs nowszej wersji aplikacji użytkownika. Umieszczono go na początku wbudowanej w procesor pamięci FLASH, tak aby był domyślnie uruchamiany po włączeniu mikrokontrolera. Dzięki temu jest zawsze uruchamiany przed aplikacją użytkownika, przez co może wystartować nawet wtedy, gdy aplikacja użytkownika jest w jakiś sposób uszkodzona lub wadliwa. Po uruchomieniu sprawdza się, czy bootloader ma przejść do aktualizacji. Jeżeli nie, to wyłącza on wszystkie uaktywnione interfejsy procesora i wywołuje aplikację użytkownika.

W artykule opisywane są dwie wersje zbudowane na szkielecie bootloadera dedykowanego do aplikacji przemysłowych, pracującego z interfejsem USART, z którego wykorzystano jedynie pliki startowe. Dostęp do karty SD i system plików FAT w wersji tylko do odczytu zostały napisane na od podstaw. Są one na tyle uniwersalne, że z łatwością można je stosować w różnych urządzeniach. Oprogramowanie obsługujące USB, powstało trochę wcześniej. Jest ono na tyle uniwersalne i łatwe w konfiguracji, że w prosty sposób można je przenosić na różne procesory. Do obsługi standardowych peryferiów (z wyjątkiem USB) zastosowano zmodyfikowane w celu zmniejszenia kodu wynikowego biblioteki producenta.

Jak wspomniano wcześniej, sam bootloader jest umieszczony w pamięci FLASH, a dokładniej na jej początku. Rozwiązanie to jest konieczne, gdyż po restarcie procesor musi mieć możliwość uruchomienia go, niezależnie od tego, czy jest w nim poprawny program, czy nie. Jako że po restarcie procesor skacze do początku pamięci FLASH (przy odpowiednim ustawieniu linii BOOT0 i BOOT1) dlatego najłatwiej było umieścić bootloader na początku tego obszaru. Daje to nam również dużą uniwersalność, gdyż nie trzeba się już martwić wgraniem go w odpowiednie miejsce – ładujemy go raz, tak jak zwykły program, za pomocą programatora JTAG lub przez USART1. Bootloader po starcie procesora uruchamia się z pamięci FLASH, ale zaraz na początku swojego działania kopiuje samego siebie do pamięci SRAM, a następnie skacze do swojej „kopii" i od tego momentu działa już z poziomu pamięci RAM. Rozwiązanie to jest konieczne ze względu na niemożność równoczesnego czytania i programowania pamięci FLASH.

Następnie wykonywana jest inicjalizacja niezbędnych peryferiów i sprawdzenie, czy bootloader ma być wywołany. Dzięki temu rozwiązaniu nie dokonujemy za każdym razem aktualizacji oprogramowania, lecz ro-



Rys. 1. Procedura programowania pamięci FLASH mikrokontrolera STM32

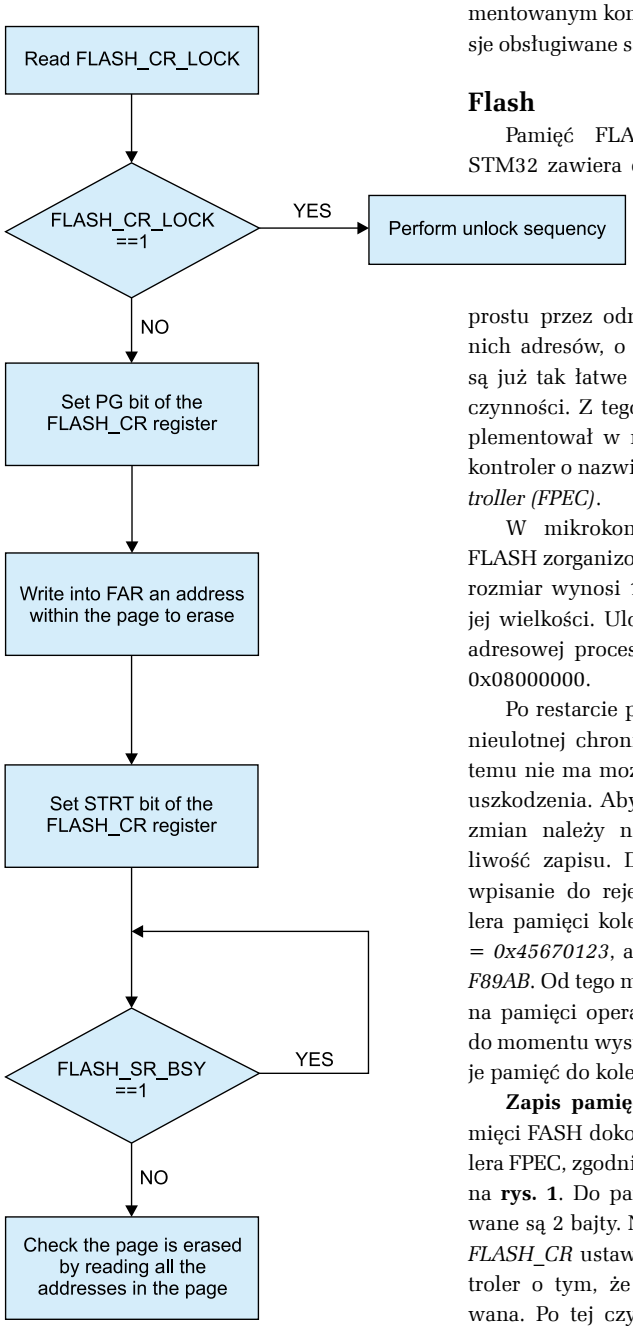
bimy to tylko w ściśle określonych przypadkach. Są to:

- Brak jakiegokolwiek oprogramowania użytkownika w pamięci FLASH.
- Spełnienie warunku wymuszenia przejścia w tryb bootloadera.
- Bootloader został wywołany z poziomu aplikacji użytkownika.

Sprawdzenie pierwszego warunku polega na odczytania pierwszych komórek wektora przerwań programu użytkownika (wskaźnik stosu i wektor *RESET*) i sprawdzenia czy nie są przypadkiem skasowane. Ten prosty test oczywiście nie jest w stanie wykryć, czy aplikacja nie zawiera błędów, niemniej jednak wystarcza do stwierdzenia, czy w ogóle znajduje się ona w pamięci.

Drugi sposób realizowany jest przez sprawdzenie stanu jednego z pinów GPIO procesora, tzn. czy podłączony tam przycisk jest wciśnięty. Używany jest do tego, aby można było zaktualizować program po starcie urządzenia niezależnie od tego, czy w pamięci znajduje się już jakiś program, czy nie. Jest to pomocne, gdy aplikacja użytkownika zawiera błędy i program zawiesza się tuż po starcie.

Trzeci sposób pozwala na umieszczenie w programie użytkownika dosłownie kilku linii kodu, które powodują wywołanie bootloadera. Jest to możliwe na dwa sposoby. Pierwszy, to zapisanie do jednego z rejestrów *Backup* procesora konkretnej wartości i restart układu. Po nim bootloader, podczas sprawdzania warunków wejścia w tryb aktualizacji, sprawdza zawartość pierwszego z rejestrów *Backup*. Jeżeli jego zawartość jest odpowiednia, wówczas kasuje ten rejestr i przechodzi do wymiany programu. W tym rozwiązaniu używany jest jeden z rejestrów *Backup*, dlatego jeżeli użytkownikowi bardzo zależy na zastosowaniu go do innego przeznaczenia, to wówczas może użyć drugiej metody – polega ona na skoku do funkcji, której adres znajduje się w wektorze przerwań samego bootloadera na pozycji 12 (SVC_Handler). Pod tym wskaźnikiem kryje się adres kopii wektora *RESET*, którego wywołanie powoduje natychmiastowe przejście do aktualizacji, bez sprawdzania jakichkolwiek warunków.



Rys. 2. Procedura kasowania pamięci FLASH mikrokontrolera STM32 metodą „strona po stronie"

Bootloader w wersji SD działa na wszystkich wersjach STM32, ponieważ potrzebuje do prawidłowej pracy jedynie jednego interfejsu SPI i jednej linii GPIO. Co do USB, to w grę wchodzą jedynie modele z zaimplementowanym kontrolerem, gdyż tylko te wersje obsługiwane są przez oprogramowanie.

Flash

Pamięć FLASH, w mikrokontrolerze STM32 zawiera dedykowany kontroler ułatwiający do niej dostęp od strony oprogramowania. O ile odczyt pamięci można realizować po prostu przez odnoszenie się do odpowiednich adresów, o tyle kasowanie i zapis nie są już tak łatwe i wymagają odpowiednich czynności. Z tego powodu producent zaimplementował w mikrokontrolerze specjalny kontroler o nazwie *Flash Program/Erase Controller (FPEC)*.

W mikrokontrolerach STM32 pamięć FLASH zorganizowana jest w strony, których rozmiar wynosi 1 kB lub 2 kB, zależnie od jej wielkości. Ulokowana jest w przestrzeni adresowej procesora, począwszy od adresu 0x08000000.

Po restarcie procesora kontroler pamięci nieulotnej chroni ją przed zapisem. Dzięki temu nie ma możliwości przypadkowego jej uszkodzenia. Aby dokonywać jakichkolwiek zmian należy najpierw odblokować możliwość zapisu. Dokonuje się tego poprzez wpisanie do rejestru *FLASH_KEYR* kontrolera pamięci kolejno dwóch wartości: *KEY1 = 0x45670123*, a następnie *KEY2 = 0xCDEF89AB*. Od tego momentu można dokonywać na pamięci operacji kasowania i zapisu, aż do momentu wystąpienia błędu, który blokuje pamięć do kolejnego restartu.

Zapis pamięci. Operacji zapisu do pamięci FASH dokonuje się za pomocą kontrolera FPEC, zgodnie z algorytmem pokazanym na **rys. 1**. Do pamięci jednocześnie zapisywane są 2 bajty. Najpierw należy w rejestrze *FLASH_CR* ustawić bit *PG* informujący kontroler o tym, że pamięć będzie programowana. Po tej czynności można dokonywać zapisu – wpisujemy 2 bajty nowej zawartości bezpośrednio pod adres, pod którym ma być ona umieszczona. Następnie czekamy

za zakończenie operacji, po czym sprawdzamy, czy zapis został dokonany poprawnie. Przy braku jakichkolwiek błędów, możemy przejść do zapisu kolejnego słowa. Po zapisaniu całej strony kasujemy bit PG.

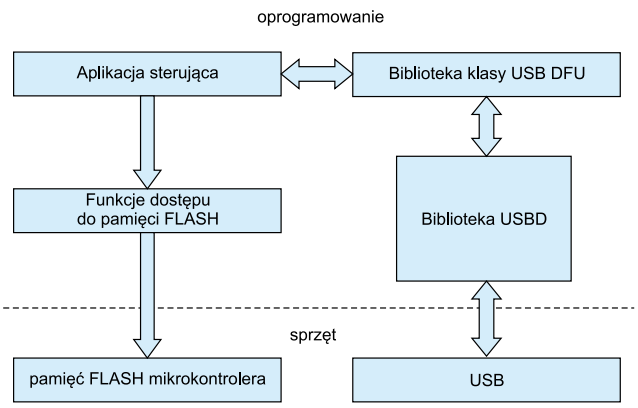
Kasowanie Pamięci. Kasowanie pamięci Flash w mikrokontrolerze STM32 może być wykonane na dwa sposoby: strona po stronie lub całość pamięci. My oczywiście używamy tylko kasowania metody „strona po stronie", gdyż w pamięci znajduje się nasz bootloader, który nie wolno usuwać. Procedurę kasowania strona po stronie przedstawiono na **rys. 2**.

Operację kasowania rozpoczynamy od ustawienia w rejestrze *FLASH_CR* bitu *PER* informującego kontroler, że będziemy dokonywać operacji kasowania strony (*Page Erase*). Następnie do rejestru *FLASH_AR* wpisujemy adres strony, którą kasujemy i ustawiamy w rejestrze *FLASH_CR* bit *STRT*. W tym momencie rozpoczyna się procedura kasowania. W kolejnym kroku czekamy na jej zakończenie, po czym sprawdzamy, czy strona została skasowana poprawnie poprzez odczytanie jej zawartości.

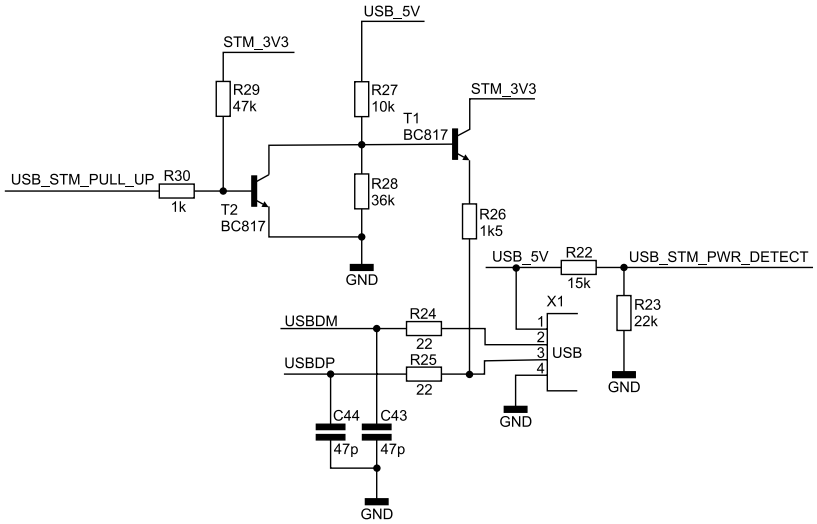
Bootloader USB

Budowę aplikacji bootloadera pracującego z portem USB przedstawiono na **rys. 3**. Rozwiązanie to bazuje na wbudowanym w część mikrokontrolerów rodziny STM32 sprzętowym kontrolerze USB Full-Speed Device. Jego główne cechy to:

- Pełna zgodność ze standardem 2.0 full-speed.
- Do 8 konfigurowalnych punktów końcowych.
- Pełne sprzętowe kodowanie/dekodowanie ramek USB (CRC, NRZI, bit-stuffing itp.).
- Wsparcie w każdym punkcie końcowym dla wszystkich typów transmisji (kontrolna, przerwaniowa, izochroniczna i masowa).
- Wsparcie sprzętowe dla przechodzenia w tryb uśpienia (Suspend) oraz wybudzenia z tego trybu (Resume).
- 512 B przestrzeni pamięci RAM dedykowanej na bufory sprzętowe.



Rys. 3. Schemat blokowy bootloadera USB



Rys. 4. Podłączenie portu USB

– 3 wektory przerwań skojarzone z kontrolerem przerwań NVIC: transmisja o niskim priorytecie, transmisja o wysokim priorytecie, wyjście procesora z trybu Suspend.

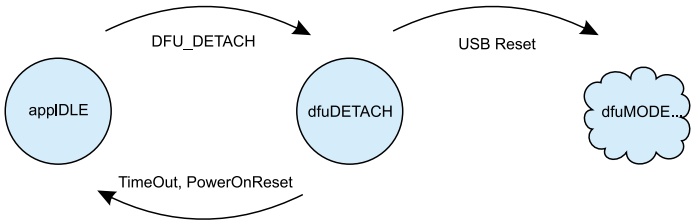
Do obsługi kontrolera zastosowano uniwersalną warstwę USB. Pozwala ona w łatwy sposób pisać programy korzystające z tego interfejsu prawie niezależnie od zastosowanego układu peryferyjnego (procesora). Na **rys. 4** pokazano przykładowe podłączenie złącza USB-B do mikrokontrolera. Poza liniami D+ i D–, do procesora podłączone są również dwie dodatkowe linie. Pierwsza z nich to *USB_STM_PWR_DETECT* służąca do wykrywania napięcia zasilającego docierającego bezpośrednio z komputera PC i przesyłanego po kablu USB. Fakt wykrycia podłączenia do Hosta zgłasza się stanem wysokim, zaś sama linia skonfigurowana jest w procesorze jako przerwanie zewnętrzne. Druga to *USB_STM_PULL_UP* wykorzystywana do programowego włączania lub wyłączania rezystora 1,5 kΩ na linii D+. Sygnały USBDM i USBDP podłączamy do dedykowanych linii mikrokontrolera, natomiast pozostałe dwa do dowolnych linii GPIO.

Do transmisji z komputerem potrzebne jest jeszcze zdefiniowanie wyższej warstwy transmisji. Można to zrobić na dwa sposoby: zdefiniowanie własnego protokołu wymiany kodu lub użycie gotowego. Z tych dwóch jedynie drugie wydaje się być sensowne, zwłaszcza że taki protokół został już kiedyś zdefiniowany i to oficjalnie jako jedna ze standardowych klas urządzeń USB. Mowa tu o klasie DFU (ang. Device Firmware Upgrade), której dokumentację można pobrać ze strony *www.usb.org*.

USB DFU. Klasa DFU pozwala na aktualizowanie oprogramowanie urządzenia bezpośrednio za pomocą interfejsu USB. Definiuje dwa tryby pracy: tryb pracy normalnej – użytkownika (Run Time Mode) i tryb DFU (DFU Mode).

W trybie pierwszym urządzenie USB działa normalnie, zgodnie z założeniami, z jakimi zostało zaprojektowane, np. jest to drukarka, skaner, konwerter USB/RS232, pamięć masowa itp. Poza tymi standardowymi możliwościami ma ono jeden dodatkowy interfejs USB – interfejs DFU, który nie ma żadnych punktów końcowych. Tworząc taki dodatkowy interfejs dodajemy do deskryptora konfiguracji dwa dodatkowe deskryptory: interfejsu i funkcjonalny DFU (*DFU Functional Descriptor*). Host komunikuje się z tym interfejsem za pomocą transmisji kontrolnej i robi to tylko w jednym celu – aby przełączyć się w tryb drugi. W tym celu wysyła do interfejsu DFU żądanie *DFU_DETACH*. Po tym żądaniu urządzenie przełącza się w tryb DFU na jeden z dwóch sposobów. Jeżeli ma ustawiony bit 3 w polu *bmAttributes* w deskryptorze funkcjonalnym DFU (*bitWillDetach*), to wówczas wykonuje sam cykl odłączenia od Hosta i połączenia, tym razem już z konfiguracją w trybie DFU. Jeżeli natomiast nie, wówczas rozpoczyna zliczanie czasu, aż dojdzie do wartości równej zawartości pola *wValue*, jednej ze składowych żądania *DFU_DETACH*. Jeżeli otrzyma w tym czasie od Hosta sygnał zerowania, wówczas zmienia konfigurację na DFU, w przeciwnym razie pozostaje w normalnym trybie pracy. Ten drugi sposób przełączenia pokazano na **rys. 5** zaczerpniętym z dokumentacji DFU.

W trybie DFU urządzenie zawiera jedną konfigurację, tylko z jednym interfejsem –

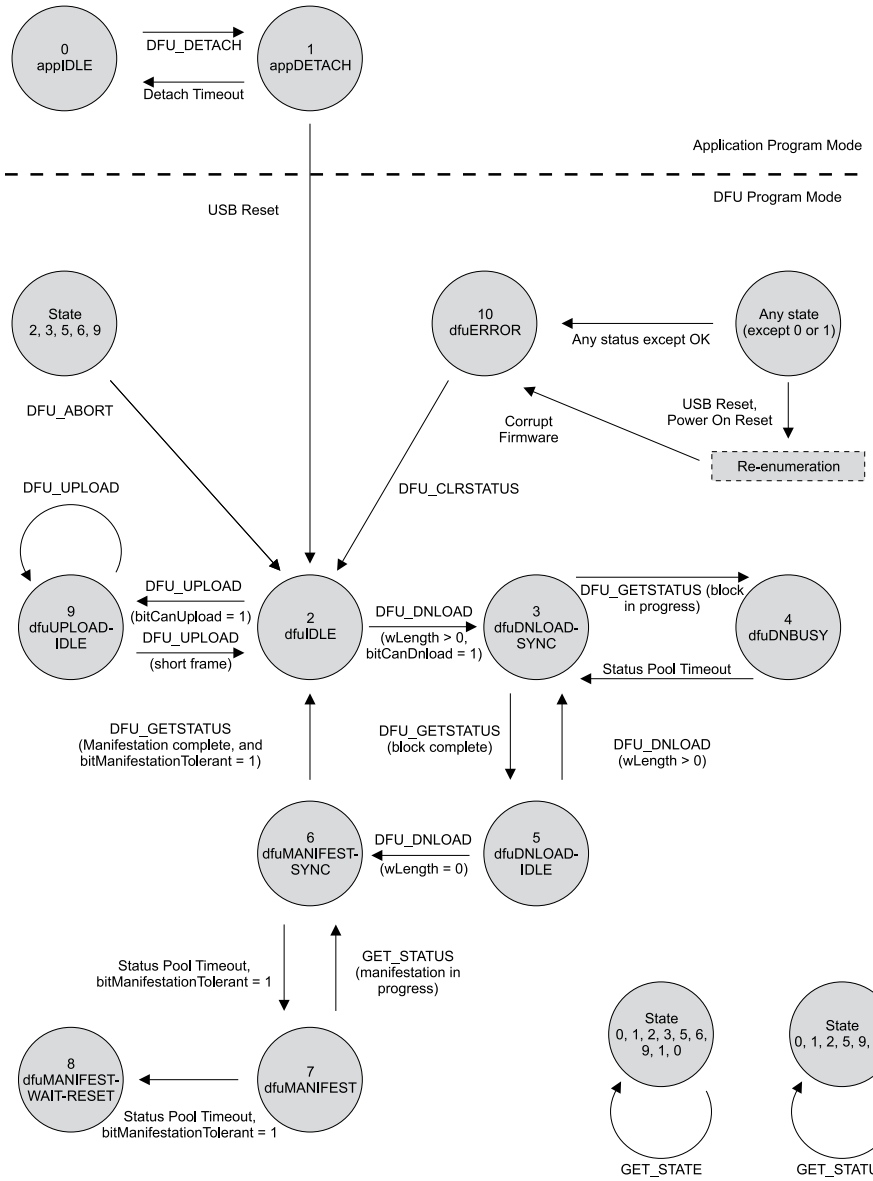


Rys. 5. Procedura przejścia urządzenia USB w tryb DFU przy wykorzystaniu resetu otrzymanego od Hosta

DFU. Możliwe są alternatywne ustawienia dla tego interfejsu np. w przypadku, gdy mamy kilka różnych pamięci do zaprogramowania w układzie. Wówczas dla każdej robimy oddzielną wersję interfejsu. Interfejs ten oraz wszystkie alternatywne ustawienia, jeżeli istnieją, muszą zawierać dwa deskryptory opisujące go i są to te same deskryptory jak w przypadku pierwszego trybu pracy urządzenia – deskryptor funkcjonalny jest identyczny, deskryptor interfejsu ma drobne różnice na polach numeru interfejsu, numeru alternatywnego ustawienia i kodu protokołu (*bInterfaceProtocol*).

Działanie w trybie DFU opiera się również na żądaniach transmisji kontrolnej, niemniej jednak tu sekwencja żądań jest trochę bardziej skomplikowana, co przedstawiono na **rys. 6**.

Pobranie od Hosta nowej wersji oprogramowania określa procedura *Dnload*. Rozpoczyna się ona w stanie *dfuIDLE*. Host przesyła do układu nowe oprogramowanie w postaci paczek danych za pomocą żądania *DFU_DNLOAD*. Rozmiar tych danych mieści się między wartością rozmiaru bufora zerowego punktu końcowego (*bMaxPacketSize0*) a maksymalnym rozmiarem paczki (parametr *wTransferSize* zawarty w deskryptorze funkcjonalnym DFU). W praktyce przesyłamy maksymalne paczki (*wTransferSize*), mniejszy rozmiar stosujemy w przypadku ostatniej paczki, gdy całkowity rozmiar danych nie jest wielokrotnością *wTransferSize*. Gdy przetransferujemy wszystkie dane, wówczas Host wysyła pakiet *DFU_DNLOAD* o rozmiarze 0, co oznacza, że transmisja danych została zakończona. Wtedy przechodzimy do kończenia programowania i uruchamiania nowego kodu.



Rys. 6. Automat stanów przedstawiający pracę urządzenia USB w trybie DFU

Do obsługi tego automatu napisana została specjalna biblioteka pozwalająca zarządzać strumieniem danych z nowym oprogramowaniem. Pozwala ona również na przełączanie alternatywnych ustawień interfejsów w celu programowania więcej niż jednej pamięci.

Aby otrzymywać dane od Hosta należy dodać do biblioteki DFU funkcję odpowiedzialną za przetworzenie otrzymanych danych. Przykładową procedurę obsługi umieszczono na **list. 1**.

Funkcja ta jest wywoływana zawsze w momencie skompletowania przez warstwę DFU kolejnego pakietu odebranego od Hosta. Pakiety te są zapisywane do bufora, który został podpięty wcześniej, w momencie inicjalizacji transmisji. W dokumentacji standardu DFU powiedziane jest jasno, że nie ma żadnych ograniczeń, czy zapisujemy otrzymane fragmenty nowego oprogramowania na bieżąco, czy też magazynujemy je np. w jakieś pamięci zewnętrznej by zaprogramować procesor dopiero po skompletowaniu całego pliku nowego oprogramowania. Ta druga metoda jest lepsza, gdy musimy mieć pewność, że cała aplikacja jest kompletna i poprawnie przesłana do układu. W naszym przypadku tak nie jest, gdyż nawet jeśli coś pójdzie nie tak, zawsze możemy rozpocząć procedurę programowania od nowa. Z tego powodu w powyższym kodzie programujemy pamięć strona po stronie. Po przesłaniu całego nowego oprogramowania odbieramy pakiet o długości 0, co jest sygnalizowane przez wywołanie naszej funkcji *boot_DNLOAD* z parametrem *length=0*, w celu zasygnalizowania tego faktu. Teraz Host inicjuje ostatnią fazę (*Manifest*), która to faza jest też obsługiwana automatycznie. Możemy oczywiście być o kolejnych krokach informowani, ale dla tego zastosowania protokołu DFU nie było to konieczne i ograniczyliśmy się jedynie do odebrania informacji o przejściu znów w tryb aplikacji. Wtedy następuje wywołanie nowo zapisanego programu.

Współpraca z systemem Windows

Niestety, w systemie Windows brak jest domyślnego sterownika dla urządzeń typu DFU, co zmusiło do jego napisania. Zawiera jedynie możliwość współpracy z urządzeniem posiadającym w trybie apli-

kacji interfejs DFU i w razie konieczności pracy z innymi interfejsami należy zmodyfikować driver. Do samego programowania procesora w zupełności ta opcja wystarcza. Do obsługi transmisji powstał program uruchamiany z wiersza poleceń. Nie są do niego przekazywane żadne parametry – jedynym warunkiem żeby zadziałał jest to, aby w tym samym katalogu co program znajdowały się: plik „file_name.txt", a w nim zapisana nazwa pliku binarnego, który program wysyła do układu oraz ów plik binarny. Tu nadmienić muszę jedną rzecz: protokół DFU definiuje standard pliku do wymiany danych. Plik taki zawiera specjalne informacje o programowanym układzie, wgrywanym programie oraz dane pozwalające na dodatkową detekcję błędów. Takie rozwiązanie wprowadza konieczność generowania jeszcze dodatkowego pliku. Z rozwiązania tego zrezygnowano, gdyż sam protokół USB ma dobrze zorganizowane rozpoznawanie błędów oraz mechanizm powtórzeń. Poza tym stworzony bootloader musi mieć mały rozmiar, przez co wprowadzanie dodatkowego sprawdzania sum kontrolnych CRC i innych elementów spowodowałoby zbyt duże powiększenie rozmiaru kodu. Program jak i sterownik na komputerze są przeźroczyste dla strumienia danych, dlatego nic nie stoi na przeszkodzie, żeby zamiast funkcji *boot_DNLOAD* do warstwy DFU podpiąć funkcję, która będzie oceniać otrzymany strumień danych pod kątem zgodności z DFU, a dopiero później wywoływać funkcje programujące procesor. Od strony komputera, nie jest ważne jaki plik wysyłamy do układu – to co podamy w pliku „file_name.txt" jest traktowane jako nazwa pliku, który będzie otwarty w trybie binarnym, a następnie w całości odczytany i wysłany.

Takie rozwiązanie można skrytykować i powiedzieć, że można zaprogramować procesor czymkolwiek – dowolnymi śmieciami – tak, zgadzam się z tym, ale ten bootloader miał być z założenia bardzo prosty w użyciu i służyć do szybkiego reprogramowania mikrokontrolera, dlatego od użytkownika wymagane jest to minimum uwagi, aby wiedział, co wysyła do układu. Poza tym, zaprogramowanie mikrokontrolera czymkolwiek spowoduje jedynie, że układ nie zadziała a aktualizację można będzie przeprowadzić ponownie, gdyż bootloader nigdy sam się nie skasuje.

List. 1.
```
static uint8_t boot_DNLOAD(uint8_t **buffer, uint16_t length, uint16_t packet_
number) {
  if(!length) {
    return(dfu_bStatus_OK);
  }

  boot_buffer_head_shift += length;
  if(boot_buffer_head_shift >= boot_page_size) {
    // get into critical section - disable interrupts
    __asm volatile („  CPSID  I");
    FlashUnlock();
    if (FlashErasePage(boot_prog_addr))
    // flash erase failed
    {
      FlashLock();
      // leave critical section - enable interrupts
      __asm volatile („  CPSIE  I");
      return(dfu_bStatus_errERASE);
    }
    if (FlashWritePage(boot_prog_addr, boot_buffer, boot_page_size))
    // flash write failed
    {
      FlashLock();
      // leave critical section - enable interrupts
      __asm volatile („  CPSIE  I");
      return(dfu_bStatus_errWRITE);
    }
    FlashLock();
    // leave critical section - enable interrupts
    __asm volatile („  CPSIE  I");
    boot_prog_addr += boot_page_size;
    boot_buffer_head_shift -= boot_page_size;
    *buffer = &boot_buffer[boot_buffer_head_shift];
    while(boot_buffer_head_shift) {
      boot_buffer_head_shift--;
      boot_buffer[boot_buffer_head_shift] = boot_buffer[boot_page_size + boot_
buffer_head_shift];
    }
  }
  else {
    *buffer = &boot_buffer[boot_buffer_head_shift];
  }
  return(dfu_bStatus_OK);
}
```

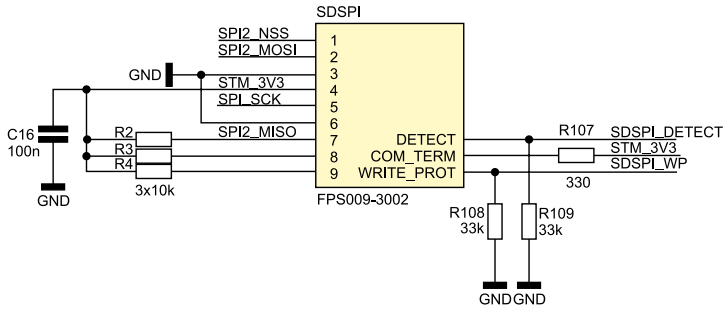
List. 2.
```
void FAT_ConnectEvent(FAT_Desc *FAT32D)
{
  FILE file_desc;
  uint32_t result;
  uint32_t address_shift = 0;
  if(!sfopen(&file_desc, (const char*)FAT32D->FAT_name_string_descriptor, „r"))
return;
  if(!sfopen(&file_desc, „stm32f10x/file_name.txt", „r")) return;
  fread(boot_read_buffer, 1, boot_page_size, &file_desc);
  fclose(&file_desc);
  if(!sfopen(&file_desc, (const char *)boot_read_buffer, „r")) return;
  FlashUnlock();
  do
  {
    if (FlashErasePage(DEF_APP_ADDRESS + address_shift)) break;
    result = fread(boot_read_buffer, 1, boot_page_size, &file_desc);
    if (FlashWritePage(DEF_APP_ADDRESS + address_shift, boot_read_buffer,
result)) break;
    address_shift += boot_page_size;
  } while(result == boot_page_size);
  FlashLock();
  fclose(&file_desc);
}
```

Bootloader SD

Jak wspomniano wcześniej, procesor komunikuje się z kartą SD przez interfejs SPI. Na **rys. 7** umieszczono schemat przykładowego podłączenie karty do procesora.

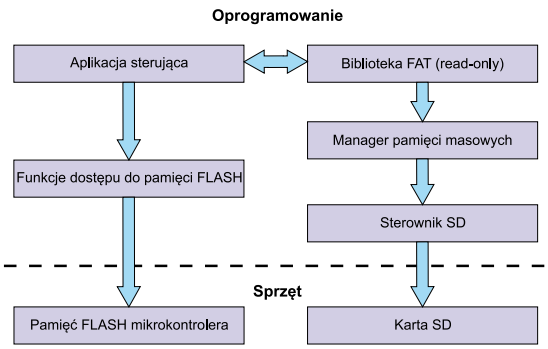
Do komunikacji używane są cztery linie SPI: SCK, MISO i MOSI, za pomocą których wymieniane są dane oraz czwarty sygnał NSS, działający jako chip select. Sygnały SDSPI_DETECT i SDSPI_WP nie są tu wykorzystywane. Służą one do detekcji umieszczenia karty w złączu i sprawdzenia, czy do karty można zapisywać dane. Schemat blokowy aplikacji bootloadera przedstawiono na **rys. 8**.

Sam dostęp do pamięci procesora i wywołania bootloadera realizowany jest w sposób identyczny jak w przypadku USB. Dostęp do karty realizuje biblioteka o budowie modułowej, dzięki czemu w razie potrzeby łatwo podmienić typ nośnika na inny niż SD.



Rys. 7. Podłączenie slotu karty SD do procesora

List. 3.
```
#ifndef BOOTLOADER_H_
#define BOOTLOADER_H_

// adres tablicy wektorow przerwan bootloadera
#define BOOTLOADER_VECTOR_ADDR      0x8000000

// definicje przeklejone z plikow bibliotecznych ST

/* --------- PWR registers bit address in the alias region ---------- */
#define PWR_OFFSET (PWR_BASE - PERIPH_BASE)

/* --- CR Register ---*/
/* Alias word address of DBP bit */
#define CR_OFFSET (PWR_OFFSET + 0x00)
#define DBP_BitNumber 0x08
#define CR_DBP_BB (PERIPH_BB_BASE + (CR_OFFSET * 32) + (DBP_BitNumber * 4))


// struktura opisujaca poczatek standardowego wektora przerwan
typedef struct {
  uint32_t SP;
  void (*RESET_ISR)(void);
  void (*NMIExc)(void);
  void (*HardFaultExc)(void);
  void (*MemManageExc)(void);
  void (*BusFaultExc)(void);
  void (*UsageFaultExc)(void);
  void (*RESRV1)(void);
  void (*RESRV2)(void);
  void (*RESRV3)(void);
  void (*RESRV4)(void);
  void (*SVC)(void);
}BOOTLOADER_CM3_ISR_TABLE;

// definicja wskaznika do tablice przerwan bootloadera
#define BOOTLOADER_TAB ((BOOTLOADER_CM3_ISR_TABLE*)BOOTLOADER_VECTOR_ADDR)

// makro sluzace do wywolania bootloadera przez skoczenie do niego
#define BOOTLOADER_CALL_BY_JUMP() (BOOTLOADER_TAB->SVC)()

// makro sluzace do wywolania bootloadera poprzez zapis sygnatury
// do pierwszego rejestru danych Backup i zresetowanie procesora
#define BOOTLOADER_CALL_BY_RESET() {\
  RCC->APB1ENR |= RCC_APB1Periph_BKP | RCC_APB1Periph_PWR;\
  RCC->APB1RSTR &= ~((uint32_t)(RCC_APB1Periph_BKP | RCC_APB1Periph_PWR));\
  *(vu32 *) CR_DBP_BB = (u32)1;\
  BKP->DR1 = 0x159D;\
  SCB->AIRCR = (SCB->AIRCR & 0xFFFF) | (0x5FA << 16) | (1 << 0);\
}

#endif /*BOOTLOADER_H_*/
```



Rys. 8. Schemat blokowy bootloadera SD

Manager pamięci masowych jest tak naprawdę zbiorem kilku prostych funkcji, które pozwalają w łatwy sposób kontrolować dostęp do nośników danych. Co do biblioteki FAT, daje ona jedynie możliwość odczytu plików, jednak ze wsparciem dla długich nazw. Napisana została ona od podstaw.

Uruchomienie bootloadera jest tu identyczne jak w przypadku USB. Aplikacja sterująca również działa bardzo podobnie. Pokazana na **list. 2** funkcja *FAT_ConnectEvent* jest związana z biblioteką FAT i wywoływana w momencie wykrycia partycji FAT16/32. W niej dokonujemy otwarcia katalogu głównego, następnie sprawdzamy czy na karcie istnieje katalog stm32f10x, a w nim plik „file_name.txt". Jeżeli tak, wówczas odczytujemy jego zawartość, którą traktujemy jako nazwę pliku binarnego znajdującego się w tym samym katalogu. Następnie otwieramy ten plik binarny i odczytujemy z niego paczki o wielkości rozmiaru strony pamięci FLASH mikrokontrolera, które następnie programujemy. Po odczytaniu całego pliku zamykamy go i kończymy pracę całej funkcji. Po wyjściu z niej następuje uruchomienie programu użytkownika.

Uruchomienie i aplikacja użytkownika

Sam bootloader wgrywamy do procesora tak jak każdy inny program, za pomocą interfejsu JTAG lub przez USART. Po tej czynności jest on od razu gotowy do pracy. Aby za jego pomocą poprawnie uruchomić aplikację użytkownika należy w niej dokonać kilka zabiegów. Pierwszy z nich to relokowanie naszej aplikacji w pamięci FLASH z adresu 0x8000000 na 0x8002000, czyli o 8 kB do przodu (np. przez zmianę w skrypcie linkera adresu początku pamięci). Drugi to usunięcie ustawienia wektora przerwań aplikacji użytkownika, gdyż bootloader wykonuje tą czynność sam.

Dodatkową opcją, wspomnianą wcześniej jest możliwość wywołania bootloadera w momencie zadeklarowanym już przez użytkownika w jego kodzie. Na **list. 3** umieszczono przykładowy plik nagłówkowy pozwalający wywołać bootloader na wspomniane wcześniej dwa sposoby: za pomocą makra *BOOTLOADER_CALL_BY_JUMP* lub *BOOTLOADER_CALL_BY_RESET*. W pliku, którym go dołączamy musimy jedynie dołączyć odpowiednią bibliotekę ST definiującą wskaźniki do struktur rejestrów peryferiów (RCC, BKP, itp). Sam nagłówek nie załącza ich z tego powodu, że w różnych wersjach bibliotek ST znajduje się to w różnych plikach.

Plik ten definiuje adres bazowy wektora przerwań samego bootloadera, który jest zgodny z adresem początku pamięci FLASH. Dalej mamy zdefiniowany początek standardowej dla tych procesorów struktury wektora przerwań. Wyjaśnienia wymaga chyba tylko to, co jest wewnątrz makra *BOOTLOADER_CALL_BY_RESET*. Pierwsze dwie linie to włączenie zegara dla kontrolera *Backup* oraz wyłączenie jego resetu, trzecia jest to odblokowanie zapisu do rejestrów Backup (pochodzi to z biblioteki stm32f10x_pwr). W czwartej linii wpisujemy do pierwszego rejestru danych *Backup* wartość 0x159D, której będzie szukać w tymże rejestrze bootloader po resecie. Piąta linia jest to wygenerowanie programowego sygnału resetu dla mikrokontrolera. Polega to na ustawieniu bitu *VECTRESET* (najmłodszy bit) w rejestrze *„Application Interrupt and Reset Control Register"*. Jest to jeden z rejestrów kontrolera NVIC, zdefiniowany w specyfikacji samego rdzenia Cortex -M3. Po wpisaniu na bit zerowy jedynki z równoczesnym wpisaniem na bity 16-31 sygnatury 0x5FA następuje zresetowanie procesora.

Podsumowanie

Bootloadery te dedykowane są dla mikrokontrolera STM32, niemniej jednak po pewnych przeróbkach można je przenieść na inne platformy. Wywoływane są w ściśle określonych momentach, dzięki czemu nie zakłócają normalnego startu programu. Dzięki dość uniwersalnej budowie istnieje możliwość poszerzenia możliwości tych programów. Można się też pokusić o obsługę magistrali zewnętrznej procesora, np. w celu programowania zewnętrznych pamięci FLASH i RAM.

Piotr Wojtowicz
piotreklc60@gmail.com

Literatura:
Dokumentacja STM32:
www.st.com/mcu/inchtml-pages-stm32.html
Strona domowa SD:
www.sdcard.org
Strona domowa USB:
www.usb.org
Specyfikacja DFU:
www.usb.org/developers/devclass_docs/DFU_1.1.pdf
Kody źródłowe:
www.wsn.agh.edu.pl