Wykorzystaj USB

Implementacja klasy HID interfejsu USB w STM32

Zmierzch ery interfejsów RS232 i LPT jest faktem niezaprzeczalnym. Interfejsy te, ze względu na prostotę obsługi programowej, były bardzo chętnie wykorzystywane do komunikacji komputera z najróżniejszymi urządzeniami elektronicznymi, zarówno tymi budowanymi przez elektroników hobbystów, jak również przez profesjonalistów. Ale ich czas się dokonał. W artykule prezentujemy sposób implementacji interfejsu USB w mikrokontrolerze STM32 z wykorzystaniem klasy HID (Human Interface Device), dla której większość systemów operacyjnych posiada wbudowane sterowniki.

Pojawienie się 10 lat temu interfejsu USB doprowadziło do wyparcia archaicznych interfejsów RS232 i LPT. Proces wypierania ich w komputerach stacjonarnych przebiegał stosunkowo wolno, w przypadku komputerów przenośnych nieco szybciej. O ile jeszcze dwa, trzy lata temu problem z brakiem portów występował praktycznie tylko w przypadku komputerów przenośnych, tak teraz jest powszechny również w przypadku komputerów stacjonarnych.

Sposoby radzenia sobie z tym problemem bywają różne: od nie zawsze działających adapterów USB/RS232 po karty PCI/ExpressCard, umożliwiające współpracę z urządzeniami wymagającymi interfejsów RS232 lub LPT.

W przypadku nowych urządzeń można zastosować specjalizowane układy scalone służące za konwertery pomiędzy łączem USB a np. portem szeregowym. Można również zastosować mikrokontroler z wbudowanym interfejsem USB, co jest chyba najbardziej uzasadnionym ekonomicznie rozwiązaniem.

Jedną z najprostszych do oprogramowania klas interfejsu USB jest klasa HID, opracowana z myślą o urządzeniach takich jak klawiatura czy mysz. W artykule przedstawiony zostanie przykład implementacji klasy HID interfejsu USB w mikrokontrolerze STM32 z grupy Performance Line (STM32F103) i wykorzystaniu jej do celów innych niż wskazuje na to pierwotne przeznaczenie klasy HID.

Oprogramowanie dla PC

Zastosowanie dowolnego interfejsu komunikującego budowane urządzenie elektroniczne z komputerem PC zawsze odbywa się na dwóch płaszczyznach: implementacji komunikacji po stronie mikrokontrolera oraz po stronie komputera PC. W przypadku interfejsu USB oprogramowanie komunikacji dla komputera PC jest stosunkowo trudne do wykonania i w większości przypadków wymaga pisania własnych sterowników sprzętu. Jednym z wyjątków od tej reguły jest klasa HID (*Human Interface Device*). Jest ona pierwszą z klas interfejsu USB i powstała z myślą o urządzeniach służących do sterowania komputerem przez człowieka, takich jak klawiatura,

myszka czy też joystick. Najważniejszą jej zaletą jest fakt, iż sterowniki są zawarte standardowo w większości współczesnych systemów operacyjnych. Dzięki temu urządzenie jest gotowe do pracy prawie natychmiast po podłączeniu.

Oczywiście bezpośrednio po podłączeniu nowego urządzenia klasy HID sterowniki zostaną zainstalowane automatycznie przez system operacyjny. Jak łatwo można się domyślić z przeznaczenia klasy HID – prędkość transmisji nie jest najwyższa. W przypadku interfejsu USB LowSpeed (1,5 Mbit/s) maksymalna prędkość transmisji danych wynosi 0,8 kB/s. W przypadku interfejsu USB FullSpeed (12 Mbit/s) prędkość transmisji danych może osiągnąć wartość do 64 kB/s.

Oprogramowanie dla komputera PC przedstawione w artykule zostało przygotowane dla systemu Windows. Komunikacja z urządzeniem klasy HID jest możliwa do zrealizowania poprzez wykorzystanie standardowych funkcji WinAPI służących do manipulacji plikami (FileCreate, FileRead, FileWrite itp.). Gdy istotny jest jak najkrótszy czas realizacji aplikacji, to można zastosować jedną z wielu gotowych bibliotek przeznaczonych do obsługi urządzeń klasy HID, które znacznie upraszczają proces pisania aplikacji dla PC. Jest to o tyle istotne, gdyż nie każdy elektronik, wykorzystujący mikrokontrolery.

Wybór narzędzia programistycznego podyktowany był zakładaną prostotą pisania aplikacji. Głównym założeniem było szybkie przygotowanie aplikacji, najlepiej z wykorzystaniem graficznego. Jednymi z najbardziej znanych i lubianych są narzędzia opracowane przez firmę Borland, czyli Delphi i C++ Builder. Niestety firma Borland nieco zmieniła branżę i przyszłość jej środowisk jest niepewna. Dlatego też zdecydowałem się na wybór platformy .NET firmy Microsoft oraz stosunkowo nowego języka programowania C#. Ciekawostką jest fakt, iż język C# został opracowany przez byłego głównego architekta środowiska Delphi Andersa Hejlsberga.

Rys. 1. Struktura drzewa projektu aplikacji

Podstawowym środowiskiem do tworzenia aplikacji dla platformy .NET jest Visual Studio firmy Microsoft. Jest to oprogramowanie komercyjne, jednak dobrą tradycją firmy Microsoft jest udostępnianie za darmo wersji Express Edition. Z punktu widzenia osoby tworzącej proste programy sterujące wykorzystywane prywatnie wersja Express Edition w zasadzie nie posiada żadnych znaczących ograniczeń. Brakuje tu przede wszystkim całej otoczki korporacyjnej, narzędzi pracy zespołowej, zaawansowanych narzędzi do obsługi baz danych itp. Jeżeli jednak ktoś z jakichkolwiek przyczyn nie chce korzystać z środowiska Visual Studio może wykorzystać dostępny na zasadach Open Source edytor SharpDevelop. Jest to IDE przeznaczone do pisania programów dla platformy .NET. Oczywiście konieczne jest posiadanie pakietu .NET SDK, który jest udostępniany za darmo przez firmę Microsoft.

Oprogramowanie dla mikrokontrolera

Mikrokontrolery STM32, podobnie jak i pozostałe produkty firmy STMicroelectronics, mają mocne wsparcie producenta w postaci bibliotek programistycznych oraz przykładowych aplikacji je wykorzystujących. Jedną z takich bibliotek jest STM32F10xUSBLib przeznaczona do obsługi interfejsu USB wraz z szeregiem przykładowych implementacji poszczególnych urządzeń interfejsu USB. Na potrzeby artykułu wykorzystano odpowiednio zmodyfikowaną aplikację `Custom_HID`. Prezentuje ona zastosowanie klasy HID do realizacji urządzenia nie będącego jej typowym przedstawicielem, lecz wykorzystującym ją do własnych, zdefiniowanych przez programistę celów. Projekt dla środowiska Ride znajduje się w katalogu o następującej ścieżce dostępu: *USBLib\demos\Custom_HID\project\RIDE*. Plik *Custom_HID.rprj* należy załadować do edytora Ride.

Strukturę drzewa projektu aplikacji przedstawiono na **rys. 1**. Projekt składa się z trzech grup plików źródłowych. W grupie „Application files" umieszczono pliki zawierające kod zasadniczej części programu oraz pliki biblioteki USB charakteryzujące konkretne urządzenie. W grupie USBLib znajdują się pozostałe pliki, których zawartość nie zależy od implementowanego urządzenia, ale zawiera funkcje podstawowej konfiguracji interfejsu USB. W grupie FWLib znajdują się pliki biblioteki programistycznej dla pozostałych układów peryferyjnych mikrokontrolera STM32 wykorzystywanych w aplikacji. Niektóre pliki z grupy „Application Files", zmodyfikowano w celu dostosowania aplikacji do zestawu ZL27ARM (*www.kamami.pl*), gdyż te dostarczone przez STMicroelectronics przystosowane są do zestawów STM3210B-EVAL i STM3210E-EVAL.

Przykładowa aplikacja w pierwotnej wersji umożliwiała włączenie 4 diod LED, odczyt wartości napięcia podawanego z potencjometru oraz stanu dwóch przycisków. Aplikacja została rozszerzona o możliwość sterowania dodatkowymi czterema diodami LED (zestaw ZL27ARM posiada ich osiem), oraz odczyt stanu dodatkowych dwóch przycisków.

Deskryptory

Najważniejszym plikiem jest `usb_desc.c` zawierający deskryptory opisujące konfigurację urządzenia USB. Pierwszym deskryptorem zawartym w pliku jest deskryptor urządzenia (Device Descriptor).

Deskryptor urządzenia

Zawiera takie informacje, jak wersję standardu USB, numery identyfikujące producenta (VendorID) oraz urządzenie (ProductID). Kod deskryptora DeviceDescriptor przedstawiono na **list. 1**

Najistotniejszymi informacjami są numery `idVendor` oraz `idProduct`, które zostaną wykorzystane do połączenia się z urządzeniem USB z poziomu aplikacji uruchomionej na komputerze PC. Numerów tych w zasadzie nie należy modyfikować, gdyż VendorID (VID) jest nadawany członkom organizacji USB-IF (www.usb.org). Numer ProductID (PID) jest określany we własnym zakresie przez posiadacza numeru VID. Oczywiście do zastosowań prywatnych, czy też testów, można przypisać dowolne numery VID i PID, jednak absolutnie nie wolno tego robić w przypadku urządzeń wprowadzanych na rynek. W przypadku prezentowanej aplikacji pozostawiamy oryginalne numery VID i PID. W systemie może pracować kilka urządzeń o identycznych numerach VID i PID, więc naprawdę nie ma potrzeby ich modyfikacji nawet, jeśli zamierzamy zbudować kilka urządzeń pełniących różne funkcje.

Deskryptor raportów

Deskryptor raportów opisuje wszystkie cechy raportów m.in. identyfikator, kierunek transmisji raportu, rozmiar raportu, wartości minimalne i maksymalne przyjmowane przez raport i inne. Aplikacja demonstracyjna wykorzystuje 13 raportów – po jednym dla każdego transmitowanego stanu elementu (8 diod LED, potencjometr i 4 przyciski). Ze względu na dużą objętość deskryptora przedstawiono tylko wybrane jego fragmenty, charakteryzujące każdy z wykorzystywanych typów raportów.

Deskryptor raportu stanu diody LED (identyfikator raportu: 1) przedstawiono na **list. 2.** Składa się on z dziesięciu pól, na każde pole przypadają dwa bajty: identyfikatora pola oraz wartości. Najistotniejsze są pola REPORT_ID oraz USAGE. Ich wartość odpowiada identyfikatorowi raportu. Deskryptor raportów zawiera osiem takich struktur, które różnią się war-

List. 1. Kod źródłowy deskryptora urządzenia

```
/* USB Standard Device Descriptor */
const u8 CustomHID_DeviceDescriptor[CUSTOMHID_SIZ_DEVICE_DESC] =
  {
    0x12,     //bLength
    USB_DEVICE_DESCRIPTOR_TYPE, //bDescriptorType
    0x00,     //bcdUSB
    0x02,
    0x00,     //bDeviceClass
    0x00,     //bDeviceSubClass
    0x00,     //bDeviceProtocol
    0x40,     //bMaxPacketSize = 40
    0x83,     //VendorID = 0x0483
    0x04,
    0x50,     //ProductID = 0x5750
    0x57,
    0x00,     //bcdDevice rel. 2.00
    0x02,
    1,        //Index of string descriptor describing manufacturer
    2,        //Index of string descriptor describing product
    3,        //Index of string descriptor describing the device serial number
    0x01      //bNumConfigurations
  }
  ; /* CustomHID_DeviceDescriptor */
```

List. 2. Kod deskryptora raportu LED

```
/* Led 1 */
0x85, 0x01,           //REPORT_ID (1)
0x09, 0x01,           //USAGE (LED 1)
0x15, 0x00,           //LOGICAL_MINIMUM (0)
0x25, 0x01,           //LOGICAL_MAXIMUM (1)
0x75, 0x08,           //REPORT_SIZE (8)
0x95, 0x01,           //REPORT_COUNT (1)
0xB1, 0x82,           //FEATURE (Data,Var,Abs,Vol)
0x85, 0x01,           //REPORT_ID (1)
0x09, 0x01,           //USAGE (LED 1)
0x91, 0x82,           //OUTPUT (Data,Var,Abs,Vol)
```

tościami REPORT_ID oraz USAGE. Deskryptor raportu stanu potencjometru (raport o identyfikatorze 9) przedstawiono na **list. 3.**

Podstawową różnicą w stosunku do poprzedniego deskryptora jest wartość pola LOGICAL_MAXIMUM, które w przypadku raportu stanu potencjometru przyjmuje wartość 255. Należy zwrócić uwagę, że wartość tego pola jest typu *signed* i musi być zapisana na 16 bitach. W przeciwnym razie stała 0xFF potraktowana zostałaby jako wartość –128 co byłoby błędem. Deskryptor raportu stanu przycisku przedstawiono na **list. 4.**

Raport ten składa się z dwóch części: zmiennej o rozmiarze jednego bitu, która odpowiada stanowi przycisku oraz stałej o rozmiarze 7-bitów.

Transfer stanu diod LED

Stan diod LED transmitowany jest z aplikacji uruchomionej na komputerze do mikrokontrolera za pomocą ośmiu raportów o numerach ID z zakresu 1-8. Po odebraniu przez mikrokontroler raportu wywoływana jest procedura EP1_OUT_Callback, w ramach której podejmowana jest reakcja na nadesłane w raporcie dane. Procedura ta jest zdefiniowana w pliku usb_endp.c. Kod procedury EP1_OUT_Callback przedstawiono na **list. 5.**

Na początku procedury danych z endpointa odbiorczego kopiowane są do tablicy Receive_Buffer, która reprezentuje odebrany raport. Następnie sprawdzany jest stan elementu tablicy o indeksie równym jeden, a więc stan diody LED. W zależności od wartości tego elementu tablicy odpowiednio modyfikowana jest zmienna Led_State. Następnie w zależności od ID raportu stan diody LED zostanie przypisany do odpowiadającego jej wyjścia GPIO mikrokontrolera za pomocą instrukcji switch. Po modyfikacji odpowiedniego wyjścia ustawiany jest odpowiedni status endpointa odbiorczego informujący o przetworzeniu odebranych danych.

Transfer stanu przycisków

Stan przycisków zestawu ZL27ARM transmitowany jest z mikrokontrolera do komputera PC z wykorzystaniem endpointa nadawczego i raportów o numerach 10-13. Przyciski SW0--SW3 są podłączone do wyprowadzeń PA0-PA3, a więc do linii EXTI0-EXTI3. W ramach procedur obsługi przerwań Exti0-Exti3 formowany jest odpowiedni raport z danymi zależnymi od stanu przycisku, który wywołał przerwanie. Kod procedury obsługi przerwania EXTI0 przedstawiono na **list. 6.** Pozostałe trzy procedury różnią się tylko numerem linii przerwania oraz ID raportu.

Transfer stanu potencjometru P2

Odczyt wyniku pomiaru przetwornika ADC zrealizowano za pomocą układu DMA. Po każdym zakończonym pomiarze wynik transmitowany jest bez pośrednictwa CPU do pamięci danych. Po zakończeniu przez układ DMA transferu danych zgłaszane jest przerwanie. Kod procedury przerwania od kanału DMA przedstawiony jest na **list. 7.**

W celu wyeliminowania cyklicznego transferu przez łącze USB identycznych wyników pomiarów sprawdzana jest różnica pomiarów: aktualnego i poprzedniego (po ograniczeniu rozdzielczości do 8 bitów). Jeżeli jest ona większa od 4, to raport z wynikiem pomiaru napięcia jest kopiowany do endpointu nadawczego.

Oprogramowanie dla komputera PC

Jak już wspomniano na wstępie, oprogramowanie dla komputera PC napisano w języku C# dla platformy .NET 2.0. Wykorzystano bibliotekę HIDLibrary, której autorem jest Michael P. O'Brien (www.mike-obrien.net). Bibliotekę tę napisano w języku Visual Basic.NET. Dostępna jest w postaci źródłowej oraz pliku DLL na stronie autora. Nie jest to oczywiście jedyna biblioteka ułatwiająca korzystanie z klasy HID interfejsu USB. Na stronie www.lvr.com/hid_page znajduje się zbiór odnośników do projektów i bibliotek oprogramowania dla klasy HID.

Czytelnik powinien posiadać zainstalowane środowisko Visual C# 2008 Express Edition, komercyjną wersję środowiska Visual Studio 2008 lub program SharpDevelop wraz z .NET 2.0 Software Development Kit (SDK). Visual C# 2008 EE umożliwia tworzenie aplikacji wyłącznie dla .NET Framework w wersji 3.5, natomiast w chwili pisania artykułu środowisko SharpDevelop było dostępne tylko dla .NET Framework w wersji 2.0. Nie ma to większego znaczenia, gdyż żadna z technologii udostępnianych przez .NET 3.5 nie będzie w projekcie wykorzystywa-

List. 3. Kod deskryptora raportu potencjometru

```
/* Potencjometr P1 */
0x85, 0x09,           //REPORT_ID (9)
0x09, 0x09,           //USAGE (P1)
0x15, 0x00,           //LOGICAL_MINIMUM (0)
0x26, 0xff, 0x00,     //LOGICAL_MAXIMUM (255)
0x75, 0x08,           //REPORT_SIZE (8)
0x81, 0x82,           //INPUT (Data,Var,Abs,Vol)
0x85, 0x09,           //REPORT_ID (9)
0x09, 0x09,           //USAGE (P1)
0xb1, 0x82,           //FEATURE (Data,Var,Abs,Vol)
```

List. 4. Kod deskryptora przycisku

```
/* Przycisk SW0  */
0x85, 0x0A,           //REPORT_ID (10)
0x09, 0x0A,           //USAGE (SW0)
0x15, 0x00,           //LOGICAL_MINIMUM (0)
0x25, 0x01,           //LOGICAL_MAXIMUM (1)
0x75, 0x01,           //REPORT_SIZE (1)
0x81, 0x82,           //INPUT (Data,Var,Abs,Vol)
0x09, 0x0A,           //USAGE (SW0)
0x75, 0x01,           //REPORT_SIZE (1
0xb1, 0x82,           //FEATURE (Data,Var,Abs,Vol)
0x75, 0x07,           //REPORT_SIZE (7)
0x81, 0x83,           //INPUT (Cnst,Var,Abs,Vol)
0x85, 0x0A,           //REPORT_ID (10)
0x75, 0x07,           //REPORT_SIZE (7)
0xb1, 0x83,           //FEATURE (Cnst,Var,Abs,Vol)
```

List. 5. Kod procedury EP1_OUT_Callback
```
void EP1_OUT_Callback(void)
{
  BitAction Led_State;
  PMAToUserBufferCopy(Receive_Buffer, ENDP1_RXADDR, 2);
  if (Receive_Buffer[1] == 0)
  {
    Led_State = Bit_RESET;
  }
  else
  {
    Led_State = Bit_SET;
  }
  switch (Receive_Buffer[0]) // jakie ID raportu ?
  {
    case 1: /* Led 1 */
     GPIO_WriteBit(GPIOB, GPIO_Pin_8, Led_State);
     break;
    case 2: /* Led 2 */
      GPIO_WriteBit(GPIOB, GPIO_Pin_9, Led_State);
      break;
    case 3: /* Led 3 */
      GPIO_WriteBit(GPIOB, GPIO_Pin_10, Led_State);
      break;
    case 4: /* Led 4 */
      GPIO_WriteBit(GPIOB, GPIO_Pin_11, Led_State);
      break;
    case 5: /* Led 5 */
      GPIO_WriteBit(GPIOB, GPIO_Pin_12, Led_State);
      break;
    case 6: /* Led 6 */
      GPIO_WriteBit(GPIOB, GPIO_Pin_13, Led_State);
      break;
    case 7: /* Led 7 */
      GPIO_WriteBit(GPIOB, GPIO_Pin_14, Led_State);
      break;
    case 8: /* Led 8 */
      GPIO_WriteBit(GPIOB, GPIO_Pin_15, Led_State);
      break;
    default:
    GPIO_Write(GPIOB, ~(GPIO_Pin_8|GPIO_Pin_9|GPIO_Pin_10|GPIO_Pin_11|GPIO_
Pin_12|GPIO_Pin_13|GPIO_Pin_14|GPIO_Pin_15));
    break;
  }
  SetEPRxStatus(ENDP1, EP_RX_VALID);
}
```

List. 6. Kod procedury obsługi przerwania
```
void EXTI0_IRQHandler(void)
{
 if(EXTI_GetITStatus(EXTI_Line0) != RESET)
  {
    Send_Buffer[0] = 0xA;    // ID raportu
    if (GPIO_ReadInputDataBit(GPIOA, GPIO_Pin_0) == Bit_RESET)
    {
      Send_Buffer[1] = 0x01; // przycisk naciśnięty
    }
    else
    {
      Send_Buffer[1] = 0x00; // przycisk zwolniony
    }
                           // skopiowanie raportu do endpointa
    UserToPMABufferCopy(Send_Buffer, ENDP1_TXADDR, 2);
    SetEPTxCount(ENDP1, 2);  // Rozmiar danych w endpoincie
    SetEPTxValid(ENDP1);     // Dane w endpoincie kompletne

    EXTI_ClearITPendingBit(EXTI_Line0);
  }
}
```

na, więc przedstawiona aplikacja powinna posiadać identyczną funkcjonalność, niezależnie od wersji frameworka .NET.

Tworzenie aplikacji wykorzystującej bibliotekę HIDLibrary.dll

Proces tworzenia aplikacji przedstawiono na przykładzie środowiska VisualC# 2008 EE. Po uruchomieniu środowiska programistycznego należy utworzyć nowy projekt typu Windows Forms Application. Główną część okna zajmie Form Designer, służący do projektowania wyglądu okna aplikacji. Po lewej stronie okna jest ukryty Toolbox zawierający komponenty graficzne. Ukaże się on po najechaniu kursorem na pole z napisem Toolbox. Po prawej stronie okna znajduje się Solution Explorer, który przedstawia strukturę plików i folderów projektu. Pierwszą czynnością należy wykonać jest dodanie referencji do biblioteki HIDLibrary.dll. W tym celu należy prawym przyciskiem myszy kliknąć na folder References drzewa projektu a następnie z menu kontekstowego wybrać opcję *Add Reference*. Ukaże się okno *Add Reference*. Należy wybrać zakładkę *Browse* i wskazać plik biblioteki `HIDLibrary.dll`. Jeśli nie wystąpiły żadne problemy to w katalogu References drzewa projektu powinna się pojawić pozycja HIDLibrary.

Okno aplikacji

Wygląd projektu okna aplikacji demonstracyjnej przedstawiony jest na **rys. 2**. Lista rozwijalna comboBox1 służy do wyboru urządzenia klasy HID, z którym będzie komunikować się aplikacja. Po wybraniu urządzenia z listy, w celu połączenia się z wybranym urządzeniem, należy kliknąć przycisk *Połącz*. Okienka `checkBox1…8` (grupa *Diody LED*) służą do ustawiania stanu diod LED zestawu ZL27ARM. Komponenty panel1…4 (grupa *Przyciski*) służą do wizualizacji stanu przycisków zestawu ZL27ARM. Kolor zielony symbolizuje przycisk zwolniony, natomiast czerwony przycisk naciśnięty. Komponent progressBar1 (grupa *Potencjometr*) służy do wizualizacji wartości podawanej na wejście przetwornika analogowo-cyfrowego z potencjometru P1 zestawu ZL27ARM.

Wykorzystywane zmienne

Aplikacja tworzy dwa wątki: do zapisu oraz do odczytu danych z urządzenia HID. Wątek zapisu raportów wyjściowych jest reprezentowany przez zmienną `WriteThread`, natomiast wątek odczytu raportów wejściowych przez zmienną `ReadThread`. Stan wszystkich wykorzystywanych elementów zestawu ZL27ARM przechowywany jest w zmiennych globalnych `LED`, `Buttons` oraz `AnalogValue`. Zmienne te wykorzystywane są do przekazywania danych o stanie poszczególnych elementów pomiędzy wątkami służącymi do zapisu i odczytu raportów a pozostałą częścią aplikacji. Tablica `HID` przechowuje wynik enumeracji wszystkich urządzeń HID zainstalowanych w systemie. Wynik ten może obejmować wszystkie urządzenia klasy HID lub też urządzenia o podanych identyfikatorach VID oraz PID. Zmienna `MyHID` reprezentuje urządzenie, z którym aplikacja będzie wymieniać dane. Deklaracje wszystkich zmiennych globalnych przedstawiono na **list. 8**.

Enumeracja urządzeń HID

Proces enumeracji urządzeń dokonywany jest na etapie ładowania formularza aplikacji (zdarzenie `FormLoad()`). Fragment kodu odpowiedzialny za enumerację urządzeń przedstawiono na **list. 9**.

Zakres poszukiwań urządzeń HID został ograniczony do urządzeń o podanych identyfi-

List. 7. Kod procedury obsługi przerwania od DMA
```
void DMA1_Channel1_IRQHandler(void)
{
  Send_Buffer[0] = 0x09;
  if((ADC_ConvertedValueX >> 4) - (ADC_ConvertedValueX_1 >> 4) > 4)
  {
    Send_Buffer[1] = (u8)(ADC_ConvertedValueX >> 4);
    UserToPMABufferCopy(Send_Buffer, ENDP1_TXADDR, 2);
    SetEPTxCount(ENDP1, 2);
    SetEPTxValid(ENDP1);
    ADC_ConvertedValueX_1 = ADC_ConvertedValueX;
  }
  DMA_ClearFlag(DMA1_FLAG_TC1);
}
```

List. 8. Deklaracje zmiennych globalnych
```
// Wątki odczytu i zapisu do urządzenia HID
Thread WriteThread, ReadThread;
// tablica urządzeń klasy HID
HIDLibrary.HidDevice[] HID;
// urządzenie klasy HID
HIDLibrary.HidDevice MyHID;
// zmienna przechowująca stan potencjometru
byte AnalogValue;
// tablica przechowująca stan przycisków
byte[] Buttons = new byte[4];
// tablica przechowująca stan diod LED
byte[][] LED = {
   new byte[1] { 0 },
   new byte[1] { 0 },
   new byte[1] { 0 },
   new byte[1] { 0 },
   new byte[1] { 0 },
   new byte[1] { 0 },
   new byte[1] { 0 },
   new byte[1] { 0 },
};
// tablica przechowująca informacje o konieczności zapisu do urządzenia HID
byte[] pUpdate = new byte[8] { 1, 1, 1, 1, 1, 1, 1, 1 };
```

katorach VID i PID, które są właściwe dla aplikacji demonstracyjnej zestawu ZL27ARM. Po dokonaniu enumeracji następuje sprawdzenie, czy w tablicy `HID` znajdują się jakieś dane, to znaczy czy znaleziono przynajmniej jedno urządzenie klasy HID. Jeśli warunek sprawdzenia jest spełniony, to następuje wypełnienie kontrolki `comboBox1` pozycjami reprezentującymi każde odnalezione urządzenie. W przeciwnym razie wyświetlany jest komunikat informujący, że nie znaleziono żadnego urządzenia HID.

Podłączenie do urządzenia HID

Podłączenie do urządzenia HID dokonywane jest w ramach obsługi zdarzenia `OnClick` przycisku `button1`. Kod handlera tego zdarzenia przedstawiono na **list. 10**.

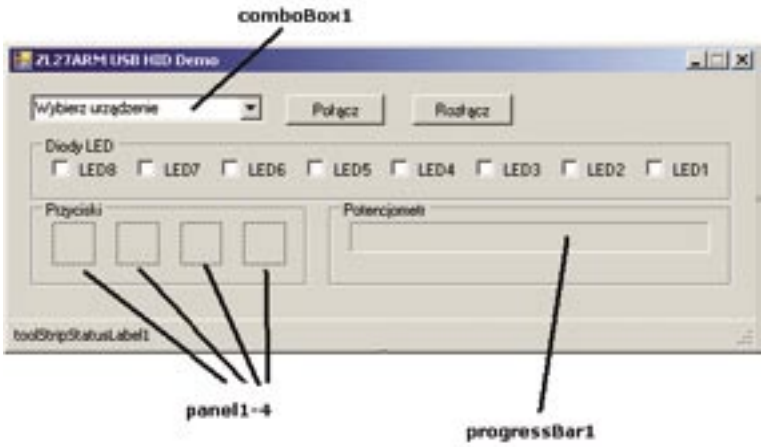
Na początku sprawdzana jest właściwość `SelectedIndex` komponentu `comboBox1`, mówiąca o tym, która pozycja listy została wybrana. Jeśli właściwość ta ma wartość większą od -1 to oznacza, że jedna z pozycji została wybrana i realizowane jest podłączenie do urządzenia HID. W przeciwnym razie wyświetlany jest komunikat mówiący, iż nie zostało wybrane żadne urządzenie z listy. Samo podłączenie sprowadza się do przypisania do zmiennej `MyHID` wybranej pozycji z tablicy `HID` i wywołaniu metody `OpenDevice()`. Następnie tworzone są i uruchamiane wątki `WriteThread` oraz `ReadThread`. Pozostała część kodu odpowiada za zmianę stanu właściwości Enabled wykorzystywanych kontrolek oraz ustawienie na pasku statusu aplikacji napisu „Połączony".

Odczyt raportów wejściowych

Działanie wątku rozpoczyna się od utworzenia zmiennej `InputReport`, która reprezentuje odczytany raport wejściowy. Argument konstruktora tej zmiennej określa rozmiar raportu w bajtach i jest odczytywany z właściwości `InputReportByteLength` podłączonego urządzenia. Ponieważ wątek uruchamiany jest tylko raz (po podłączeniu urządzenia) musi wykonywać się w pętli nieskończonej. W tej pętli następuje sprawdzenie, czy zmienna `MyUSB` jest właściwie zainicjowana. Jeśli tak, to za pomocą metody `ReadReport()` odczytywany jest raport zmiennej `MyUSB`. Następnie w zależności od wartości właściwości `ReportID` dane z raportu przypisywane są albo do zmiennej `AnalogValue`, albo do jednej z czterech pozycji tablicy `Buttons`. Kod wątku realizującego odczyt raportów wejściowych przedstawiono na **list. 11**.

Zapis raportów wyjściowych

Działanie wątku rozpoczyna się od utworzenia zmiennej `OutputReport`, która reprezentuje zapisywany do urządzenia raport. Następnie w pętli nieskończonej wykonywana jest zasadnicza część kodu odpowiedzialna za transfer do urządzenia HID raportów z danymi. W celu wyeliminowania niepotrzebnych transferów danych sprawdzany jest stan każdej pozycji



Rys. 2. Okno projektu aplikacji demonstracyjnej

List. 9. Fragment programu odpowiedzialny za enumerację
```
HID = HidDevices.Enumerate(0x0483, 0x5750);
if (HID.Length > 0)
   {
   for (int i = 0; i < HID.Length; i++)
     {
     comboBox1.Items.Add(„HID device „ + i.ToString() +
                         „ ver. „ + HID[i].Attributes.Version.ToString());
     }
   }
   else
   {
   MessageBox.Show(„Nie znaleziono żadnego urządzenia HID!");
   }
```

List. 10. Kod handlera obsługi zdarzenia
```
private void button1_Click(object sender, EventArgs e)
{
if (comboBox1.SelectedIndex > -1)
   {
   MyHID = HID[comboBox1.SelectedIndex];
   MyHID.OpenDevice();
   if (MyHID.IsOpen == true)
      {
      WriteThread = new Thread(WriteOutputReports);
      WriteThread.Start();
      ReadThread = new Thread(ReadInputReports);
      ReadThread.Start();
      button2.Enabled = true;
      button1.Enabled = false;
      groupBox1.Enabled = true;
      checkBox1.Enabled = true;
      checkBox2.Enabled = true;
      checkBox3.Enabled = true;
      checkBox4.Enabled = true;
      checkBox5.Enabled = true;
      checkBox6.Enabled = true;
      checkBox7.Enabled = true;
      checkBox8.Enabled = true;
      toolStripStatusLabel1.Text = „Połączony";
      }
    }
    else
      MessageBox.Show("Nie wybrano urządzenia");
   }
```

List. 11. Kod wątku realizującego odczyt raportów wejściowych

```
void ReadInputReports()
{
HIDLibrary.HidReport InputReport =
       new HIDLibrary.HidReport(MyUSB.Capabilities.InputReportByteLength);
while (true)
  {
  if (MyUSB != null)
     {
     InputReport = MyUSB.ReadReport();
     switch (InputReport.ReportId)
     {
     case 0x9: AnalogValue = InputReport.Data[0]; break;
     case 0xA: Buttons[0] = InputReport.Data[0]; break;
     case 0xB: Buttons[1] = InputReport.Data[0]; break;
     case 0xC: Buttons[2] = InputReport.Data[0]; break;
     case 0xD: Buttons[3] = InputReport.Data[0]; break;
     }
     Thread.Sleep(20);
     }
  }
}
```

List. 12. Kod wątku realizującego zapis raportów wyjściowych

```
void WriteOutputReports()
{
HIDLibrary.HidReport OutputReport =
new HIDLibrary.HidReport(MyUSB.Capabilities.OutputReportByteLength);
while (true)
 {
 if (MyUSB != null)
  {
  for (byte i = 0; i < 8; i++)
    {
    if (pUpdate[i] == 1)
      {
      pUpdate[i] = 0;
      OutputReport.ReportId = (byte)(i + 1);
      OutputReport.Data = LED[i];
      MyUSB.WriteReport(OutputReport);
      }
      Thread.Sleep(20);
     }
   }
 }
}
```

List. 13. Kod handlera obsługi zdarzenia generowanego przez timer

```
private void timer1_Tick(object sender, EventArgs e)
{
progressBar1.Value = AnalogValue;

if (Buttons[0] == 1)
    panel1.BackColor = Color.Red;
else
    panel1.BackColor = Color.Green;
if (Buttons[1] == 1)
    panel2.BackColor = Color.Red;
else
    panel2.BackColor = Color.Green;
if (Buttons[2] == 1)
    panel3.BackColor = Color.Red;
else
    panel3.BackColor = Color.Green;
if (Buttons[3] == 1)
    panel4.BackColor = Color.Red;
else
    panel4.BackColor = Color.Green;
}
```

tablicy `pUpdate` i w sytuacji, gdy któryś z elementów tablicy ma wartość 1, transmitowany jest odpowiedni raport zawierajacy stan diody LED. Kod wątku realizującego zapis raportów wyściowych do urządzenia HID przedstawiono na **list. 12**.

Aktualizacja stanu kontrolek wizualnych

Aktualizacja stanu kontrolek wejściowych na podstawie odczytanych danych odbywa się w ramach handlera zdarzenia `OnTimer Timera 1`. Kod handlera zdarzenia przedstawiono na **list. 13.**

Obsługa aplikacji demonstracyjnej

Zestaw ZL27ARM z zaprogramowanym mikrokontrolerem należy podłączyć do złącza USB komputera przed uruchomieniem aplikacji demonstracyjnej. System powinien automatycznie wykryć nowe urządzenie oraz zainstalować niezbędne sterowniki. Jeśli sterowniki zostały poprawnie zainstalowane, należy uruchomić aplikację demonstracyjną. W przypadku nie znalezienia urządzenia o określonych w kodzie programu numerach VID i PID, wyświetlony zostanie odpowiedni komunikat. W przeciwnym razie powinno ukazać się okno aplikacji demonstracyjnej. Z listy comboBox1 należy wybrać pozycję odpowiadającą urządzeniu HID. W przypadku podłączenia jednego zestawu na liście powinna znajdować się jedna pozycja. Po wybraniu urządzenia należy kliknąć przycisk „Połącz". Jeśli pomyślnie nawiązano połączenie z urządzeniem HID kontrolki `checkBox1…8` zostaną uaktywnione. Od tej chwili stan tych kontrolek będzie odwzorowywany na diodach LED zestawu ZL27ARM – „zaznaczony" checkBox odpowiada włączonej diodzie LED, natomiast „pusty" wyłączonej. Pozostałe elementy aplikacji demonstracyjnej służą do odwzorowania stanu przycisków i potencjometru, więc w celu sprawdzenia działania aplikacji należy nacisnąć dowolny z przycisków SW0...SW3 lub pokręcić osią potencjometru P1. Wartość napięcia podawanego przez potencjometr na wejście przetwornika analogowo-cyfrowego zostanie odwzorowana w postaci paska postępu. Z kolei naciśnięty przycisk będzie reprezentowany przez czerwony kolor odpowiadającego mu panelu. Na tym kończą się możliwości przedstawionej aplikacji demonstracyjnej a otwiera się szerokie pole do wykorzystania przedstawionej biblioteki oraz mikrokontrolerów STM32 we własnych aplikacjach wykorzystujących interfejs USB do niewymagających dużej prędkości transferów danych pomiędzy komputerem PC a urządzeniem z mikrokontrolerem STM32.

Przedstawiona aplikacja zajmuje nieco ponad 8 KB pamięci Flash, co w przypadku mikrokontrolera STM32F103VBT6 jest ułamkiem całej dostępnej pamięci programu.

Podsumowanie

Temat oprogramowania interfejsu USB, zarówno po stronie mikrokontrolera, jak i komputera PC, jest na tyle szeroki, że trudno przedstawić całość zagadnienia w ramach jednego artykułu. W artykule przedstawiono sposób na postawienie „pierwszych kroków" na drodze programowania interfejsu USB. Droga ta jest niestety bardzo długa i wymaga szczegółowego zagłębienia się tak w specyfikacje standardu, jak i informacje oraz przykładowe aplikacje dostarczane przez producenta mikrokontrolerów STM32.

Radosław Kwiecień, EP
radoslaw.kwiecien@ep.com.pl