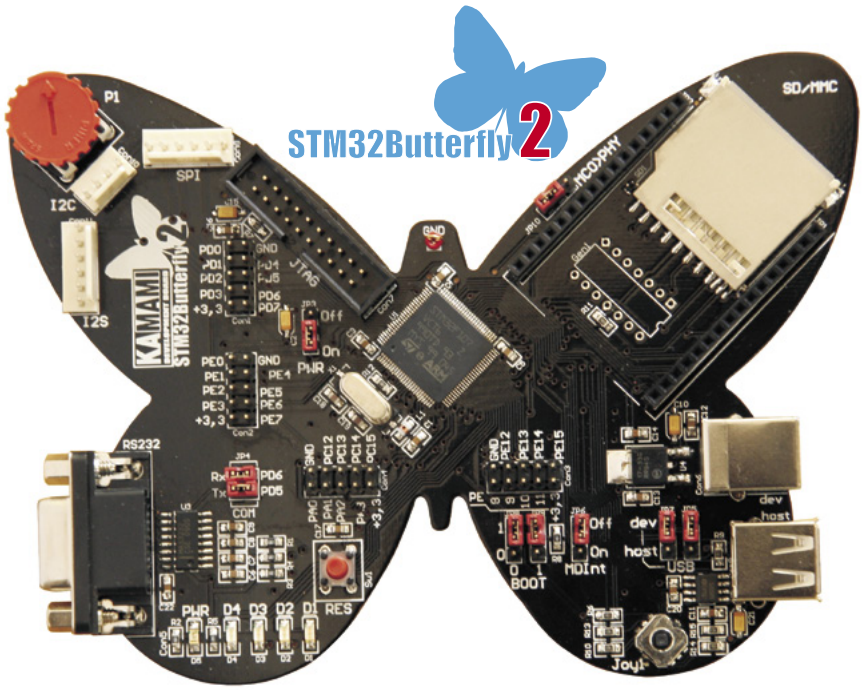
# Cyfrowy analizator 32-kanałowy (1)

## Konfigurowanie mikrokontrolera, podstawy funkcjonowania aplikacji

*Płytka ewaluacyjna zakupiona w celu zapoznania się z nowym układem scalonym po zakończeniu nauki nie musi być bezużyteczna. Można na jej bazie zbudować sterownik mikroprocesorowy lub użyteczny przyrząd laboratoryjny. W artykule opisano sposób wykonania analizatora stanów logicznych z wyświetlaniem zarejestrowanych poziomów sygnałów na ekranie PC, z którym komunikacja odbywa się przez port USB. Zaprezentowane rozwiązanie może posłużyć jako baza do budowy wielu własnych projektów.*




Analizator wykonano z użyciem zestawu ewaluacyjnego *STM32 Butterfly* z mikrokontrolerem STM32F107VBT6. Oprogramowanie dla STM32 napisano w języku C. Punktem wyjścia do implementacji był udostępniony za darmo projekt CustomHID (biblioteka *STM32F10xUSBLib*). Prezentuje on zastosowanie klasy HID do realizacji komunikacji urządzenia za pomocą USB. Jako środowisko programistyczne wybrano *Atollic True Studio STM32 Lite*. To oprogramowanie ma wygodny edytor, wbudowany kompilator oraz interfejs programatora.

Aplikację dla komputera PC napisano w języku Microsoft C# z użyciem bibliotek: *HIDLibrary* (*http://labs.mikeobrien.net/Document.aspx?id=hidlibrary*) do obsługi komunikacji przez USB oraz *ZedGraph* (*http://zedgraph.org/*) do sporządzania diagramów przebiegów. Wykorzystano środowisko programistyczne *Microsoft Visual Studio 2010 Ultimate*. Ma ono ogromną liczbę gotowych do użycia funkcji, z których z punktu widzenia projektu, najbardziej pomocną była możliwość graficznego projektowania aplikacji okienkowych.

### Oprogramowanie zestawu ewaluacyjnego

Wszystkie używane zmienne/tablice zawiera plik *stm32_eval.h*. Implementację należy rozpocząć od poprawnego skonfigurowania zegarów urządzenia. Nie ma tutaj potrzeby omawiania tego fragmentu – dzięki użyciu projektu CustomHID otrzymujemy prawidłową konfigurację. Należy tylko wiedzieć, że jest to konfiguracja przeznaczona do pracy z zewnętrznym rezonatorem kwarcowym o częstotliwości 8 MHz, którego sygnał po powieleniu daje sygnału zegarowego taktującego CPU wynoszącą 72 MHz. Jest to zarazem maksymalna szybkość pracy urządzenia.

**Podstawowe parametry:**
- Odbiór danych konfiguracyjnych za pomocą USB.
- Analizator 32-kanałowy.
- Możliwość monitorowania mniejszej liczby kanałów bez konieczności ich odłączania od urządzenia.
- Próbkowanie częstotliwością sygnału zegarowego doprowadzonego do jednego z pinów.
- Alternatywnie, odczyt z użyciem DMA.
- Funkcja zatrzaskiwania poziomów logicznych.
- Przesłanie danych do komputera PC za pomocą USB.

**Funkcjonalność aplikacji sterującej:**
- Możliwość ustawienia *trigger word* (dla każdej linii 3 poziomy: 0, 1 lub dowolny).
- Możliwość ustawienia źródła wyzwalania (DMA, sygnał zegarowy).
- Możliwość wyboru zbocza wyzwalania (opadające, rosnące, dowolne) w przypadku wyzwalania sygnałem zegarowym.
- Przesłanie danych konfiguracyjnych do analizatora.
- Odbiór danych z analizatora.
- Wyświetlanie przebiegów z zaznaczeniem *trigger word*.
- Zapis przebiegów jako obraz/plik tekstowy, drukowanie.

**Listing 1. Procedura konfigurująca rejestr GPIO**

```
//hw_config.c
  RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIO_DISCONNECT | RCC_APB2Periph_GPIO_IOAIN | RCC_APB2Periph_GPIOC | RCC_
APB2Periph_GPIOD | RCC_APB2Periph_GPIOE, ENABLE);
  GPIO_InitStructure.GPIO_Pin=GPIO_Pin_1 |GPIO_Pin_2 | GPIO_Pin_3 |GPIO_Pin_4 | GPIO_Pin_5 | GPIO_Pin_6 | GPIO_Pin_7 |
GPIO_Pin_8 | GPIO_Pin_9 | GPIO_Pin_10 | GPIO_Pin_11;                //wskazanie używanych doprowadzeń
  GPIO_InitStructure.GPIO_Mode=GPIO_Mode_IPD;                     //praca jako pull-down
  GPIO_InitStructure.GPIO_Speed=GPIO_Speed_50MHz;                 //prędkość 50 MHz
  GPIO_Init(GPIOC, &GPIO_InitStructure);
```

**Listing 2. Konfigurowanie DMA i przerwań**

```
//hw_config.c
DMA_InitTypeDef DMA_InitStructure;
RCC_AHBPeriphClockCmd(RCC_AHBPeriph_DMA1, ENABLE);                              //wlacza zegar dla DMA1
  //channel1 odczytuje rejestr GPIOC
  DMA_DeInit(DMA1_Channel1);
  DMA_InitStructure.DMA_PeripheralBaseAddr = (uint32_t)&(GPIOC->IDR);           //z rejestru C
  DMA_InitStructure.DMA_MemoryBaseAddr = (uint32_t)registerGPIOC;               //do tablicy
  DMA_InitStructure.DMA_DIR = DMA_DIR_PeripheralSRC;                            //z rejestru do tablicy
  DMA_InitStructure.DMA_BufferSize = DataSize;                                  //rozmiar tablicy
  DMA_InitStructure.DMA_PeripheralInc = DMA_PeripheralInc_Disable;              //odczytuje z tego samego miejsca
  DMA_InitStructure.DMA_MemoryInc = DMA_MemoryInc_Enable;                       //inkrementuje wskaznik w tablicy
  DMA_InitStructure.DMA_PeripheralDataSize = DMA_PeripheralDataSize_HalfWord;//slowo=32bity, rejestr ma 16 bitow
  DMA_InitStructure.DMA_MemoryDataSize = DMA_MemoryDataSize_HalfWord;           //j.w.
  DMA_InitStructure.DMA_Mode = DMA_Mode_Circular;                               //DMA dziala w petli
  DMA_InitStructure.DMA_Priority = DMA_Priority_High;                           //priorytet
  DMA_InitStructure.DMA_M2M = DMA_M2M_Enable;                                   //kopiowanie z pamieci do pamieci
  DMA_Init(DMA1_Channel1, &DMA_InitStructure);                                  //inicjalizacja
```

**Listing 3. Konfigurowanie pinu I/O jako wejścia pull-down**

```
//hw_config.c
  GPIO_InitStructure.GPIO_Pin=GPIO_Pin_6;                   //wybranie pinu
  GPIO_InitStructure.GPIO_Mode=GPIO_Mode_IPD;               //tryb pull-down
  GPIO_InitStructure.GPIO_Speed=GPIO_Speed_50MHz;           //szybkość 50 MHz
  GPIO_Init(GPIOB,&GPIO_InitStructure);
```

**Listing 4. Wskazanie funkcji obsługi przerwania**

```
//hw_config.c
  /* Enable the EXTI9_5 Interrupt */
  NVIC_InitStructure.NVIC_IRQChannel = EXTI9_5_IRQn; //wybór funkcji obsługi
  NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
  NVIC_InitStructure.NVIC_IRQChannelCmd=ENABLE;      //włączenie obsługi
  NVIC_Init(&NVIC_InitStructure);
```

**Listing 5. Konfigurowanie wywołania przerwania (zbocza opadające)**

```
//usb_endp.c
EXTI_InitStructForClock.EXTI_Line=EXTI_Line6;               //PB6
EXTI_InitStructForClock.EXTI_Mode=EXTI_Mode_Interrupt;
EXTI_InitStructForClock.EXTI_Trigger=EXTI_Trigger_Falling;  //zbocze opadające
EXTI_InitStructForClock.EXTI_LineCmd=ENABLE;                //włączenie
                                                            //obsługi
EXTI_Init(&EXTI_InitStructForClock);
```

Pracę należy rozpocząć od skonfigurowania odpowiednich rejestrów GPIOX jako wejścia. W projekcie wykorzystano 3 rejestry: GPIOC, GPIOD oraz GPIOE. Na **listingu 1** zamieszczono procedurę konfigurującą rejestr GPIO.

Aby umożliwić pracę w dwóch opisanych wcześniej trybach odczytu stanu pinów jest konieczne skonfigurowanie DMA oraz obsługi przerwania.

W projekcie użyto 3 kanałów DMA, ponieważ monitorowanie stanu 32 pinów wymaga ciągłego odczytu 3 rejestrów GPIO. DMA działa w trybie kopiowania z adresu źródłowego pamięci do adresu końcowego w pamięci. Źródłem danych jest odpowiedni rejestr GPIO, natomiast przeznaczeniem – tablica która gromadzi dane dla danej gru-

**Listing 6. Funkcja main()**
**(1)regPointer wskazuje aktualne miejsce odczytu stanu rejestrów w tablicy wszystkich danych, po osiągnięciu końca bufora wraca na początek**
**(2)captureIsWorking steruje pętlą, odczyt uruchamiany jest w momencie odebrania konfiguracji z komputera, odczyt zatrzymywany jest kiedy zostanie zatrzaśnięty zadany stan na liniach**
**(3)ilość danych do wysłania na PC, podawane z konfiguracji, wysyłana jest określona liczba próbek przed trigger word, trigger word oraz taka sama liczba próbek po trigger word, taka metoda gwarantuje największą spójność danych**
**(4)sprawdza czy odczytany stan pinów odpowiada trigger word, nakładane maski są przesyłane z PC ponieważ tam użytkownik zaznacza których linii nie chce monitorować (stan X linii)**
**(5)po znalezieniu trigger word zatrzymywane jest DMA lub wyłączana jest obsługa przerwania(w zależności od tego którą opcję wybrano), zabezpiecza to przed nadpisaniem danych zgromadzonych w buforze danych**
**(6)po zatrzymaniu pętli, dane są wysyłane do PC przez wykorzystanie przerwania programowego, z obserwacji wynika że funkcję wysyłającą przez USB można obsłużyć tylko w funkcji przerwania**

```
//main.c
while (1) {
  regPointer=0;                                                                                          //(1)
  while(captureIsWorking==1){                                                                            //(2)
    sendDataTotalLength=2*sendlength+1;                                                                  //(3)
    sendStart=(regPointer>=sendlength) ? regPointer-sendlength : regPointer-sendlength+DataSize;
    ptrInterr=sendDataTotalLength;
    if((captureIsWorking==1) &&                                                                          //(4)
      ((registerGPIOC[regPointer] & maskGPIOC)==captureGPIOC) &&
      ((registerGPIOD[regPointer] & maskGPIOD)==captureGPIOD) &&
      ((registerGPIOE[regPointer] & maskGPIOE)==captureGPIOE)){
      if(mode==0){                                                                                       //(5)
        DMA_Cmd(DMA1_Channel1, DISABLE);
        DMA_Cmd(DMA1_Channel2, DISABLE);
        DMA_Cmd(DMA2_Channel1, DISABLE);
      }
      else{
        EXTI_InitStructForClock.EXTI_LineCmd=DISABLE;
        EXTI_Init(&EXTI_InitStructForClock);
      }
      captureIsWorking=0;
```

py wyprowadzeń. Jedna komórka tej tablicy gromadzi stany wszystkich pinów danego rejestru (rejestry GPIO są 16-bitowe). Aby umożliwić rejestrowanie większej ilości danych, DMA działa w trybie inkrementacji wskaźnika adresu przeznaczenia, a po osiągnięciu końca tablicy wraca na jej początek. Rozmiar kopiowanych danych to pół słowa (16 bitów). Warto wspomnieć, iż jest możliwość generowania przez DMA przerwań po zapełnieniu połowy i/lub całości tablicy, ale w omawianym projekcie nie było to potrzebne. Poniżej przedstawiono konfigurację jednego z kanałów DMA. W pozostałych dwóch przypadkach należy odpowiednio zmienić numer kanału lub DMA oraz adresy źródłowe i końcowe pamięci, co zaprezentowano na **listingu 2**.

Deklaracja tablicy przeznaczenia:

```
uint16_t registerGPIOC[DataSize];//
stm32_eval.h
```

Definicja rozmiaru tablicy:

```
#define DataSize 5000 //stm32_eval.h
```

Prędkość pracy urządzenia w włączonym DMA nie jest w projekcie kontrolowana. Układa działa z maksymalną prędkością (dzieląc matrycę szyn-Bus Matrix z CPU cyklicznie – na przemian CPU i DMA). Można natomiast podłączyć do urządzenia wejście zegarowe. Wtedy odczyt rejestrów będzie wykonywany w takt zegara podczas wywołania przerwania na skonfigurowanym zboczu(malejące, rosnące lub dowolne).

Jako wejście sygnału zegarowego użyto wyprowadzenia PB6 rejestru GPIOB. Do żądanego działania należy skonfigurować PB6, przerwanie oraz funkcję obsługi przerwania.

Ciągły monitoring stanu wyprowadzeń odbywa się w identyczny sposób, zarówna dla DMA, jak i dla przerwania. Całość znajduje się w funkcji *main()*, którą zamieszczono na **listingu 6**.

## Odczyt poziomów wejść i ich zapis do bufora danych

W wypadku DMA poza konfiguracją nie trzeba ręcznie pisać żadnej obsługi. Działające DMA kopiuje w podanym trybie ze źródła do celu. Dodatkowo można wykorzystać przerwania po zapełnieniu połowy i/lub całości bufora ale tutaj nie jest to potrzebne. W przypadku wyzwalania na pinie zegarowym należy napisać funkcję obsługi przerwania (**listing 7**).

## Komunikacja przez USB

Konfiguracja interfejsu USB jest już prawidłowo zaimplementowana w projekcie CustomHID. Do przesyłania danych niezbędne jest zdefiniowanej deskryptorów dla urządzenia oraz endpointów. Deskryptor analizatora zamieszczono na **listingu 8**. Najważniejsze pola to VendorID oraz ProductID przez które urządzenie jest identyfikowane w systemie komputera do którego jest podłączone.

**Listing 6. c.d.**

```
      whichReg=1;
      STM_EVAL_LEDOff(LED1);
      STM_EVAL_LEDOff(LED2);
      while(1){                                                    //(6)
        NVIC_SetPendingIRQ(TIM3_IRQn);
        if(sendDataTotalLength==0) break;
        sendDataTotalLength--;
      }
      regPtrInterr=0;
    }
    else
    {
      regPointer=(regPointer + 1) % DataSize;
    }
  }
}
```

**Listing 7. Funkcja obsługi przerwania**
**(1)odczyty 16 bitowych rejestrów GPIO, odczyt odbywa się przez odwołanie do adresu zapisanego w odpowiedniej strukturze GPIOX, biblioteka dla STM zawiera funkcje odczytujące ale ponieważ sprawdzają one warunki poprawności są bardzo wolne( w projekcie po rezygnacji z funkcji bibiotecznych i wykorzystanie odwołania do adresu uzyskano przyrost szybkości odczytu rejestrów o 2 MHz!)**
**(2)przesunięcie wskaźnika na następną pozycję w buforze**
**(3)czyszczenie flagi przerwania, bardzo ważne!, bez tego obsługa przerwania nie działa prawidłowo**

```
//stm32f10x_it.c
void EXTI9_5_IRQHandler(void)
{
//STM_EVAL_LEDOn(LED1);
  registerGPIOC[regPtrInterr]=(uint16_t)GPIOC->IDR;                //(1)
  registerGPIOD[regPtrInterr]=(uint16_t)GPIOD->IDR;
  registerGPIOE[regPtrInterr]=(uint16_t)GPIOE->IDR;
  regPtrInterr=(regPtrInterr + 1) % DataSize;                      //(2)
  EXTI_ClearITPendingBit(EXTI_Line6);                              //(3)
}
```

**Listing 8. Deskryptor USB**

```
//usb_desc.c
/* USB Standard Device Descriptor */
const uint8_t CustomHID_DeviceDescriptor[CUSTOMHID_SIZ_DEVICE_DESC] =
  {
    0x12,                                   //bLength
    USB_DEVICE_DESCRIPTOR_TYPE,             //bDescriptorType
    0x00,                                   //bcdUSB
    0x02,
    0x00,                                   //bDeviceClass
    0x00,                                   //bDeviceSubClass
    0x00,                                   //bDeviceProtocol
    0x40,                                   //bMaxPacketSize40
    0x83,                                   //idVendor (0x0483)
    0x04,
    0x50,                                   //idProduct = 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
```

**Listing 9. Konfiguracja deskryptora**

```
//usb_desc.c
    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)
```

**Listing 10. Wysyłanie danych przez USB za pomocą przerwania**

```
//hw_config.c
  NVIC_PriorityGroupConfig(NVIC_PriorityGroup_1);          //wybranie grupy
                                                           //priorytetow
                                                           //konfiguracja i
                                                           //wlaczenie
                                                           //przerwania TIM3
  NVIC_InitStructure.NVIC_IRQChannel=TIM3_IRQn;
  NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority=0;
  NVIC_InitStructure.NVIC_IRQChannelSubPriority=0;
  NVIC_InitStructure.NVIC_IRQChannelCmd=ENABLE;
  NVIC_Init(&NVIC_InitStructure);
```

**Listing 11. Zmienne sterujące odczytem**
**(1)odczyt danych, dane przesyłane są w pakietach 2*8 bitów, w pierwszym bajcie znajduje się id definiowane w deskryptorze, w drugim właściwe dane**
**(2)odczyt trigger word, ponieważ rejestry są 16 bitowe, przesłanie stanów pojedynczego rejestru odbywa się przez przesłanie osobno części HIGH i LOW**
**(3)odczyt maski nakładanej na rejestry podczas szukania trigger word, przesłanie ich jest konieczne żeby użytkownik mógł wyłączyć monitoring określonych linii bez fizycznego odłączania sondy**
**(4)rozpoczęcie pracy, włączenie DMA lub przerwania oraz uruchomienie głównej pętli w main**

```
//usb_endp.c
void EP1_OUT_Callback(void)
{
  captureIsWorking=0;
  USB_SIL_Read(EP1_OUT, Receive_Buffer);
//(1)
  if(nowmask==0){ //wczytuje trigger word
    switch (Receive_Buffer[0])
//(2)
    {
    case 1:
      if(whichReg==1){
        if(whichPart==0){
          captureGPIOC=Receive_Buffer[1];
          ++whichPart;
        }
        else if(whichPart==1){
          captureTEMPH=Receive_Buffer[1];
          captureGPIOC |=(captureTEMPH << 8);
          --whichPart;
          ++whichReg;
        }
      }
      break;
    case 2:
      if(whichReg==2){
        if(whichPart==0){
          captureGPIOD=Receive_Buffer[1];
          ++whichPart;
        }
        else if(whichPart==1){
          captureTEMPH=Receive_Buffer[1];
          captureGPIOD |=(captureTEMPH << 8);
          --whichPart;
          ++whichReg;
        }
      }
      break;
    case 3:
      if(whichReg==3){
        if(whichPart==0){
          captureGPIOE=Receive_Buffer[1];
          ++whichPart;
        }
        else if(whichPart==1){
          captureTEMPH=Receive_Buffer[1];
          captureGPIOE |=(captureTEMPH << 8);
          --whichPart;
          whichReg=1;
          nowmask=1;
        }
      }
      break;
    }
  }
  else if(nowmask==1){ //wczytuje maski
    switch (Receive_Buffer[0]) //(3)
    {
      case 1:
        if(whichReg==1){
          if(whichPart==0){
            maskGPIOC=Receive_Buffer[1];
            ++whichPart;
          }
          else if(whichPart==1){
            captureTEMPH=Receive_Buffer[1];
            maskGPIOC |=(captureTEMPH << 8);
            --whichPart;
            ++whichReg;
          }
        }
```

**Listing 11. c.d.**

```
        break;
      case 2:
        if(whichReg==2){
          if(whichPart==0){
            maskGPIOD=Receive_Buffer[1];
            ++whichPart;
          }
          else if(whichPart==1){
            captureTEMPH=Receive_Buffer[1];
            maskGPIOD |=(captureTEMPH << 8);
            --whichPart;
            ++whichReg;
          }
        }
        break;
      case 3:
        if(whichReg==3){
          if(whichPart==0){
            maskGPIOE=Receive_Buffer[1];
            ++whichPart;
          }
          else if(whichPart==1){
            captureTEMPH=Receive_Buffer[1];
            maskGPIOE |=(captureTEMPH << 8);
            --whichPart;
            whichReg=1;
          }
        }
        break;
      case 4: //(4)
      for(i=0;i<DataSize;++i){
        registerGPIOC[i]=0;
        registerGPIOD[i]=0;
        registerGPIOE[i]=0;
      }
      mode=Receive_Buffer[1] & 3;
      sendlength=probesCount[Receive_Buffer[1] >> 2];
      switch(mode){
        case 0: //DMA
        DMA_Cmd(DMA1_Channel1, ENABLE);
        DMA_Cmd(DMA1_Channel2, ENABLE);
        DMA_Cmd(DMA2_Channel1, ENABLE);
        captureIsWorking=1;
        break;
      case 1: //zegar, opadajace
        EXTI_InitStructForClock.EXTI_Line=EXTI_Line6;
        EXTI_InitStructForClock.EXTI_Mode=EXTI_Mode_
Interrupt;
        EXTI_InitStructForClock.EXTI_Trigger=EXTI_Trigger_
Falling;
        EXTI_InitStructForClock.EXTI_LineCmd=ENABLE;
        EXTI_Init(&EXTI_InitStructForClock);
        captureIsWorking=1;
        break;
        case 2: //zegar, rosnace
        EXTI_InitStructForClock.EXTI_Line=EXTI_Line6;
        EXTI_InitStructForClock.EXTI_Mode=EXTI_Mode_
Interrupt;
        EXTI_InitStructForClock.EXTI_Trigger=EXTI_Trigger_
Rising;
        EXTI_InitStructForClock.EXTI_LineCmd=ENABLE;
        EXTI_Init(&EXTI_InitStructForClock);
        captureIsWorking=1;
        break;
        case 3: //zegar, dowolne
        EXTI_InitStructForClock.EXTI_Line=EXTI_Line6;
        EXTI_InitStructForClock.EXTI_Mode=EXTI_Mode_
Interrupt;
        EXTI_InitStructForClock.EXTI_Trigger=EXTI_Trigger_
Rising_Falling;
        EXTI_InitStructForClock.EXTI_LineCmd=ENABLE;
        EXTI_Init(&EXTI_InitStructForClock);
        captureIsWorking=1;
        break;
      }
      nowmask=0;
      STM_EVAL_LEDOff(LED1);
      STM_EVAL_LEDOn(LED2);
      break;
    }
  }
}
```

Do odbierania danych wykorzystano deskryptor endpointa dla diod LED (np. **listing 9**). Konieczne jest określenie pola REPORT_ID. Każdy pakiet przesłany przez USB musi być opatrzony odpowiednim identyfikatorem, w przeciwnym wypadku wysłane dane nie zostaną odebrane przez aplikację.

Do wysyłania danych wykorzystano deskryptory przycisku, poniżej konfiguracja jednego z deskryptorów:

Tu również należy zdefiniować identyfikator (REPORT_ID) z jakim zostanie przesłany pakiet z PC do urządzenia. Do wysyłania danych przez USB skonfigurowano przerwanie programowe (**listing 10**). Po odpowiednim skonfigurowaniu należy zaimplementować obsługę zapisu i odczytu danych.

Odbiór danych odbywa się w funkcji obsługi endpointa. Po konfiguracji USB nie trzeba podejmować żadnych dodatkowych działań, po pojawieniu się danych na USB obsługa automatycznie przenosi się do tej funkcji. Funkcja zawiera odczyta konfiguracji przesłanej z PC, włączenie DMA lub przerwania oraz uruchomienie pętli w funkcji main. Jak wspomniano, do odbioru/wysyłania danych jest konieczne zdefiniowanie deskryptorów. Niestety, ich liczba jest ograniczona. Ponieważ konieczne było wielokrotne użycie tych samych deskryptorów, zostały zastosowane odpowiednie zmienne sterujące odczytem (**listing 11**). Z obserwacji wynika że nie ma pewności przesłania przez USB wszystkich danych (tzn pakiety mogą zostać zagubione), żeby uniknąć tego problemu należy wprowadzać opóźnienia pomiędzy wysyłanymi danymi z PC. Wysyłanie danych odbywa się funkcji obsługi przerwania programowego zaprezentowanej na **listingu 12**.

**Listing 12. Funkcja obsługi przerwania Timera 3**

```
//stm32f10x_it.c
void TIM3_IRQHandler(void){
  STM_EVAL_LEDOff(LED2);
  if(ptrInterr==0){
    Send_Buffer[0]=0x06;
    Send_Buffer[1]=0x06;
    USB_SIL_Write(EP1_IN, (uint8_t*) Send_Buffer, 2);
    STM_EVAL_LEDOn(LED1);
    STM_EVAL_LEDOff(LED2);
  } else{
    Send_Registers[0]=0x01;
    Send_Registers[1]=registerGPIOC[sendStart];
    Send_Registers[2]=0x02;
    Send_Registers[3]=registerGPIOC[sendStart]>>8;
    Send_Registers[4]=0x03;
    Send_Registers[5]=registerGPIOD[sendStart];
    Send_Registers[6]=0x04;
    Send_Registers[7]=registerGPIOD[sendStart]>>8;;
    Send_Registers[8]=0x05;
    Send_Registers[9]=registerGPIOE[sendStart];;
    USB_SIL_Write(EP1_IN, (uint8_t*) Send_Registers, 10);
    --ptrInterr;
    sendStart=(sendStart+1) % DataSize;
  }
}
```

**Listing 13. Wprowadzanie opóźnienień pomiędzy pakietami**

```
//usb_desc.c
const uint8_t CustomHID_ConfigDescriptor[CUSTOMHID_SIZ_CONFIG_DESC] =
{
...
  0x01,  /* bInterval: Polling Interval (1 ms) */
}
```

Paczki przesyłane z urządzenia do PC to także 2*8 bitów czyli osobno są przesyłane części HIGH i LOW. Bardzo ważne jest nadanie w pierwszym elemencie tablicy odpowiedniego id zdefiniowanego w deskryptorach. Bez właściwego ID dane fizycznie pojawią się na porcie USB komputera ale nie zostaną odebrane przez aplikację. Nie ma gwarancji że wszystkie dane dotrą więc trzeba wprowadzać opóźnienia między pakietami (**listing 13**).

W projekcie wystarczyło opóźnienie 1 ms. Teoretycznie większe opóźnienie daję lepszą gwarancję ale znacznie wydłuża wysyłanie danych (standardowo interwał ustawiony na 32 ms!), dlatego należy odpowiednio dobrać czas.

## Podsumowanie

W tym artykule podano podstawy wiedzy niezbędnej do wykonania aplikacji. W kolejnym numerze EP zostanie opisane oprogramowanie analizatora przeznaczone dla STM32 Butterfly, natomiast w następnym dla komputera PC.

**Jarosław Dąbrowski**
**Sebastian Feduniak**