Spis treści

1. V-USB od A do Z – „po polsku”

„**V-USB**” to po prostu nazwa biblioteki w języku C (AVR GCC), która przeznaczona jest dla zwykłych 8-bitowych mikrokontrolerów AVR (nieposiadających sprzętowej obsługi komunikacji USB). Została ona stworzona przez Objective Development i można ją pobrać ze strony projektu:

http://www.obdev.at/products/vusb/index.html

1.1. Podstawy połączeń elektrycznych V-USB

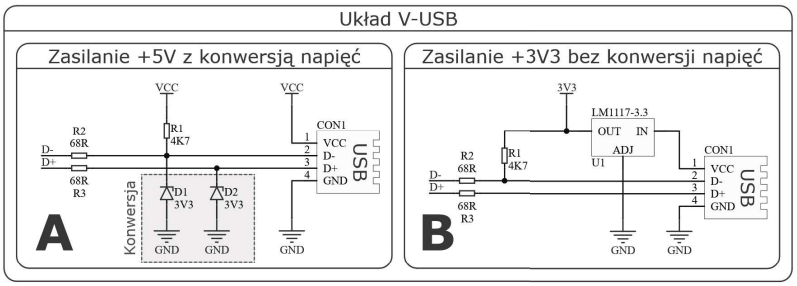
Aby można było korzystać z biblioteki V-USB, należy zapewnić minimum połączeń pomiędzy mikrokontrolerem a złączem USB, jak np. poprawne zasilanie układu docelowego (urządzenia) oraz konwersję napięć, gdy układ docelowy zasilany będzie napięciem +5 V. W przypadku gdy układ docelowy nie będzie zasilany ze złącza USB, wystarczy wykorzystać linie **D+** i **D-** oraz bezwzględnie masę **GND**.

1.1.1. Sposoby zasilania urządzenia/mikrokontrolera ze złącza USB

Na rysunku 1 przedstawiono schematy prezentujące dwa podstawowe warianty zasilania sterownika ze złącza USB. Podzielono ja na dwie części: A oraz B.

W części **A** rysunku 1 przedstawiono najczęściej spotykane rozwiązanie wykorzystujące bezpośrednio napięcie zasilania +5 V (dla mikrokontrolera i jego peryferiów) dostępne w złączu USB. Z uwagi na to, że na liniach sygnałowych D+ i D- panuje poziom napięć +3,3 V, należy bezwzględnie zastosować układ konwersji napięć, zrealizowany na podstawie dwów diod Zenera (C3V3): D1 oraz D2.

Uwaga! Warto zastosować diody Zenera o wartości napięcia 3,3 V. *Spotykane w internecie rozwiązanie z diodami Zenera z napięciem +3,6 V niejednokrotnie powoduje problemy przy współpracy V-USB z niektórymi komputerami stacjonarnymi, a najczęściej z laptopami.*



Rysunek 1. Przykłady zasilania z USB

W części **B** rysunku 1 przedstawiono poprawny sposób zasilania +3,3 V z USB. Polega on na zastosowaniu stabilizatora (najkorzystniej) typu LDO (Low Dop Output), np. LM1117-3.3 (albo przetwornicy), który zapewni odpowiednią wydajność prądową dla układu z mikrokontrolerem i jego peryferiami. W przypadku zasilania mikrokontrolera napięciem +3,3 V nie trzeba stosować układu konwersji napięć z wykorzystaniem diod Zenera D1 i D2 (przedstawionego w części A).

Obie części rysunku 1 posiadają wspólne elementy, takie jak:

1. Rezystor **PULL-UP** (R1,R1*) o wartości 4,7 K, podciągający linię USB **D-** do VCC (podciąganie linii D- do VCC oznacza dla hosta, że podłączono urządzenie typu low speed). W części A rysunku 1 rezystor R1 podłączony jest bezpośrednio do linii zasilania USB +5 V z uwagi na zastosowaną konwersję napięć, w części B (R1*) podłączony jest zaś do napięcia wyjściowego stabilizatora +3,3 V.
2. Rezystory szeregowe na liniach sygnałowych D+ i D- (R2,R2*, R3,R3*) z rezystancją 68R.

Zdecydowanie najgorszym podejściem do zasilania jest zastosowanie zwykłych diod krzemowych D1 i D2, jak zaprezentowano na rysunku 2. Tego typu układ powinien w teorii zapewnić obniżenie napięcia z linii USB +5 V do poziomu ok. +3,3 V. W praktyce powoduje on mnóstwo kłopotów związanych z różną wartością spadku napięcia na diodach w zależności od łącznego poboru prądu przez cały układ.

Nie ma się jednak czym przejmować, ponieważ to już w zasadzie ostatni błąd na drodze przed finalnym, pierwszym prawidłowym uruchomieniem projektu. Błąd ten mówi tylko tyle, że brakuje funkcji o nazwie **`usbFunctionSetup`** gdzieś w naszym projekcie (w pliku `main.c`). Za chwilę dowiesz się, co to za funkcja, do czego służy oraz że musi być bezwzględnie użyta wewnątrz naszego projektu. Stanowić to będzie prawidłową i minimalną konfigurację projektu do pracy z biblioteką V-USB.

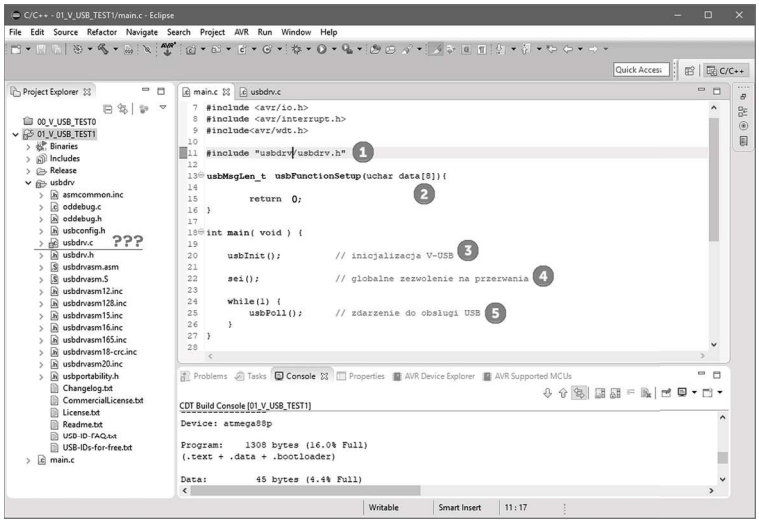
1.2.3. Pierwszy kod urządzenia opartego na V-USB

Możemy przystąpić do napisania najprostszej aplikacji dla mikrokontrolera. Zanim to zrobimy, spróbujmy wspólnie ustalić założenia i działania niezbędne do przygotowania nie tylko aplikacji dla mikrokontrolera, bo ona nie będzie warta przysłowiowego funta kłaków, jeśli nie napiszemy za chwilę kolejnej aplikacji, ale tym razem na komputer PC, która będzie umiała, kolokwialnie mówiąc, „rozmawiać" z naszą aplikacją zaszytą w mikrokontrolerze.

1. Zadaniem aplikacji testowej na AVR będzie włączanie/wyłączanie diod: LED1 i LED2, a także przesyłanie jakiegoś napisu do komputera, na żądanie wysłane ze strony naszej aplikacji na komputerze. W związku z tym musimy mieć świadomość, że w tym przykładzie to komputer, w zasadzie aplikacja napisana na komputerze, będzie inicjował działania, czyli żądał włączania lub wyłączania diod LED, a także zwracania informacji o stanie naszego urządzenia.
2. Instalacja driverów dla systemu Windows potrzebnych do sterownika.
3. Zadaniem aplikacji testowej na PC wykorzystującej bibliotekę LibUSB będzie wysyłanie żądań/zapytań za pomocą USB do naszego układu opartego na mikrokontrolerze AVR.

Uwaga! W materiałach dołączonych do książki znajduje się aplikacja testowa do każdego z opisanych ćwiczeń. W dalszej części publikacji zajmiemy się również nauką pisania tego typu aplikacji we własnym zakresie. Użyjemy do tego celu środowiska Delphi.

Na rysunku 10 przedstawiono widok projektu po pierwszej udanej kompilacji, która nie zwraca już żadnych błędów poza drobnym komunikatem samego środowiska Eclipse. Komunikat oznaczono trzema znakami zapytania (zostanie to omówione w dalszej części rozdziału). W punktach na rysunku 10 przedstawiono, z jakich elementów powinien składać się szkielet programu, gdy zamierzamy korzystać z biblioteki V-USB.



Rysunek 10. Szkielet programu w pliku main.c

Uwaga! Zanim przystąpimy do pisania pierwszego kodu źródłowego z wykorzystaniem biblioteki V-USB, należy zwrócić uwagę, aby nigdy w kodzie nie definiować stałej dosłownej odpowiedzialnej za częstotliwość taktowania F_CPU za pomocą np. *#define F_CPU 12000000*. Częstotliwość taktowania mikrokontrolera zawsze należy zdefiniować w ustawieniach projektu w ramach środowiska programistycznego, np. Eclipse lub Atmel Studio. Jeśli nie posługujemy się tego typu środowiskami, zawsze definicję częstotliwości można dodać do pliku sterującego kompilacją „makefile”.

Dla większej przejrzystości zamieszczono również kod źródłowy.

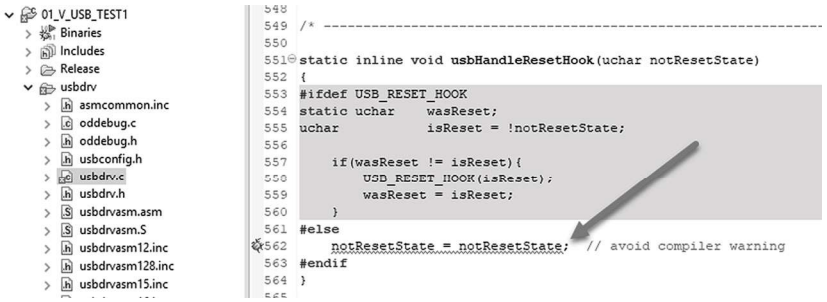
```
#include <avr/io.h>
#include <avr/interrupt.h>
#include <avr/wdt.h>
#include "usbdrv/usbdrv.h"        // (1) dołączenie biblioteki

usbMsgLen_t usbFunctionSetup( uchar data[8] ) {     // (2)
    return  0;
}
int main( void ) {
    usbInit();                    // (3) inicjalizacja V-USB
```

```
    sei();              // (4) globalne zezwolenie na przerwania
    while(1) {
        usbPoll();          // (5) zdarzenie do obsługi USB
    }
}
```

Poniżej opisano po kolei najważniejsze fragmenty omawianego kodu źródłowego:

1. Dołączamy bibliotekę V-USB do projektu za pomocą dyrektywy #include.
2. Koniecznie dodajemy do naszego projektu funkcję `usbFunctionSetup`.
3. Inicjalizujemy bibliotekę V-USB.
4. Koniecznie odblokowujemy globalne przerwania.
5. W pętli głównej cyklicznie wywołujemy funkcję (główne zdarzenie) biblioteki V-USB.

Tak napisany program kompiluje się już poprawnie, ponieważ dodaliśmy wymaganą funkcję `usbFunctionSetup`, która jest praktycznie wywoływana za każdym razem, gdy cokolwiek zdarzy się na magistrali USB i zostanie to wykryte przez zdarzenie `usbPoll()`. Za chwilę przejdziemy do omawiania funkcji `usbFunctionSetup`, ale najpierw wyjaśnimy, co oznaczają trzy znaki zapytania z rysunku 10 – dlaczego Eclipse zgłasza to jako swój błąd. Na szczęście ten fakt nie ma żadnego znaczenia, jeśli chodzi o prawidłową kompilację programu. Pomimo to warto zawsze doprowadzać kod źródłowy do takiego stanu, aby nie były widoczne żadne ostrzeżenia, czy to z poziomu kompilatora, czy też Eclipse.

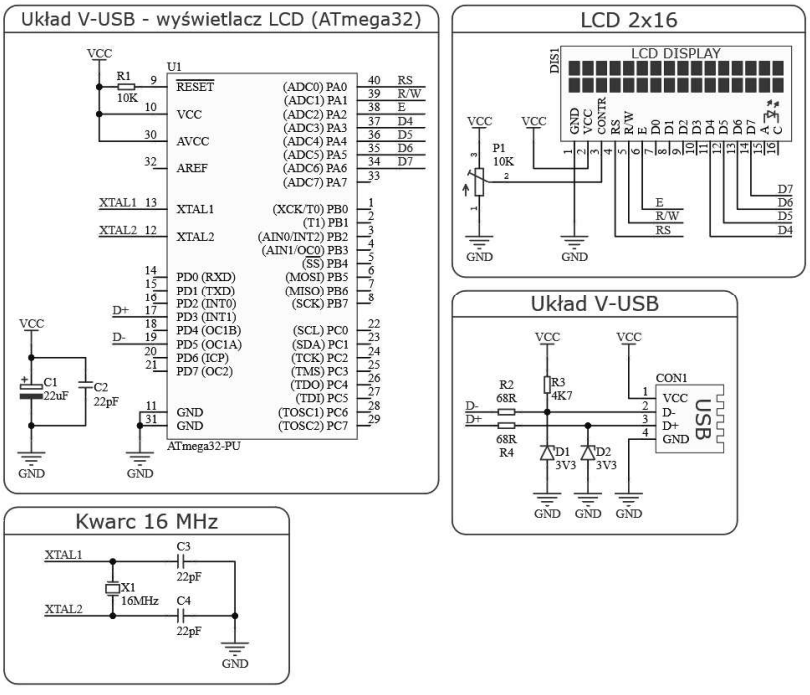


Rysunek 11. Problem zgłaszany przez środowisko Eclipse, a nie kompilator

Problemem dla Eclipse okazuje się zapis w jednej z funkcji w pliku `usbdrv.c`, który tak naprawdę dodany został z uwagi na to, aby nie generować tzw. ostrzeżeń (warnings) kompilatora. Na rysunku 11 widać problematyczną linię dla Eclipse.

1.2.10. Funkcja `usbFunctionWrite()` – odbiór danych z hosta – obsługa LCD

Do tej pory poznaliśmy sposób, za pomocą którego można było przesłać dane z komputera do mikrokontrolera za pomocą V-USB, korzystając z zaledwie 7 bajtów (za pomocą pakietu typu „control message", co było dość sporym ograniczeniem). Wyobraźmy sobie, że zechcemy wysyłać dowolne teksty wprost z aplikacji na komputerze, które mają się pojawiać na dowolnym wyświetlaczu podłączonym do mikrokontrolera. Tak się składa, że na zestawie ATB mamy do dyspozycji prosty dwuwierszowy wyświetlacz alfanumeryczny LCD 2x16. Wystarczy zatem dołączyć do projektu bibliotekę do tego typu wyświetlacza z jednej z naszych poprzednich publikacji wydawnictwa Atnel, a następnie zaimplementować funkcję **`usbFunctionWrite()`**. Ćwiczenie proste, ale już samo podłączanie wyświetlacza na płytce stykowej może często przysparzać wielu problemów, dlatego na zestawie ATB realizacja zadania zajmie dosłownie kilka chwil. Tym, którzy nie dysponują jeszcze zestawem uruchomieniowym ATB, przedstawiamy schemat połączeń do tego ćwiczenia na rysunku 22.



Rysunek 22. Schemat układu V-USB z wyświetlaczem LCD 2x16

Jak widać na rysunku 22, tym razem układ oparty na bibliotece **V-USB** zrealizowano na mikrokontrolerze **ATmega32**, gdzie do całego portu A zostały podłączone linie sterujące wyświetlaczem **LCD** oraz jego podświetleniem. Tor wejściowy V-USB został podłączony do pinów **PD3**(INT1) oraz **PD5** (zgodnie z zasadą przedstawioną na rysunku 20, punkt 3). Dla odmiany do taktowania mikrokontrolera wykorzystano rezonator kwarcowy 16 MHz.

W związku ze zmianą taktowania mikrokontrolera w porównaniu z wcześniej omawianymi projektami należy zmienić odpowiednie ustawienie w pliku `usbconfig.h`.

```
/* ---------------------- Hardware Config ------------------------ */
#define USB_CFG_CLOCK_KHZ       16000     /* 16 MHz */
```

Poza tym dokonamy zmiany numerów VID oraz PID, jak niżej:

```
/* ---------------------- Device Description --------------------- */
#define  USB_CFG_VENDOR_ID      0xc0, 0x15 /* = 0x15c0 */
#define  USB_CFG_DEVICE_ID      0xdc, 0x06 /* = 0x06dc */
```

Kolorem szarym i nieco większym fontem zaznaczono zmiany w ramach VID oraz PID w porównaniu z tymi z poprzednich ćwiczeń, będących odpowiednikami dla programatorów USBasp. Można teraz użyć programu **MkAvrCalculator** w pełnej wersji, który pozwoli zainstalować sterowniki dla nowego urządzenia, które pojawi się na magistrali USB. Tym razem nazwa projektu (ćwiczenia) na nośniku dołączonym do książki to: **04_V_USB_ATB_TEST4**. Naturalnie każdy może w ramach własnego kodu źródłowego zmienić nazwę nadaną przez nas na dowolną, wg własnego uznania, pamiętając również o poprawnym podaniu liczby znaków nazwy, np. „**V-USB-TEST**”:

```
/* -------------------- Device Description -------------------- */
#define USB_CFG_DEVICE_NAME     'V','-','U','S','B','-
','T','E','S','T'
#define USB_CFG_DEVICE_NAME_LEN 10
```

Tym razem przykładowa nazwa liczyć będzie 10 znaków ASCII.

Skoro nowy projekt ma umożliwiać wyświetlanie tekstów, to należy przyjąć założenia dotyczące przygotowania chociaż kilku komend, które zostaną do tego celu wykorzystane. Komendy powinny uwzględniać nie tylko czyszczenie całego ekranu LCD, lecz także np. każdej jego osobnej linii. Potrzebna będzie komenda do ustawiania kursora na LCD, czyli miejsca, od którego mają

się wyświetlić przesłane znaki oraz komenda odpowiadająca za przesłanie samego tekstu.

```
// usb user requests
enum {
     cmd_cls,       // kasowanie całego ekranu
     cmd_cls1,      // kasowanie górnej linii
     cmd_cls2,      // kasowanie dolnej linii
     cmd_locate,    // ustawienie kursora w pozycji Y,X
     cmd_text,      // przesłanie tekstu do LCD
     cmd_bck_on,    // włączanie/wyłączanie podświetlenia LCD
};
```

Łatwo zauważyć, że występuje tu pewna nadmiarowość komend, ponieważ trzy pierwsze można byłoby sprowadzić do jednej, ale z możliwością przekazywania do niej argumentu, od którego będzie zależeć, którą z trzech operacji będzie wykonywać. Od tego momentu każdy będzie mógł już we własnym zakresie wprowadzać do kodu ćwiczenia własne modyfikacje i rozbudowywać go wg własnych potrzeb. Na końcu jest jeszcze komenda odpowiedzialna za włączanie i wyłączanie podświetlenia LCD. Trzy ostatnie komendy będą wymagały dodatkowych danych, przy czym tylko jedna z nich (przedostatnia) zostanie wykorzystana w kodzie źródłowym nowej funkcji **`usbFunctionWrite()`**. Poniżej kod źródłowy z omawianym przykładem.

```
#include <avr/io.h>
#include <avr/interrupt.h>
#include <string.h>
#include <util/delay.h>
#include <avr/wdt.h>
#include "LCD/lcd44780.h"
#include "usbdrv/usbdrv.h"

// usb user requests – komendy
enum {
     cmd_cls,
     cmd_cls1,
     cmd_cls2,
     cmd_locate,
     cmd_text,
     cmd_bck_on,
};
uint8_t data_length;
```

```
//---- LCD Backlight macros ----------
#define LCD_BK          (1<<PA7)
#define LCD_BK_OFF PORTA &= ~LCD_BK
#define LCD_BK_ON  PORTA |= LCD_BK
#define LCD_BK_TOG      PORTA ^= LCD_BK

// nagłówek funkcji do wymuszenia reenumeracji na USB
void usb_reset( void );
usbMsgLen_t usbFunctionSetup( uchar data[8] ) {
     usbMsgLen_t len = 0;
     usbRequest_t * urq = (usbRequest_t*)data;
     switch( urq->bRequest ) {
          case cmd_cls:
               lcd_cls();
               break;
          case cmd_cls1:
               lcd_locate(0,0);
               for( uint8_t i=0; i<16; i++ ) lcd_char( ' ' );
               break;
          case cmd_cls2:
               lcd_locate(1,0);
               for( uint8_t i=0; i<16; i++ ) lcd_char( ' ' );
               break;
          case cmd_locate:
               lcd_locate( urq->wValue.bytes[0], urq->wValue.
bytes[1] );
               break;
          case cmd_text:
               data_length = urq->wLength.word; // liczba znaków
               return USB_NO_MSG;  // ciąg dalszy w usbFunctionWrite()
               break;
          case cmd_bck_on:
               if( urq->wValue.word ) LCD_BK_ON;
               else LCD_BK_OFF;
               break;
     }
    return  len;
}
uchar usbFunctionWrite( uchar * data, uchar len ) {
     char buf[9];                    // podręczny bufor
     memcpy( buf, data, len );       // kopiuj dane do bufora
     buf[len] = 0;                   // ostatni znak w buforze = 0
```

3. Deskryptory urządzeń USB – podstawy

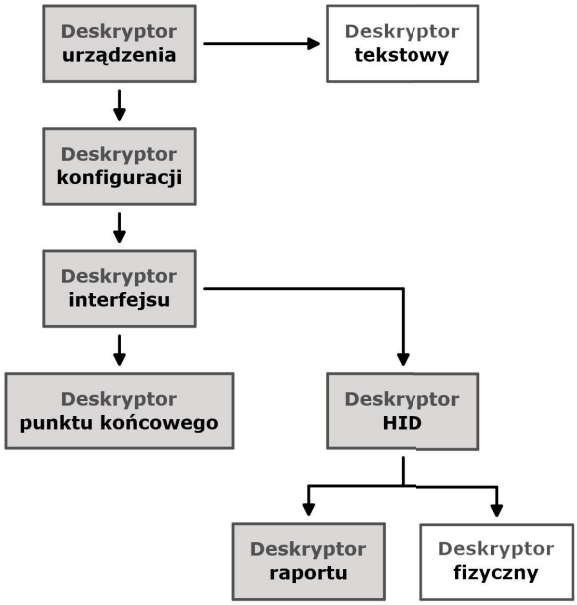
Proces rozpoznania, wykrycia nowego urządzenia podłączanego do magistrali USB realizowany jest w dwóch etapach. Pierwszy działa na poziomie elektrycznym i realizowany jest przez podciągnięcie linii D- do VCC dla urządzeń low speed lub linii D+ do VCC dla urządzeń high speed. Tuż po wykryciu przez komputer (host) nowo podłączonego urządzenia pracującego w standardzie USB rozpoczyna się drugi etap. Host pobiera od wykrytego urządzenia informacje, które są niezbędne do jego prawidłowego działania. Każde urządzenie zgodne ze standardem USB posiada zapisane w swojej pamięci tzw. deskryptory, które umożliwiają rozpoznanie przez host jego rodzaju oraz funkcji, jakie ono ma spełniać. Deskryptory są po prostu krótkimi paczkami bajtów, przesyłanymi w odpowiedniej kolejności do hosta. To właśnie dzięki nim host dowiaduje się m.in.:

- Do jakiej klasy i podklasy zalicza się urządzenie?
- Z jaką prędkością pracuje?
- Jakie są numery VID i PID nowo podłączonego urządzenia?
- Jakiego rodzaju i w jaki sposób będą przesyłane informacje?

Wspomniane zagadnienia stanowią niewielki procent wszystkich informacji zapisanych w deskryptorach. Ze względu na sporą ilość informacji o urządzeniu występuje kilka rodzajów deskryptorów, które uporządkowano w standardzie USB w sposób hierarchiczny. Strukturę przedstawiono na rysunku 31.

Na rysunku 31 białym kolorem zaznaczono deskryptory opcjonalne, których wykorzystanie nie jest wymagane do prawidłowego działania urządzenia. Pozostałe deskryptory są niezbędne, jeśli jest to urządzenie typu HID. W przypadku gdy jest to inny rodzaj urządzenia, mogą być pominięte również deskryptory: deskryptor HID, deskryptor raportu i deskryptor fizyczny. Każdy deskryptor stanowi pewien ściśle uporządkowany pakiet bajtów i bitów zawierających informacje o urządzeniu. Rolą programisty jest przygotowanie każdego pakietu, szczególnie w przypadku deskryptorów zaznaczonych kolorem szarym na rysunku 31. Deskryptor raportu wymaga specjalnej uwagi, jeśli chodzi o urządzenia typu HID, ponieważ jego zawartość definiuje ostatecznie i szczegółowo funkcje urządzenia. W dalszych częściach tego rozdziału opisano poszczególne rodzaje deskryptorów (przedstawionych na rysunku 31),

uwzględniając opis każdego z pakietów, włącznie z opisem, za co są odpowiedzialne poszczególne bajty i bity.



Rysunek 31. Struktura deskryptorów urządzenia USB

3.1. Deskryptor urządzenia (ang. Device Descriptor)

Na szczycie struktury przedstawionej na rysunku 31 widnieje deskryptor urządzenia, który przechowuje najważniejsze informacje pozwalające hostowi na przydzielenie odpowiedniego sterownika do systemu operacyjnego (jest to pewnego rodzaju podstawowa „wizytówka" urządzenia podłączanego do magistrali USB). Zawarte są w nim informacje takie jak m.in. klasa i podklasa urządzenia, numery VID i PID, liczba możliwych konfiguracji urządzenia oraz opis producenta lub produktu. **Urządzenie może posiadać tylko jeden deskryptor** tego typu. W tabeli 1 przedstawiono wszystkie parametry przechowywane w omawianym deskryptorze.

Tabela 1. Struktura deskryptora urządzenia – Device Descriptor

Nazwa pola	Offset (bajt)	Rozmiar (bajt)	Typ pola	Opis
bLength	0	1	Liczba	Rozmiar deskryptora urządzenia, wyrażany w bajtach.
bDescriptorType	1	1	Stała	Dla deskryptora urządzenia to pole zawsze przyjmuje wartość 0x01.
bcdUSB	2	2	BCD	Identyfikator wersji specyfikacji USB, z którą zgodne jest urządzenie. Dla wersji 1.1 wartość tego pola to 0x0110, dla wersji 2.0 – 0x0200.
bDeviceClass	4	1	Klasa	Kod klasy urządzenia. – 0x00: każdy interfejs w danej konfiguracji działa niezależnie i określa swoją własną klasę. – 0x01 – 0xFE: kod klasy (spis kodów klas znajduje się na stronie usb.org). – 0xFF: typ klasy jest określony przez producenta urządzenia (vendor specific).
bDeviceSubClass	5	1	Subklasa	Kod subklasy urządzenia. Wartości tego pola są zależne od bDeviceClass: – Dla wartości 0x00 bieżące pole przyjmuje wartość 0. – Dla wartości od 0x01 do 0xFE kod subklasy można znaleźć na stronie usb.org. – Dla wartości 0xFF typ subklasy jest określony przez producenta urządzenia (vendor specific).
bDeviceProtocol	6	1	Protokół	Kod protokołu urządzenia. Wartości tego pola są zależne od bDeviceClass i bDeviceSubClass: – Dla wartości 0x00 urządzenie nie korzysta z protokołu określonego przez bDeviceClass, ale może bazować na protokołach poszczególnych interfejsów. – Dla wartości od 0x01 do 0xFE kod protokołu można znaleźć na stronie usb.org. – Dla wartości 0xFF typ protokołu jest określony przez producenta urządzenia (vendor specific).

Nazwa pola	Offset (bajt)	Rozmiar (bajt)	Typ pola	Opis
bMaxPacketSize0	7	1	Liczba	Maksymalny rozmiar pakietu dla podstawowego punktu końcowego – endpoint 0. To pole może przyjmować następujące wartości: 8, 16, 32 i 64 (bajty). Dla urządzeń low speed – maksymalnie 8.
idVendor	8	2	ID	Identyfikator producenta (VID). Przyznawany przez organizację na stronie usb.org.
idProduct	10	2	ID	Identyfikator produktu. Wartość określana przez producenta urządzenia.
bcdDevice	12	2	BCD	Numer wersji urządzenia (zapis taki sam jak w polu bcdUSB).
iManufacturer	14	1	Indeks	Indeks deskryptora tekstowego, który zawiera opis producenta.
iProduct	15	1	Indeks	Indeks deskryptora tekstowego, który zawiera opis produktu.
iSerialNumber	16	1	Indeks	Indeks deskryptora tekstowego, który zawiera numer seryjny urządzenia.
bNum-Configurations	17	1	Liczba	Liczba możliwych konfiguracji urządzenia.

Pole ***bDescriptorType*** znajduje się w niemal wszystkich tabelach opisujących deskryptory w tym rozdziale. Typy deskryptorów zostały odpowiednio ponumerowane przez twórców standardu USB, zgodnie z tabelą 2 (na podstawie dokumentacji ze strony usb.org).

Tabela 2. Typy deskryptorów

Typ deskryptora	Wartość
Urządzenia	1
Konfiguracji	2
Tekstowy	3
Interfejsu	4
Punktu końcowego	5
DEVICE_QUALIFIER	6
OTHER_SPEED_CONFIGURATION	7
INTERFACE_POWER	8

W tabeli 2 przedstawiono również typy deskryptorów, które nie zostały zawarte na rysunku przedstawiającym strukturę deskryptorów urządzenia USB (rysunek 31). Deskryptor kwalifikatora urządzenia **DEVICE_QUALIFIER** wykorzystywany jest w urządzeniach USB pracujących w innych trybach niż *low speed* i zawiera informacje na temat tego, w jaki sposób zmieni się zachowanie urządzenia podczas pracy z inną prędkością, np. kwalifikator urządzenia pracującego w trybie *full speed* opisuje, jak zmienią się jego właściwości podczas pracy w trybie *high speed*.

W tabeli 3 przedstawiono deskryptor kwalifikatora, który występuje w urządzeniach operujących na innych prędkościach niż ***low speed***.

Tabela 3. Struktura deskryptora kwalifikatora urządzenia – Device Qualifier Descriptor

Nazwa pola	Offset (bajt)	Rozmiar (bajt)	Typ pola	Opis
bLength	0	1	Liczba	Rozmiar deskryptora kwalifikatora urządzenia, wyrażany w bajtach.
bDescriptorType	1	1	Stała	Dla deskryptora kwalifikatora urządzenia to pole zawsze przyjmuje wartość 0x06.
bcdUSB	2	2	BCD	Identyfikator wersji specyfikacji USB, z którą zgodne jest urządzenie. Dla wersji 1.1 wartość tego pola to 0x0110, dla wersji 2.0 – 0x0200.
bDeviceClass	4	1	Klasa	Kod klasy (jak w tabeli 1).
bDeviceSubClass	5	1	Subklasa	Kod subklasy (jak w tabeli 1).
bDeviceProtocol	6	1	Protokół	Kod protokołu (jak w tabeli 1).
bMaxPacketSize0	7	1	Liczba	Maksymalny rozmiar pakietu dla konfiguracji urządzenia typu ***other speed***.
bNum-Configurations	8	1	Liczba	Liczba konfiguracji urządzenia typu ***other speed*** (*high speed* i *full speed*).
bReserved	9	1	Stała	Pole zarezerwowane dla przyszłych rozszerzeń, przyjmuje wartość 0.

Pozostałe deskryptory: **OTHER_SPEED_CONFIGURATION** i **INTERFACE_POWER**, zostaną opisane w następnych podrozdziałach.

3.2. Deskryptor konfiguracji (ang. Configuration Descriptor)

Urządzenie USB może pracować w różnych konfiguracjach, np. w jednym trybie tzw. uśpienia ograniczony jest pobór mocy, w drugim trybie urządzenie zaś pracuje na „pełnych obrotach", przez co zwiększony jest pobór mocy. W takim przypadku omawiane urządzenie powinno zawierać co najmniej dwa deskryptory konfiguracji. W zależności od funkcji urządzenia USB wspierających różne konfiguracje host podejmuje decyzje, którą konfigurację włączyć, i **tylko jedna konfiguracja może być aktywna w danym czasie**. Deskryptor konfiguracji przechowuje informacje takie jak: ilość energii zużywanej przez daną konfigurację, rodzaj źródła zasilania (samodzielne lub z magistrali USB) oraz liczbę posiadanych interfejsów. Wszystkie parametry zapisane w deskryptorze konfiguracji zostały przedstawione w tabeli 4.

Tabela 4. Struktura deskryptora konfiguracji – Configuration Descriptor

Nazwa pola	Offset (bajt)	Rozmiar (bajt)	Typ pola	Opis
bLength	0	1	Liczba	Rozmiar deskryptora konfiguracji, wyrażany w bajtach.
bDescriptorType	1	1	Stała	Dla deskryptora konfiguracji to pole zawsze przyjmuje wartość 0x02.
wTotalLength	2	2	Liczba	Całkowity rozmiar wszystkich deskryptorów opisujących daną konfigurację. Jest to suma rozmiarów deskryptorów: konfiguracji, interfejsów, punktów końcowych i pozostałych opcjonalnych lub specyficznych dla danej klasy urządzenia.
bNumInterfaces	4	1	Liczba	Liczba interfejsów obsługiwanych przez bieżącą konfigurację.
bConfiguration-Value	5	1	Liczba	Identyfikator deskryptora konfiguracji.
iConfiguration	6	1	Indeks	Indeks deskryptora tekstowego opisującego daną konfigurację.
bmAttributes	7	1	Bitowe	Charakterystyki konfiguracji bitów D7–D0: – D7: wartość zarezerwowana (zawsze 1). – D6: jeżeli urządzenie zasilane jest z magistrali USB, to należy w polu *bMaxPower* ustawić wartość natężenia prądu, jakie będzie pobierane z magistrali USB.

Nazwa pola	Offset (bajt)	Rozmiar (bajt)	Typ pola	Opis
				W takim wypadku pole D6 przyjmuje wartość 1, w przeciwnym – wartość 0. – D5: urządzenie obsługuje zdalne wybudzanie: nie – 0, tak – 1. – D4…D0: wartości zarezerwowane (zawsze 0).
bMaxPower	8	1	Natężenie prądu (mA)	Maksymalna wartość natężenia prądu, jaka może zostać pobrana z magistrali USB przez urządzenie pracujące w bieżącej konfiguracji (podstawowa jednostka to 2 mA; wartość 50 w tym polu oznacza, że urządzenie może pobrać z magistrali maksymalnie 100 mA).

Podobnie jak w przypadku deskryptora urządzenia obok standardowego deskryptora konfiguracji pojawia się tzw. deskryptor innych prędkości konfiguracji (ang. **OTHER_SPEED_CONFIGURATION**) przedstawiony w tabeli 5. Jest on jednak stosowany dla urządzeń wspierających prędkości **high speed**.

Tabela 5. Struktura deskryptora innych prędkości konfiguracji – Other Speed Configuration Descriptor

Nazwa pola	Offset (bajt)	Rozmiar (bajt)	Typ pola	Opis
bLength	0	1	Liczba	Rozmiar deskryptora konfiguracji (***other speed***), wyrażany w bajtach.
bDescriptorType	1	1	Stała	Dla deskryptora **Other Speed Configuration** to pole zawsze przyjmuje wartość 0x07.
wTotalLength	2	2	Liczba	Całkowity rozmiar wszystkich deskryptorów opisujących daną konfigurację.
bNumInterfaces	4	1	Liczba	Liczba interfejsów obsługiwanych przez bieżącą konfigurację.
bConfiguration-Value	5	1	Liczba	Identyfikator deskryptora konfiguracji.
iConfiguration	6	1	Indeks	Indeks deskryptora tekstowego opisującego daną konfigurację.

Nazwa pola	Offset (bajt)	Rozmiar (bajt)	Typ pola	Opis
bmAttributes	7	1	Bitowe	Charakterystyki konfiguracji (jak w tabeli 4).
bMaxPower	8	1	Natężenie prądu (mA)	Maksymalna wartość natężenia prądu, która może zostać pobrana z magistrali USB przez urządzenie pracujące w bieżącej konfiguracji (podstawowa jednostka to 2 mA; wartość 50 w tym polu oznacza, że urządzenie może pobrać z magistrali maksymalnie 100 mA).

3.3. Deskryptor interfejsu (ang. Interface Descriptor)

Urządzenie może posiadać tylko jedną aktywną konfigurację w tym samym czasie. Każda konfiguracja urządzenia może posiadać kilka interfejsów, które mogą być aktywne w tym samym czasie. Przykładem może być rozbudowana **klawiatura multimedialna**, która umożliwia użytkownikowi korzystanie jednocześnie z np. klawiatury standardowej, myszki (jako touchpada) i obsługi multimediów (ang. consumer). Tego typu klawiatura multimedialna mogłaby realizować kilka różnych funkcji: obsługę przycisków ***Keyboard Interface Descriptor***, poruszanie kursorem ***Mouse Interface Descriptor*** i kontrolowanie plików multimedialnych (np. odtwarzanie, pauzowanie i przewijanie: filmów, zdjęć, muzyki itd.) ***Consumer Interface Descriptor***. Na rysunku 32 przedstawiono poglądową strukturę deskryptorów do omawianej klawiatury multimedialnej.

W deskryptorze urządzenia pole bNumConfigurations opisuje liczbę możliwych konfiguracji urządzenia. Omawiana klawiatura multimedialna posiada jedną konfigurację z trzema interfejsami. Informacja o liczbie interfejsów dostępnych w ramach danej konfiguracji znajduje się w deskryptorze konfiguracji w polu bNumInterfaces. W deskryptorze każdego interfejsu znajduje się z kolei informacja o liczbie punktów końcowych, z których dany interfejs może korzystać w celu wymiany danych z hostem. Informacja o liczbie dostępnych punktów końcowych dla danego interfejsu znajduje się w polu bNumEndpoints. Niezależnie od liczby dostępnych i wykorzystywanych przez dany interfejs punktów końcowych każdy interfejs korzysta z podstawowego **– zerowego punktu końcowego** (który nie jest brany pod uwagę w polu bNumEndpoints), opcjonalnie może wykorzystać dodatkowe punkty końcowe.

4. Zasady budowy deskryptora raportu dla urządzeń HID

Deskryptor raportu musi zawsze wystąpić, jeśli mamy do czynienia z urządzeniem HID. Definiuje precyzyjnie sposób, w jaki będą wymieniane dane pomiędzy urządzeniem HID a hostem. To na podstawie zawartości deskryptora raportu później przygotowuje się w kodzie programu niewielkie struktury danych (grupy bajtów), tzw. **raporty**. Raporty zwykle definiowane są jako zmienne strukturalne w pamięci RAM. Jeśli urządzenie chce przesłać dane do hosta, to najpierw napełnia raport wejściowy (strukturę danych w RAM) odpowiednimi wartościami, a następnie przesyła raport w całości do hosta w trybie przerwaniowym za pomocą określonego punktu końcowego (endpoint in). Na podobnej zasadzie host przesyła dane do urządzenia HID. Najpierw napełnia raport wyjściowy odpowiednimi danymi, a następnie przesyła go w całości do urządzenia za pomocą wybranego punktu końcowego (endpoint out). W przeciwieństwie do wcześniej opisanych deskryptorów takich jak: deskryptor urządzenia, konfiguracji, interfejsu, których budowa jest ściśle uwarunkowana przez standard USB, to deskryptor raportu może być swobodnie tworzony przez projektanta urządzenia HID, pod warunkiem ścisłego korzystania z zasad określonych w standardzie USB, dotyczących jego budowy.

Deskryptory raportów mogą być proste i złożone. Proste zwykle definiują jedno urządzenie HID, złożone zaś pozwalają na zdefiniowanie funkcji kilku różnych urządzeń HID. Przykładem urządzenia, które wykorzystuje złożony deskryptor raportu, może być klawiatura multimedialna, która składać się może z trzech różnych urządzeń. W takiej sytuacji w ramach jednego deskryptora znaleźć się mogą: deskryptor „Standard Keyboard 101", deskryptor „Consumer" do obsługi multimediów i deskryptor „standard Mouse" do obsługi np. touchpada.

```
prefix  data                prefix  data
                             name    value/name
0x05, 0x01,                 // USAGE_PAGE (Generic Desktop)
0x09, 0x06,                 // USAGE (Keyboard)
0xa1, 0x01,                 // COLLECTION (Application)
0x95, 0x08,                 //   REPORT_COUNT (8)
0x75, 0x01,                 //   REPORT_SIZE (1)
0x05, 0x07,                 //   USAGE_PAGE (Keyboard)
0x19, 0xe0,                 //   USAGE_MINIMUM (Keyboard LeftControl)
0x29, 0xe7,                 //   USAGE_MAXIMUM (Keyboard Right GUI)
0x81, 0x02,                 //   INPUT (Data,Var,Abs) ; Modifier byte
0x95, 0x01,                 //   REPORT_COUNT (1)
0x75, 0x08,                 //   REPORT_SIZE (8)
0x81, 0x03,                 //   INPUT (Cnst,Var,Abs) ; Reserved byte
0x95, 0x03,                 //   REPORT_COUNT (3)
0x75, 0x01,                 //   REPORT_SIZE (1)
0x05, 0x08,                 //   USAGE_PAGE (LEDs)
0x19, 0x01,                 //   USAGE_MINIMUM (Num Lock)
0x29, 0x03,                 //   USAGE_MAXIMUM (Scroll Lock)
0x91, 0x02,                 //   OUTPUT (Data,Var,Abs) ; LED report
0x95, 0x01,                 //   REPORT_COUNT (1)
0x75, 0x05,                 //   REPORT_SIZE (5)
0x91, 0x03,                 //   OUTPUT (Cnst,Var,Abs); LED rep padd
0x95, 0x06,                 //   REPORT_COUNT (6 - USE_REPORTS_ID)
0x75, 0x08,                 //   REPORT_SIZE (8)
0x05, 0x07,                 //   USAGE_PAGE (Keyboard)
0x19, 0x00,                 //   USAGE_MINIMUM (Reserved
0x29, 0x65,                 //   USAGE_MAXIMUM (Keyboard Application)
0x15, 0x00,                 //   LOGICAL_MINIMUM (0)
0x25, 0x65,                 //   LOGICAL_MAXIMUM (101)
0x81, 0x00,                 //   INPUT (Data,Ary,Abs)
0xc0                        // END_COLLECTION
```

Rysunek 36. Poglądowy deskryptor standardowej klawiatury HID

Proces budowy deskryptora raportu można byłoby porównać do języka programowania, w którym występują pewne słowa kluczowe, za pomocą których można realizować wiele różnych funkcji. Standard USB przewiduje formy opisowe dla elementów/pozycji (ang. items), z jakich powinien składać się każdy nowo tworzony deskryptor raportu. W tej publikacji będziemy stosowali nazwę „elementy". Stanowią one coś w rodzaju słów kluczowych, pewnych mnemonik, w których każdy element posiada swój ściśle określony kod liczbowy, tzw. prefiks liczący sobie 6 bitów. Prefiks mieści się w ramach 1 bajtu na 6 najstarszych bitach, w związku z tym 2 najmłodsze bity zostały wykorzystane do ustalenia, ile bajtów danych może wystąpić łącznie w jednym elemencie po pierwszym bajcie prefiksu.

Występują trzy typy elementów (items) służących do budowy deskryptorów raportu:

1. **Elementy główne** (Main Items), np.: Input, Output, Feature, Collection (hid1_11.pdf 6.2.2.4).
2. **Elementy globalne** (Global Items), np.: Usage Page, Logical Min/Max (hid1_11.pdf 6.2.2.7).
3. **Elementy lokalne** (Local Items), np.: Usage, Usage Min/Max (hid1_11. pdf 6.2.2.8).

(Na końcu każdego punktu podano dokładny numer rozdziału w dokumencie **hid1_11.pdf**, gdzie znaleźć można dokładne prefiksy wymienionych elementów). Elementy są zapisane w formacie tzw. **Short Items.** Tworzenie elementów w tym formacie wiąże się z tym, że każdy element może składać się minimalnie z 1 bajtu, a maksymalnie z 4 bajtów (wliczając w to pierwszy bajt z prefiksem). W formacie Short Items pierwszy bajt zawsze zawiera prefiks danego elementu, który opisany jest na 6 starszych bitach pierwszego bajtu, 2 najmłodsze bity oznaczają liczbę bajtów, która może wystąpić w dalszej części elementu. Jeden element może nie mieć żadnych bajtów danych, wtedy będzie reprezentował go tylko 1 bajt.

Format xxxx xx nn – gdzie znakiem **x** oznaczono bity prefiksu, znakiem **n** oznaczono zaś bity oznaczające liczbę bajtów w dalszej części opisu elementu. Zgodnie z tym każdy element może sobie liczyć minimalnie 1 bajt, wtedy wartość **nn** będzie równa 0b00 (0). Jeśli będzie potrzebny kolejny i tylko 1 bajt opisu elementu, to **nn** przyjmie wartość 0b01 (1) itd. Na przykład element globalny służący do opisu numeru ID dla raportu (jeśli ma zostać użyty) posiada prefiks: **1000 01nn**. W związku z tym, że dla określenia numeru ID wystarczy zwykle 1 bajt, to **nn** będzie równe 0b01 (1), a pierwszy bajt będzie równy **1000 0101**. Wraz z numerem ID = 1 cały element będzie składał się z 2 bajtów:
1. Pierwszy bajt: 1000 0101 (**0x85**).
2. Drugi bajt: 0000 0001 (**0x01**).

W celu wygodniejszej dla oka prezentacji element taki w całej strukturze stosuje się wraz z krótkim komentarzem, który informuje o nazwie użytego elementu i wartości argumentu w nawiasach okrągłych:

```
0x85, 0x01,                    // Report ID (1)
```

Gdyby z jakichś powodów numer ID miał być większy niż 255, wtedy nie starczyłoby 1 bajtu do przekazania wartości ID. Załóżmy hipotetycznie (choć nigdy nie zdarzają się tak duże numery ID), że numer ID ma być równy 513 (0x0201). W tej sytuacji w pierwszym bajcie prefiks pozostanie ten sam, lecz zmieni się wartość bitów **nn** na 0b10 (2). Dlatego pierwszy bajt przyjmie postać: 1000 0110 (0x86), co oznacza, że tuż za pierwszym bajtem pojawią się 2 kolejne bajty niosące wartość ID = 513 (0x0201). Zawsze najmłodszy bajt wartości będzie występował jako pierwszy po bajcie prefiksu, a kod całego elementu będzie wyglądał w następujący sposób:

```
0x86, 0x01, 0x02,              // Report ID (513)
```