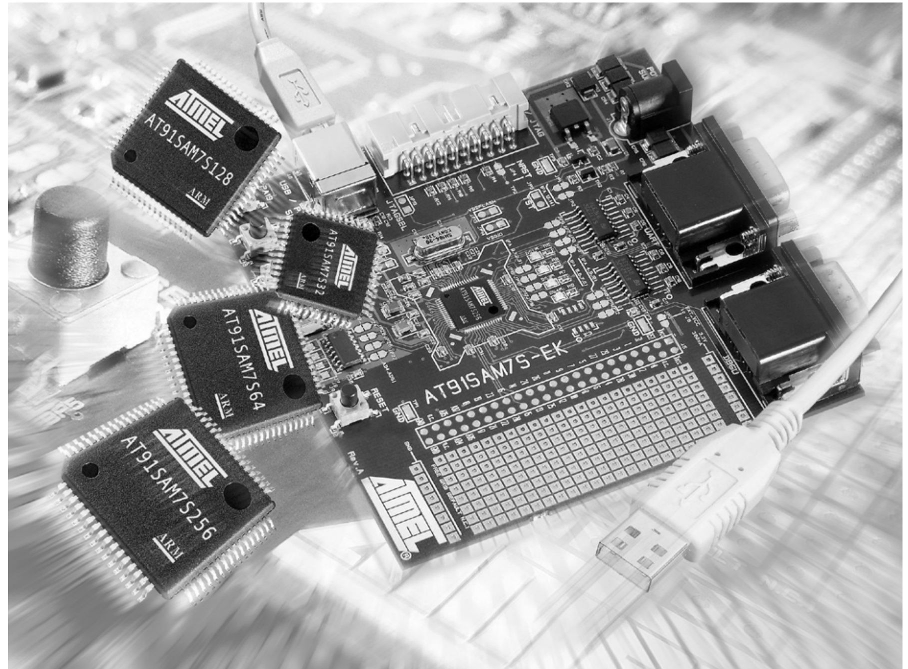
Opis implementacji klasy magazynującej USB na przykładzie mikrokontrolera AT91SAM7S64, część 1

Specyfikacja interfejsu USB jest dość trudna do zrozumienia, stąd wynika olbrzymia popularność układów firmy FTDI, które sprowadzają obsługę tego portu do poziomu transmisji przez znany „na wylot" RS232. Chcąc jednak wykorzystać wszystkie możliwości, jakie udostępnia nam USB trzeba sięgnąć po inne rozwiązania.



Moim szczególnym obszarem zainteresowań w dziedzinie elektroniki są interfejsy oraz wykorzystanie pamięci masowych. Do opublikowania projektu „Czytnik kart SD" (EPxxxxx) skłoniła mnie chęć podzielenia się swoją pasją. Celem niniejszego opracowania jest natomiast przybliżenie szczegółów implementacji klasy magazynującej USB w urządzeniu *embedded* zbudowanym na jednym z mikrokontrolerów popularnej rodziny SAM7S. Artykuł wraz z kodem programu może być przydatny dla konstruktorów poszukujących informacji o implementacji klasy magazynującej zainteresowanych samym protokołem USB, SD (via SPI) lub elementami interfejsu SCSI. Artykuł może być również inspiracją do wszelkich modyfikacji projektu czytnika kart SD – dostosowania go do własnych potrzeb lub udoskonalenia. Zachęcam do dzielenia się także udoskonalonymi wersjami „biblioteki" Mass Storage Class dla SAM7, ponieważ projekt jest typu *open–source*. Dedykuję ten projekt do zastosowań typu *non–profit* i edukacyjnych.

Bazą hardware'ową do wszelkich eksperymentów może być płytka (niejako ewaluacyjna) wspomnianego tutaj czytnika kart SD. Software był pisany i kompilowany w pakiecie WinARM (dokładniejsze informacje znajdują się w artykule o czytniku). Aby zapoznać się z implementacją Mass Storage Class nie jest konieczne ani doświadczenie przy pracy z kompilatorem gcc zawartym w Win ARMie, ani umiejętność programowania mikrokontrolerów ARM – dla mikrokontrolera AT91SAM7S64 będą przedstawione jedynie przykłady, lecz nic nie stoi na przeszkodzie przenieść kod na inną platformę.

Założeniem wstępnym jest przyswojenie przez Czytelnika informacji z artykułu „Czytnik kart SD", ponieważ zawarta jest w nim ogólna idea działania całego urządzenia dysku zewnętrznego. Pojawią się tu liczne (nie zawsze odnotowane) nawiązania do opisu czytnika. Samo opracowanie implementacji może wydawać się „akademicko" zagmatwane bez znajomości ogólnego zarysu działania czytnika. W artykule zostaną przedstawione mechanizmy i specyfikacje konieczne do implementacji MSC, zalecana jest więc literatura przedstawiona w artykule „Czytnik kart SD".

Nie będą tu omawiane bardzo szczegółowo znaczenia wszystkich flag i rejestrów. Według mnie najlepszą formą „rozgryzania" działania czytnika kart i klasy magazynującej będzie zapoznanie się z zasadą działania urządzenia i następnie odnalezienie w kodzie programu fragmentów odpowiadających opisowi i ich późniejsze rozczytywanie z notą aplikacyjną AT91SAM7S64 i specyfikacjami USB oraz SCSI (ich tłumaczenie jest całkowicie pozbawione sensu). Odnalezienie odpowiednich fragmentów kodu postaram się ułatwić podając nazwy kluczowych funkcji i ewentualnie plików .c i .h, w których te funkcje występują.

USB

USB jest zdaniem wielu elektroników i programistów interfejsem trudnym do opanowania. Niestety jest to prawda. Mimo to, jest on coraz lepiej rozpracowywany także przez elektroników–amatorów, m.in.

dzięki projektom typu *open–source* i implementacji najniższych warstw transmisji w tanich mikrokontrolerach. Wiele osób może budować urządzenia z USB wykorzystując popularne i łatwo konfigurowalne układy FTDI, nie koniecznie wnikając w istotę działania samego interfejsu. Mimo wielu barier jest on bardzo szybki i wszechstronny, stąd jego duża popularność w sprzęcie powszechnego użytku i coraz większa liczba zastosowań w przemyśle.

W niniejszym opisie poziom szczegółowości postaram się dostosować do wymagań implementacji MSC używającej stosunkowo łatwego (jak na USB) protokołu *Bulk–Only–Transport*.

Najniższe warstwy protokołu USB

Warstwy te mogą sprawiać wrażenie najbardziej skomplikowanych, lecz są one zaimplementowane sprzętowo w mikrokontrolerze AT91SAM7S, dzięki czemu użytkownik nie musi się o nie troszczyć. Dodatkowo są w dużej mierze wspólne dla urządzeń pracujących w różnych klasach USB. Z powyższych powodów przedstawię je jedynie poglądowo z niewielkim naciskiem na zagadnienia typowe dla Mass Storage Class.

Endpointy i pakiety

Dane przesyłane przez USB trafiają do (oraz wychodzą ze) specjalnych buforów przypisanych do tzw. *endpoints*. Kryjący się za endpointem bufor nazywany jest Bank. W mikrokontrolerze AT91SAM7S mamy do wykorzystania 4 endpointy: EP0...EP3, a w czytniku kart SD użyte są 3 z nich: EP0, EP1, EP2.

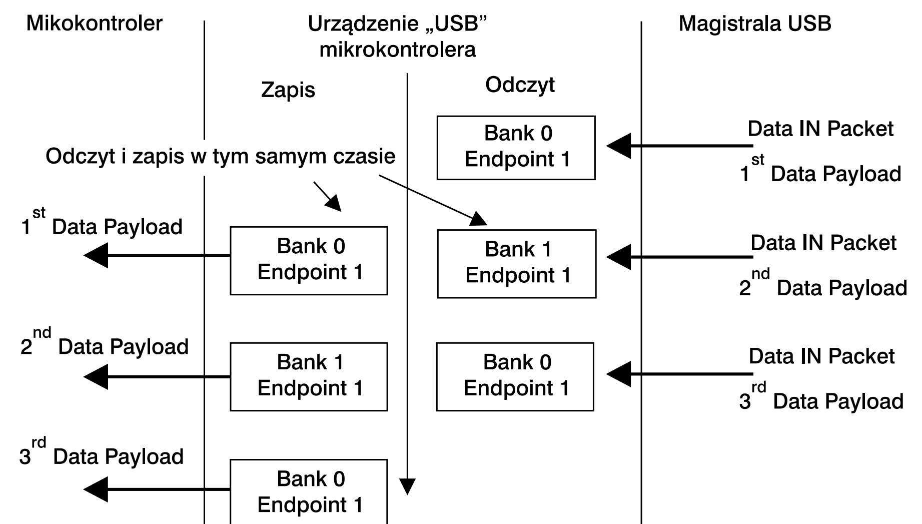
Endpoint EP0 można wykorzystać do kontroli oraz ustawiania transmisji. W projekcie Mass Storage EP0 użyty jest jako tzw. Control Endpoint – jego funkcją jest sterowanie transmisją. Podczas enumeracji i konfiguracji (*set–up*) urządzenia MSC dane są przekazywane przez ten właśnie endpoint. Pozostałe dwa endpointy (EP1 oraz EP2) zostały skonfigurowane do pracy w trybie bulk: EP1 jako bulk–IN, a EP2 jako bulk–OUT. Te endpointy mają większy rozmiar buforów niż EP0 (64 bajty dla EP1 i EP2, gdy rozmiar EP0 to tylko 8 bajtów), ponieważ służą do przesyłania danych właściwych, a nie tylko sterujących.

Zadaniem mikrokontrolera i hosta jest zapewnienie „wyłowienia” z pakietu (lub wpisania do pakietu) przesłanego przez USB danych, które znajdą się w określonym endpoincie oraz – w przypadku mikrokontrolera – ustawienie odpowiednich flag, np. do którego endpointu dane zostały wpisane przez hosta oraz jaki typ informacji one niosą. Flagi te mogą np. zgłaszać przerwanie, w którym zostanie zawarta reakcja na nadchodzące dane. Sam skład pakietu danych USB jest dość skomplikowany i nie będzie tutaj omawiany, ponieważ zajmuje się nim mikrokontroler. Użytkownik – przez rejestry kontrolne i sterujące mikrokontrolera – dostaje informacje, jakie dane przyszły, z którego endpointu itp. Wysyłanie danych USB nie odbywa się bezpośrednio przez np. wpisywanie do portu USB, lecz – w dużym uproszczeniu – program w mikrokontrolerze wpisuje dane do odpowiedniego endpointu i zgłasza, że zostały wpisane, a host odczytuje je w odpowiednim czasie.

Tym sposobem przybliżone zostały Czytelnikowi zagadnienia związane z przesyłaniem pakietów przez interfejs USB. Podsumowując, można powiedzieć, że pakiet USB to dane (trafiające do buforów endpointu), które chcemy przesłać, „opakowane” w dodatkowe, kontrolne i sterujące ciągi bitów, które następnie „filtruje” mikrokontroler lub host. W tym momencie, gdy po odebraniu danych w mikrokontrolerze mamy już ustawione odpowiednie flagi w rejestrach i ewentualne przychodzące dane znajdują się w buforze (Banku) któregoś endpointu, startujemy z wyższego poziomu komunikacji USB. Przykładowo: w przypadku, gdy w banku znajdują się dane, należy je sekwencyjnie odczytywać np. do tablicy w programie, aż do uzyskania informacji o końcu transmisji. Teraz przejdziemy do wyższej warstwy komunikacji: odczytamy – jak to zostało wspomniane – sekwencyjnie dane z odpowiednich banków do tablic w programie. Użyjemy do tego celu funkcji USB_DataIn (wysyłanie danych z endpointu bulk–in), USB_DataOut (odbieranie danych z endpointu bulk–out) i USB_SendData (wysyłanie danych z EP0 do hosta). Odbieranie i interpretacja danych z EP0 odbywa się w funkcji USBRequestProcessor wywoływanej w main.

Przesyłanie danych

W projekcie czytnika kart realizacja wymiany właściwych danych pomiędzy hostem i urządzeniem jest realizowana przez funkcje USB_DataOut oraz USB_DataIn. Warto zwrócić uwagę na fakt, że w endpointach EP1 i EP2 są do dyspozycji podwójne bufory (*Dual–Bank*). Dzięki temu – w uproszczeniu – gdy host zapełnia jeden z buforów, np. Bank0, to drugi (Bank1) może być odczytywany i dane z niego przetwarzane. Po zapełnieniu pierwszego bufora (Bank0) host wpisuje do drugiego (Bank1), a mikrokontroler odczytuje i przetwarza dane z pierwszego (Bank0) itd. Tym sposobem otrzymuje się znaczne przyspieszenie transmisji. Dodatkowo redukowane są opóźnienia wynikające z niezastosowania systemu przerwań



Rys. 1. Schematyczna prezentacja przepływu danych w czasie

List. 1. Tekst wysyłany z terminala podczas enumeracji

```
RESET REQUEST
get DEVICE_DESCRIPTOR, expected length =  0x40
RESET REQUEST
set address
get DEVICE_DESCRIPTOR, expected length =  0x12
get CONFIGURATION_DESCRIPTOR, expected length =  0x09
get LANGUAGE DESCRIPTOR, expected length =  0xFF
get CONFIGURATION_DESCRIPTOR, expected length =  0xFF
get LANGUAGE DESCRIPTOR, expected length =  0xFF
get LANGUAGE DESCRIPTOR, expected length =  0xFF
get DEVICE_DESCRIPTOR, expected length =  0x12
get CONFIGURATION_DESCRIPTOR, expected length =  0x09
get CONFIGURATION_DESCRIPTOR, expected length =  0x20
get LANGUAGE DESCRIPTOR, expected length =  0x02
get LANGUAGE DESCRIPTOR, expected length =  0x04
get MANUFACTURER DESCRIPTOR, expected length =  0x02
get MANUFACTURER DESCRIPTOR, expected length =  0x06
set configuration
!class specific request...
-GET MAX LUN
```

List. 2. Deskryptor „Device Descriptor"

```
const char USB_DeviceDescriptor[] = {
  0x12,   // bLength
  0x01,   // bDescriptorType
  0x00,   // bcdUSBL
  0x02,   //
  0x00,   // bDeviceClass
  0x00,   // bDeviceSubclass
  0x00,   // bDeviceProtocol
  0x08,   // bMaxPacketSize0
  0x01,   // idVendorL
  0x00,   //
  0x01,   // idProductL
  0x00,   //
  0x01,   // bcdDeviceL
  0x00,   //
  0x01,   // iManufacturer
  0x02,   // iProduct
  0x01,   // SerialNumber
  0x01    // bNumConfigs
};
```

w programie sterującym czytnikiem. Na **rys. 1** przedstawiono schematycznie przepływ danych w czasie. Data Payload oznacza właściwe dane niosące informacje. Należy pamiętać o nomenklaturze oznaczania kierunku transmisji: otóż kierunek jest oznaczany tak, jak „widzi" go host, czyli DATA IN będzie oznaczało wysyłanie danych z urządzenia DO HOSTA, natomiast DATA OUT będzie odbieraniem danych w urządzeniu, czyli wysyłaniem danych OD HOSTA. Z jednej strony jest to mylące dla konstruktora urządzenia, z drugiej strony jednak pozwala na pełną zgodność opisu kierunku danych w standardzie.

Ze względu na inną specyfikę wysyłania danych do hosta (DATA IN) podczas transferów kontrolnych niż w przypadku transferów właściwych danych (*data payloads*), inna funkcja realizuje transfery kontrolne (via EP0). Jest to funkcja USB_SendData. Jest ona prostsza, dodatkowo hardware mikrokontrolera AT91SAM7S64 nie posiada w endpoincie EP0 dwóch banków. Tutaj oznaczenie „send" odpowiada rzeczywistemu kierunkowi transmisji danych: urządzenie wysyła dane do hosta.

Odbieranie przez mikrokontroler danych kontrolnych z EP0 (oficjalnie DATA OUT) odbywa się w funkcji USBRequestProcessor wywoływanej bezpośrednio z pętli głównej programu demonstracyjnego czytnika kart. Funkcja ta jest bardzo rozbudowana, przede wszystkim ze względu na różnorodność żądań hosta i odpowiedzi z nimi związanych. Ważny element tej funkcji to sprawdzenie (blisko jej początku), czy nadszedł pakiet „setup". Jak zostało wspomniane, to peryferium UDP mikrokontrolera (jego port USB innymi słowy) realizuje rozpoznawanie rodzaju pakietu. Dane, które nadeszły w pakiecie setup, znajdują się w banku endpointu EP0. Następnie na podstawie zawartości bufora EP0 z ciągu bajtów w nim się znajdujących „wyławiane" są odpowiednie informacje, czyli flagi zdefiniowane w standardzie USB (w naszym przypadku USB 2.0). Te flagi to bmRequestType, bRequest, wValue, wIndex, wLength. Na podstawie położeń poszczególnych bitów w tych flagach oraz wartości samych flag rozpoznajemy m.in. czego żąda od naszego urządzenia host, lub ile bajtów odpowiedzi na żądanie należy wysłać (np. w czasie pobierania deskryptorów, co zostało opisane dalej). Wszystkie rodzaje requestów (żądań) nie będą tutaj omawiane, ponieważ artykuł rozrósłby się do sporych rozmiarów. Dodatkowo nie są to requesty kluczowe dla zrozumienia działania klasy magazynującej USB i są podobne dla wszystkich urządzeń USB, więc przy pracy z własnym urządzeniem, pracującym nawet w innej klasie niż MSC, wystarczy większość ciała tej lub podobnej funkcji wkleić do własnego kodu. Żądaniami, które zostaną bardziej szczegółowo omówione, są żądania wysłania deskryptorów, czyli tablic danych zawierających podstawowe informacje o urządzeniu USB, ponieważ są one charakterystyczne dla urządzenia i w każdym urządzeniu należy je odpowiednio skonfigurować.

Podczas przejścia przez funkcję USBRequestProcessor należy wspomnieć jedynie, że program wykonuje właściwie wszystkie polecenia dla endpointu EP0 (kontrolnego) konieczne do rozpoznania urządzenia w systemie i rozpoczęcia właściwej transmisji danych USB. Po omówieniu ważnych, z punktu widzenia użytkownika, deskryptorów przejdę

Tab. 1. Deskryptor urządzenia (Device Descriptor)

Bajt	Nazwa	Opis	Wartość (dla czytnika kart SD)
0	bLength	Długość deskryptora w bajtach	0x12
1	bDescriptorType	Numer rodzaju deskryptora	0x01
2	bcdUSB	Niższy bajt oznaczający wersję standardu USB	0x00
3	bcdUSB	Wyższy bajt oznaczający wersję standardu USB	0x02
4	bDeviceClass	Klasa urządzenia	0x00
5	bDeviceSubClass	„podklasa" urządzenia	0x00
6	bDeviceProtocol	Protokół urządzenia	0x00
7	bMaxPacketSize0	Największy rozmiar pakietu danych dla endpointu EP0	0x08
8	idVendor	Kod producenta – przyznawany przez USB–IF (L)	0xCD
9	idVendor	Kod producenta (H)	0xAB
10	idProduct	Kod produktu (L)	0x01
11	idProduct	Kod produktu (H)	0xEF
12	bcdDevice	Kod wersji urządzenia (BCD) (L)	0x00
13	bcdDevice	Kod wersji urządzenia (BCD) (H)	0x01
14	iManufacturer	Indeks do tekstowego deskryptora producenta	0x01
15	iProduct	Indeks do tekstowego deskryptora produktu	0x02
16	iSerialNumber	Indeks do tekstowego deskryptora numeru seryjnego	0x03
17	bNumConfigurations	Liczba możliwych konfiguracji	0x01

do tego, co się dzieje, gdy pojawiają się requesty typowe dla klasy oraz dane w endpoincie EP0 i EP2 (*Bulk Out*), czyli miejscu, gdzie rozpoczyna się obsługa poleceń hosta typowych dla Mass Storage Class. Dla lepszego zobrazowania czego żąda system operacyjny od urządzenia USB, można umieścić w programie sterującym polecenia wysłania ciągów znaków (tekstów) do terminala, np. przez port DBGU mikrokontrolera AT91SAM7S64 (port DBGU zawiera w sobie m.in. „okrojony" UART). Jest to chyba najprostszy i dość skuteczny sposób na debugging oraz analizę przebiegu programu nawet przy korzystaniu z systemu operacyjnego. **List. 1** zawiera tekst z terminala wysłany w czasie enumeracji czytnika kart z dołożonymi odpowiednimi poleceniami DBGU.

Deskryptory USB i Mass Storage – Device Descriptor

Jak zostało wspomniane wyżej, deskryptory zawierają opis urządzenia dla hosta, aby ten wiedział, z jakim urządzeniem „rozmawia". Są one pobierane zaraz po rozpoczęciu enumeracji lub kolokwialnie: po włożeniu wtyczki USB do gniazdka lub zerowaniu urządzenia.

W czasie enumeracji host może żądać różnych deskryptorów. Ciekawe jest to, że host nie zawsze zgłasza żądanie wysłania całego deskryptora, a także, że kilkukrotnie żąda jednego z deskryptorów (jak na list. 1). Podczas każdego requestu enumeracji host podaje wartość, ile bajtów powinno mu odesłać urządzenie. W komentarzach do kodu zostały zawarte nazwy flag używane w specyfikacji USB 2.0, a znaczenie najważniejszych z nich zostanie tutaj omówione. Litera „L" po nazwie bajtu w listingach oznacza, że jest to bajt mniej znaczący (niższy) – w takim przypadku na wartość składa się więcej bajtów. Wartości „składane" z bajtów starałem się oznaczać odpowiednimi komentarzami w kodzie programu.

Podstawowym deskryptorem jest deskryptor urządzenia – Device Descriptor. Jego zawartość w przypadku czytnika kart została przedstawiona na **list. 2** w takiej formie, jak jest zapisany w programie (czyli w formie tablicy stałych odsyłanej odpowiednimi fragmentami lub w całości do hosta). Znaczenia poszczególnych wartości przedstawiono skrótowo w **tab. 1**.

Rzeczą wymagającą komentarza w Device Descriptor są zerowe wartości pól bDeviceClass, bDeviceSubclass i bDeviceProtocol. Według specyfikacji USB 2.0, każdy interfejs obsługiwany przez urządzenie będzie niósł informację o klasie w jakiej pracuje urządzenie. Dla elektronika–programisty oznacza to mniej więcej tyle, że wartości te zostaną zdefiniowane dopiero w deskryptorze konfiguracji Configuration Descriptor (o którym będzie mowa niżej) przy okazji definiowania także interfejsu urządzenia.

Deskryptory USB i Mass Storage – dygresje

W tym miejscu można wspomnieć o różnicach pomiędzy USB 1.1, USB 2.0, USB High speed i USB Full speed. Nazwy te są bardzo często mylone. USB 1.1 i USB 2.0 to wersje specyfikacji USB (*Revisions* – pełna nazwa dokumentu to Universal Serial Bus Specification Revision 2.0). Oczywiście standard USB 1.1 jest starszy, a USB 2.0 jest nowszy i obszerniejszy. Transmisja USB high speed została wprowadzona do specyfikacji USB 2.0 (specyfikacja USB 1.1 jeszcze jej nie obejmowała). „Magiczne" high speed oznacza – oprócz tego, że urządzenie szybko wymienia dane z hostem – trochę inny protokół, nawet na poziomie ułożenia bitów w pakietach, konieczność obsługi odpowiedzi na inne zapytania hosta niż w full speed itd. Dla przykładu (najniższa warstwa) pole bitowe synchronizujące transmisję full speed ma 8 bitów, a w standardzie high speed ma ono 32 bity. Inna różnica: urządzenie pracujące w high speed musi posiadać dodatkowe deskryptory. Według specyfikacji USB 2.0 wyróżniamy 3 szybkości pracy urządzeń:

Low speed – 1,5 Mb/s
Full speed – 12 Mb/s
High speed – 480 Mb/s.

Jeśli producent urządzenia napisał na opakowaniu „USB 2.0", nie koniecznie musi to oznaczać, że urządzenie pracuje z przepływnością high speed. W drugą stronę, oznaczenie high speed powinno implikować zgodność ze standardem USB 2.0. Czytnik kart pracuje jako urządzenie full speed w standardzie USB 2.0, ponieważ taki rodzaj wymiany danych obsługuje niskopoziomowo mikrokontroler AT91SAM7S64.

Ten akapit może się wydawać nieco zagmatwany i chwilowo warto przyjąć go „na wiarę". Urządzenie może mieć więcej niż jedną konfigurację (bNumConfigurations > 1) – może być jakby kilkoma urządzeniami USB w jednym. Wtedy oczywiście urządzenie będzie musiało posiadać więcej niż jeden deskryptor konfiguracji. Podobna sytuacja będzie miała miejsce w samym deskryptorze konfiguracji: jedna konfiguracja będzie mogła obsługiwać więcej niż jeden interfejs. A wybiegając całkiem daleko, każdy interfejs będzie mógł mieć przypisany więcej niż jeden endpoint. Na szczęście, w takim ujęciu endpoint już nie jest „podzielny". W czytniku kart SD wykorzystany jest jeden deskryptor konfiguracji oraz jeden deskryptor interfejsu i dwa deskryptory endpointów – ale po kolei...

Robert Brzoza-Woch