Wielokanałowy, dwukierunkowy interfejs COM/USB

Za pomocą zestawu Freescale KwikStik K40

Przedstawiamy sposób implementacji na zestawie KwikStik K40 z mikrokontrolerem Kinetis MK40X256VLQ100 urządzenia kompozytowego USB, emulującego wiele portów COM. Każdy port ma przypisana własną funkcję obsługi przychodzących do niego danych, co umożliwia imitowanie niezależnych urządzeń i aplikacji na nich działających.

Nie ma dużego znaczenia, który mikrokontroler z rodziny KINETIS zostanie wybrany do realizacji symulatora, jednak musi mieć cechy, takie jak możliwość dołączenia przez USB i niewielkie wymiary. Ważna jest też dostępność urządzenia oraz znajomość środowiska programistycznego. Zgodny z wymaganiami jest mikrokontroler Freescale MK40X256VLQ100 stosowany w zestawie Kwikstik K40, pokazany na **fotografii 1**. Ma on następujące parametry: szybkość taktowania wynosząca 100 MHz, pamięć Flash o pojemności 256 kB oraz wbudowany debugger SEGGER J-Link. Projekt dla mikrokontrolera został przygotowany przy użyciu środowiska programistycznego CodeWarrior 10.1 oraz systemu czasu rzeczywistego MQX w wersji 3.7. Opiera się on na przykładowym projekcie dostarczonym przez Freescale Semiconductor, zawartym w pakiecie demo *Kinetis KwikStik Demo Software Lab Guide Rev. 0.4*, który po instalacji znajduje się w katalogu: *„${MQX_ROOT_DIR}/demo/Kwikstik_Demo/cw10”* oraz kodzie z przykładu wirtualnego portu COM znajdującego się domyślnie w katalogu: *„${MQX_ROOT_DIR}/usb/device/examples/cdc/virtual_com”*. Wszystkie biblioteki użyte w projekcie są dostarczone razem z systemem operacyjnym MQX 3.7.

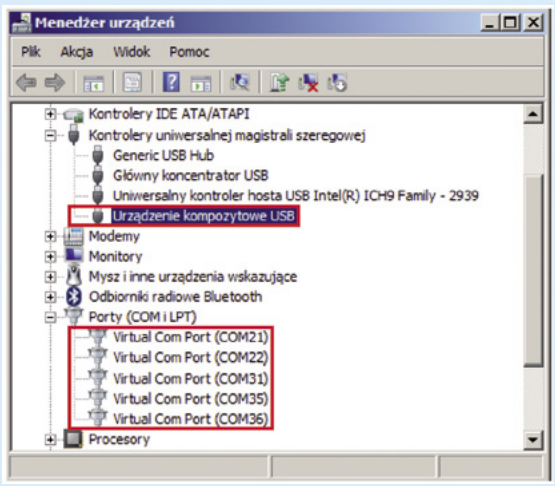
Aby napisać aplikację, która na mikrokontrolerze obsługuje wiele punktów końcowych USB, należy zdefiniować wiele struktur identyfikujących, oraz napisać odpowiednie funkcje umożliwiające osobną obsługę każdego zestawu punktów końcowych składających się na imitowane urządzenie. Ponieważ będę chciał obsługiwać imitowane urządzenia przez wirtualny port COM zostanie użyta do tego klasa CDC i przedefiniowana, tak aby była możliwość osobnej obsługi każdego z punktów końcowych.

Nie będą opisywane wszystkie pola struktur i wszystkie własności komunikacji USB,a tylko te, które wyjaśniają pobieżnie zagadnienie oraz te, które należy zmodyfikować w celu osiągnięcia, określonego przez temat projektu, efektu. Dla lepszego śledzenia zmian wprowadzanych w kodzie, przy każdej wymaganej modyfikacji znajduje się komentarz: „*// !!! zmień aby dodać*”, wraz z opisem.

Fotografia 1. Widok zestawu KwikStik K40

Urządzenie kompozytowe jest to pojedyncze urządzenie pozwalające na obsługę wielu niezależnych aplikacji USB. Efektem niniejszego projektu będzie wykrycie przez system urządzenia kompozytowego oraz kilku wirtualnych portów COM jak pokazano na **rysunku 2**.

Aby zaimplementować urządzenie kompozytowe należy zintegrować ze sobą kilka interfejsów w urządzeniu (rys. 1.3). W skrócie, aby dodać nowy interfejs w celu obsługi dodatkowej aplikacji używając tego samego kontrolera należy: zmodyfikować liczbę interfejsów obsługiwanych przez deskryptor konfiguracji, dodać nowy deskryptor interfejsu, deskryptory punktów końcowych oraz powiązane z nimi funkcje obsługi żądań.



Rysunek 2. Widok urządzenia kompozytowego oraz imitowanych przez niego wirtualnych portów COM w Menedżerze Urządzeń systemu Windows

Tabela 1. Pola deskryptora urządzenia	
Pole	**Opis**
bLength	wielkość deskryptora w bajtach
bDescriptorType	typ deskryptora
bcdUSB	wersja specyfikacji USB w formacie BCD (binary-coded decimal)
bDeviceClass	kod klasy
bDeviceSubclass	kod podklasy
bDeviceProtocol	kod protokołu
bMaxPacketSize0	maksymalny rozmiar pakietów dla zerowego punktu końcowego
idVendor	numer identyfikacyjny sprzedawcy
idProduct	numer identyfikacyjny produktu
bcdDevice	numer serii urządzenia
iManufacturer	indeks deskryptora znakowego producenta
iProduct	indeks deskryptora znakowego produktu
iSerialNumber	indeks deskryptora znakowego numeru seryjnego
bNumConfigurations	liczba możliwych konfiguracji

Urządzenie USB raportuje swoje atrybuty za pośrednictwem deskryptorów. Deskryptor jest strukturą o zdefiniowanym formacie. Każdy rozpoczyna się od pola zawierającego całko- wity rozmiar w bajtach, a następnie po nim posiada pole o wielkości bajtu identyfikujące typ deskryptora. Urządzenie USB potrzebuje zdefiniowania deskryptorów urządzenia, deskryptorów konfiguracji i interfejsów.

Deskryptor urządzenia (**tabela 1**) zawiera ogólne informacje o urządzeniu USB, dotyczące jego globalnych własności. Urządzenie USB ma tylko jeden deskryptor urządzenia, znajduje się on w tablicy g_device_descriptor.

Dla urządzenia kompozytowego należy ustawić klasę urządzenia (*Device Class*), podklasę (*Device Subclass*) oraz protokół (*Device Protocol*) na wartość 0. Wtedy każdy interfejs będzie mógł ustawić własne wartości, a host będzie rozpoznawał podłączony przez USB mikrokontroler jako urządzenie kompozytowe. Należy też ustawić pola Vendor ID oraz Product ID na właściwe wartości. Wszystkie urządzenia mające tą samą parę numerów VID/PID powinny używać tych samych sterowników. Przykład deskryptora urządzenia pokazano na **listingu 1**.

Listing 1. Deskryptor urządzenia
```
uint_8 g_device_descriptor[DEVICE_DESCRIPTOR_SIZE]  =
{
  DEVICE_DESCRIPTOR_SIZE,  /*  Device Dexcriptor  Size  */
  USB_DEVICE_DESCRIPTOR,  /*  Device Type of descriptor  */
  USB_uint_16_low(BCD_USB_VERSION),  /*    BCD USB version  */
  USB_uint_16_high(BCD_USB_VERSION),
  0,  /*  Device Class  */
  0,  /*  Device Subclass  */
  0,  /*  Device Protocol  */
  CONTROL_MAX_PACKET_SIZE,  /*  Max  Packet size  */
  0xXX,0xXX,     /*  Vendor  ID  */
  0xXX,0xXX,  /*  Product ID  */
  0x02,0x00,  /*  BCD Device version */
  0x01,  /*  Manufacturer string  index */
  0x02,  /*  Product string  index */
  0x00,  /*  Serial number  string  index */
  DEVICE_DESC_NUM_CONFIG_SUPPOTED  /*  Number  of configurations */
};
```

Deskryptor konfiguracji (**tabela 2**) opisuje bardziej specyficzną konfigurację urządzenia. Definiuje liczbę interfejsów dostarczonych w danej konfiguracji.

Należy odpowiednio zmienić wartości definicji w pliku usb_descriptor.h, mianowicie wielkość struktury z deskryptorami CONFIG_DESC_SIZE oraz liczbę zadeklarowanych interfejsów CONFIG_DESC_NUM_INTERFACES_SUPPORTED. Przykładowy deskryptor konfiguracji zamieszczono na **listingu 2**.

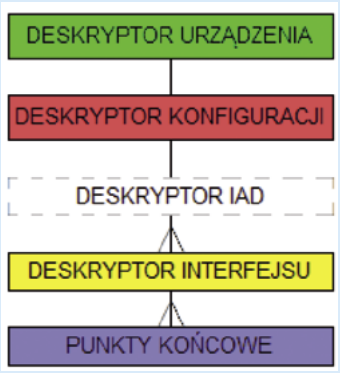
Tabela 2. Pola deskryptora konfiguracji	
Pole	**Opis**
bLength	wielkość deskryptora w bajtach
bDescriptorType	typ deskryptora
wTotalLength	rozmiar w bajtach deskryptora konfiguracji i wszystkich deskryptorów mu podporządkowanych
bNumInterfaces	liczba interfejsów w konfiguracji
bConfigurationValue	identyfikator dla wywołań Get_Configuration i Set_Configuration
iConfiguration	indeks deskryptora znakowego konfiguracji
bmAttributes	atrybuty zasilania i zdalnego wywołania (ang. wakeup)
bMaxPower	(maksymalne natężenie prądu)/2, wyrażone w amperach

Deskryptor interfejsu (**tabela 3**) opisuje specyfikę interfejsów w ramach deskryptora konfiguracji. Rozdziela on punkty końcowe w funkcjonalne grupy, z których każda realizuje jedną z funkcji urządzenia. Ponieważ jest to obszerna struktura, została ona wyizolowana do pliku new_interface.desc. Warto nadmienić, że zerowy punkt końcowy nie jest uwzględniany w liczbie punktów końcowych dla danego deskryptora interfejsu.

Zadeklarowanie dodatkowego interfejsu odbywa się przez zdefiniowanie kolejnego numeru interfejsu _INTERFACE_ oraz kolejnych zestawów punktów końcowych: _CIC_NOTIF_ENDPOINT_, _DIC_BULK_IN_ENDPOINT_, _DIC_BULK_OUT_ENDPOINT_ załączenie pliku new_interface.desc. W każdym deskryptorze interfejsu zmieniany jest przede wszystkim numer interfejsu bInterfaceNumber oraz numery punktów końcowych (**listing 3**).

Listing 2. Deskryptor konfiguracji
```
uint_8 g_config_descriptor[CONFIG_DESC_SIZE]  =
{
  /*     Configuration Descriptor Size - always 9  bytes*/
CONFIG_ONLY_DESC_SIZE,
  /*  "Configuration" type of descriptor  */
  USB_CONFIG_DESCRIPTOR,
  USB_uint_16_low(CONFIG_DESC_SIZE),
  /*     Total length of the Configuration descriptor  */
  USB_uint_16_high(CONFIG_DESC_SIZE),
  /*     NumInterfaces */
  CONFIG_DESC_NUM_INTERFACES_SUPPOTED,
  0x01,          /* Configuration  Value */
  0x00,          /* Configuration Description String  Index*/
  /*     Attributes.support RemoteWakeup and  self power  */
  BUS_POWERED|SELF_POWERED|(REMOTE_WAKEUP_SUPPORT<<REMOTE_WAKEUP_SHIFT),
  /*     Current  draw from bus */
  CONFIG_DESC_CURRENT_DRAWN
  ...
  /*  deklaracje umieszczone w  piku new_interface.desc    */
  ...
}
```

Deskryptor IAD (**tabela 4**) umożliwia powiązanie wielu interfejsów z jedną funkcją logiczną urządzenia. W tym celu została dodana klasa na poziomie urządzenia, która musi zostać dołączona do implementacji urządzenia korzystającego z IAD. W ten sposób możliwa jest identyfikacja urządzenia korzystającego z IAD podczas fazy enumeracji. Pozwoli to na zainstalowanie sterownika dla urządzenia o specjalnym przeznaczeniu, dzięki któremu będzie możliwe właściwe przekazywanie konfiguracji



Rysunek 3. Hierarchia deskryptorów w projekcie

oraz lokalizowanie odpowiednich sterowników dla urządzeń korzystających z IAD. Obsługa omawianego deskryptora została dodana wraz z Windows XP SP2.

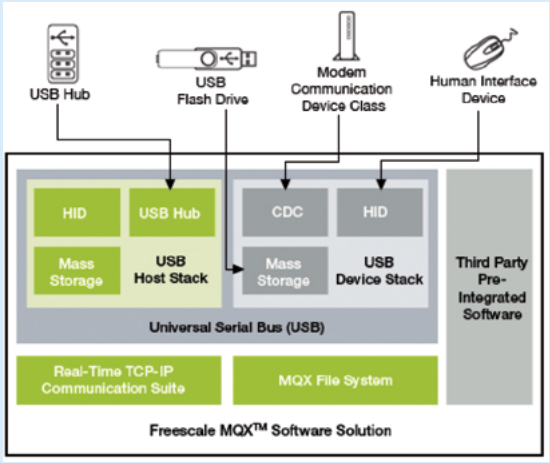
Kody klasy, podklasy i protokołu powinny mieć takie wartości jak w interfejsach specyfikujących funkcję urządzenia. Przykład deskryptora IAD zamieszczono na **listingu 4**.

Każdy punkt końcowy użyty w interfejsie ma własny deskryptor. Wszystkie punkty końcowe oprócz punktu zerowego są zdefiniowane w tablicy ep struktur USB_EP_STRUCT (**tabela 5**). Zerowy punkt końcowy spełnia rolę kontrolną i dla każdego urządzenia jest konfigurowany przed resztą punktów końcowych. Deklarację struktur trzech punktów końcowych dla pierwszego wirtualnego portu COM zamieszczono na **listingu 5**.

Dla nowo podłączanego urządzenia USB system operacyjny musi dobrać odpowiednie sterowniki. Do wyboru sterownika w systemie Windows używane są pliki INF. Menedżer oraz rejestr systemowy przechowują informacje o urządzeniach i przypisanych do nich sterownikach. Pliki INF znajdują się w folderze *%SystemRoot%/inf*, a każdy nowy plik jest tam kopiowany.

Składnia pliku INF:

- Tekst po średniku „;” jest traktowany jako komentarz.
- Wszystkie informacje są zorganizowane w sekcje. Nazwy sekcji zawierają się w nawiasach kwadratowych [ ].
- Tekst pomiędzy znakami procentu %text% traktowany jest jako String. W sekcji „[Strings]” jest ograniczany cudzysłowem.



Rysunek 4. Stos USB Host/Urządzenie

Tabela 3. Pola deskryptora interfejsu

Pole	Opis
bLength	wielkość deskryptora w bajtach
bDescriptorType	typ deskryptora
bInterfaceNumber	numer identyfikujący interfejs
bAlternateSetting	wartość używana do wyboru alternatywnych ustawień
bNumEndpoints	liczba obsługiwanych punktów końcowych nie licząc zerowego
bInterfaceClass	kod klasy
bInterfaceSubclass	kod podklasy
bInterfaceProtocol	kod protokołu
iInterface	indeks deskryptora znakowego interfejsu

Listing 3. Sposób zadeklarowania pierwszego interfejsu

```
#define  _INTERFACE_                1
#define  _CIC_NOTIF_ENDPOINT_       CIC_NOTIF_ENDPOINT1
#define  _DIC_BULK_IN_ENDPOINT_     DIC_BULK_IN_ENDPOINT1
#define  _DIC_BULK_OUT_ENDPOINT_    DIC_BULK_OUT_ENDPOINT1
#include  „new_interface.desc”
```

Tabela 4. Pola deskryptora IAD

Pole	Opis
bLength	wielkość deskryptora w bajtach
bDescriptorType	typ deskryptora
bFirstInterface	numer pierwszego interfejsu w funkcji
bInterfaceCount	liczba interfejsów w kolekcji
bFunctionClass	kod klasy
bFunctionSubClass	kod podklasy
bFunctionProtocol	kod protokołu
iFunction	indeks deskryptora znakowego

Listing 4. Implementacja deskryptora IAD

```
/*  INTERFACE ASSOCIATION DESCRIPTOR */ IAD_ONLY_DESC_SIZE, USB_IAD_DESCRIPTOR,
(uint_8)((_INTERFACE_-1)*2),  /*  bFirstInterface */
0x02, /*  bInterfaceCount */
CDC_CLASS,  /*  Communication Interface  Class */
CIC_SUBCLASS_CODE,
CIC_PROTOCOL_CODE,
0x00, /*  Interface  Description String Index*/
```

Tabela 5. Struktura USB_EP_STRUCT

Pole	Opis
ep_num	numer punktu końcowego
Type	typ punktu końcowego USB_INTERRUPT_PIPE lub USB_BULK_PIPE
direction	kierunek USB_SEND lub USB_RECV
Size	przepustowość punktu końcowego

Listing 5. Deklaracja struktur trzech punktów końcowych dla pierwszego wirtualnego portu COM

```
USB_EP_STRUCT ep[CDC_DESC_ENDPOINT_COUNT] = {
  #if CIC_NOTIF_ELEM_SUPPORT
  {
    CIC_NOTIF_ENDPOINT,
    USB_INTERRUPT_PIPE,
    USB_SEND,
    CIC_NOTIF_ENDP_PACKET_SIZE
  }
  #endif
  #if DATA_CLASS_SUPPORT
  #if CIC_NOTIF_ELEM_SUPPORT
  ,
  #endif
  {
    DIC_BULK_IN_ENDPOINT,
    USB_BULK_PIPE,
    USB_SEND,
    DIC_BULK_IN_ENDP_PACKET_SIZE
  },
  {
    DIC_BULK_OUT_ENDPOINT,
    USB_BULK_PIPE,
    USB_RECV,
    DIC_BULK_OUT_ENDP_PACKET_SIZE
  }
  #endif
  …
  /*  zestawy  punktów końcowych dla kolejnych urządzeń  */
  …
}
```

Listing 6. Struktura _USB_EVENT_STRUCT
```
typedef struct  _USB_EVENT_STRUCT
{
   _usb_device_handle handle;      /* controler  device handle */
   uint_8             ep_num;
   boolean            setup;       /* is  setup packet */
   boolean            direction;   /* direction of endpoint */
   uint_8_ptr         buffer_ptr;  /* pointer to buffer */
   uint_32            len;         /* buffer size of endpoint */
}  USB_EVENT_STRUCT, *PTR_USB_EVENT_STRUCT
```

- Elementy o postaci: HKR, ,NTMPDriver„usbser.sys dostarczają informacji do przechowania w rejestrze.
- Niektóre elementy są wartościami początkowymi jak np. Class=Ports.

Urządzenia z wieloma interfejsami mogą wyspecyfikować sterowniki dla każdego z interfejsów. W tym wypadku urządzenie ma wiele identyfikatorów w następującej postaci: USB \VID_xxxx&PID_yyyy&MI_ww.

Do projektu zostanie użyta klasa CDC (*Communication Device Class*) służąca do implementacji protokołów komunikacji przez łącze USB. Jej miejsce w stosie host/urządzenie zostało zaprezentowane na **rysunku 4**.

Aby móc użyć klasy CDC warstwy API należy:

- Wywołać USB_Class_CDC_Init() w celu inicjalizacji sterownika klasy, wszystkich klas podrzędnych i sterownika urządzenia.
- Gdy funkcja zwrotna zostaje wywołana przez zdarzenie USB_APP_ENUM_COMPLETE, stan aplikacji może zostać zmieniony na gotowy.
- Należy wywołać funkcję USB_Class_CDC_Send_Data() w celu wysłania danych do hosta.
- Należy wywołać funkcję USB_Class_CDC_Recv_Data(), gdy zostaje wywołana funkcja zwrotna ze zdarzeniem USB_APP_DATA_RECEIVED.

Klasa CDC, jak już zostało wspomniane, sama inicjalizuje klasy od niej niższe. Jednak aby osiągnąć cel projektu należy zajrzeć warstwę głębiej i przekonać się w jaki sposób się to realizuje. Właściwy proces inicjalizacji API urządzenia USB przebiega w następujący sposób:

- Wywołać funkcję _usb_device_init() w celu inicjalizacji sterownika niskiego poziomu oraz kontrolera.
- Wywołać funkcję _usb_device_register_service(), aby zarejestrować funkcję zwrotną dla zdarzeń na magistrali.
- Wywołać funkcję _usb_device_register_service(), aby zarejestrować wywołania zwrotne dla punktów końcowych.
- Wywołać funkcję _usb_device_init_endpoint(), aby zainicjalizować żądane punkty końcowe.

Warstwa urządzenia musi być zainicjalizowana w celu wysyłania zarejestrowanych wywołań zwrotnych w momencie zajścia zdarzenia na magistrali USB. Urządzenia muszą rozpocząć odbieranie wywołań opisanych w specyfikacji USB [6].

Klasa CDC oczywiście realizuje wszystkie punkty krok po kroku, jednak należy zwrócić uwagę, że punkty końcowe są rejestrowane przez funkcję USB_Class_CDC_Event(), która wszystkim punktom końcowym przypisuje te same funkcje zwrotne. Ponieważ celem jest odseparowanie funkcji obsługi dla każdego portu wirtualnego, należy punkty końcowe ponownie zarejestrować, tym razem z własnymi funkcjami zwrotnymi. Do tego celu posłuży nam nowa funkcja _usb_device_reinit_endpoint().

Najłatwiejszym sposobem osobnej obsługi punktów końcowych jest ich przerejestrowanie tak, aby były obsługiwane przez nowe osobne funkcje zwrotne. Dlatego zostanie utworzona nowa funkcja ingerująca w inicjalizację klasy CDC. Nowa funkcja _usb_device_reinit_endpoint(handle, out_endpoint, callback) powoduje zarejestrowanie własnej funkcji zwrotnej dla wyjściowego punktu końcowego. Funkcja przyjmuje trzy argumenty:

- handle – uchwyt do identyfikacji kontrolera,
- out_endpoint – wyjściowy punkt końcowy,
- callback – wskaźnik na własną funkcję obsługi wywołania zwrotnego.

Funkcja _usb_device_reinit_endpoint() przerejestrowuje w trzech krokach:

- Zeruje zainicjowane struktury dla danego punktu końcowego oraz wyłącza go za pomocą funkcji _usb_device_deinit_endpoint().
- Inicjuje struktury dla danego punktu końcowego oraz włącza go za pomocą funkcji _usb_device_init_endpoint().
- Rejestruje obsługę zdarzeń dla danego punktu końcowego za pomocą funkcji _usb_device_register_service().

Nowa (własna) funkcja zwrotna ma prototyp o postaci void USB_Service_Dic_Bulk_OutX(PTR_USB_EVENT_STRUCT event, pointer arg);

Argument event dostarcza informacji na temat odebranych informacji przez punkt końcowy. Można wydobyć z niego wielkość bufora przez odwołanie event->len oraz informacje w nim zawarte dzięki wskaźnikowi event->buffer_ptr. Struktura _USB_EVENT_STRUCT została zaprezentowana na **listingu 6**.

W nowej funkcji można przeprowadzić własne obliczenia. Na końcu powinno znaleźć się wywołanie funkcji USB_Class_CDC_Recv_Data(), przygotowujące bufor odbiorczy na kolejne zdarzenie oraz wywołanie funkcji USB_Class_CDC_Send_Data(), wysyłające wynik funkcji na konkretny punkt końcowy.

Obsługa komunikacji przez USB jest dość skomplikowana, jednak podzielenie stosu protokołu na warstwy sprzyja jej logicznemu rozdzieleniu, a zdefiniowane klasy automatycznie realizują przypisane im funkcje. Dzięki drobnej ingerencji w podstawowe funkcje możliwe było zrealizowanie osobnej komunikacji dla każdego imitowanego urządzenia. Problem stanowiło odpowiednie zdefiniowanie deskryptorów USB, jednak po ich wyspecyfikowaniu uzyskany został pożądany efekt.

David Obrycki