helion.pl

Idź do

- Spis treści
- Przykładowy rozdział

Katalog książek

- Katalog online
- Zamów drukowany katalog

Twój koszyk

- Dodaj do koszyka

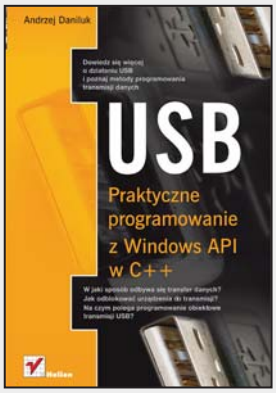
Cennik i informacje

- Zamów informacje o nowościach
- Zamów cennik

Czytelnia

- Fragmenty książek online

USB. Praktyczne programowanie z Windows API w C++



Autor: Andrzej Daniluk
ISBN: 978-83-246-2032-6
Format: 158x235, stron: 280

Dowiedz się więcej o działaniu USB i poznaj metody programowania transmisji danych

- W jaki sposób odbywa się transfer danych?
- Jak odblokować urządzenia do transmisji?
- Na czym polega programowanie obiektowe transmisji USB?

USB (skrót od ang. Universal Serial Bus – uniwersalna magistrala szeregowa) to rodzaj bardzo praktycznego portu komunikacyjnego, dzięki któremu możliwe jest podłączanie do komputera wielu różnych urządzeń, takich jak mysz, kamera, aparat fotograficzny, telefon komórkowy, modem, skaner, przenośna pamięć. Podłączone w ten sposób urządzenia są od razu wykrywane i rozpoznawane przez system, dzięki czemu instalacja sterowników i konfiguracja sprzętu odbywają się zwykle automatycznie.

Książka „USB. Praktyczne programowanie z Windows API w C++" w zwięzły sposób przedstawia wszelkie zagadnienia, dotyczące użytkowania i programowania transmisji USB. Korzystając z tego podręcznika, poznasz nie tylko teoretyczne podstawy działania USB, ale także zdobędziesz praktyczne umiejętności w tym zakresie. Książka zawiera bowiem zarówno konkretne przykłady, jak i ćwiczenia do samodzielnego wykonania dla wszystkich, którzy chcą zyskać wiedzę na zaawansowanym poziomie. Dowiesz się między innymi, jakie są rodzaje transferów danych i transakcji USB, za co odpowiadają komponenty i jak wykorzystać wzorce projektowe. Zrozumiesz także metody projektowania obiektowego oraz implementacji oprogramowania sterującego łączem USB.

- środowisko fizyczne i sygnałowe USB
- Klasy instalacji urządzeń
- Rejestr systemowy
- Transmisja danych
- Struktura systemu USB
- Detekcja i identyfikacja urządzeń
- Odczyt i zapis danych w formie raportu
- Programowanie obiektowe transmisji USB
- Wewnętrzne struktury danych
- Programy wielowątkowe
- Konwertery USB

Tu znajdziesz wszystko o działaniu USB i możliwościach jego wykorzystania!

Kontakt

Helion SA
ul. Kościuszki 1c
44-100 Gliwice
tel. 032 230 98 63
e-mail: helion@helion.pl
© Helion 1991-2008

Spis treści

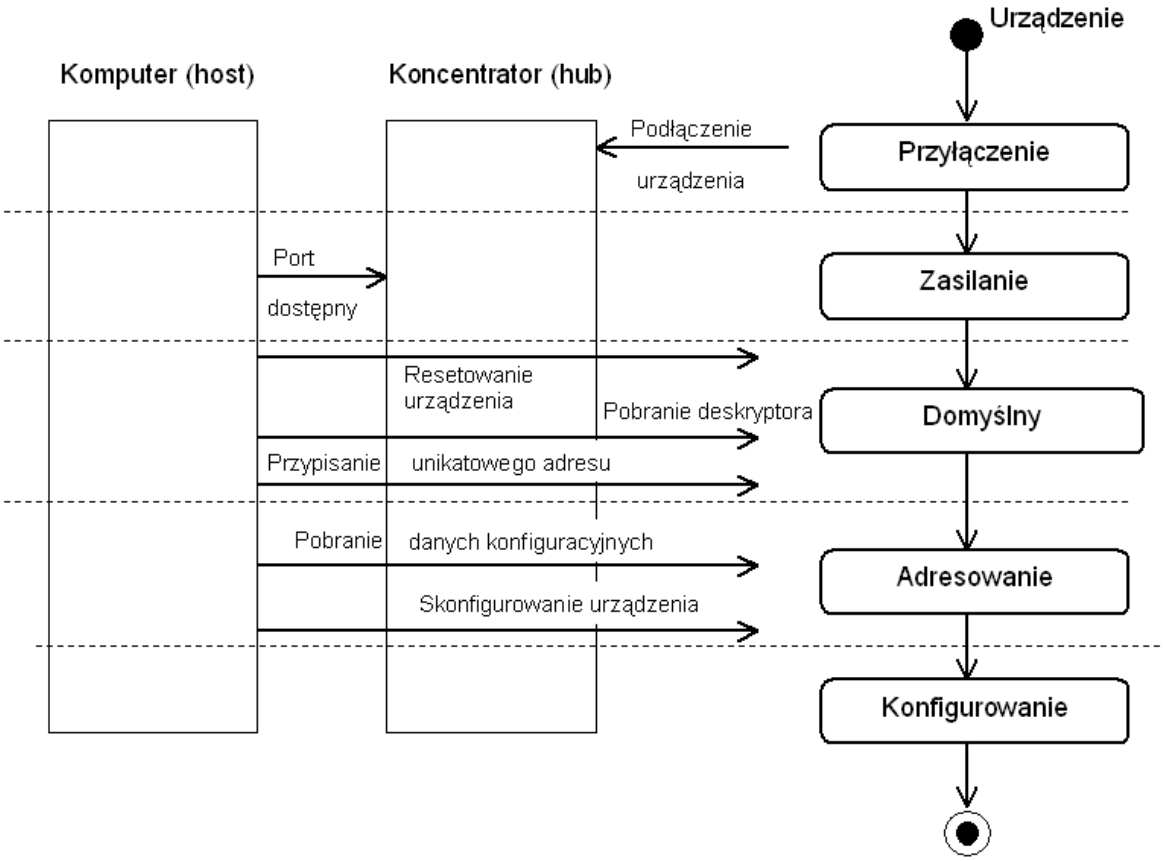
Rozdział 4.

Detekcja i identyfikacja urządzeń dołączonych do magistrali USB

Urządzenia USB są automatycznie wykrywane przez system operacyjny po ich podłączeniu i włączeniu zasilania. Kiedy w systemie pojawi się nowy sprzęt, aktywowane są procedury jego detekcji i identyfikacji . Zespół tego typu operacji często określany jest jako enumeracja (ang. *bus enumeration*). Rozpoczęcie procesu enumeracji powoduje przejście urządzenia między czterema podstawowymi stanami, jak to pokazano na rysunku 4.1.

Za pośrednictwem kilkunastu czynności, z których najważniejsze zostały przedstawione poniżej, system operacyjny wykonuje enumerację urządzenia w ramach poszczególnych stanów.

- Użytkownik przyłącza urządzenie do portu USB hosta (macierzystego komputera) lub huba (koncentratora) — urządzenie pozostaje w stanie przyłączenia (ang. *attached state*).
- Po odblokowaniu linii zasilającej urządzenie przechodzi w stan zasilania (ang. *powered state*).
- Po sprawdzeniu stanu linii zasilających oprogramowanie hosta przystępuje do konfigurowania nowego sprzętu.
- Hub poprzez testowanie stanu linii sygnałowych sprawdza, z jaką prędkością przesyłu danych urządzenie może pracować. Informacja zostaje przekazana do hosta w odpowiedzi na wysłany przez niego rozkaz `Get_Port_Status`.
- Kiedy nowe urządzenie zostaje rozpoznane, kontroler hosta wysyła do huba rozkaz `Set_Port_Feature`. Port zostaje zresetowany (przez 10 ms linie D– oraz D+ pozostają w niskim stanie logicznym).



Rysunek 4.1. *Podstawowe stany urządzenia w trakcie enumeracji*

- Poprzez dalsze testowanie aktualnego stanu linii D+ i D– host sprawdza, czy urządzenie pracujące z pełną szybkością przesyłu danych może pracować też z szybkością wysoką.
- Ponownie wysyłając rozkaz `Get_Port_Status`, host sprawdza, czy urządzenie pozostaje w stanie *Reset*. Jeżeli nie, utworzony zostaje potok zerowy przeznaczony do celów konfiguracji urządzenia. Urządzeniu przypisany zostaje domyślny adres `00h`, po czym przechodzi ono do stanu domyślnego (ang. *default state*).
- Host wysyła rozkaz `Get_Descriptor`, aby otrzymać informacje o maksymalnym rozmiarze pakietu danych, który może być transmitowany potokiem domyślnym. Rozkaz ten kierowany jest do zerowego punktu końcowego `EP0` urządzenia. Oprogramowanie hosta może identyfikować w danym czasie tylko jedno urządzenie, zatem tylko jedno urządzenie w danym czasie może pozostawać z adresem `00h`.
- W następnej kolejności za pośrednictwem rozkazu `Set_Address` urządzeniu przypisywany jest unikatowy adres — urządzenie przechodzi do stanu adresowania (ang. *addressed state*). Nowy adres pozostaje aktualny, dopóki urządzenie przyłączone jest do portu USB. W momencie odłączenia urządzenia port jest resetowany.
- W dalszej kolejności za pośrednictwem na nowo adresowanego rozkazu `Get_Descriptor` oprogramowanie hosta pobiera kompletne deskryptory urządzenia (zob. rysunek 3.6).

- Po odczytaniu deskryptorów urządzenia, oprogramowanie hosta wyszukuje dla urządzenia najlepiej pasujący sterownik i zapisuje stosowane informacje (Vendor ID, Product ID, ...) w pliku *.inf*.
- Sterownik urządzenia wysyła rozkaz `Set_Configuration` w celu ostatecznego skonfigurowania nowego sprzętu. Od tego momentu urządzenie pozostaje w stanie skonfigurowania (ang. *configured state*)[1].

Podstawowe zasoby systemowe

Kompilator C++ w module *setupapi.h* udostępnia szereg użytecznych funkcji i struktur, które w znakomity sposób umożliwiają samodzielne przeprowadzenie detekcji i identyfikacji ścieżek dostępu do interfejsów urządzeń aktualnie dołączonych do magistrali USB. W tym podrozdziale zostały przedstawione najważniejsze z nich.

Funkcja SetupDiGetClassDevs()

Funkcja zwraca identyfikator klasy podłączonych urządzeń, których lista i opis konfiguracji znajduje się w rejestrze systemowym w kluczu `HKEY_LOCAL_MACHINE` (zob. rozdział 2).

```
HDEVINFO
  SetupDiGetClassDevs(
    IN LPGUID  ClassGuid,  OPTIONAL
    IN PCTSTR  Enumerator,  OPTIONAL
    IN HWND  hwndParent,  OPTIONAL
    IN DWORD  Flags
    );
```

Parametr `ClassGuid` wskazuje strukturę GUID klasy urządzeń. Użycie tego parametru jest opcjonalne. Aplikacje użytkownika mogą pobierać adres identyfikatora GUID dla klasy urządzeń HID za pomocą funkcji `HidD_GetHidGuid()`. Wskaźnik `Enumerator` wskazuje łańcuch znaków (zakończony zerowym ogranicznikiem) przechowujący dane konkretnych urządzeń (zob. rozdział 2, rys. 2.4). Użycie tego parametru w programie jest opcjonalne. Jeżeli wskaźnikowi przypiszemy wartość `NULL`, funkcja zwróci listę urządzeń typu PnP (ang. *Plug and Play*). Opcjonalnie wykorzystywany identyfikator `hwndParent` wskazuje okno odpowiedzialne za interakcję z otrzymanym zestawem urządzeń. Znacznik `Flags` przyjmuje postać bitowej alternatywy wybranego zestawu następujących stałych symbolicznych:

`DIGCF_ALLCLASSES` — określa listę wszystkich zainstalowanych w systemie urządzeń;

`DIGCF_DEVICEINTERFACE` — określa listę zainstalowanych urządzeń z danym interfejsem;

`DIGCF_DEFAULT` — zwraca listę urządzeń z domyślnym interfejsem;

[1] Jeżeli w trakcie transmisji urządzenie przez 3 ms nie wykrywa znacznika początku ramki danych SOF, przechodzi do stanu zawieszenie (ang. *suspended state*).

DIGCF_PRESENT — określa urządzenia aktualnie dostępne w systemie;

DIGCF_PROFILE — określa listę urządzeń będących częścią aktualnego zestawu sprzętowego.

Jeżeli wykonanie funkcji nie powiedzie się, zwracana jest wartość INVALID_HANDLE_VALUE. Kod ewentualnego błędu można zdiagnozować za pomocą funkcji GetLastError().

Funkcja SetupDiEnumDeviceInterfaces()

Funkcja wyszukuje interfejsy urządzeń identyfikowanych przez wskaźnik DeviceInfoSet zwracany przez funkcję SetupDiGetClassDevs().

```
WINSETUPAPI BOOL WINAPI
  SetupDiEnumDeviceInterfaces(
    IN HDEVINFO  DeviceInfoSet,
    IN PSP_DEVINFO_DATA  DeviceInfoData,  OPTIONAL
    IN LPGUID  InterfaceClassGuid,
    IN DWORD  MemberIndex,
    OUT PSP_DEVICE_INTERFACE_DATA  DeviceInterfaceData
    );
```

Wskaźnik DeviceInfoData wskazuje strukturę SP_DEVINFO_DATA (zob. tabela 4.1), umożliwiając ograniczenie przeszukiwań istniejących urządzeń. Opcjonalnie funkcji można przekazać wskaźnik pusty. W takim wypadku funkcję należy wywoływać cyklicznie, tak aby przeszukała wszystkie interfejsy danego urządzenia. Wskaźnik InterfaceClassGuid wskazuje strukturę GUID. Parametr wejściowy MemberIndex jest numerem odpytywanego interfejsu. Jego wartości zaczynają się od 0 (zerowy indeks pierwszego interfejsu). Jeżeli funkcja wywoływana jest w pętli cyklicznie, przy każdym wywołaniu należy odpowiednio zwiększyć wartość MemberIndex. Jeżeli SetupDiEnumDeviceInterfaces() zwróci wartość FALSE oraz funkcja GetLAstError() zwróci ERROR_NO_MORE_ITEMS, oznacza to, że nie znaleziono interfejsu o podanym indeksie. Wskaźnik DeviceInterfaceData wskazuje strukturę SP_DEVICE_INTERFACE_DATA (zob. rozdział 3), której rozmiar należy wpisać w polu cbSize:

```
deviceInterfaceData.cbSize = sizeof(SP_DEVICE_INTERFACE_DATA);
```

Struktura SP_DEVINFO_DATA

W polach struktury przechowywane są informacje na temat egzemplarza urządzenia należącego do klasy urządzeń USB. W tabeli 4.1 zamieszczono jej opis.

Tabela 4.1. *Specyfikacja struktury USB_DEVINFO_DATA*

Typ	Element struktury	Znaczenie
DWORD	cbSize	Rozmiar struktury w bajtach.
GUID	ClassGuid	Identyfikator GUID klasy urządzeń.
DWORD	DevInst	Identyfikator wewnętrznej struktury opisującej urządzenie w systemie.
ULONG_PTR	Reserved	Zarezerwowane.

Windows DDK strukturę tę definiuje jako:

```
typedef struct _SP_DEVINFO_DATA {
  DWORD  cbSize;
  GUID  ClassGuid;
  DWORD  DevInst;
  ULONG_PTR  Reserved;
} SP_DEVINFO_DATA, *PSP_DEVINFO_DATA;
```

Definicja ta tworzy dwa nowe słowa kluczowe SP_DEVINFO_DATA (struktura) i SP_DEVINFO_
➥DATA (wskaźnik do struktury).


Uwaga

Funkcje rodziny SetupDiXx(), używając struktury SP_DEVINFO_DATA jako parametru, automatycznie sprawdzają poprawność określenia jej rozmiaru. Aktualny rozmiar struktury należy określić za pomocą operatora sizeof() i wpisać do pola cbSize. Jeżeli rozmiar struktury nie zostanie określony w ogóle lub zostanie określony nieprawidłowo, to w przypadku użycia struktury jako parametru wejściowego IN zostanie wygenerowany błąd ERROR_INVALID_PARAMETER, natomiast podczas używania struktury jako parametru wyjściowego OUT zostanie wygenerowany błąd ERROR_INVALID_USER_BUFFER.

Struktura SP_DEVICE_INTERFACE_DATA

Zasoby struktury SP_DEVICE_INTERFACE_DATA zaprezentowane w tabeli 4.2 przechowują dane interfejsu należącego do klasy urządzeń USB.

Tabela 4.2. *Specyfikacja struktury SP_DEVICE_INTERFACE_DATA*

Typ	Element struktury	Znaczenie
DWORD	cbSize	Rozmiar struktury w bajtach.
GUID	InterfaceClassGuid	Identyfikator GUID interfejsu klasy urządzeń.
DWORD	Flags	Znaczniki interfejsu. Wartość SPINT_ACTIVE oznacza, że interfejs jest aktualnie dostępny. Wartość SPINT_DEFAULT oznacza domyślny interfejs dla klasy urządzeń. Wartość SPINT_REMOVED określa usunięty interfejs.
ULONG_PTR	Reserved	Parametr zarezerwowany i aktualnie nieużywany.

Windows DDK strukturę tę definiuje jako:

```
typedef struct _SP_DEVICE_INTERFACE_DATA {
  DWORD cbSize;
  GUID  InterfaceClassGuid;
  DWORD Flags;
  ULONG_PTR  Reserved;
} SP_DEVICE_INTERFACE_DATA, *PSP_DEVICE_INTERFACE_DATA;
```

Definicja ta tworzy dwa nowe słowa kluczowe SP_DEVICE_INTERFACE_DATA (struktura) i PSP_DEVICE_INTERFACE_DATA (wskaźnik do struktury).


Uwaga

Funkcje zdefiniowane w module *setupapi.h*, używając struktury `SP_DEVICE_INTERFACE_DATA` jako parametru, automatycznie sprawdzają poprawność określenia jej rozmiaru. Aktualny rozmiar struktury należy określić za pomocą operatora `sizeof()` i wpisać do pola `cbSize`. Jeżeli rozmiar struktury nie zostanie określony w ogóle lub zostanie określony nieprawidłowo, system wygeneruje błąd `ERROR_INVALID_USER_BUFFER`.

Struktura SP_DEVICE_INTERFACE_DETAIL_DATA

Struktura `SP_DEVICE_INTERFACE_DETAIL_DATA` zawiera informacje o postaci ścieżki dostępu do interfejsu wybranego urządzenia USB. W tabeli 4.3 przedstawiono znaczenie poszczególnych pól tej struktury.

Tabela 4.3. *Specyfikacja struktury SP_DEVICE_INTERFACE_DETAIL_DATA*

Typ	Element struktury	Znaczenie
`DWORD`	`cbSize`	Rozmiar struktury w bajtach.
`TCHAR`	`DevicePath[ANYSIZE_ARRAY]`	Łańcuch znaków zakończony zerowym ogranicznikiem (tzw. `NULL` — *terminated string*) zawierający pełną nazwę symboliczną urządzenia (ścieżkę dostępu do interfejsu). Parametr ten wykorzystywany jest przez funkcję `CreateFile()`.

Windows DDK strukturę tę definiuje jako:

```
typedef struct _SP_DEVICE_INTERFACE_DETAIL_DATA {
  DWORD  cbSize;
  TCHAR  DevicePath[ANYSIZE_ARRAY];
} SP_DEVICE_INTERFACE_DETAIL_DATA, *PSP_DEVICE_INTERFACE_DETAIL_DATA;
```

Definicja ta tworzy dwa nowe słowa kluczowe `SP_DEVICE_INTERFACE_DETAIL_DATA` (struktura) i `PSP_DEVICE_INTERFACE_DETAIL_DATA` (wskaźnik do struktury).


Uwaga

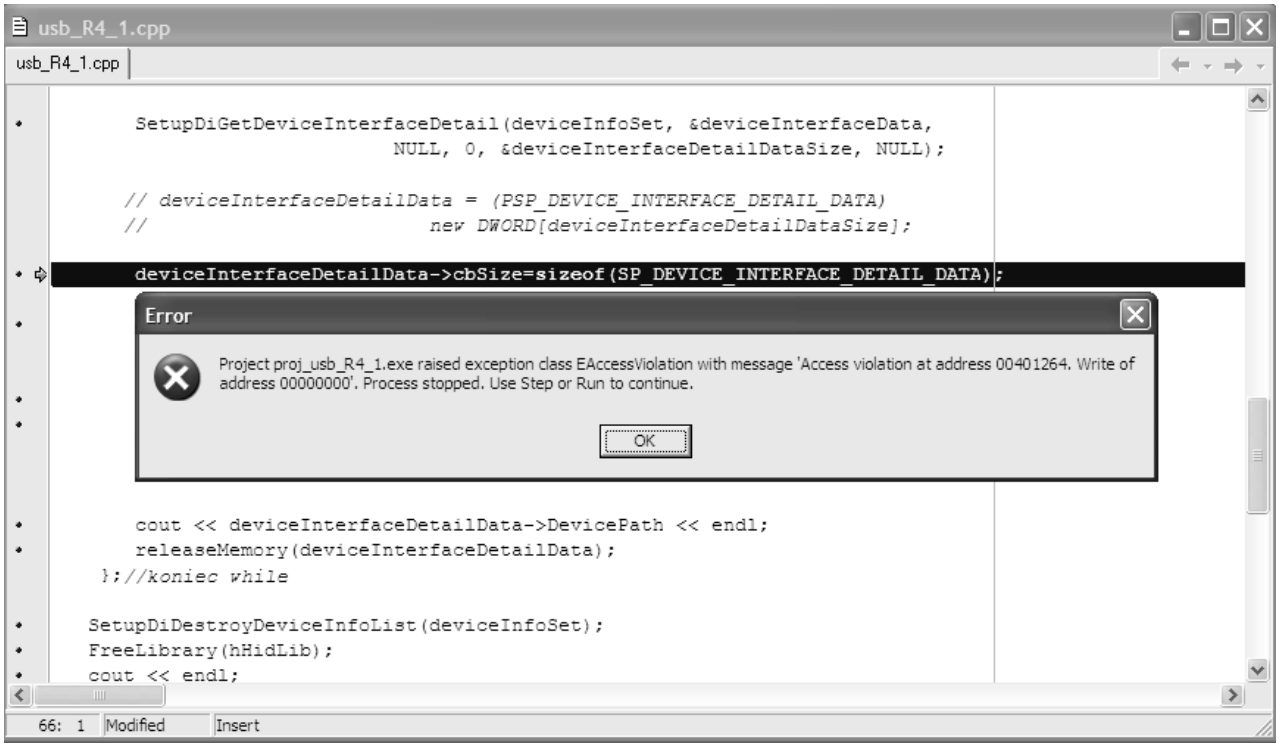
Niekiedy ścieżkę dostępu do interfejsu urządzenia utożsamia się z jego nazwą symboliczną, którą można odczytać z rejestru systemowego (zob. rozdz. 2). Chociaż te dwa łańcuchy znaków mają bardzo podobną postać, to jednak mogą różnić się długością, dlatego w programach bezpiecznej jest posługiwać się kompletnymi danymi zapisanymi w polu `DevicePath` struktury `SP_DEVICE_INTERFACE_DETAIL_DATA`.

Funkcja SetupDiGetDeviceInterfaceDetail()

Funkcja zwraca szczegółowe informacje na temat interfejsu urządzenia.

```
WINSETUPAPI BOOL WINAPI
  SetupDiGetDeviceInterfaceDetail(
    IN HDEVINFO  DeviceInfoSet,
    IN PSP_DEVICE_INTERFACE_DATA  DeviceInterfaceData,
    OUT PSP_DEVICE_INTERFACE_DETAIL_DATA  DeviceInterfaceDetailData, OPTIONAL
    IN DWORD  DeviceInterfaceDetailDataSize,
    OUT PDWORD  RequiredSize,  OPTIONAL
    OUT PSP_DEVINFO_DATA  DeviceInfoData  OPTIONAL
    );
```

Wskaźnik DeviceInfoSet zwracany jest przez funkcję SetupDiGetClassDevs(). Parametr DeviceInterfaceData wskazuje strukturę SP_DEVICE_INTERFACE_DATA. Wskaźnik DeviceInterfaceDetailData wskazuje strukturę SP_DEVICE_INTERFACE_DETAIL_DATA (zob. rozdział 3); opcjonalnie zamiast niego do funkcji może być przekazana wartość NULL. W przypadku jawnego wskazania struktury SP_DEVICE_INTERFACE_DETAIL_DATA wskaźnik powinien być poprawnie zainicjowany, a jej pole cbSize musi być prawidłowo określone. W przeciwnym razie kompilator zgłosi błędy naruszenia pamięci, w sposób podobny do tych pokazanych na rysunkach 4.2 i 4.3.

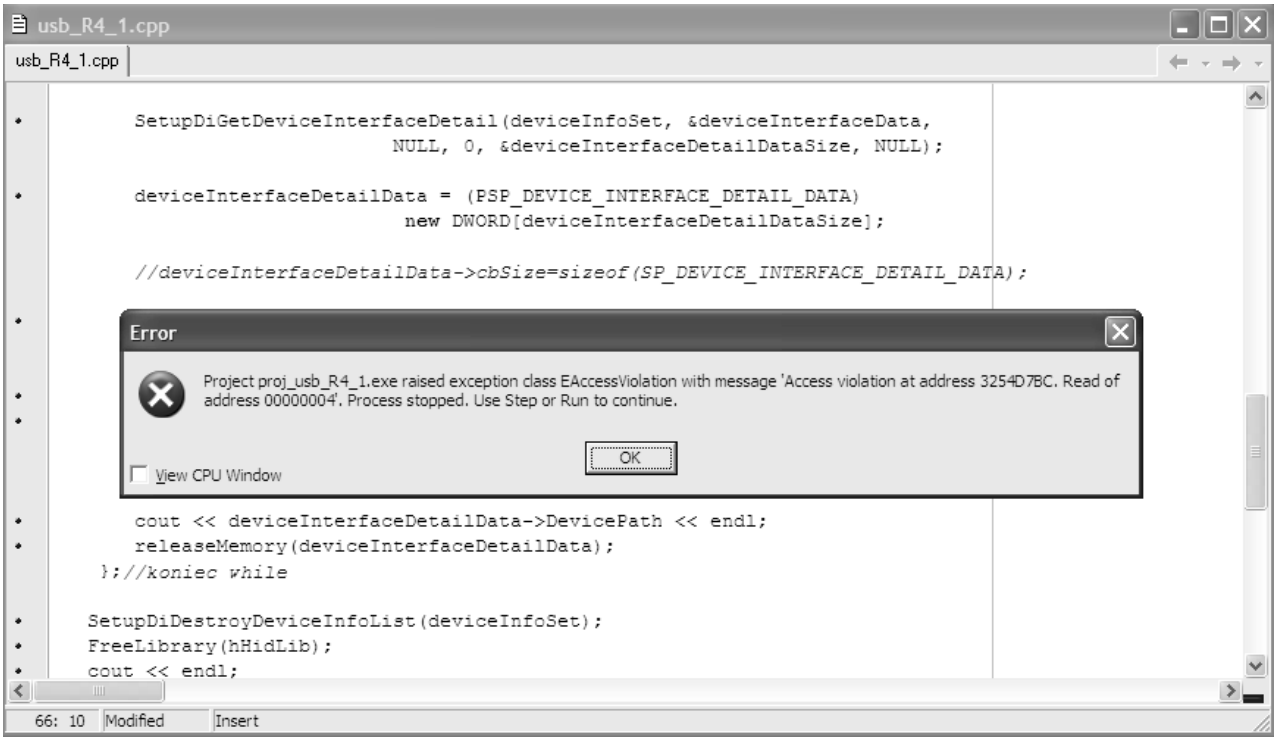


Rysunek 4.2. *Błąd naruszenia pamięci dla nieprawidłowo zainicjowanego wskaźnika do struktury SP_DEVICE_INTERFACE_DETAIL_DATA*

Argument DeviceInterfaceDetailDataSize funkcji SetupDiGetDeviceInterfaceDetail() ma wartość zerową, jeżeli DeviceInterfaceDetailData=NULL; w przeciwnym razie określa rozmiar bufora: (offsetof(SP_DEVICE_INTERFACE_DETAIL_DATA, DevicePath) + sizeof ↪(TCHAR)). Parametr RequiredSize jest wskaźnikiem do danej typu DWORD, której przypisuje się żądany rozmiar bufora wskazywanego przez DeviceInterfaceDetailData. Parametr DeviceInfoData jest wskaźnikiem do bufora danych przechowującego informacje na temat interfejsu urządzenia. Jeżeli wskaźnikowi nie przypisano wartości NULL, rozmiar danych powinien zostać określony za pomocą operatora sizeof(): Device ↪InfoData.cbSize=sizeof(SP_DEVINFO_DATA).

Funkcja SetupDiDestroyDeviceInfoList()

Funkcja usuwa wszystkie zaalokowane zasoby zawierające informacje o urządzeniu i zwalnia przydzieloną im pamięć. Kolejne urządzenia podłączane do systemu mogą korzystać ze zwolnionych zasobów.



Rysunek 4.3. *Błąd naruszenia pamięci dla nieprawidłowo określonego pola cbSize wskaźnika do struktury SP_DEVICE_INTERFACE_DETAIL_DATA*

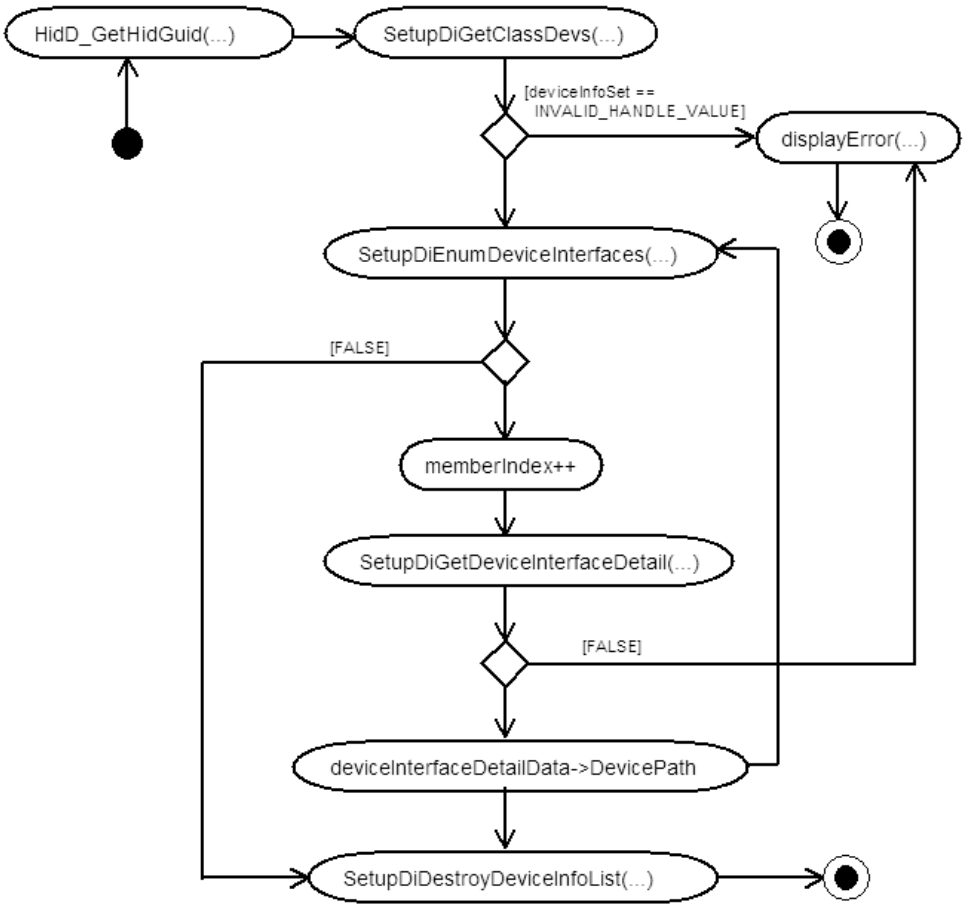
```
WINSETUPAPI BOOL WINAPI
  SetupDiDestroyDeviceInfoList(
    IN HDEVINFO  DeviceInfoSet
    );
```

Wskaźnik `DeviceInfoSet` zwracany jest przez funkcję `SetupDiGetClassDevs()`. W przypadku prawidłowego zwolnienia zasobów funkcja zwraca wartość `TRUE`, w przeciwnym razie wartość `FALSE`. Kod wystąpienia błędu zwracany jest przez funkcję `GetLastError()`.

Detekcja interfejsów urządzeń

Na rysunku 4.4 w postaci diagramu czynności przedstawiono ogólną sieć działań, za pomocą których można programowo samodzielnie wykonać procedurę detekcji urządzeń klasy HID aktualnie podłączonych do systemu, co w efekcie powinno skutkować odzyskaniem *pełnych* nazw symbolicznych (pełnych ścieżek dostępu do interfejsów) urządzeń zapisanych w polu `DevicePath` struktury `SP_DEVICE_INTERFACE_DETAIL_DATA`.

W pierwszej kolejności należy odczytać postać identyfikatora GUID interfejsu klasy urządzeń występujących w systemie. Wskaźnik `deviceInfoSet` wskaże dane zawierające informacje na temat wszystkich zainstalowanych i aktualnie dostępnych (przyłączonych) urządzeń danej klasy. Następnie wyszukiwane są interfejsy poszczególnych urządzeń. Poprzez odczytanie zawartości pola `DevicePath` struktury `SP_DEVICE_INTERFACE_DETAIL_DATA` wydobywana jest pełna ścieżka dostępu do interfejsu istniejącego urządzenia. Na koniec dotychczas używane przez program zasoby są zwalniane.



Rysunek 4.4. *Ogólny diagram czynności dla operacji wstępnej enumeracji urządzeń klasy HID*

Na listingu 4.1 zamieszczono kod modułu projektu będącego uszczegółowioną implementacją diagramu z rysunku 4.4.

Listing 4.1. *Kod modułu usb_R4_1.cpp jako przykład zaprogramowania wstępnej enumeracji urządzeń na podstawie identyfikatora GUID klasy urządzeń*

```
#include <windows>
#include <setupapi>
#include <assert>
#include <iostream>

using namespace std;

void displayError(const char* msg){
  cout << msg << endl;
  system("PAUSE");
  exit(0);
};
//-------------------------------------------------------
template <class T>
inline void releaseMemory(T &x)
{
  assert(x != NULL);
  delete x;
  x = NULL;
}
//-------------------------------------------------------
GUID classGuid;
```

```
HMODULE hHidLib;
DWORD /*unsigned long*/ memberIndex = 0;
DWORD deviceInterfaceDetailDataSize;
DWORD requiredSize;

HDEVINFO deviceInfoSet;
SP_DEVICE_INTERFACE_DATA deviceInterfaceData;
PSP_DEVICE_INTERFACE_DETAIL_DATA deviceInterfaceDetailData = NULL;

int main(){

  void (__stdcall *HidD_GetHidGuid)(OUT LPGUID HidGuid);

  hHidLib = LoadLibrary("C:\\Windows\\system32\\HID.DLL");
  if (!hHidLib)
    displayError("Błąd dołączenia biblioteki HID.DLL.");

  /*(void __stdcall*/(FARPROC&) HidD_GetHidGuid = GetProcAddress(hHidLib,
                                                    "HidD_GetHidGuid");
   if (!HidD_GetHidGuid){
      FreeLibrary(hHidLib);
      displayError("Nie znaleziono identyfikatora GUID");
   }

   HidD_GetHidGuid(&classGuid);

   deviceInfoSet = SetupDiGetClassDevs(&classGuid, NULL, NULL,
                   DIGCF_PRESENT | DIGCF_INTERFACEDEVICE);
   if (deviceInfoSet == INVALID_HANDLE_VALUE)
      displayError("Nie zidentyfikowano podłączonych urządzeń.\n");

   deviceInterfaceData.cbSize = sizeof(SP_DEVICE_INTERFACE_DATA);

   while(SetupDiEnumDeviceInterfaces(deviceInfoSet, NULL, &classGuid,
                                     memberIndex, &deviceInterfaceData)){
       memberIndex++; //inkrementacja numeru interfejsu

       SetupDiGetDeviceInterfaceDetail(deviceInfoSet, &deviceInterfaceData,
                            NULL, 0, &deviceInterfaceDetailDataSize, NULL);

       deviceInterfaceDetailData = (PSP_DEVICE_INTERFACE_DETAIL_DATA)
                             new DWORD[deviceInterfaceDetailDataSize];

       deviceInterfaceDetailData->cbSize=sizeof(SP_DEVICE_INTERFACE_DETAIL_DATA);

       if (!SetupDiGetDeviceInterfaceDetail(deviceInfoSet, &deviceInterfaceData,
                       deviceInterfaceDetailData, deviceInterfaceDetailDataSize,
                       &requiredSize, NULL)){
           releaseMemory(deviceInterfaceDetailData);
           //SetupDiDestroyDeviceInfoList(deviceInfoSet);
           //displayError("Nie można pobrać informacji o interfejsie.\n");
       }

       cout << deviceInterfaceDetailData->DevicePath << endl;
       releaseMemory(deviceInterfaceDetailData);
    };//koniec while
```

```
    SetupDiDestroyDeviceInfoList(deviceInfoSet);
    FreeLibrary(hHidLib);
    cout << endl;
    system("PAUSE");
    return 0;
}
//---------------------------------------------------------
```

Różne odmiany wskaźnika do funkcji `FARPROC` zdefiniowane są w module *windef.h*. W Linuksie `FARPROC` należy zastąpić wskaźnikiem ogólnym `void*`. Windows SDK API stosuje konwencję `__stdcall`, co zapewnia, że funkcja zostanie wywołana zgodnie z wymogami systemu operacyjnego. Oznacza to, że w funkcji wywołującej liczba i typ argumentów muszą być poprawne. Funkcje i typy danych definiowane w zasobach DDK API często posługują się następującymi makrodefinicjami:

```
#define DDKAPI __stdcall
```

lub

```
#define DDKAPI_PTR __stdcall*
```

W tego typu konwencjach deklaracja wskaźnika do funkcji może zostać zapisana na jeden z dwóch sposobów:

```
void (DDKAPI *HidD_GetHidGuid)(OUT LPGUID HidGuid);
```

lub

```
void (DDKAPI_PTR HidD_GetHidGuid)(OUT LPGUID HidGuid);
```

Na rysunku 4.5 przedstawiono działający program z listingu 4.1. Wynik działania programu należy porównać z odpowiednimi zapisami w edytorze rejestrów (zob. rysunek 2.1).

Rysunek 4.5. *Aplikacja proj_usb_R4_1 w trakcie działania*

Odczytane ścieżki dostępu do poszczególnych interfejsów urządzeń klasy HID mogą być przekazane do funkcji `CreateFile()` w celu otrzymania identyfikatora pliku sterownika urządzenia USB aktualnie dostępnego w systemie.