Visual C++. Gotowe rozwiązania dla programistów Windows

Autorzy: Jacek Matulewski, Maciej Pakulski, Dawid Borycki, Bartosz Biały, Piotr Pepłowski, Michał Matuszak, Daniel Szlag, Dawid Urbański
ISBN: 978-83-246-1928-3
Format: 158×235, stron: 536

Zostań znawcą środowiska programistycznego Visual C++

- Podstawowe funkcje i technologie systemu Windows w oczach programistów
- Praktyczne użycie funkcji WinAPI i biblioteki MFC
- Programowanie współbieżne dla procesorów wielordzeniowych

Środowisko programistyczne Microsoft Visual C++ idealnie nadaje się do wykorzystania w przypadku pisania programów dla platformy Win32 i jest chętnie wykorzystywane przez profesjonalnych programistów, tworzących aplikacje dla systemu Windows. Zarówno biblioteka MFC, jak i wbudowane funkcje WinAPI oraz możliwości programowania współbieżnego świetnie sprawdzają się w codziennej pracy programistycznej, oszczędzając czas, pozwalając na wykorzystanie mnóstwa kontrolek i funkcji, a także elastycznie dopasowując się do potrzeb tworzonej aplikacji.

Autorzy książki „Visual C++. Gotowe rozwiązania dla programistów Windows" skupiają się w niej nie tyle na opisie samego środowiska programistycznego, ile na możliwościach, jakie oferuje ono swoim użytkownikom. Po krótkim wprowadzeniu do projektowania interfejsu aplikacji przechodzą do kontroli stanu systemu, obsługi tworzonego programu, omówienia systemów plików, multimediów i rejestru, komunikatów Windows, bibliotek DLL oraz automatyzacji i wielu innych zagadnień. W publikacji tej znajdziesz gotowe odpowiedzi na wiele pytań dotyczących konkretnych kwestii programistycznych, rzeczowe porady oraz sposoby wykorzystania funkcji i technologii dostępnych podczas programowania w środowisku Visual C++.

- Projektowanie interfejsu aplikacji przy użyciu biblioteki MFC
- Kontrola stanu systemu
- Uruchamianie i kontrolowanie aplikacji oraz ich okien
- Systemy plików, multimedia i inne funkcje WinAPI
- Rejestr systemu Windows
- Komunikaty Windows
- Biblioteki DLL
- Automatyzacja i inne technologie oparte na COM
- Sieci komputerowe
- Programowanie współbieżne z OpenMP
- Biblioteka Threading Building Blocks

Dla tych, którzy w środowisku Visual C++ chcą się poczuć jak ryby w wodzie!

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

Spis treści

Rozdział 4.

Systemy plików, multimedia i inne funkcje WinAPI

Pliki i system plików (funkcje powłoki)

Interfejs użytkownika systemu Windows pozwala na uruchamianie programów, kontrolę plików i katalogów (z funkcjami kosza systemowego włącznie), drukowanie dokumentów, tworzenie skrótów do nich itp. W interfejsie programisty WinAPI tym operacjom odpowiadają tzw. funkcje powłoki, gdzie przez powłokę (ang. *shell*) rozumie się tę najwyższą warstwę systemu, która odpowiada za komunikację z użytkownikiem[1]. W ten sposób powłoka przesłania jądro i warstwy, do których użytkownik nie musi sięgać[2].

Funkcje WinAPI dotyczące powłoki są zazwyczaj prostsze w użyciu i bardziej zautomatyzowane niż ich głębsze odpowiedniki. Najlepszym przykładem jest opisana w poprzednim rozdziale funkcja `ShellExecute`, która jest znacznie łatwiejsza w użyciu od `CreateProcess`. Teraz Czytelnik pozna inne funkcje pozwalające na wygodniejsze manipulowanie plikami, w tym m.in. na korzystanie z kosza, operacje na grupach plików i całych katalogach. Omówimy także interfejsy COM należące do powłoki, które pozwalają na tworzenie skrótów, oraz interfejs *IFileOperation*, który jest dostępny w systemie Windows Vista.

---

[1] Słowo „interfejs" w tym i w poprzednim zdaniu oznacza oczywiście coś innego. W pierwszym przypadku chodzi o GUI (ang. *graphic user interface*), a więc okna, menu i inne graficzne elementy aplikacji widoczne na ekranie, podczas gdy w drugim mowa o bibliotece funkcji pozwalających na kontrolę systemu Windows. Funkcje powłoki to podzbiór funkcji interfejsu WinAPI, które pozwalają na kontrolę interfejsu GUI.

[2] Zob. „Blokowanie dostępu do komputera" w rozdziale 2.

Odczytywanie ścieżek do katalogów specjalnych

> Ścieżki do katalogów specjalnych użytkownika (np. katalogu z dokumentami czy pulpitu) można odczytać z rejestru (por. rozdział 5.). Jednak nie jest to sposób zalecany. Przedstawione tutaj rozwiązanie korzystające z funkcji powłoki jest poprawnym sposobem odczytywania ścieżki do tych katalogów.

Do odczytania katalogów specjalnych systemu i profilu użytkownika służy funkcja `SHGetSpecialFolderPath`[3] zdefiniowana w nagłówku *shlobj.h*. Jej trzeci argument wskazuje interesujący nas katalog. Najbardziej popularne to: `CSIDL_PERSONAL` (*Moje dokumenty*), `CSIDL_DESKTOP` (*Pulpit*), `CSIDL_WINDOWS` (*C:\Windows*) i `CSIDL_SYSTEM` (*C:\ Windows\ System32*). Część stałych odpowiadających katalogom definiowanym dla każdego użytkownika ma wersje zawierające `_COMMON_`. Odnoszą się one do odpowiednich katalogów zawierających elementy dostępne w profilach wszystkich użytkowników (w Windows XP są to podkatalogi katalogu *C:\Documents and Settings\All Users*, a w Windows Vista są to podkatalogi katalogu *C:\Users* (*Użytkownicy*)), np. `CSIDL_`
`↪COMMON_DESKTOPDIRECTORY`[4]. Listing 4.1 zawiera kilka przykładowych funkcji zwracających uzyskane dzięki wywołaniu funkcji `SHGetSpecialFolderPath` ścieżki do katalogów specjalnych w postaci obiektu `CString` — łańcucha wygodnego do użycia w aplikacjach korzystających z biblioteki MFC.

Listing 4.1. *Zbiór funkcji zwracających ścieżki do katalogów specjalnych*

```
CString Katalog_Windows()
{
    TCHAR path[MAX_PATH];
    SHGetSpecialFolderPath(NULL, path, CSIDL_WINDOWS, FALSE);
    return CString(path);
}

CString Katalog_System()
{
    TCHAR path[MAX_PATH];
    SHGetSpecialFolderPath(NULL, path, CSIDL_SYSTEM, FALSE);
    return CString(path);
}

CString Katalog_MojeDokumenty()
{
    TCHAR path[MAX_PATH];
    SHGetSpecialFolderPath(NULL, path, CSIDL_PERSONAL, FALSE);
    return CString(path);
}

CString Katalog_AllUsers_Pulpit()
{
    TCHAR path[MAX_PATH];
```

[3] Funkcja ta działa w każdej wersji systemu Windows, jednak w Windows 95 i NT 4.0 wymaga zainstalowania Internet Explorera 4.0.

[4] Wszystkie stałe CSIDL znajdzie Czytelnik w dokumentacji MSDN pod hasłem CSIDL.

```
    SHGetSpecialFolderPath(NULL, path, CSIDL_COMMON_DESKTOPDIRECTORY, FALSE);
    return CString(path);
}

CString Katalog_Pulpit()
{
    TCHAR path[MAX_PATH];
    SHGetSpecialFolderPath(NULL, path, CSIDL_DESKTOPDIRECTORY, FALSE);
    return CString(path);
}
```

Tworzenie skrótu (.lnk)

W tym projekcie po raz pierwszy będziemy mieli do czynienia z obiektem zdefiniowanym w systemie Windows i udostępnionym programistom w ramach mechanizmu COM (ang. *Component Object Model*). Pełniejsze wprowadzenie do COM i związanych z nim technologii znajdzie Czytelnik w rozdziale 4. Tutaj ograniczę się zatem jedynie do omówienia funkcji wykorzystanych w poniższym kodzie (w komentarzu na końcu tego projektu).

Zgodnie z zasadami przedstawionymi we wstępie zasadnicze funkcje napisane zostaną w taki sposób, aby mogły być użyte w dowolnym projekcie, lecz w przykładach ich użycia skorzystamy z typów zdefiniowanych w MFC, w szczególności dotyczy to łańcuchów.

1. Tworzymy nowy projekt aplikacji MFC z oknem dialogowym o nazwie *PlikSkrotu*.
2. Do projektu dodajemy pliki *Skrot.h* i *Skrot.cpp* (oczywiście do odpowiednich gałęzi w drzewie plików projektu widocznym w *Solution Explorer*).
3. W pliku nagłówkowym `Skrot.h` definiujemy strukturę pomocniczą `CParametrySkrotu`, której pola będą przechowywały następujące własności skrótu: pełną ścieżkę wraz z nazwą pliku, do którego chcemy utworzyć skrót, opis, katalog roboczy, klawisz skrótu (litera, która razem z klawiszami *Ctrl* i *Alt* będzie uruchamiała skrót, jeżeli będzie umieszczony na pulpicie lub na pasku szybkiego uruchamiania), ścieżkę do pliku zawierającego ikonę skrótu oraz numer ikony w tym pliku (listing 4.2).

Listing 4.2. *Zawartość pliku nagłówkowego Skrot.h*

```
#pragma once

struct CParametrySkrotu
{
   TCHAR sciezkaPliku[MAX_PATH], katalogRoboczy[MAX_PATH], sciezkaIkony[MAX_PATH];
   TCHAR argumenty[256], opis[256];
   int rodzajOkna, numerIkony;
   wchar_t klawiszSkrotu;
};

BOOL TworzSkrot(LPCTSTR sciezkaLinku, CParametrySkrotu parametrySkrotu);
```

4. Listing 4.2 zawiera również deklaracje dwóch funkcji, które zdefiniujemy w pliku *Skrot.cpp*, a które służyć będą do tworzenia i odczytywania pliku skrótu.

5. Przechodzimy do edycji pliku *Skrot.cpp*. Umieszczamy w nim dyrektywę dołączającą nagłówek *shlwapi.h*, zawierający deklarację m.in. funkcji służących do operacji na ścieżkach do pliku, w szczególności funkcji `PathRemoveFileSpec`, która usuwa z pełnej ścieżki do pliku nazwę pliku, a pozostawia jedynie ścieżkę katalogu. Definiujemy w pliku również funkcję tworzącą skrót (listing 4.3).

Listing 4.3. *Omówienie funkcji znajduje się w komentarzu poniżej*

```
#include "stdafx.h"
#include "Skrot.h"
#include <shlwapi.h> // PathRemoveFileSpec

BOOL TworzSkrot(LPCTSTR sciezkaLinku, CParametrySkrotu parametrySkrotu)
{
   CoInitializeEx(NULL, COINIT_MULTITHREADED);

   IShellLink* pISLink;
   if (CoCreateInstance(CLSID_ShellLink,
                NULL,
                CLSCTX_INPROC_SERVER,
                IID_IShellLink,
                (void**) &pISLink) != S_OK) return FALSE;

   IPersistFile* pIPFile;
   pISLink->QueryInterface(IID_IPersistFile,(void**) &pIPFile);

   //przygotowanie parametrów
   if (wcscmp(parametrySkrotu.sciezkaPliku,L"")==0) THROW("Brak nazwy pliku, do
   ↪którego ma zostać utworzony skrót");
   if (wcscmp(parametrySkrotu.katalogRoboczy,L"")==0)
   {
      wcscpy_s(parametrySkrotu.katalogRoboczy, MAX_PATH, parametrySkrotu.sciezkaPliku);

PathRemoveFileSpecW(parametrySkrotu.katalogRoboczy);//parametrySkrotu.katalogRoboczy.
↪GetBuffer());
   }
   if (parametrySkrotu.rodzajOkna == 0) parametrySkrotu.rodzajOkna = SW_SHOWNORMAL;
   //nie dopuszczamy SW_HIDE=0 ze względu na taką domyślną inicjację
   parametrySkrotu.klawiszSkrotu = toupper(parametrySkrotu.klawiszSkrotu);

   //przygotowanie obiektu
   pISLink->SetPath(parametrySkrotu.sciezkaPliku);
   pISLink->SetWorkingDirectory(parametrySkrotu.katalogRoboczy);
   pISLink->SetArguments(parametrySkrotu.argumenty);
   if (parametrySkrotu.opis != L"") pISLink->SetDescription(parametrySkrotu.opis);
   pISLink->SetShowCmd(parametrySkrotu.rodzajOkna);
   if (parametrySkrotu.sciezkaIkony != L"") pISLink->SetIconLocation(parametry
   ↪Skrotu.sciezkaIkony, parametrySkrotu.numerIkony);
   if (parametrySkrotu.klawiszSkrotu != NULL) pISLink->SetHotkey(((HOTKEYF_ALT |
   ↪HOTKEYF_CONTROL) << 8) | parametrySkrotu.klawiszSkrotu);
   BOOL wynik = (pIPFile->Save(sciezkaLinku, FALSE) == S_OK);

   pISLink->Release();
```

```
    CoUninitialize();

    return wynik;
}
```

6. Aby przetestować funkcję `TworzSkrot`, przechodzimy do widoku projektowania i na podglądzie formy umieszczamy przycisk. Klikając go dwukrotnie, tworzymy domyślną metodę zdarzeniową, w której umieszczamy polecenia z listingu 4.4. W pliku nagłówkowym *PlikSkrotuDlg.h* należy wcześniej dołączyć nowy nagłówek za pomocą dyrektywy prekompilatora: `#include "Skrot.h"`.

Listing 4.4. *Tworzymy skrót do bieżącej aplikacji w bieżącym katalogu*

```
void CPlikSkrotuDlg::OnBnClickedButton1()
{
   CParametrySkrotu parametrySkrotu;
   GetModuleFileName(GetModuleHandle(NULL), parametrySkrotu.sciezkaPliku, MAX_PATH);
   wcscpy_s(parametrySkrotu.katalogRoboczy,MAX_PATH,parametrySkrotu.sciezkaPliku);
   PathRemoveFileSpec(parametrySkrotu.katalogRoboczy);
   wcscpy_s(parametrySkrotu.argumenty,260,L"");
   wcscpy_s(parametrySkrotu.opis,260,AfxGetApp()->m_pszAppName);
   parametrySkrotu.rodzajOkna = SW_SHOWNORMAL;
   wcscpy_s(parametrySkrotu.sciezkaIkony,MAX_PATH,parametrySkrotu.sciezkaPliku);
    parametrySkrotu.numerIkony = 0;
    parametrySkrotu.klawiszSkrotu = 'y';

   TworzSkrot(L"Skrot.lnk", parametrySkrotu);
}
```

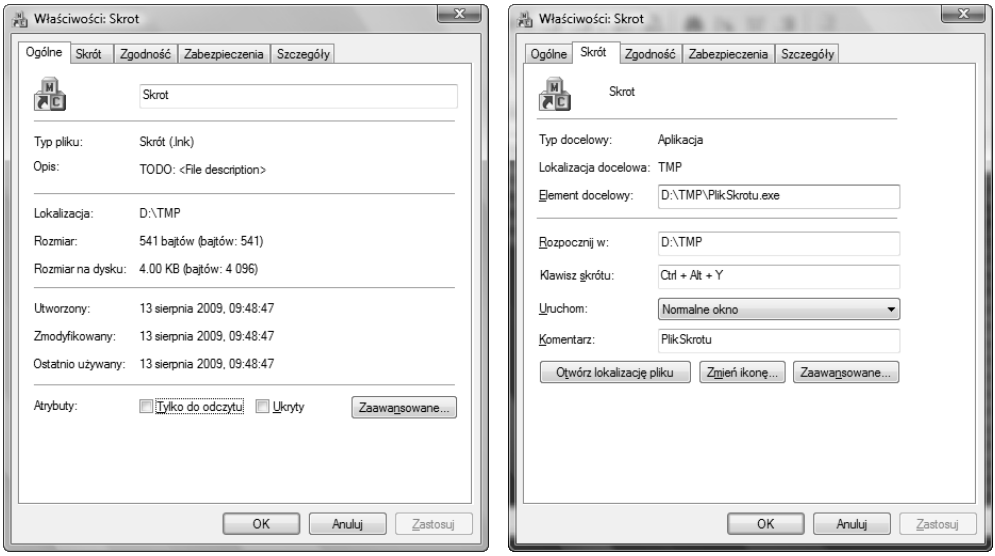
Listing 4.3 zawiera zasadniczą funkcję `TworzSkrot`. W pierwszej linii kodu tej funkcji inicjujemy bibliotekę COM, korzystając z funkcji `CoInitializeEx`. Zwykle polecenie to umieszcza się w jednej z funkcji inicjujących aplikacji, np. na początku funkcji `OnInitDialog`, w pliku `PlikSkrotuDlg.cpp`. My umieściliśmy ją w funkcji `TworzSkrot`, aby zwrócić uwagę Czytelnika, że powinna być wywołana przed utworzeniem obiektu COM, a poza tym, aby funkcja `TworzSkrot` była bardziej autonomiczna, co ułatwi jej użycie w projektach Czytelnika. Następnie tworzymy instancję obiektu COM, posługując się funkcją `CoCreateInstance` z identyfikatorem obiektu `CLSID_ShellLink`. Funkcja ta zapisuje we wskaźniku typu `IShellLink*` (nazwa użytej przez nas zmiennej to `pISLink`) wskaźnik do utworzonego obiektu COM. Typ `IShellLink` oraz użyty później `IPersist`
`↪File` to interfejsy, czyli w nomenklaturze technologii COM zbiory funkcji (metod), które podobnie jak klasy mogą dziedziczyć z innych interfejsów (w tym przypadku z `IUnknown`). Interfejsy te umożliwiają dostęp do metod utworzonego przez nas obiektu COM. Interfejs `IShellLink` udostępnia metody pozwalające na ustalenie lub odczytanie własności skrótu (pliku z rozszerzeniem *.lnk*). Natomiast `IPersistFile`[5] zawiera metody `Save` i `Load`, które pozwalają zapisać w pliku i odczytać z niego atrybuty ustalone przez interfejs `IShellLink`. Po zakończeniu korzystania z obiektu należy go jeszcze zwolnić, używając metody `Release`.

[5] Oba interfejsy dostępne są we wszystkich 32-bitowych wersjach Windows, poza Windows NT 3.x.

Jeżeli skrót o podanej nazwie już istnieje, powyższa funkcja nadpisze go bez pytania o zgodę.

Do wskazania ścieżki pliku, do którego tworzymy skrót, wykorzystujemy metodę `IShellLink::SetPath`; do wskazania katalogu roboczego — `IShellLink::SetWorking↪Directory`; do opisu — `ISLink::SetDescription` itd. Klawisz skrótu (w przypadku plików *.lnk* obowiązkowa jest kombinacja klawiszy *Ctrl+Alt*) ustalamy metodą `ISLink::SetHotKey`. Jej argument to liczba typu `Word`, w której mniej znaczący bajt zajmuje znak typu `char`, a w górnym, bardziej znaczącym bajcie zapalamy bity wskazane przez stałe `HOTKEYF_ALT` i `HOTKEYF_CONTROL` (czyli w efekcie 00000110).

Plik zapisany przez metodę z listingu 4.4 można sprawdzić za pomocą systemowego edytora skrótów (rysunek 4.1).



Rysunek 4.1. *Systemowy edytor skrótów. Dziwny opis pliku na lewym rysunku jest domyślnym opisem aplikacji w zasobach projektu; można go oczywiście z łatwością zmienić*

Odczyt i edycja skrótu .lnk

Zdefiniujemy funkcję `CzytajSkrot`, która umieści informacje o skrócie w strukturze `CParametrySkrotu` zdefiniowanej w listingu 4.2. Edycja tej struktury nie powinna sprawić żadnych trudności. Po modyfikacji informacje o skrócie można ponownie zapisać, korzystając z funkcji `TworzSkrot`.

1. Funkcja `CzytajSkrot` z listingu 4.5 powinna znaleźć się w pliku *Skrot.cpp*, natomiast do pliku nagłówkowego *Skrot.h* należy dodać jej deklarację. Nie zawiera ona zasadniczo nowych elementów. Jeszcze raz wykorzystujemy obiekt identyfikowany przez stałą `CLSID_ShellLink` i interfejsy `IShellLink` oraz `IPersistFile`.

Listing 4.5. *Definicja funkcji CzytajSkrot w wersji dla Win32*

```
BOOL CzytajSkrot(LPCTSTR sciezkaLinku, CParametrySkrotu& parametrySkrotu)
{
   CoInitializeEx(NULL, COINIT_MULTITHREADED);
   IShellLink* pISLink;
   if (CoCreateInstance (CLSID_ShellLink,
      NULL,
      CLSCTX_INPROC_SERVER,
      IID_IShellLink,
      (void**) &pISLink) != S_OK) return FALSE;

   IPersistFile* pIPFile;
   pISLink->QueryInterface(IID_IPersistFile,(void**) &pIPFile);

   if (pIPFile->Load(sciezkaLinku, 0) != S_OK)
   {
      pISLink->Release();
      return FALSE;
   }

   TCHAR cstr[MAX_PATH];
   WIN32_FIND_DATA informacjeOPliku; //tu nie wykorzystywane
   pISLink->GetPath(parametrySkrotu.sciezkaPliku, MAX_PATH, &informacjeOPliku,
   ↪SLGP_UNCPRIORITY);

   pISLink->GetWorkingDirectory(parametrySkrotu.katalogRoboczy , MAX_PATH);

   pISLink->GetArguments(cstr, MAX_PATH);
   wcscpy_s(parametrySkrotu.argumenty,260,cstr);

   pISLink->GetDescription(cstr, MAX_PATH);
   wcscpy_s(parametrySkrotu.opis,260,cstr);

   pISLink->GetShowCmd(&(parametrySkrotu.rodzajOkna));

   pISLink->GetIconLocation(parametrySkrotu.sciezkaIkony, MAX_PATH,
    ↪&(parametrySkrotu.numerIkony));

   WORD klawiszSkrotu;
   pISLink->GetHotkey(&klawiszSkrotu);
   parametrySkrotu.klawiszSkrotu = (klawiszSkrotu & 255);

   pISLink->Release();

   return TRUE;
}
```

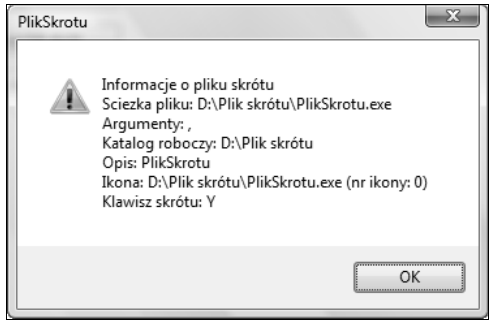
2. Aby przetestować funkcję `CzytajSkrot`, umieszczamy na podglądzie formy kolejny przycisk i tworzymy jego domyślną metodę zdarzeniową. Umieszczamy w niej polecenia z listingu 4.6.

Listing 4.6. *Ograniczymy się do zaprezentowania parametrów skrótu w oknie komunikatu*

```
void CPlikSkrotuDlg::OnBnClickedButton2()
{
   CParametrySkrotu parametrySkrotu;
   if (CzytajSkrot(L"Skrot.lnk", parametrySkrotu))
```

```
    {
        CString temp;
        temp.Format(L"Informacje o pliku skrótu\nSciezka pliku: %s\nArgumenty:
        ↪%s,\nKatalog roboczy: %s\nOpis: %s\nIkona: %s (nr ikony: %d)\nKlawisz
        ↪skrótu: %c",
            parametrySkrotu.sciezkaPliku,
            parametrySkrotu.argumenty,
            parametrySkrotu.katalogRoboczy,
            parametrySkrotu.opis,
            parametrySkrotu.sciezkaIkony,
            parametrySkrotu.numerIkony,
            parametrySkrotu.klawiszSkrotu);
        AfxMessageBox(temp);
    }
}
```

3. Po uruchomieniu aplikacji możemy kliknąć nowy przycisk. Powinniśmy wówczas zobaczyć opis skrótu jak na rysunku 4.2.



Rysunek 4.2. *Odczytane z pliku parametry skrótu*

Umieszczenie skrótu na pulpicie

Aby umieścić skrót na pulpicie, wystarczy połączyć wiedzę z dwóch pierwszych projektów w rozdziale. Listing 4.7 pokazuje, jak to zrobić. Zupełnie analogicznie wyglądałoby umieszczenie skrótu np. w menu *Start*.

Listing 4.7. *Wykorzystujemy funkcję TworzSkrot, wskazując ścieżkę pliku skonstruowaną za pomocą funkcji Katalog_Pulpit*

```
void CPlikSkrotuDlg::TworzSkrotNaPulpicie(LPCTSTR sciezkaLinku, CParametrySkrotu
↪parametrySkrotu)
{
    CString temp;
    temp.Format(L"%s\\%s", Katalog_Pulpit(), sciezkaLinku);
    TworzSkrot(temp, parametrySkrotu);
}
```

Operacje na plikach i katalogach (funkcje WinAPI)

Użytkownik ma do wyboru trzy sposoby, za pomocą których może wykonywać operacje na plikach. Po pierwsze, może wykorzystać standardowe funkcje C++ (ten sposób omówiono niżej). Po drugie, może użyć zbioru funkcji WinAPI (`CopyFile`, `MoveFile`, `DeleteFile` itp.). Te niestety zostały dodane do WinAPI dopiero od Windows 2000, co oczywiście ogranicza przenośność korzystających z nich aplikacji. I po trzecie, użytkownik może użyć funkcji powłoki o nazwie `SHFileOperation`, która pozwala między innymi na operacje na grupach plików, całych katalogach, czy na przeniesienie pliku do kosza. Ten ostatni sposób zostanie omówiony w następnych projektach. Wybór między pierwszym i drugim sposobem nie jest rozłączny; nie wszystkie operacje da się łatwo wykonać za pomocą funkcji C++. Nie ma na przykład gotowej funkcji pozwalającej na kopiowanie pliku. Do tego koniecznie trzeba użyć funkcji WinAPI, np.

```
CopyFile(L"D:\\TMP\\Log.txt",L"D:\\TMP\\Log.bak",FALSE)
```

lub bardziej złożonej konstrukcji:

```
if (!CopyFile(L"D:\\TMP\\Log.txt",L"D:\\TMP\\Log.bak",FALSE))
      ::MessageBox(NULL,L"Operacja kopiowania nie powiodła się!",L"Błąd ",MB_OK);
   else
      ::MessageBox(NULL,L"Kopiowanie zakończone",L"Informacja",MB_OK);
```

Jak łatwo się domyślić, pierwsze dwa argumenty wskazują nazwę pliku źródłowego i nową nazwę pliku. Natomiast trzeci argument to wartość logiczna określająca, czy możliwe jest nadpisywanie istniejącego pliku. Funkcja ta pozwala kopiować także katalogi razem z zawartością i podkatalogami.

Istnieje również funkcja `CopyFileEx`, pozwalająca na śledzenie postępu kopiowania pliku, który można np. pokazać na pasku postępu.

Podobnie działa funkcja `MoveFile` (także z WinAPI), przenosząca plik (zmieniająca jego położenie w tablicy alokacji plików):

```
MoveFile(L"D:\\TMP\\Log.txt",L"D:\\TMP\\Log_nowy.txt");
```

Funkcja `MoveFileWithProgress` pozwala na śledzenie postępu przenoszenia pliku, a także na ustalenie sposobu przenoszenia (np. opóźnienie do momentu ponownego uruchamiania). Ta ostatnia możliwość dostępna jest także w funkcji `MoveFileEx`.

Aby usunąć plik, można skorzystać z funkcji `DeleteFile`:

```
DeleteFile(L"D:\\TMP\\Log.txt");
```

Podobnie wygląda sprawa z katalogami. Do tworzenia katalogu można użyć funkcji C++ `mkdir` lub funkcji WinAPI `CreateDirectory`. Ta ostatnia dołączona została jednak dopiero w Windows 2000. Do zmiany bieżącego katalogu można użyć funkcji `chdir`, natomiast do usuwania pustego katalogu — `rmdir` lub wspomnianej już funkcji `DeleteFile`. Funkcje te są zadeklarowane w nagłówku *dir.h* i „od zawsze" należą do standardu C++. Nie należy się jednak obawiać używania ich w 32-bitowych wersjach Windows, gdyż obecnie są one po prostu „nakładkami" na analogiczne funkcje WinAPI. Do pobrania ścieżki bieżącego katalogu można użyć funkcji `GetCurrentDirectory`.

Operacje na plikach i katalogach (funkcje powłoki)

W poprzednim projekcie użyliśmy niskopoziomowych funkcji interfejsu programistycznego Windows (WinAPI) `CopyFile`, `MoveFile` czy `DeleteFile` do wykonywania podstawowych operacji na plikach. Chciałbym jednak zwrócić uwagę Czytelnika na inny sposób wykonania tych operacji, który może wydawać się z początku nieco bardziej skomplikowany, ale za to daje dodatkowe korzyści i przy wykonywaniu bardziej złożonych operacji okazuje się o wiele prostszy niż korzystanie z funkcji niskopoziomowych. Użyjemy do tego funkcji WinAPI `SHFileOperation` z biblioteki *shell32.dll*. Jedną z ich zalet jest to, że dostępne są już od Windows 95 i NT 4.0, czyli we wszystkich 32-bitowych wersjach Windows. Biblioteka ta wykorzystywana jest m.in. przez *Eksploratora Windows* i dlatego funkcje odwołują się do mechanizmów charakterystycznych dla eksploratora, m.in. kosza systemowego czy katalogów specjalnych. W odróżnieniu od poprzednio użytych funkcji niskopoziomowych funkcja `SHFileOperation` należy do warstwy powłoki (interfejsu graficznego) i dlatego jej działaniu towarzyszą okna żądające potwierdzenia chęci utworzenia katalogu, ostrzegające przed nadpisaniem pliku itp.

1. Tworzymy nowy projekt aplikacji MFC z oknem dialogowym o nazwie *OperacjeNaPlikach*.
2. W pliku nagłówkowym *OperacjeNaPlikachDlg.h* importujemy potrzebny moduł:

```
#include <shlwapi.h>
```

3. W tym samym pliku deklarujmy również funkcje składowe:

```
BOOL KopiowaniePliku(HWND uchwyt, LPCTSTR szZrodlo, LPCTSTR szCel);
BOOL PrzenoszeniePliku(HWND uchwyt, LPCTSTR szZrodlo, LPCTSTR szCel);
BOOL UsuwaniePliku(HWND uchwyt, LPCTSTR szZrodlo);
```

4. Przejdźmy do pliku źródłowego i zdefiniujmy funkcję pomocniczą `OperacjaNaPliku`, dzięki której korzystanie z `SHFileOperation` będzie łatwiejsze (listing 4.8).

Listing 4.8. *Funkcja „prywatna" pozwalająca na uniknięcie powtarzania kodu*

```
BOOL COperacjeNaPlikachDlg::OperacjaNaPliku(HWND uchwyt, LPCTSTR szZrodlo,
↪LPCTSTR szCel, DWORD operacja, DWORD opcje)
{
   if(!PathFileExists(szZrodlo)) // Czy plik źródłowy istnieje?
   {
      AfxMessageBox(L"Nie odnaleziono pliku");
      return FALSE;
   }

   SHFILEOPSTRUCT parametryOperacji;
   parametryOperacji.hwnd = uchwyt;
   parametryOperacji.wFunc = operacja;

   if (szZrodlo != L"")
      parametryOperacji.pFrom = szZrodlo;
  else
      parametryOperacji.pFrom = NULL;

   if (szCel != L"")
      parametryOperacji.pTo = szCel;
```

```
   else
   {
      parametryOperacji.pTo = NULL;
      parametryOperacji.fFlags = opcje;
      parametryOperacji.hNameMappings = NULL;
      parametryOperacji.lpszProgressTitle = NULL;
   }
   return (SHFileOperation(&parametryOperacji) == 0);
}
```

5. Wreszcie definiujemy zadeklarowany w pliku nagłówkowym zestaw metod (zadeklarowaliśmy je w nagłówku) wykonujących konkretne czynności na plikach (listing 4.9).

Listing 4.9. *Zbiór funkcji „publicznych”*

```
BOOL COperacjeNaPlikachDlg::KopiowaniePliku(HWND uchwyt, LPCTSTR szZrodlo,
↪LPCTSTR szCel)
{
   TCHAR lpBuffer[MAX_PATH] = {0};
   GetFullPathName(szZrodlo, MAX_PATH, lpBuffer, NULL);

   return OperacjaNaPliku(uchwyt, lpBuffer, szCel, FO_COPY, 0);
}

BOOL COperacjeNaPlikachDlg::PrzenoszeniePliku(HWND uchwyt, LPCTSTR szZrodlo,
↪LPCTSTR szCel)
{
   TCHAR lpBuffer[MAX_PATH] = {0};
   GetFullPathName(szZrodlo, MAX_PATH, lpBuffer, NULL);

   return OperacjaNaPliku(uchwyt, lpBuffer, szCel, FO_MOVE, 0);
}

BOOL COperacjeNaPlikachDlg::UsuwaniePliku(HWND uchwyt, LPCTSTR szZrodlo)
{
   TCHAR lpBuffer[MAX_PATH] = {0};
   GetFullPathName(szZrodlo, MAX_PATH, lpBuffer, NULL);

   return OperacjaNaPliku(uchwyt, lpBuffer, L"", FO_DELETE, 0);
}
```

6. Listing 4.10 zawiera domyślną metodę zdarzeniową kliknięcia przycisku `Button1`, która testuje działanie powyższych funkcji.

Listing 4.10. *Aby poniższy test zadziałał, musimy dysponować dyskiem d:. Można to oczywiście łatwo zmienić*

```
void COperacjeNaPlikachDlg::OnBnClickedButton1()
{
   TCHAR path[MAX_PATH] = {0};
   GetModuleFileName(GetModuleHandle(NULL), path, MAX_PATH);
   KopiowaniePliku(m_hWnd, path, L"d:\\Kopia projektu.sln");
   PrzenoszeniePliku(m_hWnd, L"d:\\Kopia projektu.sln", L"d:\\Kopia pliku.xml");
   UsuwaniePliku(m_hWnd, L"d:\\Kopia pliku.xml");
}
```

Operacje na plikach i katalogach w Windows Vista (interfejs IFileOperation)

W systemie Windows Vista oddano do użytku interfejs `IFileOperation`, który zastępuje opisaną wcześniej strukturę `SHFILEOPSTRUCT`. Nie oznacza to jednak, iż nie możemy już korzystać z tej struktury w nowych wersjach Windows. Interfejs `IFileOperation` wydaje się jednak wygodniejszy w użyciu. Ułatwia śledzenie postępu wykonywanych operacji oraz zapewnia możliwość wykonywania wielu operacji jednocześnie, a także bardziej szczegółowo informuje o błędach. W celu użycia funkcji udostępnianych przez interfejs `IFileOperation` należy korzystać z obiektu `IShellItem` przy określeniu ścieżki do plików i katalogów. Dzięki temu można wykonywać operacje nie tylko na plikach i katalogach, ale również na takich obiektach, jak foldery wirtualne. Przykład wykorzystania omawianego interfejsu przedstawiają poniższe projekty.

1. Tworzymy nowy projekt aplikacji MFC z oknem dialogowym o nazwie *OperacjeNaPlikachIF*.
2. Na początku pliku *OperacjeNaPlikachIFDlg.cpp* umieszczamy trzy dyrektywy:

```
#include <shobjidl.h>
#include <shlobj.h>
#include <shlwapi.h>
```

3. Natomiast w pliku nagłówkowym deklarujemy prywatną funkcję `OperacjaNaPlikuIF` i definiujemy ją zgodnie z listingiem 4.11.

Listing 4.11. *Funkcja wykonująca operacje na pliku (kopiowanie, przenoszenie, usuwanie) wykorzystująca interfejs IFileOperation*

```
HRESULT COperacjeNaPlikachIFDlg::OperacjaNaPlikuIF(LPCTSTR szZrodlo, LPCTSTR szCel,
↪DWORD operacja, DWORD opcje)
{
   if(!PathFileExists(szZrodlo)) // Czy plik źródłowy istnieje?
   {
      AfxMessageBox(L"Nie odnaleziono pliku");
      return E_POINTER;
   }

   CT2W wszZrodlo(szZrodlo);
   CT2W wszCel(szCel);
   CString wszNowaNazwa = PathFindFileNameW(wszCel);
   PathRemoveFileSpec(wszCel);

   // Tworzenie nowej instancji interfejsu IFileOperation
   IFileOperation *iFo;
   HRESULT hr = CoCreateInstance(CLSID_FileOperation,
                                 NULL,
                                 CLSCTX_LOCAL_SERVER,
                                 IID_PPV_ARGS(&iFo));
   if(!SUCCEEDED(hr))
      return hr;

   iFo->SetOperationFlags(opcje);

   // Tworzenie obiektów IShellItem
```

```
    IShellItem *psiCel = NULL, *psiZrodlo = NULL;
    SHCreateItemFromParsingName(wszZrodlo, NULL, IID_PPV_ARGS(&psiZrodlo));
    if(wszCel != NULL)
       SHCreateItemFromParsingName(wszCel, NULL, IID_PPV_ARGS(&psiCel));

    // Kopiowanie, przenoszenie czy usuwanie?
    switch(operacja)
    {
       case FO_COPY: iFo->CopyItem(psiZrodlo, psiCel, wszNowaNazwa, NULL); break;
       case FO_MOVE: iFo->MoveItem(psiZrodlo, psiCel, wszNowaNazwa, NULL); break;
       case FO_DELETE: iFo->DeleteItem(psiZrodlo, NULL); break;
    }

    // Potwierdzenie wykonania operacji
    hr = iFo->PerformOperations();
    if(!SUCCEEDED(hr))
       return hr;

    psiZrodlo->Release();
    if(psiCel != NULL)
       psiCel->Release();

    // Zwolnienie interfejsu
    iFo->Release();

    return hr;
}
```

4. Analogicznie jak w poprzednim projekcie deklarujemy i definiujemy publiczne metody odpowiedzialne za kopiowanie, przenoszenie i usuwanie plików (listing 4.12). Jednakże teraz zamiast zwracać wartość `BOOL`, zwracamy `HRESULT`.

Listing 4.12. *Metody realizujące kopiowanie, przenoszenie i usuwanie plików*

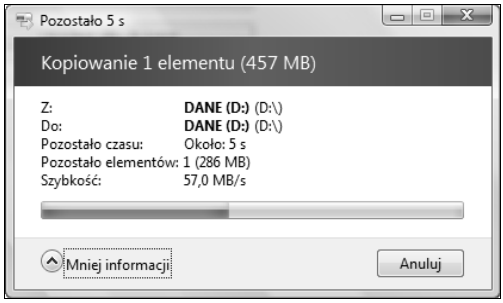
```
HRESULT COperacjeNaPlikachIFDlg::KopiowaniePliku(LPCTSTR szZrodlo, LPCTSTR szCel)
{
   TCHAR lpBuffer[MAX_PATH] = {0};
   GetFullPathName(szZrodlo, MAX_PATH, lpBuffer, NULL);

   return OperacjaNaPlikuIF(lpBuffer, szCel, FO_COPY, 0);
}

HRESULT COperacjeNaPlikachIFDlg::PrzenoszeniePliku(LPCTSTR szZrodlo, LPCTSTR szCel)
{
   TCHAR lpBuffer[MAX_PATH] = {0};
   GetFullPathName(szZrodlo, MAX_PATH, lpBuffer, NULL);

   return OperacjaNaPlikuIF(lpBuffer, szCel, FO_MOVE, 0);
}

HRESULT COperacjeNaPlikachIFDlg::UsuwaniePliku(LPCTSTR szZrodlo)
{
   TCHAR lpBuffer[MAX_PATH] = {0};
   GetFullPathName(szZrodlo, MAX_PATH, lpBuffer, NULL);

   return OperacjaNaPlikuIF(lpBuffer, NULL, FO_DELETE, 0);
}
```

5. W podglądzie okna umieszczamy przycisk i tworzymy jego domyślną metodę zdarzeniową, w której umieszczamy wywołanie funkcji z listingu 4.13.

Listing 4.13. *Na potrzeby naszego przykładu tworzymy plik test.txt, który wykorzystujemy do testowania funkcji opartych na interfejsie IFileOperation*

```
CString sciezkaPliku = L"d:\\test.txt";
HANDLE hFile = CreateFileW(sciezkaPliku, GENERIC_READ | GENERIC_WRITE, 0, NULL,
CREATE_NEW, FILE_ATTRIBUTE_NORMAL, NULL);
CloseHandle(hFile);
KopiowaniePliku(sciezkaPliku, L"d:\\test kopia.txt");
PrzenoszeniePliku(L"d:\\test kopia.txt", L"d:\\kopia xml.xml");
UsuwaniePliku(L"d:\\kopia xml.xml");
```

Czytelnik powinien zauważyć, że w przypadku kopiowania większych plików pojawia się teraz okno dialogowe, charakterystyczne dla systemu Windows Vista, informujące o postępie kopiowania (rysunek 4.3). Jeśli korzystamy ze struktury `SHFILEOPSTRUCT`, to nie mamy dostępu do własności specyficznych dla Visty.



Rysunek 4.3. *Kopiowanie elementu za pomocą IFileOperation*

Przy usuwaniu pliku (funkcja `UsuwaniePliku`) pojawi się okno dialogowe z prośbą o potwierdzenie operacji. Jeżeli nie jest ono pożądane, należy w ciele funkcji `Usuwanie↪Pliku`, w ostatnim argumencie funkcji `OperacjaNaPlikuIF`, zamiast zera użyć stałej `FOF_NOCONFIRMATION` (listing 4.14). Poza bardzo szczególnymi sytuacjami nie jest to jednak rozwiązanie godne polecenia.

Listing 4.14. *Usuwanie pliku bez konieczności potwierdzenia*

```
HRESULT COperacjeNaPlikachIFDlg::UsuwaniePlikuBezPotwierdzenia(LPCTSTR szZrodlo)
{
   TCHAR lpBuffer[MAX_PATH] = {0};
   GetFullPathName(szZrodlo, MAX_PATH, lpBuffer, NULL);

   return OperacjaNaPlikuIF(lpBuffer, NULL, FO_DELETE, FOF_NOCONFIRMATION);
}
```

Jak usunąć plik, umieszczając go w koszu?

Listing 4.15 zawiera funkcję, która różni się od funkcji `UsuwaniePliku` z poprzedniego projektu jednym szczegółem. Użyta została opcja `FOF_ALLOWUNDO`, która nakazuje przeniesienie pliku do kosza zamiast usunięcia.

Listing 4.15. *Usuwanie pliku z wykorzystaniem mechanizmu powłoki kosza systemowego*

```
HRESULT COperacjeNaPlikachIFDlg::UsuwaniePlikuDoKosza(LPCTSTR szZrodlo)
{
   TCHAR lpBuffer[MAX_PATH] = {0};
   GetFullPathName(szZrodlo, MAX_PATH, lpBuffer, NULL);

   return OperacjaNaPlikuIF(lpBuffer, NULL, FO_DELETE, FOF_ALLOWUNDO);
}
```

Teraz przed skasowaniem pliku wyświetlone zostanie okno dialogowe z pytaniem o umieszczenie pliku w koszu. Tej samej stałej można użyć w metodzie `OperacjeNaPliku` w rozwiązaniu nie korzystającym z interfejsu `IFileOperation`.

Operacje na całym katalogu

Kopiowanie całego katalogu z podkatalogami jest również możliwe i równie łatwe jak kasowanie pliku. W przypadku funkcji WinAPI i funkcji C++, które poznaliśmy wcześniej, programowanie tej operacji jest możliwe, ale wszelkie informacje prezentowane w trakcie użytkownikowi wymagałyby sporej dodatkowej pracy. A gdy korzysta się z funkcji powłoki, wystarczy jedno polecenie. W listingu 4.16 zaprezentowane są funkcje dla Windows Vista, tj. korzystające z metody `OperacjeNaPlikuIF`, ale analogiczne polecenia można bez problemu przygotować dla wcześniejszej metody `OperacjeNaPliku`.

Listing 4.16. *Zestaw funkcji służących do kopiowania, przenoszenia i usuwania katalogów*

```
HRESULT COperacjeNaPlikachIFDlg::KopiowanieKatalogu(LPCTSTR szZrodlo, LPCTSTR szCel)
{
   return OperacjaNaPlikuIF(szZrodlo, szCel, FO_COPY, FOF_NOCONFIRMMKDIR);
}

HRESULT COperacjeNaPlikachIFDlg::PrzenoszenieKatalogu(LPCTSTR szZrodlo, LPCTSTR szCel)
{
   HRESULT wynik = OperacjaNaPlikuIF(szZrodlo, szCel, FO_MOVE, FOF_NOCONFIRMMKDIR);
   RemoveDirectory(szZrodlo);
   return wynik;
}

HRESULT COperacjeNaPlikachIFDlg::UsuwanieKatalogu(LPCTSTR szZrodlo)
{
   return UsuwaniePliku(szZrodlo);
}

HRESULT COperacjeNaPlikachIFDlg::UsuwanieKataloguDoKosza(LPCTSTR szZrodlo)
{
   return UsuwaniePlikuDoKosza(szZrodlo);
}
```

Aby utworzyć katalog bez dialogu potwierdzenia, należy użyć opcji `FOF_NOCONFIRMMKDIR`, a nie `FOF_NOCONFIRMATION`. Ważną opcją jest `FOF_FILESONLY`. Powoduje ona, że operacje są wykonywane jedynie na plikach pasujących do użytej maski. Wówczas nie zostaną utworzone podkatalogi.

W przypadku dwóch funkcji usuwających katalog utworzyliśmy tak naprawdę alias do funkcji usuwających pliki. Okazuje się, że działają one równie dobrze dla pojedynczych plików, jak i dla katalogów.

Argumentami wszystkich funkcji powinny być oczywiście ścieżki do katalogów, np.

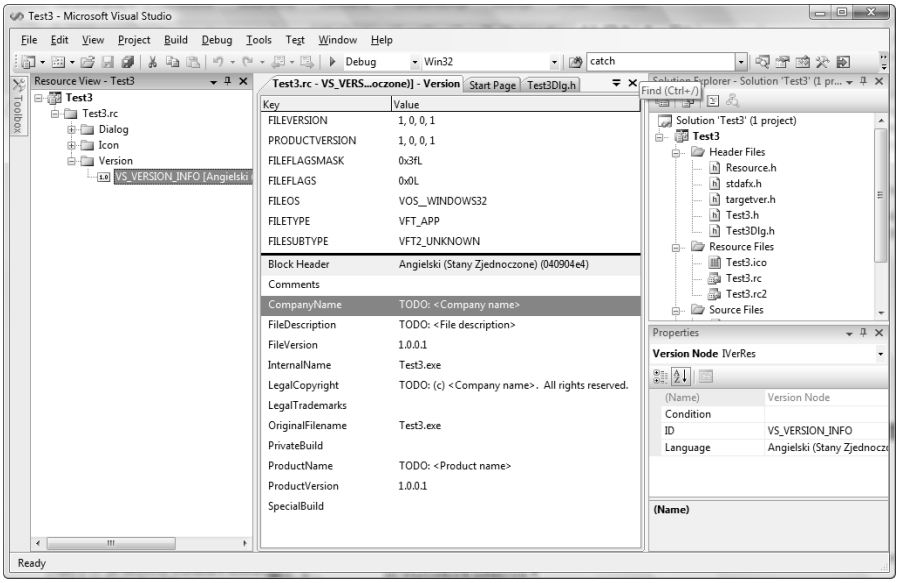
```
KopiowanieKatalogu("d:\\","d:\\Kopia projektu");
UsuwanieKataloguDoKosza("d:\\Kopia projektu");
```

Jak wspomniałem wcześniej, analogiczne metody można przygotować, korzystając z funkcji z projektu opartego na metodzie `OperacjeNaPliku`.

Odczytywanie wersji pliku .exe i .dll

Pliki wykonywalne *.exe* oraz pliki bibliotek *.dll* mogą zawierać informacje o wersji, producencie, prawach autorskich itp.[6] O ile zapisywanie tych informacji w Visual Studio jest proste (pozwalają na to ustawienia projektu), o tyle przy ich odczycie trzeba się trochę pomęczyć.

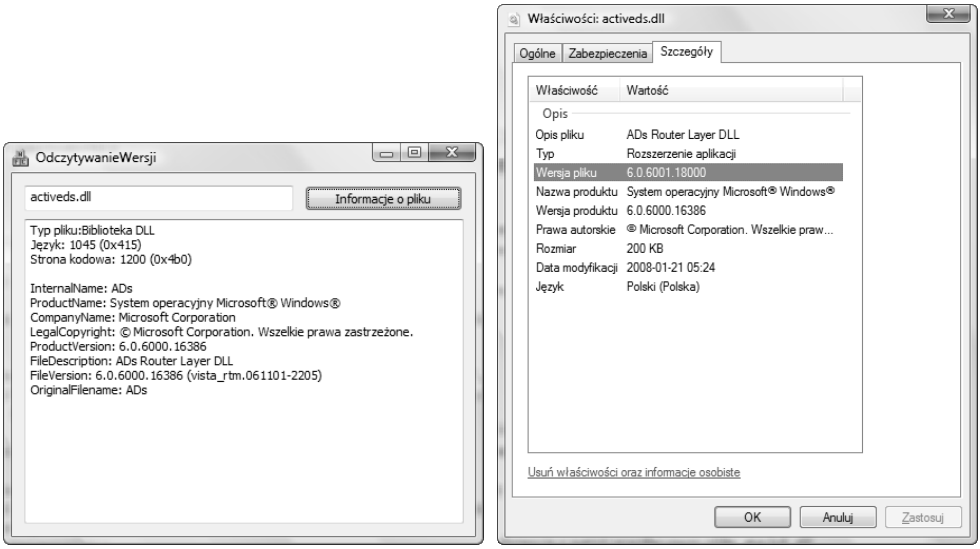
1. Tworzymy nowy projekt aplikacji MFC z oknem dialogowym o nazwie *OdczytywanieWersji*.
2. Klikając dwukrotnie plik *OdczytywanieWersji.rc* w podoknie *Solution Explorer*, otwieramy podokno *Resource View* — zasoby projektu. W gałęzi *Version* znajduje się pozycja *VS_VERSION_INFO [Angielski (Stany Zjednoczone)]*. Klikając ją dwukrotnie, otworzymy edytor (rysunek 4.4), który pozwala na zmianę jej wszystkich ustawień.



Rysunek 4.4. *Edytor wersji umieszczanej w zasobach aplikacji*

[6] Windows pozwala na oglądanie tych informacji w oknie właściwości pliku, na zakładce *Wersja*.

3. Po powrocie do widoku projektowania umieszczamy na podglądzie formy dwa pola edycyjne oraz przycisk (rysunek 4.5). Z polami edycyjnymi należy związać zmienne `Edit1` i `Edit2`.



Rysunek 4.5. *Informacje wyświetlane przez nasz program oraz w oknie właściwości Windows Vista*

4. Własność `MultiLine` drugiego pola edycyjnego ustawiamy na *True*.

5. Definiujemy metodę `PobierzInformacjeOPliku` zgodnie z listingiem 4.17.

Listing 4.17. *Funkcja odczytująca informacje o wersji ze wskazanego pliku .exe lub .dll*

```
CString COdczytywanieWersjiDlg::PobierzInformacjeOPliku(LPCTSTR nazwaPliku)
{
   CString Wynik, temp;
   DWORD uchwyt, rozmiarBufora;
   UINT rozmiarWartosci;
   LPTSTR lpBufor;
   VS_FIXEDFILEINFO *pInformacjeOPliku;
   rozmiarBufora = GetFileVersionInfoSize(nazwaPliku, &uchwyt);

   if (!rozmiarBufora)
   {
      AfxMessageBox(L"Brak informacji o wersji pliku");
      return NULL;
   }
   lpBufor = (LPTSTR)malloc(rozmiarBufora);
   if (!lpBufor)
      return NULL;

   if(!GetFileVersionInfo(nazwaPliku, uchwyt, rozmiarBufora, lpBufor))
   {
      free (lpBufor);
      return NULL;
   }
   if(VerQueryValue(lpBufor, L"\\", (LPVOID *)&pInformacjeOPliku,
   ↪(PUINT)&rozmiarWartosci))
```

```
{
   CString typPliku;

   switch (pInformacjeOPliku->dwFileType)
   {
      case VFT_UNKNOWN: typPliku="Nieznany"; break;
      case VFT_APP: typPliku="Aplikacja"; break;
      case VFT_DLL: typPliku="Biblioteka DLL"; break;
      case VFT_STATIC_LIB: typPliku="Biblioteka ładowana statycznie"; break;
      case VFT_DRV:
         switch (pInformacjeOPliku->dwFileSubtype)
         {
            case VFT2_UNKNOWN: typPliku="Nieznany rodzaj sterownika"; break;
            case VFT2_DRV_COMM: typPliku="Sterownik komunikacyjny"; break;
            case VFT2_DRV_PRINTER: typPliku="Sterownik drukarki"; break;
            case VFT2_DRV_KEYBOARD: typPliku="Sterownik klawiatury"; break;
            case VFT2_DRV_LANGUAGE: typPliku="Sterownik języka"; break;
            case VFT2_DRV_DISPLAY: typPliku="Sterownik karty graficznej"; break;
            case VFT2_DRV_MOUSE: typPliku="Sterownik myszy"; break;
            case VFT2_DRV_NETWORK: typPliku="Sterownik karty sieciowej"; break;
            case VFT2_DRV_SYSTEM: typPliku="Sterownik systemowy"; break;
           case VFT2_DRV_INSTALLABLE: typPliku="Sterownik do instalacji"; break;
            case VFT2_DRV_SOUND: typPliku="Sterownik karty dźwiękowej"; break;
         }; break;
      case VFT_FONT:
         switch (pInformacjeOPliku->dwFileSubtype)
         {
            case VFT2_UNKNOWN: typPliku="Unknown Font"; break;
            case VFT2_FONT_RASTER: typPliku="Raster Font"; break;
            case VFT2_FONT_VECTOR: typPliku="Vector Font"; break;
            case VFT2_FONT_TRUETYPE: typPliku="Truetype Font"; break;
         } break;
      case VFT_VXD:
            typPliku.Format(L"Virtual Defice Identifier = %04x",
            ↪pInformacjeOPliku->dwFileSubtype); break;
      }
   Wynik.Append(L"Typ pliku:");
   Wynik.Append(typPliku);
   Wynik.Append(L"\r\n");
}
unsigned short jezyk, stronaKodowa;
if (VerQueryValue(lpBufor, L"\\VarFileInfo\\Translation", (LPVOID
↪*)&pInformacjeOPliku, (PUINT)&rozmiarWartosci))
{
   unsigned short* wartosc=(unsigned short*)pInformacjeOPliku;
   jezyk = *wartosc;
   stronaKodowa = *(wartosc+1);
}
else
{
   jezyk = 0;
   stronaKodowa = 0;
}
temp.Format(L"Język: %d (0x%x)\r\n", jezyk, jezyk);
Wynik.Append(temp);
temp.Format(L"Strona kodowa: %d (0x%x)\r\n\r\n", stronaKodowa, stronaKodowa);
Wynik.Append(temp);
```

```
   const CString InfoStr[12]= {L"Comments", L"InternalName", L"ProductName",
   ↪L"CompanyName",
      L"LegalCopyright", L"ProductVersion", L"FileDescription", L"LegalTrademarks",
      ↪L"PrivateBuild",
      L"FileVersion", L"OriginalFilename", L"SpecialBuild"};

   for(int i=0;i<12;i++)
   {
      CString kategoria;
      kategoria.Format(L"\\StringFileInfo\\%04x%04x\\%s", jezyk, stronaKodowa,
      ↪InfoStr[i]);
      if (VerQueryValue(lpBufor, kategoria, (LPVOID *)&pInformacjeOPliku,
      ↪&rozmiarWartosci) != 0)
      {
         char* wartosc=(char*)pInformacjeOPliku;
         temp.Format(L"%s: %s\r\n", InfoStr[i], wartosc);
         Wynik.Append(temp);
      }
   }
   free(lpBufor);
   return Wynik;
}
```

6. Tworzymy domyślną metodę zdarzeniową dla przycisku i umieszczamy w niej kod z listingu 4.18.

Listing 4.18. *Wybór pliku i odczytywanie zapisanych w nim informacji. Kod znajdujący się w komentarzu pozwala przetestować działanie funkcji PobierzInformacjeOPliku na pliku bieżącej aplikacji*

```
void COdczytywanieWersjiDlg::OnBnClickedButton1()
{
   //WCHAR szPath[MAX_PATH] = {0};
   //GetModuleFileNameW(NULL, szPath, MAX_PATH);
   //CString info;
   //info = PobierzInformacjeOPliku(szPath);
   //AfxMessageBox(info);

   CFileDialog fd(TRUE);
   if(fd.DoModal())
   {
      Edit1.SetWindowTextW(fd.GetFileName());
      Edit2.SetWindowTextW(PobierzInformacjeOPliku(fd.GetFileName()));
   }
   else
      AfxMessageBox(L"Nie wybrano pliku!");
}
```

Jak działa funkcja `PobierzInformacjeOPliku` (listing 4.17)? Zaczynamy od pobrania do bufora znaków zestawu informacji. Służą do tego funkcje `GetFileVersionInfoSize` oraz `GetFileVersionInfo`[7]. Pierwsza zwraca wielkość bufora, a druga sam bufor. Informacje z bufora można „rozkodować" za pomocą funkcji `VerQueryValue`. W zależności od podanego argumentu zwraca ona informacje o poszczególnych elementach opisu umieszczonego w pliku *.exe* lub pliku *.dll*. Możliwe wartości zostały

[7] Obie dostępne we wszystkich 32-bitowych wersjach Windows.

wymienione w tabeli 4.1. Poza pierwszą pozycją w tabeli, która zwraca informacje niezależne od języka, i drugą, która zwraca informacje o samym języku i zestawie znaków, pozostałe identyfikatory budowane są zgodnie z szablonem, w którym do łańcucha *\StringFileInfo* dodajemy „zlepione" kody szesnastkowe strony kodowej i zestawu znaków, a do nich łańcuch identyfikujący konkretną informację. Informację tę dołączamy do zwracanego przez `PobierzInformacjeOPliku` łańcucha `CString`. W przypadku ogólnych zastosowań byłoby zapewne wygodniej, aby nasza funkcja zamiast listy łańcuchów zwracała zdefiniowaną przez nas strukturę. Wówczas zamiast pętli po elementach tablicy `InfoStr` należałoby umieścić w niej serię przypisań wypełniających pola struktury. Wtedy jednak kod funkcji wydłużyłby się jeszcze bardziej i stałby się jeszcze mniej przejrzysty.

Tabela 4.1. *Zakładamy, że język systemowy to polski (strona kodowa 1045, czyli w zapisie szesnastkowym 0415), a zestaw znaków to 1250 (szesnastkowo 04E5)*

Drugi argument funkcji VerQueryValue	Zwracana informacja
\	Struktura `VS_FIXEDFILEINFO`
\VarFileInfo\Translation	Język i strona kodowa
\StringFileInfo\041504E5\Comments	Komentarz
\StringFileInfo\041504E5\InternalName	Nazwa wewnętrzna pliku
\StringFileInfo\041504E5\ProductName	Nazwa programu
\StringFileInfo\041504E5\CompanyName	Nazwa firmy
\StringFileInfo\041504E5\LegalCopyright	Informacje o prawie własności
\StringFileInfo\041504E5\ProductVersion	Wersja produktu
\StringFileInfo\041504E5\FileDescription	Opis pliku
\StringFileInfo\041504E5\LegalTrademarks	Znak towarowy
\StringFileInfo\041504E5\PrivateBuild	Plik utworzony do użytku prywatnego
\StringFileInfo\041504E5\FileVersion	Wersja pliku
\StringFileInfo\041504E5\OriginalFilename	Oryginalna nazwa pliku
\StringFileInfo\041504E5\SpecialBuild	Wersja o specjalnym przeznaczeniu

Ostatnia uwaga dotyczy wykorzystania znaków specjalnych `\r\n` zamiast samego `\n`. Taka sekwencja jest wymagana przez funkcję `SetWindowTextW`. W przeciwnym wypadku kontrolka `CEdit` wyświetli cały łańcuch w jednym wierszu.

Jak dodać nazwę dokumentu do listy ostatnio otwartych dokumentów w menu Start?

Windows umożliwia umieszczanie dokumentów otwieranych w edytorach w menu *Start*, w podmenu Moje bieżące dokumenty (taka nazwa obowiązuje przynajmniej w Windows XP /Dokumenty dla Visty[8]/). Nic prostszego. Wystarczy użyć funkcji powłoki `SHAddToRecentDocs`[9,10]; jej drugim argumentem jest ścieżka do pliku, który chcemy tam umieścić:

[8] Nazwa zależy od konfiguracji stylu menu Start.

[9] Dostępne od Windows 95 i NT 4.

```
SHAddToRecentDocs(SHARD_PATH, nazwa pliku dokumentu);
```

Funkcja ta jest zadeklarowana w nagłówku *shlobj.h*, dlatego należy go włączyć do kodu bieżącego projektu.

Odczytywanie informacji o dysku

Zajmiemy się teraz przygotowaniem komponentu, który będzie udostępniał zestaw szczegółowych informacji o dysku: od jego typu, poprzez nazwę systemu plików, po numer seryjny. W ostatecznej wersji komponent będzie miał postać graficzną — będzie prezentował ilość wolnego miejsca na pasku postępu oraz dodatkowe informacje na etykietach `Static`. Do odczytu parametrów dysku użyjemy oczywiście funkcji WinAPI.

Komponent przygotujemy w dwóch krokach, podobnie jak często robi się to w praktyce. Skupiając się na wykorzystywanych funkcjach WinAPI, napiszemy najpierw klasę pobierającą potrzebne informacje. Potem wykorzystamy ją w komponencie wizualnym.

Odczytywanie danych

Pierwszym krokiem będzie napisanie funkcji `PobierzInformacjeODysku`. Korzystając z funkcji WinAPI, będzie ona odczytywać parametry fizyczne dysku, jego wielkość oraz ilość wolnego miejsca. Jej pierwszym argumentem będzie litera dysku, a drugim wskaźnik do struktury, w której zapisana zostanie informacja o dysku. Ową strukturę, nazwijmy ją `DiskInfoStruct`, zdefiniujemy sami.

1. Tworzymy nowy projekt aplikacji MFC z oknem dialogowym o nazwie *DiskInfo*.
2. Dodajemy do niego plik nagłówkowy *InformacjeODysku.h* i plik źródłowy *InformacjeODysku.cpp*.
3. W pliku nagłówkowym umieszczamy (listing 4.19):
   a) definicję struktury `DaneODysku`, do której będziemy zapisywali informacje o dysku;
   b) definicję stałej określającej maksymalną długość niektórych łańcuchów w tej strukturze;
   c) dyrektywy definiujące makra, które pomogą nam w prawidłowym zaokrąglaniu liczb;
   d) i wreszcie deklarację funkcji `PobierzInformacjeODysku`.

[10] Aby zaobserwować działanie tej funkcji, należy się upewnić, że funkcja przechowywania i wyświetlania listy niedawno otwieranych plików jest aktywna.

Listing 4.19. *Zawartość pliku nagłówkowego InformacjeODysku.h*

```
#pragma once

#define ROUND(x) ((x)>=0?(long)((x)+0.5):(long)((x)-0.5))
#define ROUND1(x) (ROUND(10*x)/10.0) //zaokrąglenie z częścią dziesiętną
#define ROUND2(x) (ROUND(100*x)/100.0) //zaokrąglenie z setnymi

#define DI_MAX_LENGTH 256

struct DaneODysku
{
   char literaDysku;
   BOOL czyDyskDostepny;

   int typDysku;
   TCHAR typDyskuOpis[DI_MAX_LENGTH];

   unsigned __int64 calkowitaPrzestrzen;
   unsigned __int64 wolnaPrzestrzen;
   unsigned __int64 zajetaPrzestrzen;

   double wolnaPrzestrzenUlamek;
   unsigned char wolnaPrzestrzenProcenty;

   TCHAR nazwaDysku[DI_MAX_LENGTH];
   ULONG numerSeryjnyDysku;
   TCHAR nazwaFAT[DI_MAX_LENGTH];
   ULONG maksymalnaDlugoscPlikuLubKatalogu;

   ULONG maksymalnaDlugoscSciezki;
};

BOOL PobierzInformacjeODysku(char literaDysku, DaneODysku& diskInfo);
```

4. Natomiast do pliku źródłowego *InformacjeODysku.cpp* dodajemy kod funkcji `PobierzInformacjeODysku` według listingu 4.20.

Listing 4.20. *Pełny kod pliku nagłówkowego*

```
#include "stdafx.h"

#include "InformacjeODysku.h"

BOOL PobierzInformacjeODysku(char literaDysku, DaneODysku &diskInfo)
{
   diskInfo.literaDysku = toupper(literaDysku);

   //Ustalanie wstępnych wartości
   diskInfo.czyDyskDostepny = TRUE;

   diskInfo.typDysku = 0;
   wcscpy_s(diskInfo.typDyskuOpis,L"");
   diskInfo.calkowitaPrzestrzen = 0;
   diskInfo.wolnaPrzestrzen = 0;
   diskInfo.zajetaPrzestrzen = 0;
   diskInfo.wolnaPrzestrzenUlamek = 0;
```

```
   diskInfo.wolnaPrzestrzenProcenty = 0;
   wcscpy_s(diskInfo.nazwaDysku,L"");
   diskInfo.numerSeryjnyDysku = 0;
   wcscpy_s(diskInfo.nazwaFAT,L"");
   diskInfo.maksymalnaDlugoscPlikuLubKatalogu = 0;
   diskInfo.maksymalnaDlugoscSciezki = 0;

   //Ścieżka katalogu głównego na dysku
   TCHAR katalogGlownyDysku[4];
   katalogGlownyDysku[0]=diskInfo.literaDysku;
   katalogGlownyDysku[1]='\0';
   wcscat_s(katalogGlownyDysku,L":\\");

   //Typ napędu (drive type)
   diskInfo.typDysku = GetDriveType(katalogGlownyDysku);
   switch(diskInfo.typDysku)
   {
      case 0:
         wcscpy_s(diskInfo.typDyskuOpis,L"Napęd nie istnieje");
         diskInfo.czyDyskDostepny = FALSE;
         break;
      case 1:
         wcscpy_s(diskInfo.typDyskuOpis,L"Dysk nie jest sformatowany");
         diskInfo.czyDyskDostepny = FALSE;
         break;
      case DRIVE_REMOVABLE: wcscpy_s(diskInfo.typDyskuOpis,L"Dysk wymienny");
break;
      case DRIVE_FIXED: wcscpy_s(diskInfo.typDyskuOpis,L"Dysk lokalny"); break;
      case DRIVE_REMOTE: wcscpy_s(diskInfo.typDyskuOpis,L"Dysk sieciowy"); break;
      case DRIVE_CDROM: wcscpy_s(diskInfo.typDyskuOpis,L"Płyta CDROM"); break;
      case DRIVE_RAMDISK: wcscpy_s(diskInfo.typDyskuOpis,L"RAM Drive"); break;
      default: wcscpy_s(diskInfo.typDyskuOpis,L"Typ dysku nierozpoznany"); break;
   }

   //Jeżeli dysk niedostępny, to kończymy
   if (!diskInfo.czyDyskDostepny) return FALSE;

   //Ilość wolnego miejsca na dysku (disk free space)
   //Typy argumentów niezgodne z Win32 SDK
   BOOL Wynik = ::GetDiskFreeSpaceEx(
       katalogGlownyDysku,
       NULL,
       (ULARGE_INTEGER*)&(diskInfo.calkowitaPrzestrzen),
       (ULARGE_INTEGER*)&(diskInfo.wolnaPrzestrzen));

   diskInfo.zajetaPrzestrzen = diskInfo.calkowitaPrzestrzen -
diskInfo.wolnaPrzestrzen;

   if (Wynik && (diskInfo.calkowitaPrzestrzen != 0))
   {
      diskInfo.wolnaPrzestrzenUlamek = diskInfo.wolnaPrzestrzen/
      (double)diskInfo.calkowitaPrzestrzen;
      diskInfo.wolnaPrzestrzenProcenty = (unsigned char)ROUND(100 *
diskInfo.wolnaPrzestrzenUlamek);
   }
   else
   {
```

```
        diskInfo.wolnaPrzestrzenUlamek = 0;
        diskInfo.wolnaPrzestrzenProcenty = 0;
        diskInfo.czyDyskDostepny = FALSE;
        return FALSE;
    }

    //Nazwa dysku, typ FAT, numer seryjny (GetVolumeInformation)
    unsigned long wlasnosciSystemuPlikow;

    Wynik = GetVolumeInformation(katalogGlownyDysku,
                diskInfo.nazwaDysku,
                DI_MAX_LENGTH,
                &(diskInfo.numerSeryjnyDysku),
                &(diskInfo.maksymalnaDlugoscPlikuLubKatalogu),
                &(wlasnosciSystemuPlikow),
                diskInfo.nazwaFAT,
                DI_MAX_LENGTH);

    diskInfo.maksymalnaDlugoscSciezki = MAX_PATH;

    return Wynik;
}
```

5. Korzystanie z funkcji tego typu jest dość naturalne dla osób posługujących się na co dzień WinAPI. Osoby programujące w C++ zapewne chętniej widziałyby klasę, która w konstruktorze pobierać będzie literę dysku i po utworzeniu udostępniać będzie informacje o dysku. Bez problemu możemy przekształcić powyższą strukturę w taką klasę, choć kosztem tego, że wszystkie jej pola są publiczne. Wystarczy do pliku nagłówkowego dodać definicję klasy `InformacjeODysku` widoczną na listingu 4.21. Może bardziej elegancko byłoby uczynić `DaneODysku` polem nowej klasy, a nie jej klasą bazową, ale na dłuższą metę rozwiązanie takie jest mniej wygodne.

Listing 4.21. *Dodajemy konstruktor inicjujący klasę udostępniającą informacje o dysku*

```
class InformacjeODysku : public DaneODysku
{
public:
    InformacjeODysku(char literaDysku='C')
    {
        this->literaDysku=literaDysku;
        PobierzInformacjeODysku(this->literaDysku,*this);
    }
};
```

Omówię po kolei użyte w funkcji `PobierzInformacjeODysku` funkcje WinAPI:

`GetDriveType`[11] — zwracana przez nią wartość to liczba naturalna określająca typ napędu. Zdefiniowane stałe kodujące poszczególne typy napędów są wymienione i opisane w instrukcji `switch`, następującej po wywołaniu tej funkcji. Nierozpoznawane

[11] Dostępna w Windows 95/98/Me oraz NT/2000/XP.

pozostawiamy typy stacji dyskietek (3.5" lub 5.25")[12]. Jedynym argumentem funkcji jest wskaźnik do łańcucha zawierającego ścieżkę do katalogu głównego dysku w badanym napędzie.

`GetDiskFreeSpaceEx` — funkcja dostępna od wersji systemu Windows 95 OSR2[13], a więc od wersji, która pozwalała na obsługę dysków większych niż 2 GB[14] . Wcześniejsza wersja funkcji `GetDiskFreeSpace` zwraca niepoprawne wartości dla tak dużych dysków. Pierwszym argumentem, podobnie jak w poprzedniej funkcji i w większości funkcji związanych z dyskami, jest wskaźnik do łańcucha zawierającego ścieżkę katalogu głównego dysku. W kolejnych trzech podawane są wskaźniki do typu `ULARGE_INTEGER`, w których zapisana zostanie ilość wolnego miejsca dostępnego dla aplikacji wywołującej funkcję, całkowita ilość bajtów dostępna na dysku i całkowita ilość wolnych bajtów. Typ `ULARGE_INTEGER` jest zdefiniowany w WinAPI jako unia, która w zależności od używanego kompilatora może być parą dwóch liczb 32-bitowych lub jedną liczbą 64-bitową. Visual Studio zawiera typ `unsigned __int64`, który wykorzystujemy do przechowania zwracanych przez `GetDiskFreeSpaceEx` wartości, a więc dotyczy nas druga opcja unii. Konieczne jest jednak rzutowanie wskaźników z `unsigned __int64` na `ULARGE_INTEGER` przy wywoływaniu funkcji.

`GetVolumeInformation`[15] to funkcja zwracająca informacje o systemie plików na dysku, tj. nazwę dysku, numer seryjny, maksymalną długość nazwy katalogu lub pliku, typ FAT, informacje o kompresji itp. Dane pobieramy bezpośrednio do elementów struktury `DaneODysku`.

Stała `MAX_PATH` przechowuje maksymalną długość ścieżki do pliku łącznie z jego nazwą z rozszerzeniem. Zapisujemy tę wartość w polu `maksymalnaDlugoscSciezki` struktury. Jest ona własnością systemu, a nie dysku, jest więc identyczna dla każdego dysku.

Testy

Najprostszy sposób przetestowania powyższej funkcji `PobierzInformacjeODysku` (ewentualnie klasy `InformacjeODysku`) polega na umieszczeniu w oknie przycisku, z którego wywołujemy funkcję widoczną na listingu 4.22.

Listing 4.22. *Funkcja wyświetlająca informacje o dysku*

```
void WyswietlInformacje(char literaDysku)
{
   InformacjeODysku informacjeODysku(literaDysku);

   if(!informacjeODysku.czyDyskDostepny)
```

[12] Można je oczywiście, nieco nieelegancko, rozpoznać, korzystając z objętości dyskietki.

[13] W takiej postaci program nie będzie działał w „czystym" Windows 95. Można temu zaradzić, sprawdzając wersję systemu za pomocą `GetVersionEx` i w tym systemie wywołując starszą wersję funkcji — system i tak nie obsługuje większych dysków.

[14] Łatwo sprawdzić, że 2 giga = 2 • 1024 mega = 2 • 1024 • 1024 kilo = 2 • 1024 • 1024 • 1024 = 2147483648; to więcej niż zakres 32-bitowej liczby całkowitej long (ze znakiem). Konieczne więc jest użycie liczb 64-bitowych typu __int64. I tu należy uważać, żeby przypadkowo nie przeprowadzić konwersji na liczby 32-bitowe przez przypisywanie lub operacje na zmiennych.

[15] Dostępna w Windows 95/98/Me oraz NT/2000/XP.

```
    {
       MessageBox(NULL,L"Dysk nie jest dostępny",L"Ostrzeżenie",MB_ICONWARNING);
       return;
    }
    CString komunikat;
    komunikat.Format(L"Informacje o dysku\nNazwa: %s\nTyp dysku: %s\nTyp FAT:
    ➥%s\nWielkość dysku: %.2f GB\nIlość wolnego miejsca: %.0f %%",
       informacjeODysku.nazwaDysku,
       informacjeODysku.typDyskuOpis,
       informacjeODysku.nazwaFAT,
       informacjeODysku.calkowitaPrzestrzen/1024.0/1024.0/1024.0,
       100*informacjeODysku.wolnaPrzestrzenUlamek
       );
    MessageBox(NULL,komunikat,L"Informacja o dysku",MB_OK);
}
```

Wyświetla ona okno z komunikatem zawierającym podstawowe informacje o dysku lub informacje, że dysk jest niedostępny, jeżeli podamy złą literę dysku[16]. Możemy jednak pójść o krok dalej — przygotujmy projekt okna z listą `CListBox` (rysunek 4.6) i wyświetlmy na niej informacje o wszystkich dostępnych w komputerze dyskach logicznych, zarówno lokalnych, jak i sieciowych.

1. Przechodzimy do widoku projektowania okna.
2. Na formie umieszczamy kontrolkę `CListBox`, z którą wiążemy zmienną `ListBox1`.
3. W metodzie `OnInitDialog` umieszczamy polecenia z listingu 4.23.

Listing 4.23. *Sprawdzamy po prostu kolejno wszystkie litery od C: do Z:*

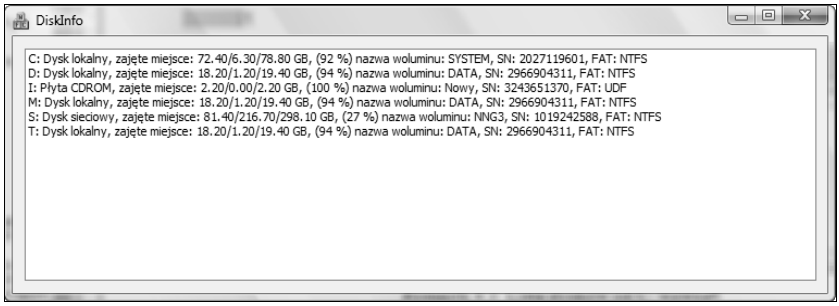
```
BOOL CDiskInfoDlg::OnInitDialog()
{
   CDialog::OnInitDialog();

   // Set the icon for this dialog.  The framework does this automatically
   //  when the application's main window is not a dialog
   SetIcon(m_hIcon, TRUE);         // Set big icon
   SetIcon(m_hIcon, FALSE);      // Set small icon

   const int BwGB=1024*1024*1024; //ilość bajtów w GB

   for(char litera = 'c'; litera <= 'z'; litera++)
   {
      DaneODysku informacjeODysku;
      PobierzInformacjeODysku(litera, informacjeODysku);
      if (informacjeODysku.czyDyskDostepny)
      {
         double calkowitaPrzestrzenGB =
         ➥ROUND1((double)informacjeODysku.calkowitaPrzestrzen/BwGB);
         double wolnaPrzestrzenGB =
         ➥ROUND1((double)informacjeODysku.wolnaPrzestrzen/BwGB);
```

[16] Jeżeli zamiast klasy chcemy użyć funkcji, należy usunąć pierwszą linię kodu, a zamiast niej wstawić:

```
DaneODysku informacjeODysku;
PobierzInformacjeODysku(literaDysku,informacjeODysku);
```

```
            double zajetaPrzestrzenGB =
             ↪ROUND1((double)informacjeODysku.zajetaPrzestrzen/BwGB);

            CString temp;

            temp.AppendFormat(L"%c: %s, zajęte miejsce: %.2lf/%.2lf/%.2lf GB, (%d %%)
             ↪nazwa woluminu: %s, SN: %I64d, FAT: %s",
               informacjeODysku.literaDysku, informacjeODysku.typDyskuOpis,
               zajetaPrzestrzenGB, wolnaPrzestrzenGB,
               calkowitaPrzestrzenGB, 100 - informacjeODysku.wolnaPrzestrzenProcenty,
               informacjeODysku.nazwaDysku, (unsigned __int64)informacjeODysku.
               ↪numerSeryjnyDysku,
               informacjeODysku.nazwaFAT);

            listBox1.AddString(temp);
        }
    }

    return TRUE;  // return TRUE  unless you set the focus to a control
}
```

Rysunek 4.6. *Lista dysków od C: wzwyż*



Jeżeli nie chcemy sprawdzać wszystkich dysków w pętli z literą dysku jako indeksem, możemy ograniczyć się do rzeczywiście obecnych dysków, korzystając z funkcji WinAPI `GetLogicalDriveStrings`[17] (funkcja ta pojawi się w dalszej części rozdziału).

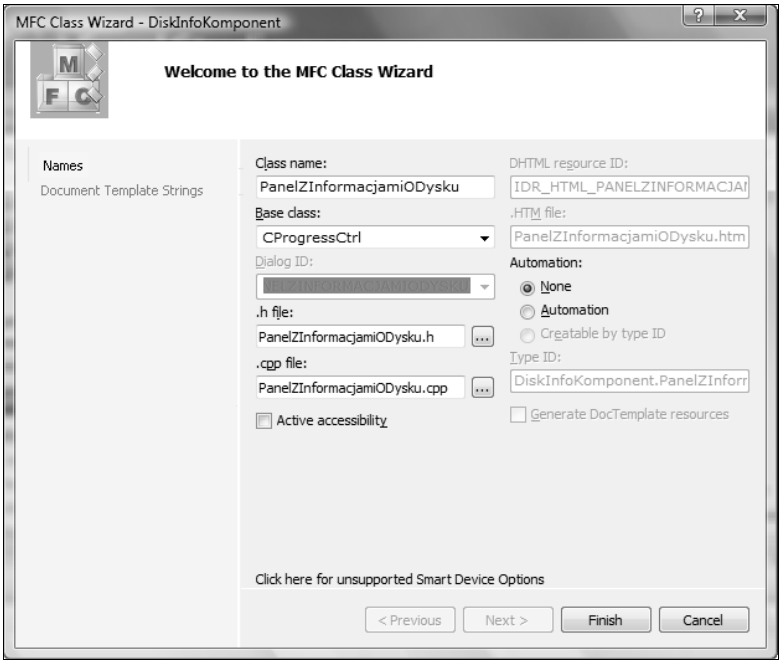
Kontrolka MFC

Przygotujemy teraz kontrolkę `CDiskInfoPanel`, która nie tylko będzie udostępniać informacje o dysku, ale również prezentować je w postaci paska postępu z towarzyszącymi mu napisami. Postępowanie to polegać będzie na zmodyfikowaniu kontrolki `CProgressCtrl` w taki sposób, aby oprócz paska postępu widoczne były jeszcze dwa opisy informujące o symbolu dysku, jego nazwie, nazwie systemu plików oraz ilości wolnego miejsca w gigabajtach.

1. Tworzymy nowy projekt aplikacji MFC z oknem dialogowym o nazwie *DiskInfoKomponent*.

[17] Dostępna w Windows 95/98/Me oraz NT/2000/XP.

2. Natychmiast po utworzeniu plików projektu wybieramy z menu *Project* polecenie *Add Class…*. Pojawi się okno dialogowe, w którym zaznaczamy pozycję *MFC Class* i klikamy *Add*. Pojawi się okno kreatora, które wypełniamy zgodnie ze wzorem z rysunku 4.7 i klikamy przycisk *Finish*.



Rysunek 4.7. *MFC Class Wizard dla klasy CDiskInfoPanel*

3. Do katalogu projektu kopiujemy pliki *InformacjeODysku.h* i *InformacjeODysku.cpp*, które przygotowaliśmy w poprzednim podrozdziale, i dodajemy je do projektu.

4. Przechodzimy do edycji pliku nagłówkowego *PanelZInformacjamiODysku.h* i modyfikujemy go zgodnie z listingiem 4.24.

Listing 4.24. *Pełny kod pliku nagłówkowego klasy PanelZInformacjamiODysku z wyróżnieniem zmian, jakich dokonaliśmy względem klasy wygenerowanej przez kreator*

```
#pragma once

#include "InformacjeODysku.h"

// PanelZInformacjamiODysku

class PanelZInformacjamiODysku : public CProgressCtrl
{
   DECLARE_DYNAMIC(PanelZInformacjamiODysku)

private:
   bool obiektIstnieje;
   char literaDysku; //litera dysku
   DaneODysku daneODysku; //struktura przechowująca informacje o dysku
   CString opisLewy,opisPrawy; //pola wykorzystywane do umieszczenia
   int pozycjaPaska; //informacji o dysku na pasku postępu
```

```
public:
   PanelZInformacjamiODysku(char LiteraDysku='c');
   virtual ~PanelZInformacjamiODysku();

   //publiczne metody i własności
   DaneODysku GetDaneODysku();
   char GetLiteraDysku();
   void SetLiteraDysku(char LiteraDysku);
   __declspec (property (get = GetDaneODysku)) DaneODysku Informacje;
   __declspec (property (get = GetLiteraDysku, put = SetLiteraDysku)) char LiteraDysku;

protected:
   DECLARE_MESSAGE_MAP()
};
```

5. Zagadkowe może wydawać się pole `objektIstnieje`. Jest ono związane z faktem, iż nasz obiekt jest kontrolką, a zatem niektóre jego własności nie są dostępne przed utworzeniem go. Na przykład nie można zmienić pozycji paska postępu (metoda `SetPos`) w konstruktorze. Spowodowałoby to wyjątek. Pole to zainicjujemy wartością *false*, a przełączymy na *true* po załadowaniu kontrolki.
6. W pliku *PanelZInformacjamiODysku.cpp* definiujemy zadeklarowaną w nagłówku metodę `SetLiteraDysku` (niemal identyczną jak metoda z listingu 4.25) oraz dwie proste metody-akcesory.

Listing 4.25. *Pełny kod pliku źródłowego z zaznaczonymi zmianami*

```
// PanelZInformacjamiODysku.cpp : implementation file
//

#include "stdafx.h"
#include "DiskInfoKomponent.h"
#include "PanelZInformacjamiODysku.h"

// PanelZInformacjamiODysku

IMPLEMENT_DYNAMIC(PanelZInformacjamiODysku, CProgressCtrl)

PanelZInformacjamiODysku::PanelZInformacjamiODysku(char LiteraDysku)
   :objektIstnieje(false)
{
   SetLiteraDysku(LiteraDysku);
}

PanelZInformacjamiODysku::~PanelZInformacjamiODysku()
{
}


BEGIN_MESSAGE_MAP(PanelZInformacjamiODysku, CProgressCtrl)
END_MESSAGE_MAP()

void PanelZInformacjamiODysku::SetLiteraDysku(char LiteraDysku)
{
   literaDysku = LiteraDysku;
   PobierzInformacjeODysku(literaDysku, daneODysku);
```

```
    const int BwGB=1024*1024*1024;

    double calkowitaPrzestrzenGB = ROUND1(daneODysku.calkowitaPrzestrzen/BwGB);
    double wolnaPrzestrzenGB = ROUND1(daneODysku.wolnaPrzestrzen/BwGB);
    double zajetaPrzestrzenGB = ROUND1(daneODysku.zajetaPrzestrzen/BwGB);

    opisLewy.Format(L" %c: %s (%s)", daneODysku.literaDysku, daneODysku.nazwaDysku,
    ↪daneODysku.nazwaFAT);
    opisPrawy.Format(L"%.2f/%.2f/%.2f (%d%%) ", zajetaPrzestrzenGB,
    ↪wolnaPrzestrzenGB, calkowitaPrzestrzenGB, 100 -
    ↪daneODysku.wolnaPrzestrzenProcenty);
    pozycjaPaska = 100 - daneODysku.wolnaPrzestrzenProcenty;

    if (!daneODysku.czyDyskDostepny)
    {
       opisLewy.Format(L"%c: Dysk niedostępny!",daneODysku.literaDysku);
       opisPrawy.Format(L"");
       pozycjaPaska = 100;
    }

    if(obiektIstnieje) this->SetPos(pozycjaPaska);
}

DaneODysku PanelZInformacjamiODysku::GetDaneODysku()
{
    return daneODysku;
}

char PanelZInformacjamiODysku::GetLiteraDysku()
{
    return literaDysku;
}

// PanelZInformacjamiODysku message handlers
```

7. Pole `objektIstnieje` wykorzystaliśmy na końcu metody `SetLiteraDysku`. Jeżeli obiekt już powstał, np. gdy jawnie wywołał tę metodę użytkownik, aktualizowana jest pozycja paska postępu. W konstruktorze pozycja nie zostanie ustawiona. Musimy to zrobić po powstaniu obiektu. Wykorzystamy do tego pierwsze odświeżenie kontrolki, tj. pierwsze wywołanie metody związanej z otrzymaniem komunikatu `WM_PAINT`[18]. Będzie to o tyle wygodne, że w metodzie tej przygotujemy także dodatkowy opis umieszczony na kontrolce.

   a) W pliku *PanelZInformacjamiODysku.cpp* do makr mapujących komunikaty dodajemy wywołanie makra wiążącego komunikat `WM_PAINT` z metodą `OnPaint`:

```
BEGIN_MESSAGE_MAP(PanelZInformacjamiODysku, CProgressCtrl)
   ON_WM_PAINT()
END_MESSAGE_MAP()
```

[18] Użycie komunikatu `WM_CREATE` w systemach wcześniejszych niż Windows Vista może powodować kłopoty, dlatego staram się go unikać.

b) W pliku nagłówkowym *PanelZInformacjamiODysku.h* umieszczamy deklarację chronionej metody `OnPaint`:

```
protected:
   void OnCreate();
```

c) Definiujemy tę metodę w pliku źródłowym zgodnie ze wzorem z listingu 4.26.

Listing 4.26. *Metoda uruchamiana po utworzeniu kontrolki*

```
void PanelZInformacjamiODysku::OnPaint()
{
   //ustawianie pozycji paska przy pierwszym uruchamianiu
   if(!obiektIstnieje)
   {
      obiektIstnieje=true;
      this->SetPos(pozycjaPaska);
   }

   CClientDC dc(this); //device context for painting
   CRect rc, rcUpdate, rcProgressBar;
   CRgn rgn;

   GetUpdateRect(rcUpdate);

   CProgressCtrl::OnPaint();

   rgn.CreateRectRgn(rcUpdate.left, rcUpdate.top, rcUpdate.right, rcUpdate.bottom);
   dc.SelectClipRgn(&rgn);
   GetClientRect(rc);
   dc.SetBkMode(TRANSPARENT);
   dc.SelectClipRgn(NULL);
   dc.DrawText(opisLewy, -1, rc, DT_SINGLELINE | DT_VCENTER | DT_LEFT); // lewy
   ↪opis paska postępu
   dc.DrawText(opisPrawy, -1, rc, DT_SINGLELINE | DT_VCENTER | DT_RIGHT); // prawy
   ↪opis paska postępu

   dc.SelectClipRgn(NULL);
}
```

8. Kontrolka jest gotowa. Przejdźmy teraz do okna dialogowego, aby wykorzystać nowy komponent do prezentacji informacji o dyskach. Zacznijmy od usunięcia obecnych w oknie dwóch przycisków i etykiety.

9. Następnie w pliku *DiskInfoKomponentDlg.cpp* zadeklarujmy dostęp do plików komponentu, dodając dyrektywę:

```
#include "PanelZInformacjamiODysku.h"
```

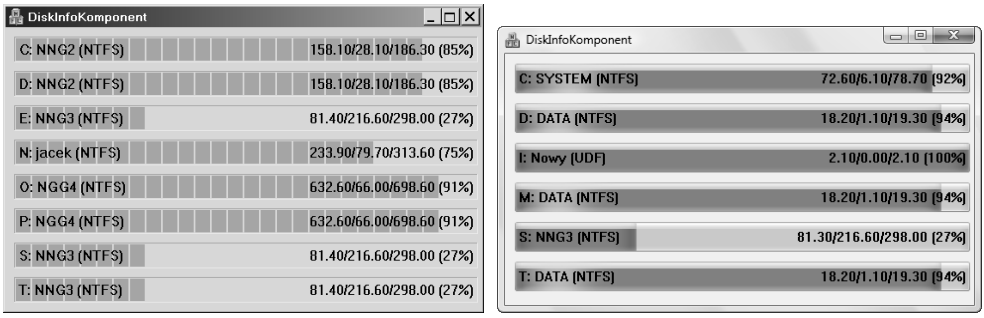
10. Następnie w metodzie `OnInitDialog` utwórzmy i umieśćmy w oknie panele (listing 4.27). Po umieszczeniu paneli dostosowujemy wielkość okna, tak żeby wszystkie dyski były widoczne.

Listing 4.27. *Dynamiczne tworzenie kontrolek CDiskInfoPanel*

```
BOOL CDiskInfoKomponentDlg::OnInitDialog()
{
   CDialog::OnInitDialog();
```

```
    // Set the icon for this dialog.  The framework does this automatically
    //  when the application's main window is not a dialog
    SetIcon(m_hIcon, TRUE);         // Set big icon
    SetIcon(m_hIcon, FALSE);      // Set small icon

    //przygotowanie paneli
    const int przes = 10, wys = 30;
    int iloscPaneli = 0;
    CRect clientRect;
    GetClientRect(clientRect);
    PanelZInformacjamiODysku *panel;

    for(char litera = 'c'; litera <= 'z'; litera++)
    {
       panel = new PanelZInformacjamiODysku(litera);
       if(panel->Informacje.czyDyskDostepny)
       {
          panel->Create(WS_CHILD | WS_VISIBLE, clientRect, this, iloscPaneli);
          panel->MoveWindow(przes, przes + iloscPaneli * (wys + przes),
          ↪clientRect.Width() - 2*przes, wys);
          iloscPaneli++;
       }
       else delete panel;
    }

    // Dopasowanie wysokości okna dialogowego do ilości paneli
    int newHeight = (iloscPaneli+1) * (wys + przes);
    SetWindowPos(NULL, clientRect.left, clientRect.bottom, clientRect.Width(),
     ↪newHeight, SWP_SHOWWINDOW);

    return TRUE;  // return TRUE  unless you set the focus to a control
}
```

11. Kompilujemy projekt i uruchamiamy aplikację. Zobaczymy formę zapełnioną dynamicznie tworzonymi komponentami `CDiskInfoPanel` (rysunek 4.8).



Rysunek 4.8. *Ostateczna postać aplikacji w Windows XP i Windows Vista*

Stworzoną kontrolkę można osadzić na formie w sposób „statyczny”. W tym celu wystarczy przejść do widoku projektowania okna dialogowego projektu, do którego dołączone są pliki *InformacjeODysku.h*/*.cpp* oraz pliki kontrolki *PanelZInformacjamiODysku.h*/*.cpp*, i umieścić w tym oknie kontrolkę `CProgressCtrl`. Następnie należy związać z nią zmienną, której domyślny typ `CProgressCtrl` trzeba zmodyfikować na `PanelZInformacjamiODysku`.

Typ ten można zresztą zmienić już po utworzeniu w deklaracji zmiennej w pliku nagłówkowym okna dialogowego. Kontrolka udostępnia własność `LiteraDysku`, której można przypisać literę dowolnego dysku (ze względu na domyślny argument konstruktora domyślna litera to C).

Ikona w obszarze powiadamiania (zasobniku)

Powracamy na chwilę do funkcji powłoki, a dokładnie do jednej z nich, o nazwie `Shell_NotifyIcon`[19]. Pozwala ona na kontrolę ikon w obszarze powiadamiania (mowa o tym miejscu przy zegarze, które przy domyślnym ustawieniu pulpitu znajduje się z prawej strony paska zadań).

Funkcja Shell_NotifyIcon

Do obsługi ikon umieszczanych w obszarze powiadamiania służy funkcja powłoki `Shell_NotifyIcon`. Przyjmuje ona dwa argumenty, z których pierwszy może mieć wartości `NIM_ADD`, `NIM_MODIFY` lub `NIM_DELETE`, oznaczające odpowiednio: dodanie, zmianę i usunięcie ikony z obszaru powiadamiania. Drugi to wskaźnik do struktury typu `NOTIFYICONDATA` zawierającej informacje o ikonie.

Zacznijmy od umieszczenia ikony w zasobniku. Nie jest to zadanie trudne. Aby się o tym przekonać, możemy do projektu dodać przycisk i w jego metodzie zdarzeniowej umieścić wyróżnione w listingu polecenia. Klikając przycisk, dodamy do zasobnika ikonę aplikacji (tę, która używana jest w pasku zadań) z podpowiedzią o treści Podpowiedź.

1. Tworzymy nowy projekt aplikacji MFC z oknem dialogowym o nazwie *IkonaWZasobniku*.
2. Deklarujemy pole klasy `CIkonaWZasobniku`, w którym przechowywać będziemy informację o ikonie:

```
private:
   NOTIFYICONDATA informacjeOIkonie;
```

3. Następnie umieszczamy na oknie dialogowym przycisk. Klikając go dwukrotnie, tworzymy domyślną metodę zdarzeniową i umieszczamy w niej polecenia widoczne na listingu 4.28.

Listing 4.28. *Najprostsze użycie funkcji Shell_NotifyIcon*

```
void CIkonaWZasobnikuDlg::OnBnClickedButton1()
{
   informacjeOIkonie.cbSize = sizeof(informacjeOIkonie);
   informacjeOIkonie.hWnd = m_hWnd;
   wcscpy_s(informacjeOIkonie.szTip, L"Podpowiedź");
```

[19] Dostępna we wszystkich 32-bitowych wersjach Windows, poza NT 3.x.

```
    informacjeOIkonie.hIcon = LoadIcon(AfxGetInstanceHandle(),
    ↪MAKEINTRESOURCE(IDR_MAINFRAME));
    informacjeOIkonie.uID = 0;
    informacjeOIkonie.uFlags = NIF_ICON | NIF_TIP;

    Shell_NotifyIcon(NIM_ADD, &informacjeOIkonie);
}
```

4. Kładziemy na formie drugi przycisk i umieszczamy w nim polecenie usuwające ikonę z zasobnika (listing 4.29).

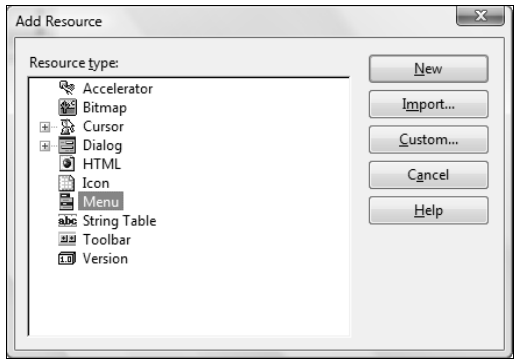
Listing 4.29. *Usuwanie z zasobnika ikony identyfikowanej przez strukturę informacjeOIkonie*

```
void CIkonaWZasobnikuDlg::OnBnClickedButton2()
{
    Shell_NotifyIcon(NIM_DELETE, &informacjeOIkonie);
}
```

Menu kontekstowe ikony

Z ikoną mogą wiązać się pewne zdarzenia. W szczególności są to pojedyncze i podwójne kliknięcie lewym przyciskiem myszy czy kliknięcie prawym przyciskiem myszy. Konsekwencją tych zdarzeń jest przesłanie komunikatów do okna aplikacji[20]. Z kliknięciem prawym przyciskiem myszy zazwyczaj związane jest rozwinięcie menu kontekstowego. Na kliknięcia lewym przyciskiem zareagujemy tylko komunikatami.

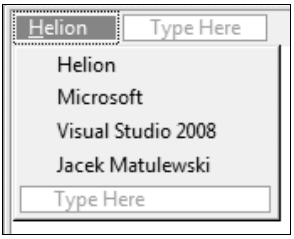
1. Zacznijmy od utworzenia menu kontekstowego (kontynuujemy rozbudowę poprzedniego projektu):
   a) Przechodzimy do edytora zasobów (podokno *Resource View*).
   b) Rozwijamy węzeł przy nazwie *IkonaWZasobniku*.
   c) Klikamy prawym przyciskiem myszy nazwę *IkonaWZasobniku.rc* i z menu podręcznego wybieramy pozycję *Add Resource…*.
   d) Wybieramy menu (rysunek 4.9). Jego identyfikator powinien być równy `IDR_MENU1`.



Rysunek 4.9. *Dodajemy menu do naszego projektu*

[20] Pełny opis obsługi komunikatów Windows zawarto w rozdziale 6.

e) Projektujemy menu według rysunku 4.10.



Rysunek 4.10. *Edytor menu Visual Studio*

2. Modyfikujemy metodę `OnBnClickedButton1` z listingu 4.28 zgodnie z listingiem 4.30.

Listing 4.30. *Włączenie obsługi komunikatów do funkcji tworzącej ikonę w zasobniku*

```
void CIkonaWZasobnikuDlg::OnBnClickedButton1()
{
   informacjeOIkonie.cbSize = sizeof(informacjeOIkonie);
   informacjeOIkonie.hWnd = m_hWnd;
   wcscpy_s(informacjeOIkonie.szTip, L"Podpowiedź");
   informacjeOIkonie.hIcon = LoadIcon(AfxGetInstanceHandle(),
   ↪MAKEINTRESOURCE(IDR_MAINFRAME));
   informacjeOIkonie.uID = 0;
   informacjeOIkonie.uFlags = NIF_TIP | NIF_ICON | NIF_MESSAGE;
   informacjeOIkonie.uVersion = NOTIFYICON_VERSION_4;
   informacjeOIkonie.uCallbackMessage = WM_USER + 1;

   Shell_NotifyIcon(NIM_ADD, &informacjeOIkonie);
}
```

3. Wiążemy metodę zdarzeniową `OnTrayClick` z identyfikatorem `WM_USER+1`, który jest używany przez system operacyjny do wysyłania powiadomień między ikoną w zasobniku a oknem o identyfikatorze `informacjeOIkonie.hWnd`.

```
BEGIN_MESSAGE_MAP(CIkonaWZasobnikuDlg, CDialog)
   ON_WM_PAINT()
   ON_WM_QUERYDRAGICON()
   //}}AFX_MSG_MAP
   ON_BN_CLICKED(IDC_BUTTON1, &CIkonaWZasobnikuDlg::OnBnClickedButton1)
   ON_BN_CLICKED(IDC_BUTTON2, &CIkonaWZasobnikuDlg::OnBnClickedButton2)
   ON_MESSAGE(WM_USER + 1, &CIkonaWZasobnikuDlg::OnTrayClick)
END_MESSAGE_MAP()
```

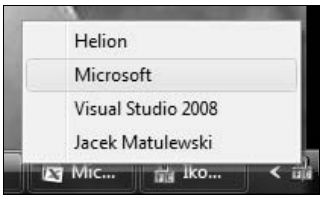
4. Dodajemy metodę `onTrayClick` i definiujemy ją według listingu 4.31.

Listing 4.31. *Obsługa komunikatów wysyłanych przez ikonę w zasobniku*

```
LRESULT CIkonaWZasobnikuDlg::OnTrayClick(WPARAM wParam, LPARAM lParam)
{
   UINT uMsg = (UINT) lParam;
   switch (uMsg)
   {
      case WM_LBUTTONDBLCLK:
         AfxMessageBox(L"Dwukrotne kliknięcie");
         break;
```

```
        case WM_LBUTTONDOWN:
            AfxMessageBox(L"Kliknięcie");
            break;
        case WM_RBUTTONUP:
            CPoint pt;
            CMenu trayMenu;
            GetCursorPos(&pt);
            trayMenu.LoadMenuW(MAKEINTRESOURCE(IDR_MENU1));
            trayMenu.GetSubMenu(0)->TrackPopupMenu(TPM_LEFTALIGN | TPM_RIGHTBUTTON,
            ↪pt.x, pt.y, this);
            break;
    }
    return TRUE;
}
```

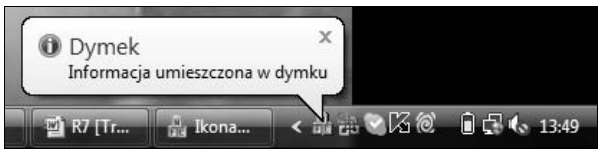
Jak wynika z kodu metody `OnTrayClick` z listingu 4.31, kliknięcia lewym przyciskiem myszy powodują wywołanie metod pokazujących komunikaty, natomiast naciśnięcie prawego klawisza myszy powoduje załadowanie menu z zasobów aplikacji i pokazanie go przy bieżącej pozycji myszy (rysunek 4.11).



Rysunek 4.11. *Przykładowe menu związane z ikoną aplikacji w zasobniku*

„Dymek”

W nowszych wersjach Windows z ikonami w obszarze powiadamiania związany może być „dymek” (ang. *balloon hint*; rysunek 4.12). Łatwo możemy go utworzyć, modyfikując funkcję z listingu 4.28. W tym celu:



Rysunek 4.12. *„Dymek” to forma powiadamiania o zdarzeniach wymagających uwagi Czytelnika. Zaletą „dymku” jest to, że nie przejmuje „focusu” bieżącej aplikacji*

Dodajemy do naszej formy kolejny przycisk. Tworzymy jego domyślną metodę zdarzeniową i definiujemy zgodnie z listingiem 4.32.

Listing 4.32. *Wywołanie „dymka” związanego z ikoną umieszczoną w zasobniku*

```
void CIkonaWZasobnikuDlg::OnBnClickedButton3()
{
    informacjeOIkonie.cbSize = sizeof(informacjeOIkonie);
    informacjeOIkonie.hWnd = m_hWnd;
    wcscpy_s(informacjeOIkonie.szInfoTitle, L"Dymek");
```

```
    wcscpy_s(informacjeOIkonie.szInfo, L"Informacja umieszczona w dymku");
    informacjeOIkonie.hIcon = LoadIcon(AfxGetInstanceHandle(),
    ↪MAKEINTRESOURCE(IDR_MAINFRAME));
    informacjeOIkonie.dwInfoFlags = NIIF_INFO;
    informacjeOIkonie.uID = 0;
    informacjeOIkonie.uFlags = NIF_INFO | NIF_ICON;

    Shell_NotifyIcon(NIM_MODIFY, &informacjeOIkonie);
}
```

Metoda jest tak napisana, że pokazuje dymek przy istniejącej ikonie. Jeżeli chcemy pokazać dymek, tworząc jednocześnie ikonę, wystarczy pierwszy argument funkcji `Shell_NotifyIcon` zmienić na `NIM_ADD`.

Multimedia (CD-Audio, MCI)

Od razu uprzedzam, że podrozdział poświęcony bibliotece MCI (ang. *Media Control Interface*), a więc podzbiorowi funkcji WinAPI służącemu do obsługi urządzeń multimedialnych, jest daleki od kompletności. Zaletą tej biblioteki jest to, że udostępnia funkcje, które pozwalają sterować wszystkimi urządzeniami multimedialnymi. W poniższych projektach skupimy naszą uwagę na obsłudze napędów CD/DVD, odtwarzaniu muzyki z płyt CD-Audio oraz kontroli poziomu głośności.

Aby wysunąć lub wsunąć tackę w napędzie CD lub DVD

Aby otworzyć lub zamknąć domyślny napęd CD-Audio, można użyć poleceń (zadeklarowane są w nagłówku `Mmsystem.h`):

```
mciSendString(L"set cdaudio door open wait", NULL, 0, 0);
mciSendString(L"set cdaudio door closed wait", NULL, 0, 0);
```

Ten prosty sposób nie pomoże nam jednak, gdy zechcemy wysunąć płytę z innego napędu niż domyślny (czyli tego z najniższą literą w symbolu dysku). Aby rozwiązać ten problem, przygotujemy funkcję `KontrolaTackiCD` oraz dwie funkcje: `OpenCD` i `CloseCD` (listing 4.33). Przygotujemy dla nich moduł plików *Multimedia.h*/ *Multimedia.cpp*. Dwie ostatnie funkcje zadeklarujemy w pliku nagłówkowym. Skorzystamy z funkcji `mciSendCommand`[21], która pozwala na przesyłanie do urządzeń multimedialnych poleceń sterujących ich działaniem. Tym razem będzie to polecenie `MCI_SET`, za pomocą którego można ustawiać niektóre ich parametry. W omawianym przypadku ustawienie będzie dotyczyło położenia tacki.

[21] Funkcje `mciSendString` i `mciSendCommand` dostępne są we wszystkich 32-bitowych wersjach Windows.

Listing 4.33. *Zawartość pliku Multimedia.cpp. Funkcje OpenCD i CloseCD należy zadeklarować w pliku nagłówkowym*

```
#include "stdafx.h"
#include "mmsystem.h"
#pragma comment(lib, "Winmm.lib")

BOOL KontrolaTackiCD(LPCTSTR Drive, BOOL Operacja)
{
   BOOL Wynik = FALSE;

   MCI_OPEN_PARMS parametry;
   parametry.dwCallback = 0; //uchwyt okna, do którego mogłyby być wysyłane
   ↪komunikaty powiadamiające o wysunięciu tacki
   parametry.lpstrDeviceType = L"CDAudio";
   parametry.lpstrElementName = Drive; //Symbol dysku w formacie X:

   //Inicjalizacja urządzenia
   mciSendCommand(0, MCI_OPEN, MCI_OPEN_ELEMENT | MCI_OPEN_TYPE, (long)&parametry);

   if (Operacja)
      //Otwieranie napędu CD-ROM
      Wynik = (mciSendCommand(parametry.wDeviceID, MCI_SET, MCI_SET_DOOR_OPEN, 0) == 0);
   else
      //Zamykanie napędu CD-ROM
      Wynik = (mciSendCommand(parametry.wDeviceID, MCI_SET, MCI_SET_DOOR_CLOSED, 0)==0);

   //zwolnienie dostępu do urządzenia
   mciSendCommand(parametry.wDeviceID, MCI_CLOSE, MCI_NOTIFY, (long)&parametry);
   return Wynik;
}

BOOL OpenCD(LPCTSTR Drive)
{
   return KontrolaTackiCD(Drive, TRUE);
}

BOOL CloseCD(LPCTSTR Drive)
{
   return KontrolaTackiCD(Drive, FALSE);
}
```

Aby uczynić kod bardziej przejrzystym, w powyższych funkcjach pominęliśmy obsługę błędów. Gdybyśmy ją uwzględnili, polecenie np. otwarcia dostępu do urządzenia powinno wyglądać następująco:

```
MCIERROR mciBlad = mciSendCommand(0,MCI_OPEN,MCI_OPEN_ELEMENT | MCI_OPEN_TYPE,
↪(long)&parametry);
if (mciBlad != 0)
{
   wchar_t opisBledu[MAXERRORLENGTH];
   mciGetErrorString(mciBlad, opisBledu, MAXERRORLENGTH);
   AfxMessageBox(opisBledu);
   return false;
}
```

gdzie `opisBledu` to tablica znaków o długości `MAXERRORLENGTH`, a `mciBlad` to zmienna typu `long`.

Użycie funkcji jest bardzo proste, np. `OpenCD(L"D:");`. Równie proste byłoby przygotowanie aplikacji konsolowej `eject` korzystającej z poniższej funkcji, a której zadaniem byłoby właśnie wysuwanie (a z parametrem `-t` wsuwanie) wskazanego napędu. Pozostawiam to Czytelnikowi jako „zadanie domowe".

Wykrywanie wysunięcia płyty z napędu lub umieszczenia jej w napędzie CD lub DVD

Aby wykryć moment wysunięcia płyty z napędu lub umieszczenia jej w napędzie, konieczne jest wykorzystanie mechanizmu komunikatów Windows. Dlatego projekt ten umieszczony został w rozdziale 6.

Sprawdzanie stanu wybranego napędu CD-Audio

Kilka kolejnych projektów dotyczy obsługi napędów z płytą CD-Audio. Oznacza to tylko i wyłącznie kontrolę napędu, który umożliwia sterowanie odtwarzaniem muzyki z płyt CD-Audio. Są napędy, które nie pozwalają na taką kontrolę (np. napędy montowane w notebookach), lub takie, które umożliwiają ją tylko w pewnym stopniu. A nawet jeżeli napęd pozwala na taką kontrolę, ale napęd optyczny nie jest połączony odpowiednim przewodem z kartą muzyczną, możemy nic nie usłyszeć. Poza tym warto wiedzieć, że muzykę można odtwarzać także w inny sposób. Przykładem jest Windows Media Player, który odczytuje dane z płyty i odtwarza je, nie korzystając ze służących do tego funkcji napędów, ale za pośrednictwem urządzeń odtwarzania plików dźwiękowych (co obciąża nieco procesor, ale jest niezależne od napędu).

Pierwsza z tej serii to funkcja `StanCDAudio`, widoczna na listingu 4.34. Funkcja ta pozwala określić, w jakim stanie jest napęd CD, a dokładnie, czy zawiera płytę i czy jest ona właśnie odtwarzana (mowa o odtwarzaniu przez napęd, a nie np. przez Windows Media Player). Musimy ponownie wykorzystać funkcję `mciSendCommand`, która za cenę mniejszej wygody obsługuje wiele typów urządzeń multimedialnych. Tym razem użyjemy polecenia `MCI_STATUS` i związanej z nim struktury `MCI_STATUS_PARMS`. Jak zwykle należy pamiętać o otwarciu i zamknięciu dostępu do urządzenia (polecenia `MCI_OPEN` i `MCI_CLOSE`).

Listing 4.34. *Funkcja sprawdzająca stan napędu CD. Należy ją umieścić w pliku Multimedia.cpp i zadeklarować w pliku nagłówkowym*

```
unsigned long StanCDAudio(LPCTSTR Drive)
{
   MCI_OPEN_PARMS parametry;
   parametry.dwCallback = 0;
   parametry.lpstrDeviceType = L"CDAudio";
   parametry.lpstrElementName = Drive; //Literą dysku musi być np. "X:"
   mciSendCommand(0, MCI_OPEN, MCI_OPEN_ELEMENT | MCI_OPEN_TYPE, (long)&parametry);
```

```
    MCI_STATUS_PARMS stan;
    stan.dwItem = MCI_STATUS_MODE;
    mciSendCommand(parametry.wDeviceID,MCI_STATUS,MCI_WAIT |
    ↪MCI_STATUS_ITEM,(long)&stan);
    unsigned long wynik=stan.dwReturn; //stan MCI zaczyna się od 524

    mciSendCommand(parametry.wDeviceID,MCI_CLOSE,MCI_NOTIFY,(long)&parametry);
    return wynik;
}
```

Wartość zwracana przez tę funkcję świadczy o stanie napędu (są to liczby od 524 wzwyż). Listing 4.35 zawiera przykład jej użycia, a jednocześnie wymienia ważniejsze zwracane przez funkcję wartości (stałe zdefiniowane są w pliku *mmsystem.h*). Przed jej testowaniem proszę pamiętać o włożeniu płyty CD-Audio do napędu.

Listing 4.35. *Przykład wykorzystania funkcji StanCDAudio*

```
#include "mmsystem.h"

void CMCIDlg::OnBnClickedButton3()
{
   wchar_t katalogNaPlycie[MAX_PATH];
   edit1.GetWindowTextW(katalogNaPlycie,MAX_PATH);
   if(katalogNaPlycie != L"")
   {
      unsigned long wynik=StanCDAudio(katalogNaPlycie);
      switch (wynik)
      {
         case MCI_MODE_NOT_READY: MessageBox(L"Napęd nie jest gotowy (brak płyty
         ↪CD-Audio)"); break;
         case MCI_MODE_PAUSE: MessageBox(L"Odtwarzanie wstrzymane (pauza)"); break;
         case MCI_MODE_PLAY: MessageBox(L"Trwa odtwarzanie"); break;
         case MCI_MODE_STOP: MessageBox(L"Odtwarzanie zatrzymane (stop)"); break;
         case MCI_MODE_OPEN: MessageBox(L"Tacka jest wysunięta"); break;
         case MCI_MODE_RECORD: MessageBox(L"Trwa zapis na płytę"); break;
         case MCI_MODE_SEEK: MessageBox(L"Szukanie"); break;
         default:
            CString temp;
            temp.Format(L"Kod błędu: %lu (prawdopodobnie napęd nie jest dyskiem
            ↪optycznym)", wynik);
            MessageBox(temp);
            break;
      }
   }
}
```

Jak zbadać, czy w napędzie jest płyta CD-Audio

Funkcja `StanCDAudio` pozwala w zasadzie na sprawdzenie, czy mamy do czynienia z płytą CD-Audio. Test taki możemy też wykonać w nieco inny, bardziej skrupulatny sposób. Możemy mianowicie sprawdzić, czy na dysku są dostępne jakieś utwory (ścieżki

muzyczne). W tym celu przygotujemy funkcję pobierającą symbol napędu i zwracającą wartość logiczną odpowiadającą odnalezieniu płyty z muzyką. Działanie funkcji polega na wielostopniowym teście zaczynającym się od sprawdzenia typu napędu, jego stanu i wreszcie na odnalezieniu ścieżek muzycznych (listing 4.36).

Listing 4.36. *Jeżeli płyta zawiera ścieżki audio, uznajemy, że jest to płyta CD-Audio*

```
bool IsCDAudio(LPCTSTR Drive)
{
   MCI_OPEN_PARMS parametry;
   parametry.dwCallback = 0;
   parametry.lpstrDeviceType = L"CDAudio";
   parametry.lpstrElementName = Drive;
   MCIERROR mciBlad = mciSendCommand(0, MCI_OPEN, MCI_OPEN_ELEMENT |
   ↪MCI_OPEN_TYPE,(long)&parametry);
   if (mciBlad != 0) return false;

   MCI_STATUS_PARMS stanNapedu;
   stanNapedu.dwCallback = 0;
   stanNapedu.dwItem = MCI_CDA_STATUS_TYPE_TRACK;
   stanNapedu.dwTrack = 1;
   mciBlad=mciSendCommand(parametry.wDeviceID,MCI_STATUS,MCI_TRACK |
   ↪MCI_STATUS_ITEM,(long)&stanNapedu);
   if (mciBlad!=0) return false;

   bool wynik;
   switch (stanNapedu.dwReturn)
   {
      case MCI_CDA_TRACK_AUDIO: wynik = true; break;
      default: wynik = false; break;
   }

   mciSendCommand(parametry.wDeviceID, MCI_CLOSE, MCI_NOTIFY, (long)&parametry);
   return wynik;
}
```

Kontrola napędu CD-Audio

Przejdźmy do zasadniczej funkcji tego podrozdziału, która pozwoli nam rozpoczynać, wstrzymywać, wznawiać i zatrzymywać odtwarzanie płyt CD-Audio. Przypominam, że istnieją napędy, nawet te nowe, które nie wspierają sprzętowego odtwarzania płyt CD-Audio. Najczęściej można je spotkać w notebookach.

Żeby uniknąć powtarzania kodu (przy wszystkich tych poleceniach należy otworzyć i zamknąć dostęp do urządzenia), przygotujemy funkcję `KontrolaCDAudio`, której nie będziemy udostępniać poza plikiem *Multimedia.cpp*, oraz zbiór czterech funkcji udostępnionych w pliku nagłówkowym, a które będą obsługiwać poszczególne czynności. Odpowiadają im cztery polecenia MCI, a więc: `MCI_PLAY`, `MCI_STOP`, `MCI_PAUSE` i `MCI_RESUME`. Listing 4.37 przedstawia wszystkie funkcje.

Listing 4.37. *Funkcje pozwalające na odtwarzanie muzyki z płyt CD-Audio*

```
bool KontrolaCDAudio(LPCTSTR Drive, ULONG Operacja)
{
   MCI_OPEN_PARMS parametry;
   parametry.dwCallback = 0;
   parametry.lpstrDeviceType = L"CDAudio";
   parametry.lpstrElementName = Drive; //Literą dysku musi być np. "X:"
   mciSendCommand(0, MCI_OPEN, MCI_OPEN_ELEMENT | MCI_OPEN_TYPE, (long)&parametry);

   bool wynik=(mciSendCommand(parametry.wDeviceID, Operacja, 0, 0) == 0);

   mciSendCommand(parametry.wDeviceID,MCI_CLOSE,MCI_NOTIFY,(long)&parametry);
   return wynik;
}

bool PlayCDAudio(LPCTSTR Drive)
{
   return KontrolaCDAudio(Drive, MCI_PLAY);
}

bool ResumeCDAudio(LPCTSTR Drive)
{
   return KontrolaCDAudio(Drive, MCI_RESUME);
}

bool PauseCDAudio(LPCTSTR Drive)
{
   if (StanCDAudio(Drive) != 525)
      return KontrolaCDAudio(Drive, MCI_PAUSE); //gdy odtwarzanie
   else
      return ResumeCDAudio(Drive); //gdy zatrzymany
}

bool StopCDAudio(LPCTSTR Drive)
{
   return KontrolaCDAudio(Drive, MCI_STOP);
}
```

Funkcja `Pause` jest na tyle sprytna, że sprawdza, czy odtwarzanie nie było wcześniej wstrzymane — jeżeli było, wznawia je za pomocą funkcji `Resume`. Korzysta w tym celu z funkcji `StanCDAudio`.

Polecam uwadze Czytelnika program *CD-Audi*, który wykorzystuje i testuje zdefiniowane w tym podrozdziale funkcje. Dostępny jest w dołączonych do książki materiałach.

Multimedia (pliki dźwiękowe WAVE)

Asynchroniczne odtwarzanie pliku dźwiękowego

Kolejnym przykładem zastosowania biblioteki MCI jest odtwarzanie różnego typu plików audio i wideo. Jeżeli jednak chcemy odtworzyć pojedynczy plik *.wav*, należy wykorzystać prostszą w obsłudze funkcję WinAPI `PlaySound`[22] . Potrafi ona odtwarzać pliki synchronicznie i asynchronicznie. W pierwszym przypadku działanie funkcji zakończy się dopiero po zakończeniu odtwarzania, w drugim tuż po jego uruchomieniu, a dźwięk odtwarzany jest w osobnym wątku. Ponadto funkcja pozwala na wskazanie jednego ze zdefiniowanych dźwięków systemowych. W końcu to też są tylko pliki *.wav*, ale do ich identyfikacji użyć można nazw-aliasów ze schematów dźwiękowych.

Dwa przedstawione niżej polecenia odtwarzają asynchronicznie (modyfikator `SND_ASYNC`) pliki *.wav*. W pierwszym wskazujemy konkretny plik znajdujący się na dysku, w drugim korzystamy z aliasu, żeby usłyszeć dźwięk odtwarzany przy uruchomieniu systemu.

```
PlaySound(L"c:\\WINDOWS\\MEDIA\\Windows Logon Sound.wav", 0, SND_FILENAME | SND_ASYNC);
PlaySound((LPCWSTR)SND_ALIAS_SYSTEMSTART, 0, SND_ALIAS_ID | SND_ASYNC);
```

Aby wszystko zadziałało, należy zaimportować nagłówek `mmsystem.h`:

```
#include "mmsystem.h"
#pragma comment(lib, "Winmm.lib")
```

Poza przełącznikami `SND_FILENAME` i `SND_ALIAS` do wskazania źródła dźwięku można użyć także `SND_RESOURCE`. Wówczas pierwszy argument musi zawierać identyfikator zasobu. Warto zwrócić uwagę także na modyfikator `SND_LOOP`, który spowoduje odtwarzanie dźwięku w pętli, aż do kolejnego wywołania funkcji `PlaySound` z pierwszym parametrem równym `NULL`, tj. `PlaySound(NULL,0,0)`.

Jak wykryć obecność karty dźwiękowej

Omówiliśmy już odtwarzanie płyt CD-Audio, plików *.wav*, ale może warto byłoby się upewnić, że w systemie w ogóle zainstalowana jest jakaś karta dźwiękowa. Możemy do tego celu użyć funkcji WinAPI `waveoutGetNumDevs` zwracającej liczbę zarejestrowanych urządzeń zdolnych do odtwarzania plików *.wav*. Jeżeli jest przynajmniej jedno, możemy być pewni co do obecności karty dźwiękowej. Oto przykład:

```
if (waveOutGetNumDevs == 0)
   AfxMessageBox(L"Brak karty dźwiękowej!");
else
   AfxMessageBox(L"Karta dźwiękowa jest zainstalowana");
```

[22] Dostępna we wszystkich 32-bitowych wersjach Windows.

Kontrola poziomu głośności odtwarzania plików dźwiękowych

Odtwarzanie plików dźwiękowych wiąże się z wykorzystaniem urządzenia, którego obecność badaliśmy w poprzednim projekcie. Urządzenie to wspomaga nie tylko odtwarzanie plików *.wav*, ale również plików *.mp3* oraz ścieżek dźwiękowych filmów *.avi*. Za pomocą funkcji WinAPI `waveoutSetVolume` możemy kontrolować poziom głośności dźwięku generowanego przez to urządzenie. Ze względu na to, że funkcja ta przyjmuje czterobajtową liczbę zawierającą poziom głośności lewego i prawego kanału, zdefiniujemy prostszą w użyciu funkcję rozdzielającą te argumenty (listing 4.38). To samo dotyczy odczytywania poziomu głośności, które jest możliwe dzięki funkcji `waveoutGetVolume`[23].

Listing 4.38. *Funkcje ułatwiające kontrolę głośności kanału WAVE*

```
void UstalPoziomGlosnosciWave(USHORT kanalLewy, USHORT kanalPrawy)
{
   ULONG glosnosc = (kanalLewy << 24) | (kanalPrawy << 8);
   waveOutSetVolume((HWAVEOUT)WAVE_MAPPER, glosnosc);
}

void CNapedOptycznyDlg::CzytajPoziomGlosnosciWave(USHORT &kanalLewy, USHORT &kanalPrawy)
{
   ULONG glosnosc;
   waveOutGetVolume((HWAVEOUT)WAVE_MAPPER, &glosnosc);
   kanalLewy = (USHORT)((glosnosc & 0xFFFF0000) >> 24);
   kanalPrawy = (USHORT)((glosnosc & 0x0000FFFF) >> 8);
}
```

Aby je przetestować:

1. Umieszczamy na formie dwa suwaki `CSliderControl`.
2. Wiążemy z nimi zmienne `slider1` i `slider2`.
3. Przechodzimy do metody `OnInitDialog()` i modyfikujemy ją według listingu 4.39.

Listing 4.39. *Ustawienie własności suwaków*

```
BOOL CPlikiDzwiekoweDlg::OnInitDialog()
{
   CDialog::OnInitDialog();

   // Set the icon for this dialog.  The framework does this automatically
   //  when the application's main window is not a dialog
   SetIcon(m_hIcon, TRUE);         // Set big icon
   SetIcon(m_hIcon, FALSE);      // Set small icon

   USHORT kanalLewy, kanalPrawy;
   CzytajPoziomGlosnosciWave(kanalLewy, kanalPrawy);
```

[23] Obie funkcje dostępne są we wszystkich 32-bitowych wersjach Windows.

```
    slider1.SetRangeMin(0);
    slider1.SetRangeMax(100);
    slider1.SetTicFreq(15);
    slider1.SetPos(kanalLewy);

    slider2.SetRangeMin(0);
    slider2.SetRangeMax(100);
    slider2.SetTicFreq(15);
    slider2.SetPos(kanalPrawy);

    return TRUE;  // return TRUE  unless you set the focus to a control
}
```

4. Przechodzimy do edycji ich własności w celu zmiany pozycji *Notify Before Move* na *True*.

5. Do pierwszego suwaka dodajemy metodę zdarzeniową, związaną z komunikatem NM_CUSTOMDRAW, i umieszczamy w niej polecenie z listingu 4.40.

Listing 4.40. *Głośność w lewym i prawym kanale kontrolować będziemy suwakami*

```
void CPlikiDzwiekoweDlg::OnNMCustomdrawSlider1(NMHDR *pNMHDR, LRESULT *pResult)
{
    LPNMCUSTOMDRAW pNMCD = reinterpret_cast<LPNMCUSTOMDRAW>(pNMHDR);
    // TODO: Add your control notification handler code here
    *pResult = 0;

    UstalPoziomGlosnosciWave(slider1.GetPos(), slider2.GetPos());
}
```

6. Przechodzimy do sekcji mapowania komunikatów i dodajemy w niej polecenie wyróżnione na listingu 4.41, które wiąże drugi suwak z istniejącą metodą.

Listing 4.41. *Sekcja mapowania komunikatów*

```
BEGIN_MESSAGE_MAP(CPlikiDzwiekoweDlg, CDialog)
    ON_WM_PAINT()
    ON_WM_QUERYDRAGICON()
    //}}AFX_MSG_MAP
    ON_BN_CLICKED(IDC_BUTTON1, &CPlikiDzwiekoweDlg::OnBnClickedButton1)
    ON_BN_CLICKED(IDC_BUTTON2, &CPlikiDzwiekoweDlg::OnBnClickedButton2)
    ON_NOTIFY(NM_CUSTOMDRAW, IDC_SLIDER1,
&CPlikiDzwiekoweDlg::OnNMCustomdrawSlider1)
    ON_NOTIFY(NM_CUSTOMDRAW, IDC_SLIDER2,
&CPlikiDzwiekoweDlg::OnNMCustomdrawSlider1)
END_MESSAGE_MAP()
```

Zwykle zamiast ustalania głośności w lewym i prawym kanale udostępnia się użytkownikowi aplikacji komponenty kontrolujące głośność obu kanałów oraz balans. Przygotowanie ich pozostawiam Czytelnikowi, ale uprzedzam, że zadanie to wcale nie jest tak banalne, jak z pozoru może się wydawać.

Kontrola poziomu głośności CD-Audio

Powróćmy jeszcze na chwilę do odtwarzania płyt CD-Audio. Jak kontrolować głośność ich odtwarzania? Służą do tego funkcje `auxSetVolume` i `auxGetVolume`, z których korzysta się zupełnie analogicznie jak z funkcji `waveoutSetVolume` i `waveoutGetVolume` poznanych w poprzednim projekcie. Wzorem poprzedniego projektu przygotujemy ponownie dwie funkcje ułatwiające kontrolę głośności (listing 4.42), jednak tym razem zwracają one wartości informujące o powodzeniu operacji.

Listing 4.42. *Nie zawsze możliwa jest kontrola CD-Audio. Może się więc okazać, że poniższe funkcje nie przynoszą żadnych efektów*

```
bool UstalPoziomGlosnosciCDAudio(USHORT kanalLewy, USHORT kanalPrawy)
{
   ULONG glosnosc = (kanalLewy << 24) | (kanalPrawy << 8);
   return (auxSetVolume(AUX_MAPPER, glosnosc) == MMSYSERR_NOERROR);
}

bool CzytajPoziomGlosnosciCDAudio(USHORT &kanalLewy, USHORT &kanalPrawy)
{
   ULONG glosnosc;
   bool wynik = (auxGetVolume(AUX_MAPPER,&glosnosc) == MMSYSERR_NOERROR);
   kanalLewy = (USHORT)((glosnosc & 0xFFFF0000) >> 24);
   kanalPrawy = (USHORT)((glosnosc & 0x0000FFFF) >> 8);
   return wynik;
}
```

Łatwo się domyślić, że do kontroli głośności urządzenia MIDI służą analogiczne funkcje `midiSetVolume` i `midiGetVolume`.

Inne

Na koniec chciałbym przedstawić zbiór kilku projektów, które trudno zakwalifikować do jednej z omówionych już kategorii, a więc różne dziwne możliwości, nie zawsze całkiem praktyczne.

Pisanie i malowanie na pulpicie

Oto ciekawostka. Znajdziemy okno związane z pulpitem, znajdziemy uchwyt do jego płótna (kontekst wyświetlania) i wykorzystamy go, żeby umieścić na pulpicie dowolny tekst, korzystając z metody `TextOut`. W tym celu:

1. Tworzymy nowy projekt o nazwie *PulpitPisanie*.
2. Formę aplikacji projektujemy według wzoru z rysunku 4.13, gdzie prostokąty pod etykietami koloru tła i koloru czcionki są kontrolkami *ActiveX Microsoft Forms Image 2.0*.

3. Z polem edycyjnym wiążemy zmienną `Edit1`, a z etykietami odpowiednio zmienne `Label1` i `Label2` (należy pamiętać o zmianie ich `ID` z `ID_STATIC` na np. `ID_STATIC10`). Kontrolkom *ActiveX* przypisujemy zmienne `Image1` i `Image2`.

Rysunek 4.13. *Widok projektowanej aplikacji*

4. Do klasy `CPulpitPisanieDlg` dodajemy następujące pola:

```
private:
   LOGFONT lf;
   CHOOSECOLOR cc, textColor;
   CFont newFont;
   CFont *oldFont;
```

5. Klikamy dwukrotnie pierwszy przycisk i umieszczamy w nim polecenia wyróżnione na listingu 4.43.

Listing 4.43. *Formatujemy czcionkę*

```
void CPulpitPisanieDlg::OnBnClickedButton1()
{
   CFontDialog fontDialog;

   if(fontDialog.DoModal() != IDCANCEL)
   {
      fontDialog.GetCurrentFont(&lf);
      newFont.CreateFontIndirectW(&lf);
      Label1.SetWindowTextW(lf.lfFaceName);
      CString temp;
      temp.Format(L"%d", fontDialog.GetSize()/10);
      Label2.SetWindowTextW(temp);
      textColor.rgbResult = fontDialog.GetColor();
      Image2.put_BackColor(textColor.rgbResult);
   }
}
```

6. Tworzymy metodę zdarzeniową dla drugiego z przycisków, którą definiujemy zgodnie z listingiem 4.44.

Listing 4.44. *Modyfikujemy kolor tła*

```
void CPulpitPisanieDlg::OnBnClickedButton2()
{
   CColorDialog colorDialog;
   if(colorDialog.DoModal() != IDCANCEL)
```

```
    {
        cc = colorDialog.m_cc;
        Image1.put_BackColor(cc.rgbResult);
    }
}
```

7. Przechodzimy wreszcie do punktu kulminacyjnego i klikamy dwukrotnie w polu edycyjnym, tworząc w ten sposób domyślną metodę zdarzeniową, w której umieszczamy polecenia z listingu 4.45.

Listing 4.45. *Napis nie będzie trwały, gdyż w żaden sposób nie zadbaliśmy o jego odświeżanie*

```
void CPulpitPisanieDlg::OnEnChangeEdit1()
{
    // TODO:  If this is a RICHEDIT control, the control will not
    // send this notification unless you override the CDialog::OnInitDialog()
    // function and call CRichEditCtrl().SetEventMask()
    // with the ENM_CHANGE flag ORed into the mask.

    CClientDC dc(GetDesktopWindow());
    CString temp;

    oldFont = dc.SelectObject(&newFont);
    dc.SetBkColor(cc.rgbResult);
    dc.SetBkMode(OPAQUE);
    dc.SetTextColor(textColor.rgbResult);
    Edit1.GetWindowTextW(temp);
    dc.TextOut(200, 200, temp);
    dc.SelectObject(oldFont);
}
```

Po uruchomieniu programu możemy wybrać krój i kolor czcionki, kolor tła i wpisać dowolny tekst w polu edycyjnym. Tekst ten pojawi się równocześnie na pulpicie (rysunek 4.14). Czcionka i kolory napisu powinny być zgodne z wybranymi.

Rysunek 4.14. *Prosta aplikacja służąca do pisania w oknie pulpitu*

Czy Windows mówi po polsku?

Dynamiczna lokalizacja programu polega na sprawdzeniu, jaki jest język zainstalowanej wersji Windows, i wyświetlaniu komunikatów w tym języku. Sprawa się upraszcza, jeżeli program operuje tylko dwoma językami: polskim, gdy mamy do czynienia z polską wersją systemu, i angielskim — w każdym innym przypadku. Listing 4.46 pokazuje, jak sprawdzić za pomocą funkcji `GetSystemDefaultLangID`[24], czy mamy do czynienia z polską wersją Windows.

Listing 4.46. *Sprawdzamy domyślny język zainstalowanego systemu Windows*

```
bool CzyJezykPolski()
{
   return (GetSystemDefaultLangID() == 0x0415);
}
```

Jak zablokować uruchamiany automatycznie wygaszacz ekranu?

Taka możliwość przydaje się przy projektowaniu aplikacji, które po uruchomieniu dalej pracują samodzielnie (bez konieczności ich kontroli myszą lub klawiaturą), a wynik ich działania jest obserwowany przez użytkownika. Są to na przykład wszelkiego rodzaju odtwarzacze wideo.

1. Na formie umieszczamy kontrolkę `CheckBox`, z którą wiążemy zmienną `checkBox1`.
2. Klikamy ją dwukrotnie i w utworzonej w ten sposób domyślnej metodzie zdarzeniowej umieszczamy polecenia z listingu 4.47.

Listing 4.47. *Blokujemy uruchamiany automatycznie przez system wygaszacz ekranu*

```
void CBlokadaWygaszaczaEkranuDlg::OnBnClickedCheck1()
{
   int aktywny=checkBox1.GetCheck()?0:1;
   SystemParametersInfo(SPI_SETSCREENSAVEACTIVE,aktywny,NULL,0);
}
```

Swoją drogą warto przejrzeć dokumentację użytej w powyższym kodzie funkcji `SystemParametersInfo`[25]. Pozwala ona nie tylko zablokować wygaszacz ekranu czy ustalić czas, po którym system go uruchomi, ale również zmodyfikować wiele innych ustawień powłoki systemu.

[24] Dostępna we wszystkich 32-bitowych wersjach Windows.

[25] Dostępna we wszystkich 32-bitowych wersjach Windows.

Zmiana tła pulpitu

Przykładem innej sztuczki, która jest możliwa dzięki zmianie ustawień systemu przeprowadzonej za pomocą funkcji `SystemParametersInfo`, jest zmiana tła pulpitu (czyli tzw. tapety).

```
SystemParametersInfo(SPI_SETDESKWALLPAPER,
                     0, nazwa pliku.GetBuffer(),
                     SPIF_UPDATEINIFILE | SPIF_SENDWININICHANGE);
```

Rozbudowany przykład prezentujący sposób użycia tej funkcji znajduje się w dołączonych do książki źródłach. Efekt działania tego projektu przedstawia rysunek 4.15.

Rysunek 4.15. *Wprawka programistyczna — program do podglądu i zmiany tła pulpitu*

Powyższa funkcja nie zadziała, jeżeli uaktywniona jest usługa *Active Desktop*. Na szczęście dla naszej metody nie stała się ona nigdy zbyt popularna. Jeżeli jednak uprzemy się, aby zmienić tło pulpitu przy uaktywnionej tej usłudze, konieczne jest wykorzystanie obiektu COM, który służy do jej kontroli. Obiekt identyfikowany jest przez stałą `TGUID CLSID_ActiveDesktop="{75048700-EF1F-11D0-9888-006097DEACF9}"`. Natomiast interfejs, który zawiera potrzebną do zmiany tła funkcję `SetWallpaper`, to `IActiveDesktop`. Pamiętać jeszcze należy o wywołaniu metody `ApplyChanges` — i gotowe.