Jacek Matulewski
http://www.phys.uni.torun.pl/~jacek/

Tworzenie aplikacji Windows

Wykorzystanie funkcji WinAPI (C++ Builder)

Ćwiczenia

Toruń, 12 grudnia 2002

Najnowsza wersja tego dokumentu znajduje się pod adresem
http://www.phys.uni.torun.pl/~jacek/dydaktyka/rad/rad4_api_cbuilder.pdf

Źródła opisanych w tym dokumencie programów znajdują się pod adresem
http://www.phys.uni.torun.pl/~jacek/dydaktyka/rad/rad4_api.zip

I. Spis treści

I. Spis treści 2
II. Funkcje obsługujące zamykanie i wstrzymywanie systemu 3
1. ExitWindowsEx 3
2. InitiateSystemShutdown 7
3. SetSystemPowerState 10
III. Proste uruchamianie i kontrola aplikacji 12
1. WinExec 12
2. ShellExecute 12
3. Zmiana priorytetu bieżącej aplikacji 13
4. Zmiana priorytetu i zamykanie innej działającej aplikacji 14
5. Lista okien 15
IV. Odczytywanie informacji o dysku 18
1. Odczytywanie parametrów dysku 18
2. Klasa pobierający i udostępniający dane o dysku 22
3. Komponent TDiskInfoPanel 23
4. Komponent niewidoczny TDiskInfo 27
V. Kontrola okien/form 28
1. Kontrola okna/formy 28
2. Okienka dialogowe 29
3. Okna o dowolnym kształcie 29
VI. Różności 32
1. Pobieranie dostępnym trybów wyświetlania karty graficznej 32
2. Zmiana trybu wyświetlania 34
VII. Gdzie szukać pomocy? 35

II. Funkcje obsługujące zamykanie i wstrzymywanie systemu

1. ExitWindowsEx

Funkcja `ExitWindowsEx` umożliwia wylogowanie użytkownika, ponowne uruchomienie i wyłączenie komputera. Wykorzystanie tej funkcji w systemach Windows 95/98/Millenium różni się od jej użycia w Windows NT/2000/XP ze względu na możliwość logowania wielu użytkowników, także zdalnie, na drugiej rodzinie systemów i różne poziomy ich uprawnień.
W systemach opartych na technologii NT przed użyciem funkcji `ExitWindowsEx`, poza wylogowywaniem, niezbędne jest potwierdzenie odpowiedniego uprawnienia użytkownika (ang. *privilege*) przez system – to czy dany użytkownik lub grupa użytkowników może wyłączać system zależy od ustawień konta zmienianych przez administratora systemu.

Zagadnienie uzyskiwania uprawnień jest dobrze zilustrowane w Win32 SDK.
Składnia funkcji `ExitWindowsEx` jest następująca:

```
bool ExitWindowsEx(UINT uAkcja, DWORD dwZarezerwowane);
```

Możliwe są następujące wartości pierwszego parametru (w nawiasach – rzeczywista wartość dziesiętna i jej reprezentacja binarna):
`EWX_LOGOFF` (0, 0000) – zamyka wszystkie procesy użytkownika wywołującego funkcję (uruchamiającego program, z którego została ona wywołana) i wylogowuje użytkownika. Nie wymaga żadnych uprawnień. Identyczne działanie ma funkcja ExitWindows.
`EWX_POWEROFF` (8, 1000) – zamyka wszystkie procesy i następnie zamyka system. W komputerach z płytą ATX wyłączane jest także zasilanie[1].
`EWX_SHUTDOWN` (1, 0001) – j.w., ale bez wyłączenia zasilania (pojawia się ekran „Teraz można bezpiecznie wyłączyć komputer").
`EWX_REBOOT` (2, 0010) – j.w., ale po zamknięciu systemu zaczyna jego ponowne uruchomienie.
Dodatkowo dostępny jest modyfikator `EWX_FORCE` (4, 0100). Jeżeli jest obecny – aplikacje zamykane są bez pytania o zachowanie danych.

Jeżeli chcemy, żeby system został ponownie uruchomiony bez pytań o zachowanie danych w aplikacji (tj. w aplikacjach C++ Buildera/Delphi bez wywoływania metody zdarzeniowej `OnCloseQuery`) należy w pierwszym parametrze podać `EWX_REBOOT | EWX_FORCE = 0110`.

Drugi parametr jest obecnie ignorowany.

Manipulacja uprawnieniami procesu[2] możliwa jest po otwarciu tokenu procesu[3]. Pozwala na to funkcja WinAPI `OpenProcessToken()`, która przyjmuje trzy argumenty: uchwyt do procesu (dla bieżącego procesu można skorzystać z funkcji `GetCurrentProcess()`), żądany przez nas rodzaj dostępu do tokenu procesu (należy skorzystać z predefiniowanych stałych) – funkcja `AdjustTokenPrivileges` z której będziemy dalej korzystać modyfikuje i pobiera uprawnienia, musimy więc ustalić TOKEN_ADJUST_PRIVILEGES | TOKEN_QUERY, trzeci parametr zwraca wskaźnik do tokenu procesu. Ostatecznie napiszemy:

```
OpenProcessToken(
```

---

[1] Dokumentacja MS SDK o tym nie wspomina, ale w niektórych wersjach systemów Windows 95/98/Millenium argument EWX_POWEROFF nie działa w prawidłowy sposób (powoduje częściowe wylogowanie – częściowe, bo niektóre programy nie zostają zamknięte). W tych systemach powinno się korzystać z EWX_SHUTDOWN, która w komputerach z płytą ATX spowodują też wyłączenie zasilania.
[2] W systemach opartych o NT proces domyślnie nie ma żadnych uprawnień do zamykania systemu, ale może uzyskać wszystkie uprawnienia użytkownika, który go uruchomił.
[3] Token procesu to nie jest może najlepsze tłumaczenie, jeżeli w ogóle można to nazwać tłumaczeniem angielskiego *process token*, tak jest jednak dość powszechnie używane. Często używa się również określenia „znacznik procesu".

```
      GetCurrentProcess(),
      TOKEN_ADJUST_PRIVILEGES | TOKEN_QUERY,
      &hToken);
```

Musimy wcześniej zadeklarować uchwyt dla tokenu procesu (`HANDLE hToken`) – funkcja `OpenProcessToken()` zwraca wskaźnik uchwytu, stąd operator referencji & przy ostatnim argumencie. Funkcja `OpenProcessToken` zwraca wartość błędu, więc można skorzystać z kontrukcji „`if`" do ewentualnego wyświetlenia komunikatu błędu.

```
if (!OpenProcessToken(...))
      {
      ShowMessage("Nie udało się otworzyć tokenu procesu");
      return false;
      }
```

Aby zmodyfikować uprawnienia procesu należy najpierw pobrać unikalny 64-bitowy kod uprawnienia (LUID[4]). Możemy go pobrać funkcją WinAPI o nazwie `LookupPrivilegeValue`. Pierwszym argumentem jest nazwa (dokładniej adres nazwy, wskaźnik do łańcucha) systemu, w którym chcemy uzyskać uprawnienia. Ponieważ funkcja `ExitWindowsEx` nie potrafi zamykać zdalnych systemów[5] Windows NT/2000/XP, możemy pominąć ten argument wpisując `NULL`. Drugi argument jest nazwą uprawnienia, którego LUID chcemy pobrać. Tu korzystamy z predefiniowanej stałej uprawnienia do zamykania systemy SE_SHUTDOWN_NAME. Uprawnienie to pozwala zarówno na zamknięcie systemu (także z wyłączeniem zasilania) jak i jego ponowne uruchomienie. Za pomocą ostatniego argumentu przekazujemy wskaźnik zmiennej 64-bitowej, do której zapisana będzie wartość kodu LUID. Zadeklarujmy więc zmienną typu LARGE_INTEGER lub równoznacznego typu LUID[6], w której będziemy przechowywali ten kod (`LUID Luid`).
Ostatecznie wpisujemy:

```
LookupPrivilegeValue(NULL, SE_SHUTDOWN_NAME, &Luid);
```

Do zmiany uprawnień musimy skorzystać z funkcji WinAPI `AdjustTokenPrivileges`, która przyjmuje sześć argumentów. Pierwsze trzy pozwalają na modyfikację uprawnień, i te będą nas interesować. Kolejne trzy służą do pobierania informacji o uprawnieniach (z ich powodu musieliśmy wpisać `TOKEN_QUERY` w funkcji `OpenTokenProcess`). Pierwszym argumentem jest uchwyt tokenu procesu, dla którego zdobywamy uprawnienia (tj. w naszym przypadku hToken). Drugim jest wartość logiczna, która, jeżeli jest ustawiona na `TRUE` odbiera wszelkie uprawnienia procesowi. My oczywiście wpisujemy `FALSE`. Trzeci jest adresem do struktury WinAPI o nazwie `TOKEN_PRIVILEGES` (typu zdefiniowanego w WinAPI o nazwie _TOKEN_PRIVILEGES) zawierającej dwa elementy: ilość uprawnień i ich tablicę. Pełna deklaracja tej struktury jest następująca:

```
typedef struct _TOKEN_PRIVILEGES
      {
      DWORD PrivilegeCount;
      LUID_AND_ATTRIBUTES Privileges[ANYSIZE_ARRAY];
      } TOKEN_PRIVILEGES;
```

Tablica elementów `LUID_AND_ATTRIBUTES` również jest strukturą WinAPI zdefiniowaną następująco:

```
typedef struct _LUID_AND_ATTRIBUTES
      {
      LUID   Luid;
      DWORD  Attributes;
      } LUID_AND_ATTRIBUTES;
```

---

[4] LUID to 64-bitowa (`LARGE_INTEGER`) wartość unikalna w danym systemie, w którym został wygenerowany od uruchomienia systemu. Za pomocą LUID oznacza się np. pobrane uprawnienia.
[5] Potrafi to funkcja `InitiateSystemShutdown` omówiona w następnym paragrafie.
[6] `typedef LARGE_INTEGER LUID`

My skorzystamy tylko z jednego elementu, ustalmy więc licznik na 1. Pierwszy element tablicy (o numerze 0) powinien posiadać pobrany przez nas numer LUID uprawnienia i wartość, którą chcemy nadać aktywującą to uprawnienie (należy skorzystać z predefiniowanej stałej `SE_PRIVILEGE_ENABLED`).

```
TOKEN_PRIVILEGES tkp;
tkp.PrivilegeCount=1;
tkp.Privileges[0].Luid=Luid;
tkp.Privileges[0].Attributes=SE_PRIVILEGE_ENABLED;
```

Tak przygotowana struktura uprawnień, a dokładniej jej adres, może posłużyć za trzeci argument funkcji `AdjustTokenPrivileges`. Pozostałe trzy tj. długość bufora do pobrania struktury `TOKEN_PRIVILEGES` zawierającej obecny stan uprawnień, adres tej struktury i zwracany rozmiar bufora, ignorujemy wpisując kolejno 0, `(PTOKEN_PRIVILEGES) NULL`[7] i 0. W efekcie wywołanie funkcji `AdjustTokenPrivileges` jest następujące:

```
AdjustTokenPrivileges(hToken, FALSE, &tkp, 0, (PTOKEN_PRIVILEGES) NULL, 0);
```

Funkcja zwraca wartość logiczną (`TRUE`, jeżeli powiodło się). Kod błędu można (co jest typowe dla WinAPI) odczytać za pomocą funkcji WinAPI `GetLastError()`). Można więc napisać dalej:

```
if (GetLastError() != ERROR_SUCCESS)
      ShowMessage("Przyznanie uprawnienia nie powiodlo sie.");
```

W tym momencie, jeżeli nie pojawił się błąd, proces uzyskał uprawnienie do zamykania systemu na lokalnej maszynie. Możemy więc wywołać funkcję `ExitWindowsEx()` z odpowiednimi argumentami opisanymi na początku paragrafu.

Następnie elegancja i bezpieczeństwo systemu wymaga, aby procesowi odebrać uzyskane przed chwilą uprawnienie. Ponieważ możemy skorzystać z przygotowanych wcześniej danych sprawa jest prosta. Zmieniamy wartość uprawnienia na 0 (co oznacza „brak uprawnienia”) i ponownie korzystamy z funkcji `AdjustTokenPrivileges()`:

```
tkp.Privileges[0].Attributes=0;
AdjustTokenPrivileges(hToken, FALSE, &tkp, 0, (PTOKEN_PRIVILEGES) NULL, 0);
if (GetLastError() != ERROR_SUCCESS)
      ShowMessage("Odebranie uprawnienia nie powiodlo sie.");
```

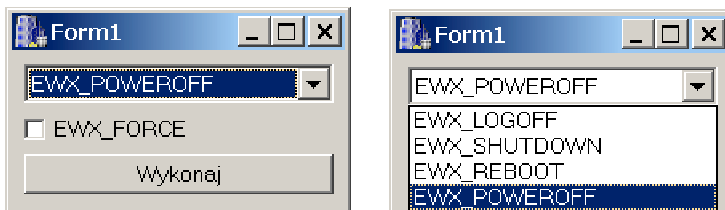
Aby uprościć obsługę programu napiszmy nakładkę na funkcję `ExitWindowsEx` uzyskującą uprawnienia (funkcja ta w umieszczona jest w oddzielnym pliku o nazwie ExitWindowsEx1.cpp):

```
bool ExitWindowsEx1(int Action)
{

//Jezeli tylko logoff (do niego nie trzeba uprawnien) to robi i wychodzi
if ((Action==EWX_LOGOFF) || (Action==(EWX_LOGOFF | EWX_FORCE)))
      return ExitWindowsEx(Action,0);

//Jezeli nie NT, to robi od razu
//(funkcje pytajace o uprawnienia sa nieobecne w Win95/98/ME)
if (!(Win32Platform==VER_PLATFORM_WIN32_NT))
      return ExitWindowsEx(Action,0);



//Jezeli jednak NT to pyta o uprawnienia

HANDLE hToken; //uchwyt do znaku procesu (handle to process token)
```

[7] Zrzutowany na wskaźnik do `TOKEN_PRIVILAGES` (istnieje zadeklarowana w WinAPI typ `PTOKEN_PRIVILEGES`) pusty wskaźnik `NULL`

```
if (!OpenProcessToken(GetCurrentProcess(),
                      TOKEN_ADJUST_PRIVILEGES | TOKEN_QUERY,
                      &hToken))
      ShowMessage("Nie udało się otworzyć tokenu procesu");

//Pobranie LUID dla uprawnienia do zamykania systemu (SE_SHUTDOWN_NAME)
LUID Luid;
LookupPrivilegeValue(NULL, SE_SHUTDOWN_NAME, &Luid);

//Ustalenie wartości dla uprawnienia
TOKEN_PRIVILEGES tkp;
tkp.PrivilegeCount=1;
tkp.Privileges[0].Luid=Luid;
tkp.Privileges[0].Attributes=SE_PRIVILEGE_ENABLED;

//Modyfikacja uprawnienia do zamkniecia systemu dla biezacego procesu
//nie pobieramy informacji o stanie uprawnien sprzed modyfikacji
AdjustTokenPrivileges(hToken, FALSE, &tkp, 0, (PTOKEN_PRIVILEGES) NULL, 0);

//Obsluga bledu
if (GetLastError() != ERROR_SUCCESS)
      ShowMessage("Przyznanie uprawnienia nie powiodlo sie.");


//Uruchomienie funkcji ExitWindowsEx
BOOL fResult = ExitWindowsEx(Action,0);
if (!fResult) ShowMessage("Zamknięcie systemu nie jest możliwe");


//Odbieranie procesowi uprawnienia do zamykania systemu
tkp.Privileges[0].Attributes = 0; //rownoznaczne z NOT_ENABLED
AdjustTokenPrivileges(hToken, FALSE, &tkp, 0, (PTOKEN_PRIVILEGES) NULL, 0);
if (GetLastError() != ERROR_SUCCESS)
      ShowMessage("Odebranie uprawnienia nie powiodlo sie.");

return fResult;
};
```

Do jej przetestowania napiszmy prostą aplikację, która będzie zawierać możliwość wyboru akcji (np. ComboBox) oraz CheckBox uwzględniający możliwość wymuszenia zamykania aplikacji oraz przycisk wywołujący funkcję `ExitWindowsEx1()`.



Metoda związana ze zdarzeniem naciśnięcia przycisku może wyglądać następująco:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
if (ComboBox1->ItemIndex==-1) return;

int Action;
switch (ComboBox1->ItemIndex)
      {
      case 0: Action=EWX_LOGOFF; break;
```

```
        case 1: Action=EWX_SHUTDOWN; break;
        case 2: Action=EWX_REBOOT; break;
        case 3: Action=EWX_POWEROFF; break;
        default: Action=EWX_LOGOFF;
}

if (CheckBox1->Checked) Action=Action | EWX_FORCE;

ExitWindowsEx1(Action);
}
```

Jeżeli `ExitWindowsEx1()` znajduje się w osobnym pliku należy pamiętać, aby dodać ten plik za pomocą dyrektywy `#include`.

2. InitiateSystemShutdown

W systemach NT istnieje jeszcze jedna funkcja WinAPI, która znacznie rozszerza możliwości programisty przy zamykaniu systemu. Jest to `InitiateSystemShutdown`, która umożliwia zamykanie i rebootowanie systemów zdalnych (w obrębie LAN), włączając oczywiście w to system lokalny, z dowolnym opóźnieniem. Jej zastosowanie powoduje wyświetlenie okienka powiadamiającego użytkownika o planowanym zamknięciu systemu; w czasie jego wyświetlania można odwołać zamknięcie systemu (nie umożliwia tego okienko wyświetlające informacje) funkcją WinAPI `AbortSystemShutdown()`.

Napiszemy funkcję `ExitWindowsEx2` analogiczną do napisanej w poprzednim paragrafie. Funkcja ma „obudowywać" funkcję WinAPI `InitiateSystemShutdown` – musi więc pobierać wszystkie potrzebne do jej uruchomienia parametry. Oryginalna deklaracja funkcji `InitiateSystemShutdown` jest następująca:

```
BOOL InitiateSystemShutdown(
        LPTSTR lpMachineName,      //adres nazwy komputera, który ma być zamknięty
        LPTSTR lpMessage,          //adres wyświetlonej informacji w okienku dialogowym
        DWORD dwTimeout,           //opóźnienie zamknięcia
        BOOL bForceAppsClosed,     //wymuszone zamknięcie aplikacji z niezachowanymi danymi
        BOOL bRebootAfterShutdown       //ponowne uruchomienie po zamknięciu systemu
        );
```

Tej samej funkcji `ExitWindowsEx2()` użyjemy do ewentualnego anulowania zamknięcia systemu. Funkcja `AbortSystemShutdown` ma o wiele prostszą składnię:

```
BOOL AbortSystemShutdown(
        LPTSTR lpMachineName       //adres nazwy komputera, który ma być zamknięty
        );
```

Jej jedyny argument pokrywa się z pierwszym argumentem funkcji inicjującej zamknięcie zdalnego systemu.

Nasza funkcją powinna więc przyjmować wszystkie argumenty `InitiateSystemShutdown` oraz wartość logiczną wskazującą na inicjowanie lub anulowanie zamknięcia systemu:

```
bool ExitWindowsEx2(
   bool InicjujLubPrzerwij,
   AnsiString nazwa_komputera,
   AnsiString komunikat,
   int opoznienie,
   bool force,
   bool reboot)
```

Ponieważ funkcja WinAPI `InitiateSystemShutdown` obecna jest tylko w systemach opartych o technologię NT warto dodać na początku naszej funkcji warunek kończący jej działanie z odpowiednim komunikatem, jeżeli system nie jest NT/2000/XP, np.

```
if (!(Win32Platform==VER_PLATFORM_WIN32_NT))
      {
      ShowMessage("Funkcja obsługuje jedynie systemy technologii NT");
      return 0;
      }
```

Następnie sprawdzamy czy podana nazwa jest nazwą pustą (przyjmujemy konwencję, że wówczas funkcja dotyczyć będzie komputera lokalnego) – w takim przypadku adres nazwy komputera przyjmujemy za `NULL`.

```
char* p_nazwa_komputera=nazwa_komputera.c_str();
if (nazwa_komputera=="") p_nazwa_komputera=NULL;
```

Analogicznie jak w funkcji opisanej w pierwszym paragrafie musimy otworzyć token procesu:

```
HANDLE hToken;
if (!OpenProcessToken(GetCurrentProcess(),
                      TOKEN_ADJUST_PRIVILEGES | TOKEN_QUERY,
                      &hToken))
      ShowMessage("Nie udało się otworzyć tokenu procesu");
```

Ponieważ zastosowanie tych funkcji możliwe jest jedynie w systemie NT/2000/XP, więc konieczne jest pobranie LUID uprawnienia w sposób analogiczny jak w powyższym paragrafie **z uwzględnieniem nazwy zdalnego komputera** jako pierwszego argumentu funkcji `LookupPrivilegeValue`. Odpowiednie uprawnienie ma nazwę `SE_REMOTE_SHUTDOWN_NAME`, jeżeli dotyczy zdalnego komputera i `SE_SHUTDOWN_NAME` dla lokalnego:

```
LUID Luid;
LPCTSTR NazwaUprawnienia=SE_REMOTE_SHUTDOWN_NAME;
if (p_nazwa_komputera==NULL) NazwaUprawnienia=SE_SHUTDOWN_NAME;
LookupPrivilegeValue(p_nazwa_komputera, NazwaUprawnienia, &Luid);

TOKEN_PRIVILEGES tkp;
tkp.PrivilegeCount=1;
tkp.Privileges[0].Luid=Luid;
tkp.Privileges[0].Attributes=SE_PRIVILEGE_ENABLED;

AdjustTokenPrivileges(hToken, FALSE, &tkp, 0, (PTOKEN_PRIVILEGES) NULL, 0);
if (GetLastError() != ERROR_SUCCESS)
      ShowMessage("Przyznanie uprawnienia nie powiodlo sie.");
```

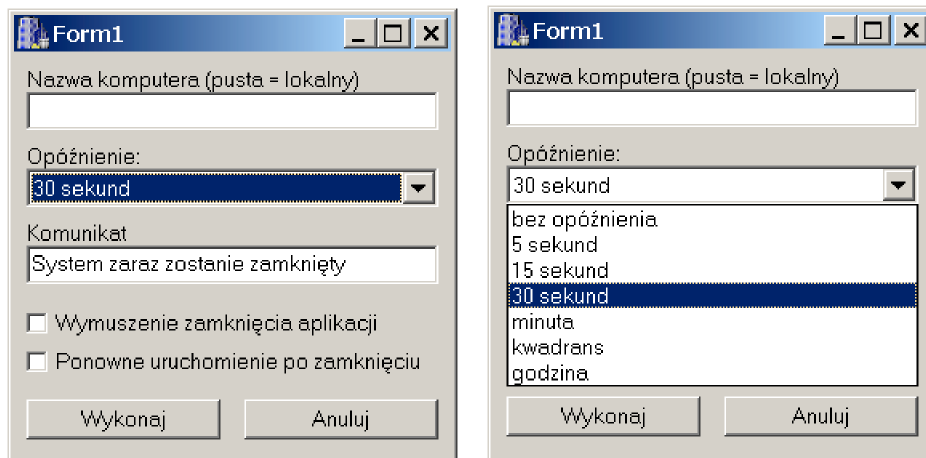
Następnie wywołujemy odpowiednią funkcję zamykającą system lub odwołującą zamykanie. Jako pierwszy argument wykorzystujemy zdefiniowaną wcześniej zmienną `p_nazwa_komputera` przechowującą nazwę zdalnego komputera lub `NULL`, jeżeli operacja ma dotyczyć komputera lokalnego:

```
BOOL fResult;
if (InicjujLubPrzerwij)
      {
      fResult=InitiateSystemShutdown(
                              p_nazwa_komputera,
                              komunikat.c_str(),
                              opoznienie,
                              force,
                              reboot);
      if (!fResult) ShowMessage("Zamknięcie zdalnego systemu niemożliwe");
      }
      else
      {
      fResult=AbortSystemShutdown(p_nazwa_komputera);
      if (!fResult) ShowMessage("Odwołanie zamknięcia nie powiodło się");
      }
```

I na końcu, podobnie jak w poprzednim paragrafie odbieramy procesowi uprawnienia do zamykania systemu:

```
tkp.Privileges[0].Attributes = 0;
AdjustTokenPrivileges(hToken, FALSE, &tkp, 0, (PTOKEN_PRIVILEGES) NULL, 0);
if (GetLastError() != ERROR_SUCCESS)
      ShowMessage("Odebranie uprawnienia nie powiodlo sie.");
```

Projekt formy może być następujący:



Metoda zdarzeniowa związana z przyciskiem „Wykonaj” może być następująca (w głównej formie):

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
int IloscSekund=0;
switch (ComboBox1->ItemIndex)
      {
    case 0: IloscSekund=0; break;
    case 1: IloscSekund=5; break;
    case 2: IloscSekund=15; break;
    case 3: IloscSekund=30; break;
    case 4: IloscSekund=60; break;
    case 5: IloscSekund=900; break;
    case 6: IloscSekund=3600; break;
    default: IloscSekund=30;
    }

ExitWindowsEx2(
            true,
            Edit1->Text,
            Edit2->Text,
            IloscSekund,
            CheckBox1->Checked,
            CheckBox2->Checked);
}
```

a służąca do anulowania:

```
void __fastcall TForm1::Button2Click(TObject *Sender)
{
ExitWindowsEx2(false,Edit1->Text,NULL,0,NULL,NULL);
}
```

W drugim przypadku pomijamy parametry, które nie są wykorzystywane do uzyskania uprawnienia. Potrzebna jest więc tylko nazwa komputera.

Zadanie
Funkcja `ExitWindowsEx2` posiada pewną lukę – po wpisaniu nazwy lokalnego komputera wywoływana jest funkcja zamykająca system z podaną nazwą jako pierwszym argumentem, co nie może się udać ponieważ uzyskane jest uprawnienie zamykania zdalnego na komputerze lokalnym, co nie jest wystarczające. Do warunku rozpoznającego komputer lokalny (jeżeli łańcuch nazwa_komputera jest pusty) dodać warunek sprawdzający, czy nie jest on nazwą komputera lokalnego. Skorzystać z funkcji WinAPI `GetComputerName()`. Porównanie należy wykonać uwzględniając fakt, że wielkość liter nie ma znaczenia (`GetComputerName()` zwraca nazwę komputera pisaną dużymi literami).

3. SetSystemPowerState

Dokumentacja MS SDK o tym nie uprzedza, ale głębokie uśpienie/wstrzymanie systemu oraz hibernacja na systemach Windows NT/2000/XP wymagają uprawnień do zamknięcia systemu. Zmodyfikujemy opisaną w pierwszym paragrafie funkcję `ExitWindowsEx1()` w taki sposób, że wywołanie `ExitWindowsEx()` zastąpimy przez `SetSystemPowerState()`. Składnia tej ostatniej jest bardzo prosta:

```
BOOL SetSystemPowerState(BOOL fSuspend, BOOL fForce);
```

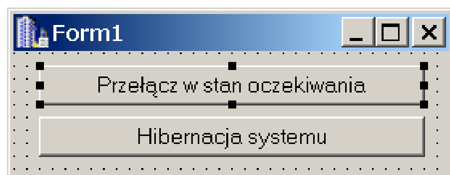
Funkcja przyjmuje dwa argumenty typu logicznego. Pierwszy odpowiada za typ wstrzymania systemu. Jeżeli jest ustawiony na `true` – system zostaje przełączony w stan wstrzymania (minimalizowane jest zużycie energii przez wyłączenie niektórych urządzeń, ale pamięć RAM stale jest podtrzymywana). W tym stanie komputer jest tak naprawdę nadal włączony i wyłączenie go z zasilania spowoduje błąd przywracania systemu. Szczegóły wstrzymywania systemu zależą od sprzętu (płyty głównej) i mogą być modyfikowane w BIOSie. Jeżeli pierwszy argument ustawiony jest na `false` system zostanie zahibernowany (nie na każdym komputerze jest to możliwe). Hibernacja oznacza całkowite wyłączenie komputera z uprzednim skopiowaniem zawartości pamięci RAM na dysk twardy. Oznacza to, że komputer może być wyłączony z prądu, ale nie powinno się w nim modyfikować sprzętu (np. zmieniać dysku) gdyż informacje o sesji zostaną w pełni przywrócone po ponownym włączeniu zasilania.

Skopiujmy funkcję `ExitWindowsEx1()` i umieśćmy w niej wywołanie `SetSystemPowerState()`:

```
bool SetSystemPowerState2(BOOL fSuspend, BOOL fForce)
{
//Komentarze zob. w ExitWindowsEx1

//Jezeli nie NT, to robi od razu
//(funkcje pytajace o uprawnienia sa nieobecne w Win95/98/ME)
if (!(Win32Platform==VER_PLATFORM_WIN32_NT))
      return SetSystemPowerState(fSuspend,fForce);


//Jezeli jednak NT to pyta o uprawnienia
HANDLE hToken;
if (!OpenProcessToken(GetCurrentProcess(),
                      TOKEN_ADJUST_PRIVILEGES | TOKEN_QUERY,
                      &hToken))
      ShowMessage("Nie udało się otworzyć tokenu procesu");

LUID Luid;
LookupPrivilegeValue(NULL, SE_SHUTDOWN_NAME, &Luid);

TOKEN_PRIVILEGES tkp;
tkp.PrivilegeCount=1;
tkp.Privileges[0].Luid=Luid;
tkp.Privileges[0].Attributes=SE_PRIVILEGE_ENABLED;
```

```
AdjustTokenPrivileges(hToken, FALSE, &tkp, 0, (PTOKEN_PRIVILEGES) NULL, 0);
if (GetLastError() != ERROR_SUCCESS)
      ShowMessage("Przyznanie uprawnienia nie powiodlo sie.");


BOOL fResult = SetSystemPowerState(fSuspend,fForce);
if (!fResult) ShowMessage("Wstrzymanie systemu nie powiodło się");


tkp.Privileges[0].Attributes = 0;
AdjustTokenPrivileges(hToken, FALSE, &tkp, 0, (PTOKEN_PRIVILEGES) NULL, 0);
if (GetLastError() != ERROR_SUCCESS)
      ShowMessage("Odebranie uprawnienia nie powiodlo sie.");

return fResult;
};
```

Teraz zajmijmy się przygotowaniem prostej formy:



Aby przełączyć komputer w stan oczekiwania wywołujemy

```
SetSystemPowerState2(true,false);
```

a żeby go zahibernować (drugi argument jest stale wyłączony):

```
SetSystemPowerState2(false,false);
```

Jak widać składnia naszej nakładki jest identyczna jak funkcji oryginalnej. Ponadto funkcja zwraca wartość błędu wywołania oryginalnej `SetSystemPowerState()` – można więc pokusić się o wyświetlenie odpowiedniego komunikatu w razie niepowodzenia.

III. Proste uruchamianie i kontrola aplikacji

1. WinExec

Najwłaściwszymi funkcjami WinAPI służącymi do uruchamiania programów jest `CreateProcess()` lub `CreateProcessAsUser()`. Pozwalają one konfigurować wiele parametrów uruchomienia aplikacji i jej środowiska. Jednak w większości wypadków nie jest to nam do niczego potrzebne i dlatego najczęściej korzysta się po prostu z funkcji WinAPI `WinExec` pomimo, że jest ona klasyfikowana jako przestarzała.

Jej wywołanie jest bardzo proste. Wystarczy podać łańcuch nazwy aplikacji razem z ewentualnymi opcjami jako pierwszy argument. Drugim parametrem jest sposób pokazania okna. Możliwych wartości drugiego parametru jest sporo, ale najczęściej używanymi są SW_NORMAL, SW_MINIMIZE (okno schowane na pasku zadań), SW_MAXIMIZE (rozciągnięte na cały ekran) i SW_HIDE (ukryte).

Wywołajmy np. pasjans:

```
WinExec("sol.exe",SW_MINIMIZE);
```

Jako parametr podana została sama nazwa pliku „sol.exe" (Pasjans) bez katalogu. Ścieżka przeszukiwania jest w takim przypadku następująca: 1) katalog uruchomienia aplikacji macierzystej, 2) bieżący katalog, 3) katalog systemowy (np. windows\system, win2000\system32), 4) główny katalog Windows, 5) katalogi wymienione w systemowej ścieżce przeszukiwania (zmienna środowiskowa PATH).

Podobny efekt można uzyskać wykorzystując inną funkcję WinAPI `ShellExecute()`. Jej szerszym opisem zajmiemy się w następnym paragrafie, a tutaj podam jedynie postać polecenia uruchamiającego aplikację:

```
ShellExecute(Application->Handle,"open","sol.exe","","",SW_NORMAL);
```

W przeciwieństwie do `WinExec()` i funkcji WinAPI wykorzystywanych w poprzednim rozdziale w przypadku funkcji `ShellExecute()` musimy wskazać na nagłówek biblioteki shellapi.h, w której znajduje się definicja tej funkcji:

```
#include <shellapi.h>
```

Funkcje `WinExec()`, `ExitWindowsEx()`, `SetSystemPowerState()` znajdują się w domyślnie dołączanych bibliotekach (`winbase.h` i `winuser.h`)

Zadanie
Funkcja `WinExec` zwraca informację o błędzie. Dodać komunikaty informujące o powodzeniu lub rodzaju błędu.

2. ShellExecute

Funkcja `ShellExecute()` służy do uruchamiania plików wykonywalnych oraz otwierania dokumentów za pomocą skojarzonych z nimi aplikacji[8].

Stworzymy bardzo prosty program pozwalający na przeglądanie zawartości katalogów na dyskach oraz umożliwiający oglądanie zawartości wskazanych dokumentów i uruchamianie aplikacji. Na formie umieśćmy komponenty `DriveComboBox`, `DirectoryListBox` i `FileListBox` (zakładka Win 3.1) oraz odpowiednio je połączmy (własność `DriveComboBox->DirList=DirectoryListBox1` oraz `DirectoryListBox1->FileList=FileListBox1`).

[8] Listę zarejestrowanych typów plików (rozpoznawalnych przez Windows po rozszerzeniu nazwy pliku) można obejrzeć korzystając z Eksploratora, menu: Narzędzia, Opcje folderów ..., zakładka Typy plików.

Chcemy aby kliknięcie pliku na `FileListBox1` uruchamiało wskazany plik/dokument (uruchomienie aplikacji domyślnej dla danego rozszerzenia pliku). Wystarczy w odpowiedniej metodzie zdarzeniowej wywołać funkcję `ShellExecute()`:

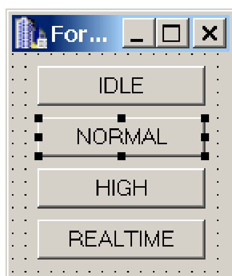
```
void __fastcall TForm1::FileListBox1DblClick(TObject *Sender)
{
ShellExecute(Application->Handle, "open",
             FileListBox1->FileName.c_str(),"","",SW_NORMAL);
}
```

Zadanie
Dodać do formy klawisze „Otwórz”, „Drukuj” i „Przeglądaj” służące odpowiednio do otwierania i drukowania pliku oraz przeglądania wybranego katalogu. Odpowiednie akcje można uzyskać nadając drugiemu argumentowi funkcji `ShellExecute()` odpowiednio „open”, „print”, „explore”.

3. Zmiana priorytetu bieżącej aplikacji

Funkcja WinAPI `SetPriorityClass()` umożliwia zmianę priorytetu działającej aplikacji – parametr niezwykle istotny w środowisku wielozadaniowym. Są cztery możliwe stany: niski priorytet, normalny (domyślny), wysoki i czasu rzeczywistego. Co zabawne w Windows 95/98/Millenium funkcja również istnieje i działa, choć z poziomu środowiska użytkownik nie ma żadnej możliwości odczytu i ustalenia priorytetów. W Windows NT/2000/XP odczyt możliwy jest za pomocą Menadżera zadań Windows (Ctrl+Alt+Del) jeżeli uwzględni się odpowiednią kolumnę w zakładce Procesy. Priorytet można w systemach NT ustalać przy uruchomieniu aplikacji z poziomu interpretatora poleceń komendą START z opcją /IDLE, /NORMAL, /HIGH lub /REALTIME, a po jej uruchomieniu priorytet można zmieniać za pomocą Menadżera zadań Windows (menu kontekstowe procesów na zakładce Procesy).

Jako pierwszy parametr wywołania funkcji `SetPriorityClass()` podaje się uchwyt do procesu, którego priorytet pragniemy zmienić (np. korzystając z funkcji `GetCurrentProcess()`). Drugim parametrem powinien być żądany priorytet procesu (predefiniowane stałe to `IDLE_PRIORITY_CLASS`, `NORMAL_PRIORITY_CLASS`, `HIGH_PRIORITY_CLASS`, `REALTIME_PRIORITY_CLASS`). Wartością funkcji jest true, jeżeli operacja się powiodła i false, jeżeli wystąpił błąd.
Poniżej znajdują się cztery metody zdarzeniowe ustawiające odpowiedni priorytet dla bieżącej aplikacji.




```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
SetPriorityClass(GetCurrentProcess(),IDLE_PRIORITY_CLASS);
}
//---------------------------------------------------------------------------
void __fastcall TForm1::Button2Click(TObject *Sender)
{
SetPriorityClass(GetCurrentProcess(),NORMAL_PRIORITY_CLASS);
}
//---------------------------------------------------------------------------
void __fastcall TForm1::Button3Click(TObject *Sender)
{
SetPriorityClass(GetCurrentProcess(),HIGH_PRIORITY_CLASS);
}
```

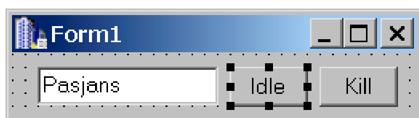
```
//---------------------------------------------------------------------------
void __fastcall TForm1::Button4Click(TObject *Sender)
{
SetPriorityClass(GetCurrentProcess(),REALTIME_PRIORITY_CLASS);
}
```

Możemy również odczytać priorytet procesu. Dodajmy do formy `Button5` oraz `Label1` i napiszmy metodę zdarzeniową:

```
void __fastcall TForm1::Button5Click(TObject *Sender)
{
switch (GetPriorityClass(GetCurrentProcess()))
      {
      case IDLE_PRIORITY_CLASS: Label1->Caption="Niski"; break;
      case NORMAL_PRIORITY_CLASS: Label1->Caption="Normalny"; break;
      case HIGH_PRIORITY_CLASS: Label1->Caption="Wysoki"; break;
      case REALTIME_PRIORITY_CLASS: Label1->Caption="Czasu rzecz."; break;
      default: Label1->Caption="Błąd!";
      }
}
```

4. Zmiana priorytetu i zamykanie innej działającej aplikacji

Funkcja `SetPriorityClass()` umożliwia zmianę priorytetów innych aplikacji. Jej argumentem jest identyfikator procesu, który musimy znaleźć, a proces przed zmianą priorytetu otworzyć[9]. Najłatwiej znaleźć identyfikator odnajdując uchwyt procesu związanego z oknem (funkcja WinAPI `FindWindow()`). Znając uchwyt można uzyskać identyfikator procesu funkcją `GetWindowThreadProcessId(hUchwyt, &ProcessId)`, gdzie pierwszym argumentem jest uchwyt do procesu, a z drugiego odczytamy jego identyfikator. Następnie otwieramy dostęp do procesu funkcją `OpenProcess()` z żądaniem dostępu PROCESS_SET_INFORMATION.



Zgodnie z zapowiedzią w poniższej metodzie identyfikujemy proces na podstawie nazwy okna, co umożliwia funkcja WinAPI `FindWindow()`. W tym celu pobieramy nazwę okna z okienka edycyjnego `Edit1`. Uchwyt uzyskany za pomocą funkcji `FindWindow()` przechowujemy w zmiennej typu `HWND` (zdefiniowanej w WinAPI) lub odpowiadającemu mu typowi `THandle`. Po sprawdzeniu, czy takie okno jest widoczne na ekranie, co w istocie nie jest konieczne, `IsWindow()` pobieramy jego numer identyfikacyjny, który pozwala nam otworzyć proces (jako żądany tryb dostępu należy podać `PROCESS_SET_INFORMATION` niezbędny przy modyfikacji priorytetu innej aplikacji). Teraz możemy ustalić jego priorytet. W przykładzie jest jedynie przełączenie jej w stan niskiego priorytetu, ale można go oczywiście rozbudować. Jeżeli zmiana się powiodła zamykany jest proces poleceniem `CloseHandle()` i metoda dobiega końca.

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
HWND hWnd = FindWindow(NULL,Edit1->Text.c_str());
if (IsWindow(hWnd))
      {
      DWORD ProcessId;
      GetWindowThreadProcessId(hWnd, &ProcessId);
      HANDLE hProc=OpenProcess(PROCESS_SET_INFORMATION, FALSE, ProcessId);
      if (hProc!=NULL)
            {
```

[9] W przypadku zmiany priorytetu bieżącej aplikacji otwieranie dostępu do procesu nie było oczywiście potrzebne.

```
            if (!SetPriorityClass(hProc,IDLE_PRIORITY_CLASS))
                  ShowMessage("Nie można zmienić priorytetu”);
            CloseHandle(hProc);
            }
      }
      else
      ShowMessage("Okno o podanej nazwie nie istnieje ("+Edit1->Text+")")
}
```

Zupełnie podobnie jak zmiana priorytetu działającej (ale nie bieżącej aplikacji) odbywa się jej zamykanie. Wykorzystuje się do tego funkcję WinAPI `TerminateProcess()`. Jej pierwszym argumentem jest identyfikator procesu (ten sam, którego używa się w funkcji zmieniającej priorytet), a drugim nie obsługiwany przez Windows kod zamknięcia programu[10]. Aby zmodyfikować powyższą metodę tak, żeby zamykała aplikację zidentyfikowaną na podstawie nazwy okna wystarczy jedynie zmodyfikować dwie linie:

```
void __fastcall TForm1::Button2Click(TObject *Sender)
{
HANDLE hWnd = FindWindow(NULL,Edit1->Text.c_str());
if (IsWindow(hWnd))
      {
      DWORD ProcessId;
      GetWindowThreadProcessId(hWnd, &ProcessId);
      HANDLE hProc = OpenProcess(PROCESS_TERMINATE, FALSE, ProcessId);
      if (hProc!=NULL)
            {
            if (!TerminateProcess(hProc,0))
                  ShowMessage("Nie można zabić ("+Edit1->Text+")");
            CloseHandle(hProc);
            }
      }
      else
      ShowMessage("Okno o podanej nazwie nie istnieje ("+Edit1->Text+")");
}
```

Należy zmienić żądane prawo dostępu na PROCESS_TERMINATE umożliwiające zakończenie procesu oraz użyć funkcji `TerminateProcess()`.

5. Lista okien

Aby skończyć krótki przegląd zagadnień dotyczących procesów i aplikacji napiszemy króciutki program wyświetlający listę okien wraz z nazwą klasy okna, jego ID w Windows i uchwytem. Umieścimy je w `StringGrid` z czterema kolumnami, bez kolumny nagłówka. `StringGrid` należy umieścić na formie, a następnie ustalić jego własności: `Align=alClient`, `ColCount=4`, `FixedCols=0`. Aby skończyć konfigurowanie siatki w metodzie `FormCreate` umieszczamy następujący kod:

```
//Ustawienia szerokości dwóch pierwszych kolumn
StringGrid1->ColWidths[0]=3*StringGrid1->DefaultColWidth;
StringGrid1->ColWidths[1]=2*StringGrid1->DefaultColWidth;

//Nagłówki kolumn
StringGrid1->Cells[0][0]="Tytuł okna";
StringGrid1->Cells[1][0]="Nazwa klasy okna";
StringGrid1->Cells[2][0]="ID okna";
StringGrid1->Cells[3][0]="Uchwyt";
```

---

[10] W najprostszym przypadku, gdy chcemy zamknąć bieżącą aplikację możemy napisać polecenie: `TerminateProcess(GetCurrentProcess(),0);`. Są jednak znacznie prostsze sposoby, żeby to zrobić.

```
//Dopasowywanie wielkości formy do siatki
Form1->ClientWidth=StringGrid1->GridWidth+3;
Form1->ClientHeight=StringGrid1->GridHeight+3;
```

Aby otrzymać listę okien posłużymy się funkcją WinAPI `EnumWindows()`. Jej działanie jest na pierwszy rzut oka dość niewygodne (choć typowe dla C++), bo zamiast zwracać np. tablicę uchwytów do okien, funkcja ta wykonuje pętle po wszystkich oknach wywołując zdefiniowaną przez nas funkcję zwrotną (ang. *callback function*) `BOOL CALLBACK EnumWindowsProc(HWND hwnd,LPARAM lParam)`, w której możemy umieścić dowolny kod. My umieścimy w niej kod dodający informacje o oknie do `StringGrid1`. Zdefiniujmy na początku prostą funkcję zwrotną:

```
BOOL CALLBACK EnumWindowsProc(HWND hwnd, LPARAM lParam)
{
const int MAX_CHAR_SIZE=128;
char tytul[MAX_CHAR_SIZE], nazwa_klasy[MAX_CHAR_SIZE];
unsigned long id_procesu;

GetWindowText(hwnd,tytul,MAX_CHAR_SIZE);
GetClassName (hwnd,nazwa_klasy,MAX_CHAR_SIZE);
GetWindowThreadProcessId(hwnd,&id_procesu);

static kolumna=1;
if (IsWindowVisible(hwnd))
      {
      Form1->StringGrid1->Cells[0][kolumna]=(AnsiString)tytul;
      Form1->StringGrid1->Cells[1][kolumna]=(AnsiString)nazwa_klasy;
      Form1->StringGrid1->Cells[2][kolumna]=(int)id_procesu;
      Form1->StringGrid1->Cells[3][kolumna]=(int)hwnd;
      kolumna++; Form1->StringGrid1->RowCount=kolumna;
      }

return true;
}
```

Jej działanie jest tak proste, że nie wymaga chyba większego komentarza. Funkcja ta będzie wywoływana przez `EnumWindows()` dla każdego okna osobno. Argumentami tej funkcji jest uchwyt do konkretnego okna oraz dowolna 32-bitowa liczba, którą możemy sterować działanie funkcji. Jak widać wywoływane są trzy funkcje WinAPI: `GetWindowText()` pobierająca nazwę okna (tekst na pasku), `GetClassName()` pobierająca nazwę klasy oraz `GetWindowThreadProcessId()` pobierająca jego ID. Korzystamy tu też z funkcji WinAPI `IsWindowVisible()`, która sprawdza, czy dane okno nie jest ukryte (wiele procesów, szczególnie systemowych korzysta z ukrytych okien, również aplikacja napisana w C++ Builderze tworzy zazwyczaj wszystkie okna od razu metodą CreateForm, choć natychmiast ich nie pokazuje[11]) Dla uproszczenia pominięto obsługę błędów, ale każda z tych funkcji zwraca wartość logiczną równą true, jeżeli jej działanie powiodło się, więc obsługę taką łatwo jest stworzyć.

Aby dokończyć dzieła należy wywołać dwuargumentową funkcję

```
BOOL EnumWindows(
    WNDENUMPROC lpEnumFunc,     //wskaźnik do funkcji zwrotnej
    LPARAM lParam               // dowolny parametr przekazywany do EnumWindowsProc()
   );
```

Pierwszym argumentem powinna być funkcja typu zdefiniowanego w winuser.h:

```
typedef BOOL (CALLBACK* WNDENUMPROC)(HWND, LPARAM);
```

[11] Można to zmienić modyfikując listę „Auto-create forms” w opcjach projektu.

a więc identycznego jak nasza funkcja EnumWindowsProc(). Jednak po wpisaniu na początku[12] `FormCreate()`:

```
EnumWindows(EnumWindowsProc,NULL);
```

pojawi się błąd informujący, że pierwszym argumentem powinien być wskaźnik do funkcji nie posiadającej argumentów. Rozwiązaniem jest zrzutowanie na typ `WNDENUMPROC`:

```
EnumWindows((WNDENUMPROC)EnumWindowsProc,NULL);
```

W wyniku otrzymamy listę okien postaci:

Form1

Tytuł okna	Nazwa klasy okna	ID okna	Uchwyt
	Shell_TrayWnd	284	196658
	AfxFrameOrView42s	944	66270
Project1	TApplication	2028	3474690
Form1	TForm1	836	2819358
E:\jacek\RAD4\rad4\WinAPI-C	TEditWindow	836	14222344
Object Inspector	TPropertyInspector	836	4129460
C++Builder - Project1 [Running] [	TAppBuilder	836	2687840
C++Builder	TApplication	836	1835838
Winamp Playlist Editor	Winamp PE	1948	328614
*** 4. Tangerine Dream - Phaedr	Winamp v1.x	1948	263088
Microsoft Word - rad4_api_cbuil	OpusApp	1496	2753844
Lister - [F:\Program Files\Borlar	TLister	992	1115378
Win32 Programmer's Reference	MS_WINDOC	1816	4064336
Znajdź Pliki	TFindFile	992	4064276
Windows Commander 5.0 - jm	TWINDOWSCMD	992	2622302
WPKontakt - Jacek	MessengerMainFram	940	459154
BCB Info - Serwis C++ Buildera [I	IEFrame	1320	1771100
	Client0	1336	197106
X-Win	XMainClass	552	393598
SystemShortcuts	TSCForm	1136	65698
Windows Commander 5.0 - jm	TWINDOWSCMD	848	263048
berkel	XWinClass	552	2163698
KaZaA - [Traffic]	KaZaA	1336	655686
Program Manager	Progman	284	65624

Na liście wymienione są także okna ukryte związane m.in. z zasobnikiem systemowym (miejsce na pasku zadań, w którym pojawia się godzina i ikonki niektórych programów).

Zadanie
Wyświetlić listę wszystkich okien (pominąć zastosowanie funkcji `IsWindowVisible`).

Zadanie
Włączyć do programu możliwość zmiany priorytetu i „zabicia" okna/procesu z poprzednich paragrafów.

---

[12] Funkcję `EnumWindows` należy wywołać przed ustaleniem wysokości formy, gdyż zmienia ona ilość wierszy w `StringGrid1`, a do niej dopasowana jest wysokość formy.

IV. Odczytywanie informacji o dysku

Celem tego rozdziału jest napisanie klasy prezentującej w postaci graficznej ilość wolnego miejsca na dysku oraz udostępniającej informacje o typie i parametrach tego dysku. Do odczytu ilości wolnego miejsca użyjemy funkcji WinAPI ze względu na błędne pokazywanie parametrów dysków większych od 2GB przez standardowe funkcje C++ (przynajmniej w obecnej jego wersji).

Komponent napiszemy w trzech etapach, podobnie jak robi się to w praktyce. Skupiając się wpierw na wykorzystywanych funkcjach WinAPI napiszemy funkcję pobierającą potrzebne informacje. Następnie zbudujemy klasę udostępniającą informacje o dyskach i wreszcie przekształcimy ją w komponent.

1. Odczytywanie parametrów dysku

Pierwszym krokiem będzie napisanie funkcji `getdiskinfo()`, którą później przekształcimy w główną metodę komponentu. Funkcja ta, korzystając z funkcji WinAPI, będzie odczytywać parametry fizyczne i ilość zajętego miejsca na dysku. Jej pierwszym argumentem funkcji będzie litera dysku, drugim wskaźnik do struktury, do której funkcja zapisze informacje o dysku. Ową strukturę, nazwijmy ją `_disk_info`, musimy oczywiście zdefiniować sami:

```
struct _disk_info
      {
      char disk_letter;
      bool disk_accesible;

      unsigned int disk_type;
      char disk_type_str[DI_MAX_NAMELENGTH];

      unsigned long total_kb;
      unsigned long free_kb;

      double free_fraction;
      unsigned int free_percentage;

      char volume_name[DI_MAX_NAMELENGTH];
      unsigned long volume_SN;
      char FAT_name[DI_MAX_NAMELENGTH];
      unsigned long max_filedir_name;

      unsigned long max_path;
      };
```

Stałą DI_MAX_NAMELENGTH ustalmy na:

```
const int DI_MAX_NAMELENGTH=255;
```

(musi znajdować się przed definicją struktury).

Uwaga!
Struktura zawiera pozycje przechowujące ilość wolnego i ilość całkowitego miejsca na dysku mierzoną w kilobajtach. Dzięki temu zabiegowi można wykorzystać standardowy typ C++ `unsigned long`. Jeżeli zależy nam na dokładnej ilości bajtów możemy zastąpić te elementy struktury przez zmienne typu `DWORDLONG`. `DWORDLONG` jest 64 bitowym typem pochodzącym z WinAPI.

Zdefiniowana powyżej struktura i stałą umieszczona powinna być w pliku nagłówkowym diskinfo.h, a następującą funkcję umieśćmy w pliku diskinfo.cpp:

```
#include "diskinfo.h"
```

```
bool getdiskinfo(char drvletter,_disk_info& disk_info)
{
//Ustalanie wstepnych wartosci
disk_info.disk_letter=toupper(drvletter);
disk_info.disk_accesible=true;
disk_info.disk_type=0;
strcpy(disk_info.disk_type_str,"");
disk_info.total_kb=0;
disk_info.free_kb=0;
disk_info.free_percentage=0;
strcpy(disk_info.volume_name,"");
disk_info.volume_SN=0;
strcpy(disk_info.FAT_name,"");
disk_info.max_filedir_name=0;
disk_info.max_path=0;

//Sciezka katalogu glownego na dysku
char drvrootstr[4]=" :\\\0"; //4 znaki bo na koncu musi byc \0
drvrootstr[0]=disk_info.disk_letter;

//Typ napedu
disk_info.disk_type=GetDriveType(drvrootstr);
switch(disk_info.disk_type)
      {
      case 0:
            strcpy(disk_info.disk_type_str,"Napęd nie istnieje");
            disk_info.disk_accesible=false;
            break;
    case 1:
            strcpy(disk_info.disk_type_str,"Dysk nie jest sformatowany");
            disk_info.disk_accesible=false;
            break;
    case DRIVE_REMOVABLE:
            strcpy(disk_info.disk_type_str,"Dysk wymienny");
            break;
    case DRIVE_FIXED:
            strcpy(disk_info.disk_type_str,"Dysk lokalny");
            break;
    case DRIVE_REMOTE:
            strcpy(disk_info.disk_type_str,"Dysk sieciowy");
            break;
    case DRIVE_CDROM:
            strcpy(disk_info.disk_type_str,"Płyta CDROM");
            break;
    case DRIVE_RAMDISK:
            strcpy(disk_info.disk_type_str,"RAM Drive");
            break;
    default:
            strcpy(disk_info.disk_type_str,"Typ dysku nierozpoznany");
    }

//Jezeli dysk niedostepny, to konczymy
if (!disk_info.disk_accesible) return false;

//Ilosc wolnego miejsca na dysku
ULARGE_INTEGER TotalBytes; //to sa unie
ULARGE_INTEGER FreeBytes;
bool Result=::GetDiskFreeSpaceEx(drvrootstr,NULL,&TotalBytes,&FreeBytes);
disk_info.total_kb=TotalBytes.QuadPart/1024;
disk_info.free_kb=FreeBytes.QuadPart/1024;
```

```
if (Result && disk_info.total_kb!=0)
    {
    disk_info.free_fraction=disk_info.free_kb/(double)disk_info.total_kb;
    disk_info.free_percentage=(unsigned int)(100*disk_info.free_fraction);
    }
    else
    {
    disk_info.free_fraction=0;
    disk_info.free_percentage=0;
    disk_info.disk_accesible=false;
    Values=disk_info;
    return false;
    }


//Nazwa dysku, typ FAT, numer seryjny
GetVolumeInformation(
      drvrootstr,
      disk_info.volume_name,
      sizeof(disk_info.volume_name),
      &disk_info.volume_SN,
      &disk_info.max_filedir_name,
      NULL,
      disk_info.FAT_name,
      sizeof(disk_info.FAT_name));

disk_info.max_path=MAX_PATH;

return Result;
}
```

Omówmy po kolei użyte funkcje WinAPI:

1) `GetDriveType()` – jej wartością jest liczba naturalna określająca typ napędu. Zdefiniowane stałe kodujące poszczególne typy napędów wymienione i opisane są w instrukcji `switch`, następującej po wywołaniu tej funkcji. Nierozpoznawane są typy stacji dyskietek[13]. Jedynym argumentem funkcji jest wskaźnik do łańcucha zawierającego ścieżkę do katalogu głównego dysku w badanym napędzie.
2) `GetDiskFreeSpaceEx()` – funkcja dostępna dopiero od wersji systemu Windows 95 OSR2[14], w którym obsługiwane są dyski większe od 2GB[15]. Wcześniejsza wersja funkcji `GetDiskFreeSpace()` zwraca niepoprawne wartości dla dużych dysków. Pierwszym argumentem, podobnie jak 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` to nie liczba, lecz unia. Jeżeli w kompilatorze nie istnieje typ liczby całkowitej 64-bitowej, zawiera ona dwie liczby 32-bitowe reprezentujące liczbę 64-bitową. W przeciwnym przypadku (i tak jest w C++ Builderze już od wersji 1.0) zawiera ona jeden element 64-bitowy o nazwie `QuadPart`. Aby uprościć dalszą arytmetykę przechowujemy w strukturze ilość miejsca na dysku w kilobajtach.
   Uwaga! Pusty namespace został dodany przy tej funkcji ze względu na zdublowanie tej funkcji przez `Sysutils::GetDiskFreeSpaceEx()` w nowszych wersjach kompilatora

---

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

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

[15] 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 (ze znakiem). Konieczne więc jest zwracanie liczb 64-bitowych. I tu należy uważać, żeby przypadkowo nie zrobić konwersji na liczby 32-bitowe przez przypisywanie lub operacje na zmiennych.

3) GetVolumeInformation() – funkcja zwracająca informacje o systemie plików na dysku (nazwa dysku, numer seryjny, maksymalną długość nazwy katalogu lub pliku, typ FAT, informacje o kompresji itp.). Dane pobieramy bezpośrednio do elementów struktury _disk_info.
4) stała MAX_PATH przechowuje maksymalną długość ścieżki do pliku włączając jego nazwę z rozszerzeniem.

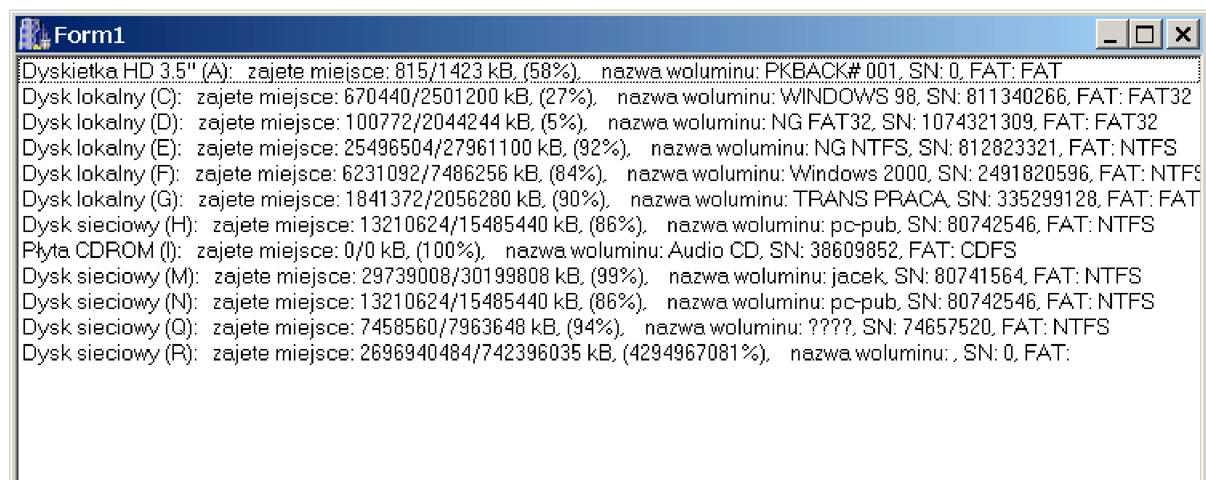
Funkcja pobiera dwa argumenty: litera oznaczająca dysk oraz referencja do struktury, do której zapisana będzie informacja o dysku. Referencja powoduje, że funkcję wywołujemy tak jakbyśmy przekazywali jako argument samą strukturę.
Aby sprawdzić działanie funkcji zaprojektujmy formę z jednym ListBoxem, w którym będziemy umieszczać informacje o dyskach lokalnych i sieciowych. W metodzie FormCreate umieśćmy pętlę po literach dysków od a do z:

```
for(char litera='a';litera<='z';litera++)
      {
      _disk_info disk_info;
      getdiskinfo(litera,disk_info);
      if (disk_info.disk_accesible)
            ListBox1->Items->Add(

//Lancuch dodawany do ListBox1
(AnsiString)disk_info.disk_type_str+" ("+
(AnsiString)disk_info.disk_letter+"):   "+
"zajete miejsce: "+
(AnsiString)(double)(disk_info.total_kb-disk_info.free_kb)+
"/"+(AnsiString)(double)(disk_info.total_kb)+" kB, "+
"("+(AnsiString)(double)(100-disk_info.free_percentage)+"%),    "+
"nazwa woluminu: "+(AnsiString)disk_info.volume_name+
", SN: "+(AnsiString)(double)disk_info.volume_SN+
", FAT: "+(AnsiString)disk_info.FAT_name

                  );
    }
```

Aby wywołanie funkcji getdiskinfo() było możliwe musimy uwzględnić plik, w którym się ona znajduje. Na górze pliku Unit1.cpp dopiszmy wobec tego:

```
#include "diskinfo.cpp"
```

Efekt działania aplikacji jest następujący (wielkości dla dysków sieciowych mogą być podane błędnie, bo błędnie odczytuje je także sam system – dotyczy to w szczególności protokołu Samba):



2. Klasa pobierająca i udostępniający dane o dysku

Kolejnym etapem jest zbudowanie klasy `TDiskInfo`, której metoda `GetDiskInfo` będzie kopią powyższej funkcji i posiadającą własność `Value` typu `_disk_info` (zdefiniowana w poprzednim paragrafie struktura przechowująca informacje o dysku). Deklaracja klasy powinna więc być następująca:

```
class TDiskInfo
{
private:
      bool GetDiskInfo(char);
public:
      TDiskInfo(char); //konstruktor klasy
      _disk_info Values; //struktura przechowujaca informacje o dysku
};
```

Widać, że obiekt nie posiada konstruktora domyślnego (bezargumentowego), gdyż zadeklarowaliśmy konstruktor, którego argumentem jest nazwa dysku. Umieśćmy tę deklarację w nowym pliku utworzonym przez edytor: z menu kontekstowego okna edycyjnego (należy prawym klawiszem myszy kliknąć na zakładki u góry okna edycyjnego) wybieramy „Open File at Cursor", a następnie „Create a new Unit". Powstanie plik nagłówkowy (.h), do którego wstawimy powyższą deklarację oraz kopię definicji struktury z poprzedniego rozdziału oraz plik źródłowy (.cpp). W tym pliku umieszczamy konstruktor, którego zadaniem jest wywołanie prywatnej metody `GetDiskInfo()`:

```
TDiskInfo::TDiskInfo(char drvletter)
      {
      GetDiskInfo(drvletter);
      }
```

Teraz kolej na metodę wywoływaną metodę. Jest ona w istocie wierną kopią napisanej wyżej funkcji, z tym, że została opakowana – zamiast podawać strukturę na dane przez referencję, informacje o dyskach będą umieszczane w własności (*data member*) `Values`. Aby zbytnio nie modyfikować funkcji deklarujemy na jej początku zmienną o nazwie identycznej jak dawny argument:

```
_disk_info disk_info;
```

a przed jej zakończeniem (przed każdym poleceniem return) dodajemy przypisanie:

```
Values=disk_info;
```

Pozostała część skopiowanej funkcji pozostaje bez zmian.

Aby sprawdzić działanie klasy musimy nieco zmodyfikować pętlę umieszczoną w metodzie `FormCreate` (tym razem ograniczymy ilość wyświetlanej informacji o dysku):

```
for(char litera='a';litera<='z';litera++)
      {
      TDiskInfo* DiskInfo=new TDiskInfo(litera);

      if (DiskInfo->Values.disk_accesible)
            ListBox1->Items->Add(

(AnsiString)DiskInfo->Values.disk_type_str+" ("+
(AnsiString)DiskInfo->Values.disk_letter+"):   "+
(AnsiString)DiskInfo->Values.volume_name+"  ("+
(AnsiString)(double)(100-DiskInfo->Values.free_percentage)+"%)"+"   "+
(AnsiString)DiskInfo->Values.FAT_name);

      delete DiskInfo;
      }
```

Zamiast deklaracji struktury i wywołania funkcji `getdiskinfo()` teraz tworzony jest za pomocą operatora `new` obiekt klasy `TDiskInfo`. Ponieważ metoda `GetDiskInfo()` wywoływana jest przez konstruktor obiekt ten zaraz po utworzeniu zawiera informacje nt. dysku o literze podanej przy tworzeniu obiektu. Po wyświetleniu informacji należy pamiętać o usunięciu obiektu.

Przez to, że metoda `GetDiskInfo()` jest prywatna – nie można jej wywołać po stworzeniu obiektu np. w celu zapisania do `Values` informacji o innym dysku. Nie stanowi to jednak problemu - można dopisać odpowiednią metodę publiczną wywołującą `GetDiskInfo()`. Pisząc komponent rozwiążemy ten problem bardziej elegancko. Udostępnimy upublicznioną (`__published`) właściwość[16] `DriveLetter`.

3. Komponent TDiskInfoPanel

Każdy komponent musi dziedziczyć z klasy `TComponent` lub jego pochodnych. Stwórzmy komponent, który będzie pokazywał ilość wolnego miejsca, informację o dysku i typ FAT. Wygodnie będzie więc, jeśli komponent będzie dziedziczył z komponentu `TCustomPanel`. Na panelu poukładamy pozostałe obiekty składowe komponentu (pasek pokazujący ilość wolnego miejsca i napisy). Wzbogacona deklaracja klasy powinna więc być następująca:

```
class TDiskInfoPanel : public TCustomPanel
{
private:
      bool GetDiskInfo(char);

      TProgressBar* ProgressBar;
      TLabel* LewyOpis;
      TLabel* PrawyOpis;
public:
      __fastcall TDiskInfoPanel(Classes::TComponent* AOwner);
      _disk_info Values;
__published:
};
```

Dodaliśmy dziedziczenie z klasy `TCustomPanel` w pierwszej linii (co zmusza nas do uwzględnienia odpowiedniego modułu `ExtCtrls`) oraz dwa wskaźniki `TLabel` i jeden do `TProgressBar` (ten z kolei wymaga dodania `ComCtrls` do `uses`). Aby obiekt mógł być dodany do klas VCL musi posiadać konstruktor, którego jedynym argumentem jest właściciel typu `TComponent`. Tylko wówczas IDE Buildera będzie umiało obsłużyć zrzucenie go na formę. Modyfikujemy wobec tego deklarację konstruktora, ale to powoduje, że musimy do obiektu dodać mechanizm wskazywania na literę dysku który nas interesuje. Dodajmy więc zapowiadaną wyżej właściwość `DriveLetter` do sekcji `published` klasy:

```
__property char DriveLetter={write=SetDriveLetter, read=ADriveLetter};
```

Właściwości i zdarzenia z sekcji opublikowanej znajdą się w Object Inspectorze po zainstalowaniu komponentu. Próba czytania `DriveLetter` spowoduje podanie wartości ze zmiennej `ADriveLetter` (typu `char`), a ustalanie wartości ma spowodować wywołanie funkcji `SetDriveLetter(char)` która, jak łatwo się domyślić wywoła `GetDiskInfo` z nową literą dysku jako argumentem. Po uzupełnieniu deklaracja klasy wygląda następująco:

```
class TDiskInfoPanel : public TCustomPanel
{
private:
      bool GetDiskInfo(char);
```

---

[16] Ustalmy na potrzeby tego paragrafu: własność (*data member*) = zmienna wewnątrz obiektu, właściwość (*property*) = specjalna konstrukcja Delphi/C++ Buildera zadeklarowana ze słowem kluczowym `property` będąca pomostem między własnościami i metodami. Zobacz plik informacje w części RAD1..

```
        char ADriveLetter;
        void __fastcall SetDriveLetter(char);
        TProgressBar* ProgressBar;
        TLabel* LewyOpis;
        TLabel* PrawyOpis;
public:
        __fastcall TDiskInfoPanel(Classes::TComponent* AOwner);
        _disk_info Values;
__published:
        __property char DriveLetter={write=SetDriveLetter, read=ADriveLetter};
};
```

Konieczna jest modyfikacja konstruktora (ponieważ konstruktor piszemy sami, musimy koniecznie wywołać samodzielnie konstruktor klasy bazowej – konstruktor domyślny robiłby to automatycznie) oraz napisać funkcję `SetDriveLetter()`.

Nowy konstruktor nie wywołuje już funkcji pobierającej informacje o dysku. W zamian powołuje obiekty związane ze wskaźnikami `ProgressBar`, `LewyOpis` i `PrawyOpis`, ustala wielkość i własności panelu oraz stworzonego paska postępu i napisów, ustala ponadto domyślną literę dysku na C, a przede wszystkim wywołuje konstruktor metody bazowej (jeszcze przed ciałem metody):

```
__fastcall TDiskInfoPanel::TDiskInfoPanel(Classes::TComponent*
AOwner):TCustomPanel(AOwner)
{
Caption="";
BevelInner=bvNone;
BevelOuter=bvNone;
Width=640-30;

ProgressBar=new TProgressBar(this);
ProgressBar->Height=Height/2;
ProgressBar->Width=Width;
ProgressBar->Left=0;
ProgressBar->Top=Height/2;
ProgressBar->Parent=this;

LewyOpis=new TLabel(this);
LewyOpis->Left=0;
LewyOpis->Top=0;
LewyOpis->Parent=this;

PrawyOpis=new TLabel(this);
PrawyOpis->Left=0; //tymczasowo
PrawyOpis->Top=0;
PrawyOpis->Parent=this;

DriveLetter='C';
}
```

Metoda `SetDriveLetter()` jest niezwykle prosta:

```
void __fastcall TDiskInfoPanel::SetDriveLetter(char drvletter)
{
ADriveLetter=toupper(drvletter);
GetDiskInfo(drvletter);

ProgressBar->Position=100-Values.free_percentage;
LewyOpis->Caption=(AnsiString)DriveLetter+":    "+
                  Values.volume_name+"    ("+Values.FAT_name+")";
PrawyOpis->Caption=
```

```
    (AnsiString)(double)((Values.total_kb-Values.free_kb)/1024)+
    "MB / "+(AnsiString)(double)(Values.total_kb/1024)+"MB
    ("+(AnsiString)(double)(100-Values.free_percentage)+"%)";
PrawyOpis->Left=ProgressBar->Left+ProgressBar->Width-PrawyOpis->Width;
}
```

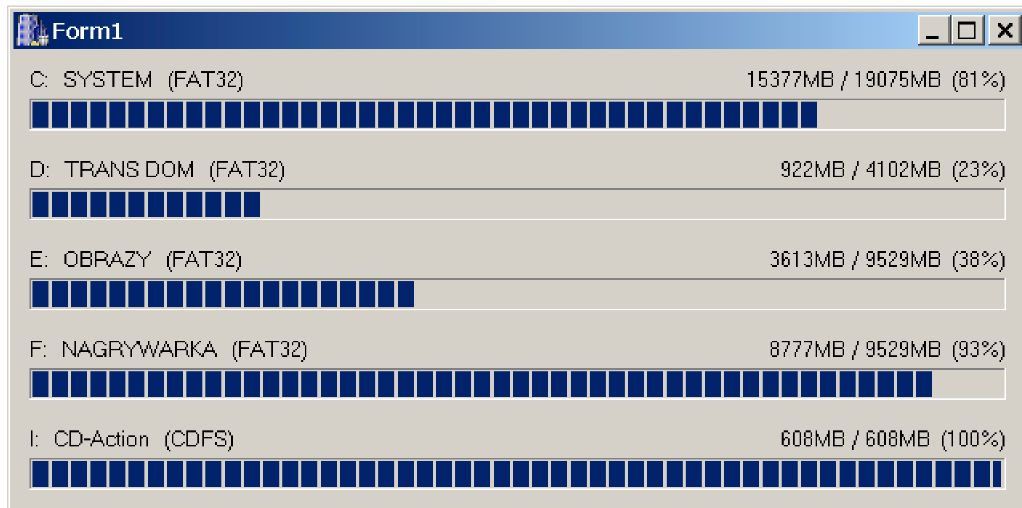
Po ustaleniu wartości zmiennej `ADriveLetter` pobierana jest informacja o wskazanym dysku za pomocą naszej metody `GetDiskInfo()`. Informacje zapisywane są oczywiście do własności `Values` i następnie wykorzystywane są do ustalenia wyglądu paska postępu tak, żeby pokazywał ilość zajętego miejsca na dysku i napisów (`TLabel LewyOpis`, `PrawyOpis`) tak, żeby pokazywały literę dysku, jego nazwę i typ FAT oraz ilość wolnych kilobajtów na dysku. Ustalana jest również pozycja prawego napisu tak, żeby przylegała do prawej strony panelu.

Ustalenie domyślnej litery dysku w konstruktorze (`DriveLetter='C';)` jest więc w istocie pośrednim wywołaniem `GetDiskInfo('C')`.

Forma, która testuje nasz obiekt może mieć następującą metodę:

```
void __fastcall TForm1::FormCreate(TObject *Sender)
{
const int drvletterno='z'-'a'+1;
TDiskInfoPanel* DiskInfoPanel[drvletterno]; //tablica wskaznikow
int polozenie_ostatniego=0;
const int margines=10;
for(char litera='a';litera<='z';litera++)
    {
    static int n=0; //numer kolejny odpowiadajacy literze
    static int drvno=0; //ilosc istniejacych fizycznie dyskow
    DiskInfoPanel[n]=new TDiskInfoPanel(this);
    DiskInfoPanel[n]->Left=margines;
    DiskInfoPanel[n]->DriveLetter=litera;
    if (DiskInfoPanel[n]->Values.disk_accesible)
        {
        DiskInfoPanel[n]->Parent=this;
        DiskInfoPanel[n]->Top=
            margines+drvno*(DiskInfoPanel[n]->Height+1.5*margines);
        polozenie_ostatniego=
            DiskInfoPanel[n]->Top+DiskInfoPanel[n]->Height;
        drvno++;
        }
        else
        {
        delete DiskInfoPanel[n];
        DiskInfoPanel[n]=NULL;
        }
    }
ClientHeight=polozenie_ostatniego+margines;
}
```

Powyższa metoda powoduje umieszczenie obiektów TDiskInfoPanel z informacjami o dyskach na formie:



Naszym głównym zadaniem jest stworzenie komponentu. Musimy dodać funkcję sprawdzającą, czy jest zarejestrowany komponent o tej samej nazwie i funkcję rejestrującą komponent (zob. opis w RAD1):

```
//Rejestracja komponentu w C++ Builder 1
static inline TDiskInfoPanel *ValidCtrCheck()
{
return new TDiskInfoPanel(NULL);
}

namespace Diskinfo
{
void __fastcall Register()
      {
      TComponentClass classes[1] = {__classid(TDiskInfoPanel)};
      RegisterComponents("JM", classes, 0);
      }
}
```

W nowszych wersjach C++ Buildera należy nieco zmodyfikować funkcję rejestrującą komponent i jego deklarację tak, żeby współpracował z mechanizmem pakietów stosowanym np. w C++ Builder 5. Do deklaracji komponentu w pliku nagłówkowym należy dodać makro `PACKAGE`:

```
class PACKAGE TDiskInfoPanel : public TCustomPanel
{
```

Tym samym makrem należy poprzedzić funkcję rejestrującą:

```
//Rejestracja komponentu w C++ Builder 5
namespace Diskinfo
{
void __fastcall PACKAGE Register()
      {
      TComponentClass classes[1] = {__classid(TDiskInfoPanel)};
      RegisterComponents("JM", classes, 0);
      }
}
```

Musimy teraz zarejestrować obiekt `TDiskInfoPanel` z pliku diskinfo.cpp. W tym celu z menu Component wybieramy pozycję Install ... i wskazujemy na nasz plik.

Jeżeli wszystko poszło dobrze w palecie komponentów pojawił się komponent `TDiskInfoPanel`, który można zrzucić na formę. Poza typowymi dla `TCustomPanel` własnościami w inspektorze obiektów znajduje się własność `DriveLetter`, której zmiana powoduje automatycznie zmianę wyświetlanych informacji na komponencie.

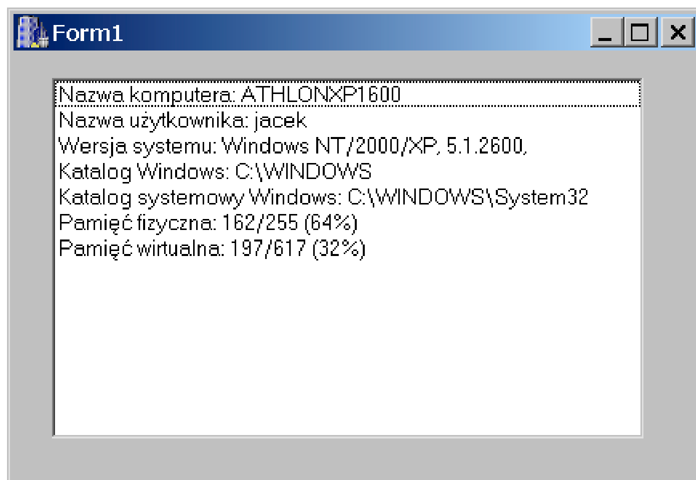
Zadanie
Dodać obsługę zdarzenia `OnResize` do obiektu `TDiskInfoPanel`.

4. Komponent niewidoczny TDiskInfo

Komponent oparty na `TCustomPanel` jest na pewno wygodny i efektowny, ale przydać się może również analogiczny komponent niewidoczny, który udostępniając informacje o dysku nie dostarczałby żadnego interface'u. Można oczywiście ustawić własność `Visible=false`, ale warto nauczyć się też przygotowania komponentów niewidocznych. Nie kryje się tu żadna trudność, należy po prostu dziedziczyć z komponentu niewidocznego, a najlepiej z `TComponent`.

Zmodyfikujmy komponent tak, żeby dziedziczył z klasy `TComponent` (w deklaracji klasy zastępujemy `: public TCustomPanel` przez `: public TComponent`). Podobnie zmieniamy wywołanie konstruktora klasy bazowej w konstruktorze naszej klasy. Ponieważ nazwa klasy `TDiskInfoPanel` traci sens możemy wrócić do `TDiskInfo` (należy zamienić każde występowanie starej nazwy przez nową, pamiętając, że należy zmienić również nazwę konstruktora i nazwę rejestrowanej w środowisku IDE klasy).
Następnie usuwamy z komponentu deklaracje, tworzenie i wszystkie odwołania do `ProgressBar`, `LewyOpis` i `PrawyOpis`. Funkcja `GetDiskInfo()` pozostaje oczywiście niezmieniona. Odpowiedni projekt można znaleźć w dołączonych do skryptu źródłach.

Zadanie
Zbuduj, analogicznie do komponentu w powyższym rozdziale[17], niewidoczny komponent `TWindowsInfo` dostarczający informacje dotyczące Windows. Należy wykorzystać funkcje WinAPI: `GetComputerName()`, `GetUserName()`, `GetSystemInfo()`, `SystemParametersInfo()`, `GetVersionEx()`,`GetKeyboardType()`, `GetWindowsDirectory()`, `GetSystemDirectory()`, `GetSystemDefaultLangID()`, `GetEnviromentVariable()` itp.



[17] Dla odmiany należy skorzystać z możliwości tworzenia szkieletu komponentu udostępnionego w C++ Builderze/Delphi (menu Component, New ..., typ pierwotny `TComponent`).

V. Kontrola okien/form[18]

Okno jest bardzo dobrze obudowane w środowisku Delphi/C++ Builder klasą TForm. Z oknem można zrobić jednak jeszcze więcej i pozwalają na to funkcje WinAPI.

1. Kontrola okna/formy

Jedną z typowych własności okna jest pozostawianie zawsze na wierzchu, nawet gdy okno nie ma „focusu". Ustalenie takiej własności okna ma sens np. wtedy gdy okno podaje istotną dla działania aplikacji lub systemu informację. Można zastosować własność obiektu `TForm->FormStyle` i napisać kod:

```
void __fastcall TForm1::CheckBox1Click(TObject *Sender)
{
//Przelaczenie powoduje mrugniecie spowodowane odmalowaniem formy
//i wszystkich komponentow wizualnych - im ich wiecej tym bardziej
//widoczne, poza tym klopoty z niektorymi komponentami
if (CheckBox1->Checked)
      Form1->FormStyle=fsStayOnTop;
      else
      Form1->FormStyle=fsNormal;
}
```

Ma on jednak taką wadę, że po zmianie stylu formy zostaje ona każdorazowo „odmalowana". Tracimy więc zawartość `Canvas` i musimy oglądać mrugnięcie widoczne tym bardziej im więcej obiektów jest na formie osadzonych. Co gorsze powoduje to wadliwe działanie niektórych z nich (pomoc odradza manipulowanie tą własnością w czasie działania aplikacji).

W zamian można skorzystać z funkcji WinAPI `SetWindowPos()`:

```
void __fastcall TForm1::CheckBox2Click(TObject *Sender)
{
if (CheckBox2->Checked)
      SetWindowPos(Form1->Handle,HWND_TOPMOST,
            Form1->Left,Form1->Top,Form1->Width,Form1->Height,
            SWP_NOMOVE | SWP_NOSIZE);
    else
    SetWindowPos(Form1->Handle,HWND_NOTOPMOST,
            Form1->Left,Form1->Top,Form1->Width,Form1->Height,
            SWP_NOMOVE | SWP_NOSIZE);
}
```

Jej argumenty to kolejno: uchwyt, styl okna, położenie oraz opcje związane z położeniem.

Funkcje WinAPI pozwalają oczywiście na znacznie więcej. Można na przykład ukryć okno formy (równoznaczne z `Form->Visible=false`):

```
ShowWindow(Form1->Handle,SW_HIDE);
```

lub okno aplikacji, które nie jest co prawda widoczne, ale to z nim związana jest informacja na pasku zadań:

---

[18] Przyznaję, że nie rozumiem dlaczego okna przezwane zostało formami. Prawdopodobnie dlatego, żeby uzgodnić nazwę tej klasy z konkurencyjnym Visual Basiciem (wg. Autorów *Delphi. Vademecum profesjonalisty* pierwotna nazwa tej klasy brzmiała TWindow). W każdym razie nazw okno i forma będę używał tutaj zamiennie.

```
ShowWindow(Application->Handle,SW_HIDE);
```

Drugie polecenie powoduje więc ukrycie aplikacji na pasku zadań. Odwrotne działanie ma:

```
ShowWindow(Application->Handle,SW_SHOW);
```

które przywraca aplikację na pasku zadań. Możliwe jest również mignięcie na pasku zadań, co ma zwrócić uwagę użytkownika na zmianę w aplikacji:

```
FlashWindow(Application->Handle,true);
```

Tą samą funkcję można wywołać z uchwytem formy (`Form1->Handle`), co spowoduje mrugnięciem paska tytułu.

2. Okienka dialogowe

WinAPI dostarcza funkcję wyświetlającą okienka dialogowe `MessageBox()`. Jej składnia jest prosta. Pierwszym argumentem jest uchwyt okna, którego nowowyświetlone okno będzie oknem modalnym. Drugim jest tekst informacji/pytania. Trzecim – tytuł okna i czwartym parametry określające ikonę i pojawiające się na oknie dialogowym przyciski. Funkcja zwraca liczbę odpowiadającą naciśniętemu przyciskowi. Najłatwiej jest chyba zrozumieć działanie tej funkcji na przykładzie:

```
void __fastcall TForm1::Button3Click(TObject *Sender)
{
MessageBox(Handle,"Informacja - Za chwilę pojawi się pytanie.",
            "Tytuł okna - Informacja",MB_OK | MB_ICONINFORMATION);
if (MessageBox(Handle,"Pytanie - Czy pokazać następne okno?",
            "Tytuł okna - Pytanie",MB_YESNO | MB_ICONQUESTION)==IDYES)
      switch(MessageBox(Handle,"Pytanie – Wybierz dowolny klawisz?",
            "Tytuł okna - Pytanie",MB_ABORTRETRYIGNORE | MB_ICONWARNING))
      {
      case ID_ABORT: ShowMessage("Nacisnąłeś \"Przerwij\""); break;
      case ID_RETRY: ShowMessage("Nacisnąłeś \"Ponów próbę\""); break;
      case ID_IGNORE: ShowMessage("Nacisnąłeś \"Ignoruj\""); break;
      }
}
```

„Opakowanie" do tej funkcji jest jedną z metod obiektu `Application->MessageBox`. Pominięto w niej pierwszy argument, którym jest domyślnie uchwyt do ukrytego okna aplikacji `Application->Handle`.

Ponadto WinAPI dostarcza funkcji `MessageBeep()`, która powoduje odtworzenie odpowiedniego dźwięku systemowego np.:

```
MessageBeep(MB_ICONEXCLAMATION);
```

3. Okna o dowolnym kształcie

WinAPI dopuszcza filtrowanie rysowanej na ekranie części okna. W praktyce oznacza to możliwość niemal dowolnego wyboru kształtu okna. Istnieje kilka funkcji pozwalających na definiowanie pola użytkownika o kształcie prostokątu (`CreateRectRgn()`), zaokrąglonego prostokątu (`CreateRoundRectRgn()`), elipsy (`CreateEllipticRgn()`) i dowolnego wielokąta (`CreatePolygonRgn()`). Możliwe jest łączenie zdefiniowanych pól (`CombineRgn()`) i w końcu ich użycie na formie (`SetWindowRgn()`). Składnia funkcji jest bardzo prosta.

Należy najpierw utworzyć odpowiedni region (w poniższym przykładzie – elipsę) i uchwyt do niego zapisać w zmiennej typu HRGN:

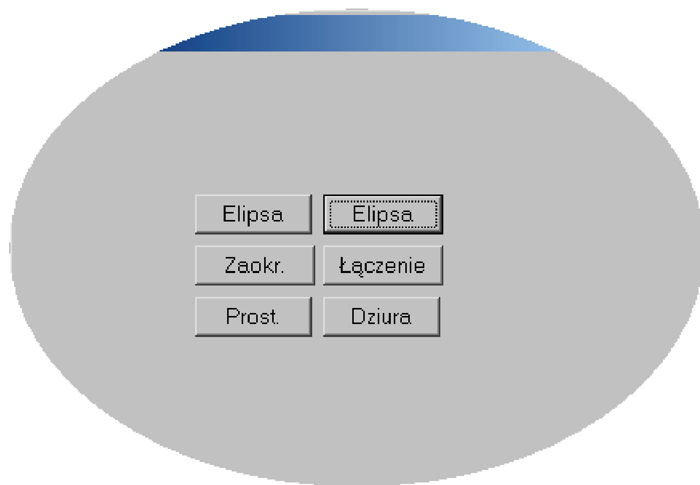
```
HRGN Rgn;
Rgn = CreateEllipticRgn(0, 0, Width, Height);
```

Funkcja `CreateEllipticRgn()` pobiera położenie i rozmiary regionu a zwraca uchwyt do tego regionu. Uchwyt ten może być argumentem funkcji `SetWindowRgn()`, której argumenty to uchwyt do okna, w którym ma być zastosowany wybrany region, uchwyt do regionu oraz wartość logiczna decydująca, czy okno ma być odmalowane. W naszym przypadku wywołanie tej funkcji może być następujące:

```
SetWindowRgn(Handle, Rgn, true);
```

Całość można skrócić do jednej linii:

```
SetWindowRgn(Handle, CreateEllipticRgn(0, 0, Width, Height), true);
```



Inne możliwe kształty regionów to prostokąt z zaokrąglonymi kątami:

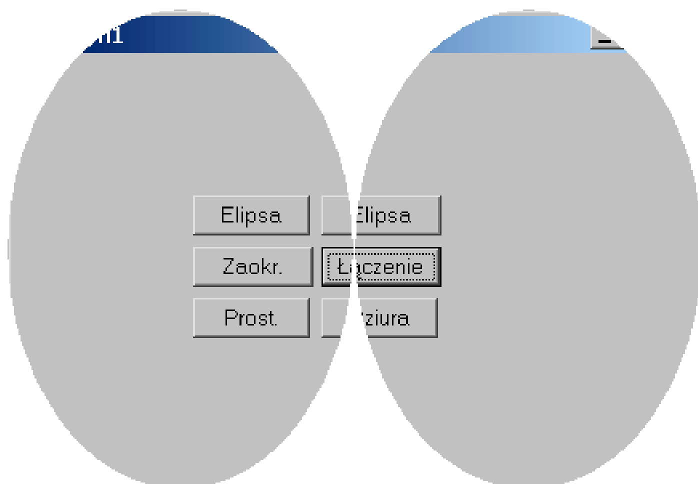
```
SetWindowRgn(Handle, CreateRoundRectRgn(0, 0, Width, Height, 200, 100),
      true);
```

zwyczajny prostokąt:

```
SetWindowRgn(Handle, CreateRectRgn(0, 0, Width, Height), true);
```

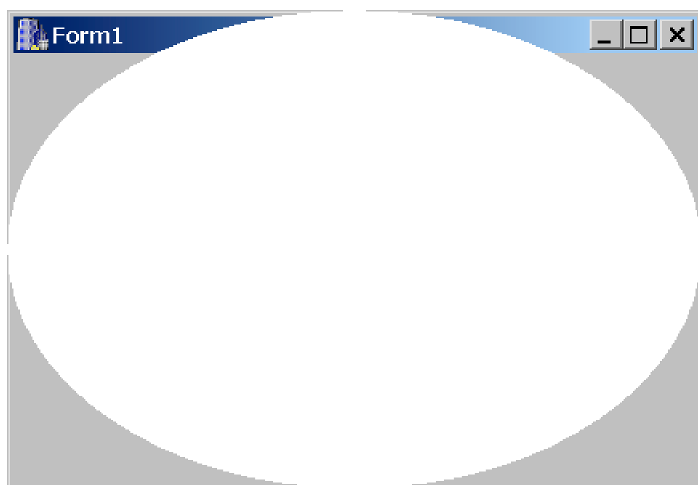
Regiony można również łączyć funkcją `CombineRgn()`. Funkcja ta pobiera dwa uchwyty do zdefiniowanych wcześniej regionów, uchwyt do istniejącego (nie tylko zadeklarowanego, ale i stworzonego funkcją `Create...Rgn()`) regionu, do którego będzie przypisany wynik kombinacji dwóch regionów wejściowych oraz sposób łączenia. Można więc stworzyć formę w kształcie oczu:

```
HRGN Rgn1, Rgn2;
Rgn1 = CreateEllipticRgn(0, 0, Width/2, Height);
Rgn2 = CreateEllipticRgn(Width/2, 0, Width, Height);
CombineRgn(Rgn1,Rgn1,Rgn2,RGN_OR);
SetWindowRgn(Handle, Rgn1, true);
```



lub formę „z dziurą":

```
HRGN Rgn1, Rgn2;
Rgn1 = CreateRectRgn(0, 0, Width, Height);
Rgn2 = CreateEllipticRgn(0, 0, Width, Height);
CombineRgn(Rgn1,Rgn1,Rgn2,RGN_DIFF);
SetWindowRgn(Handle, Rgn1, true);
```



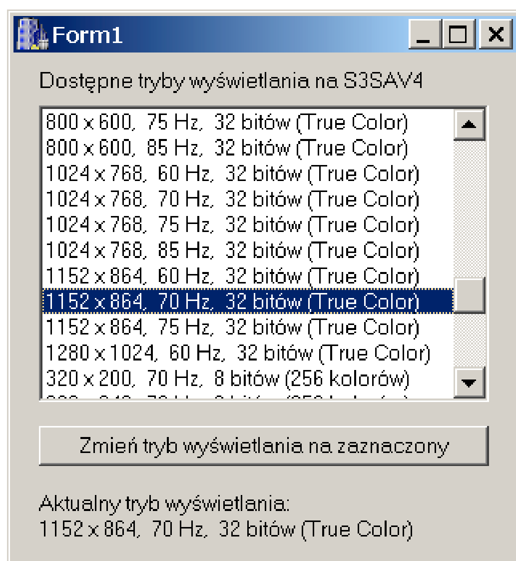
VI. Różności

1. Pobieranie dostępnych trybów pracy karty graficznej

Pobierzemy i przechowamy w tablicy listę możliwych trybów pracy karty graficznej (szerokość i wysokość ekranu w pikselach, ilość kolorów i częstość wyświetlania). Wyniki wyświetlimy w liście `TListBox`. W następnym paragrafie dodamy możliwość zmiany rozdzielczości ekranu.

Aby pobrać informacje o możliwych trybach ekranu należy wykorzystać funkcję `EnumDisplaySettings`. Jej pierwszym argumentem jest łańcuch określający kartę graficzną `"\\.\Display1"`, `"\\.\Display2"` lub `"\\.\Display2"`[19] (Windows obsługuje maksymalnie trzy). Jeżeli wpiszemy tu `NULL` – pobrane zostaną informacje dla karty bieżącej. Kolejny argument to numer bieżący trybu zwiększany od zera (wywołanie dla 0 inicjuje informacje o karcie graficznej). Funkcję należy wstawić do pętli, w której ten parametr będzie zwiększany. Ostatni parametr to wskaźnik do struktury WinApi `DEVMODE`, w której zostaną zapisane informacje o wybranym przez drugi parametr trybie wyświetlania.

Otwórzmy nowy projekt. Do klasy `TForm1` dodajmy prywatną własność `DEVMODE DevMode[1024];`. Na formę rzućmy `ListBox1`.



Właściwą funkcję pobierania informacji umieścimy w metodzie zdarzeniowej związanej z `OnCreate` formy. Zasadniczą częścią tej metody jest pętla:

```
bool Result=true;
for (int i=0;Result && i<1024;i++)
       {
       Result=EnumDisplaySettings(NULL,i,&DevMode[i]);
       }
```

Do kolejnych elementów tablicy `DevMode` pobierane są informacje. Aby równocześnie wyświetlić te informacje w `ListBox1` należy uzupełnić kod metody:

```
//Pobieranie dostepnych trybow wyswietlania
bool Result=true;
for (int i=0;Result && i<1024;i++)
       {
```

[19] W C++ należy to zakodować jako `"\\\\.\\Display1"`, ponieważ znak \ jest znakiem specjalnym zapisywanym jako „\\"

```
      Result=EnumDisplaySettings(NULL,i,&DevMode[i]);
      if (Result)
            {
            AnsiString S=(AnsiString)(double)DevMode[i].dmPelsWidth+
                  " x "+(AnsiString)(double)DevMode[i].dmPelsHeight+",
                  "+(AnsiString)(double)DevMode[i].dmDisplayFrequency+" Hz,
                  "+(AnsiString)(double)DevMode[i].dmBitsPerPel+" bitów";
            switch (DevMode[i].dmBitsPerPel)
                  {
                  case 1: S=S+" (Monochromatyczny)"; break;
                  case 2: S=S+" (4 kolory)"; break;
                  case 4: S=S+" (16 kolorów)"; break;
                  case 8: S=S+" (256 kolorów)"; break;
                  case 16: S=S+" (High Color)"; break;
                  case 32: S=S+" (True Color)"; break;
                  }
            ListBox1->Items->Add(S);
            }
      };
```

Można tę funkcję uzupełnić tak, aby zaznaczyć na liście aktualnie wybrany tryb pracy karty graficznej. W tym celu należy odczytać ilość wyświetlanych punktów na ekranie (`Screen->Height`, `Screen->Width`), ilość bitów opisujących każdy piksel (tj. ilość kolorów) i częstość odświeżania monitora (funkcja WinAPI `GetDeviceCaps`). Ze względu na to, że częstość odświeżania dostępna jest tylko w systemach NT/2000/XP – tylko w tych systemach program będzie działał poprawnie. W tym celu w metodzie `FormCreate()` należy na samym początku dodać linie odczytujące aktualne parametry wyświetlania za pomocą funkcji WinAPI `GetDeviceCaps()`. Jej pierwszym argumentem powinien być uchwyt do obiektu związanego z grafiką (np. uchwyt płótna formy `Form1->Canvas->Handle`) drugim – reprezentowana przez predefiniowaną stałą liczba wskazująca na parametr, który ma być zwrócony jako wartość funkcji. Wyniki umieszczamy w dodanym do formy komponencie `Label3`:

```
//Pobieranie aktualnego trybu wyswietlania
int sx=Screen->Width;
int sy=Screen->Height;
int sbits=GetDeviceCaps(Form1->Canvas->Handle,PLANES)*
          GetDeviceCaps(Form1->Canvas->Handle,BITSPIXEL);
int srefr=GetDeviceCaps(Form1->Canvas->Handle,VREFRESH);

AnsiString SP=(AnsiString)(double)sx+" x "+(AnsiString)(double)sy+",  "+
              (AnsiString)(double)srefr+" Hz,  "+
              (AnsiString)(double)sbits+" bitów";
switch (sbits)
      {
      case 1: SP=SP+" (Monochromatyczny)"; break;
      case 2: SP=SP+" (4 kolory)"; break;
      case 4: SP=SP+" (16 kolorów)"; break;
      case 8: SP=SP+" (256 kolorów)"; break;
      case 16: SP=SP+" (High Color)"; break;
      case 32: SP=SP+" (True Color)"; break;
      }
Label3->Caption=SP;
```

Do pętli po `i`, która wywołuje `EnumDisplaySettings()`, za dodaniem linii do `ListBox1` dodajemy instrukcję warunkową:

```
if (DevMode[i].dmPelsWidth==sx &&
    DevMode[i].dmPelsHeight==sy &&
    DevMode[i].dmBitsPerPel==sbits &&
    DevMode[i].dmDisplayFrequency==srefr) ListBox1->ItemIndex=i;
```

Ze struktury DEVMODE można również pobrać informację o nazwie kodowej karty graficznej:

```
Label1->Caption="Dostępne tryby wyświetlania na "
                     +(AnsiString)(char*)DevMode[0].dmDeviceName;
```

Zadanie
Dodać do powyższego projektu możliwość wyboru jednej z trzech kart graficznych.

2. Zmiana trybu wyświetlania

Aby zmienić tryb pracy karty graficznej (a co za tym idzie tryb pracy monitora) należy skorzystać z funkcji WinAPI ChangeDisplaySettings. Jej pierwszym argumentem jest struktura DEVMODE zawierająca opis żądanego trybu. Wykorzystajmy listę trybów uzyskaną funkcją EnumDisplaySettings (rozbudowujemy aplikację z poprzedniego paragrafu), aby mieć pewność, że wybrany przez nas argument ChangeDisplaySettings odpowiada obsługiwanemu trybowi pracy karty graficznej.

Dodajmy do formy przycisk Button1 i w jego metodzie zdarzeniowej napiszmy:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
//zob. Delphi FAQ pyt. 76
AnsiString S;
switch (ChangeDisplaySettings(&DevMode[ListBox1->ItemIndex],0))
       {
       case DISP_CHANGE_SUCCESSFUL: S="Tryb wyświetlania został zmieniony"; break;
       case DISP_CHANGE_RESTART: S="Niezbędne jest ponowne uruchomienie komputera"; break;
       case DISP_CHANGE_BADFLAGS: S="Błędne ustawienia"; break;
       case DISP_CHANGE_FAILED: S="Sterownik ekranu odrzucił ustawienia"; break;
       case DISP_CHANGE_BADMODE: S="Wybrany tryb ekranu nie jest obsługiwany"; break;
       case DISP_CHANGE_NOTUPDATED: S="Nowe ustawienia nie mogły
                                 zostać zapisane do rejestru (brak uprawnień)"; break;
       default: S="Efekt nieznany"; break;
       }
ShowMessage(S);
}
```

Najistotniejsze w tej metodzie jest wywołanie

```
ChangeDisplaySettings(&DevMode[ListBox1->ItemIndex],0),
```

w którym wybieramy tryb pracy karty odpowiadający zaznaczonej pozycji w ListBox1.
Drugi argument funkcji decyduje o tym, czy rzeczywiście zmienić tryb pracy, czy tylko wykonać test.

Można zmienić tylko wybrane elementy struktury DEVMODE. Wówczas trzeba je opisać w elemencie DEVMODE.dmFields:

```
long SetScreenResolution(int Width, int Height)
       {
       DEVMODE TmpDevMode;
       TmpDevMode.dmSize=sizeof(DEVMODE);
       TmpDevMode.dmPelsWidth=Width;
       TmpDevMode.dmPelsHeight=Height;
       TmpDevMode.dmFields=DM_PELSWIDTH | DM_PELSHEIGHT;

       return ChangeDisplaySettings(&TmpDevMode,0);
       }
```

VII. Gdzie szukać pomocy?

Kolejno:

1. Microsoft Win32 SDK (Software Development Kit) dołączony do Delphi i C++ Buildera w wersjach wyższych niż Standard/Personal
2. MSDN Library http://msdn.microsoft.com/ (najprościej wpisać hasło lub nazwę szukanej funkcji w dostępnej tam przeszukiwarce)
3. Książka pt. „Programowanie Windows” Charlsa Petzolda (http://www.charlespetzold.com/)
4. Książki dotyczące Delphi, C++ Buildera, sieć
5. Listy dyskusyjne dotyczące języków programowania aplikacji Windows (np. pl.comp.lang.delphi, pl.comp.lang.vbasic)