helion.pl
KSIĘGARNIA INTERNETOWA WYDAWNICTWO

IDŹ DO
PRZYKŁADOWY ROZDZIAŁ
SPIS TREŚCI

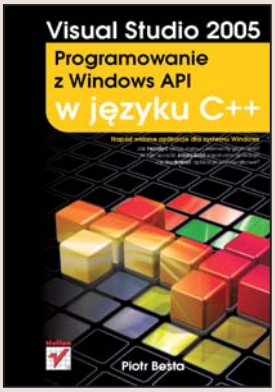
KATALOG KSIĄŻEK
KATALOG ONLINE
ZAMÓW DRUKOWANY KATALOG

TWÓJ KOSZYK
DODAJ DO KOSZYKA

CENNIK I INFORMACJE
ZAMÓW INFORMACJE O NOWOŚCIACH
ZAMÓW CENNIK

CZYTELNIA
FRAGMENTY KSIĄŻEK ONLINE

Visual Studio 2005. Programowanie z Windows API w języku C++



Autor: Piotr Besta
ISBN: 978-83-246-1567-4
Format: 158x235, stron: 728

Napisz własne aplikacje dla systemu Windows

- Jak tworzyć okna, menu i elementy graficzne?
- W jaki sposób zarządzać zasobami aplikacji?
- Jak budować aplikacje wielowątkowe?

System operacyjny to nie tylko środowisko, w którym możemy uruchamiać zainstalowane programy – to także źródło zasobów dla programów tworzonych przez nas samych. Każda aplikacja dla systemu Windows, w której można znaleźć ikony, okna dialogowe, paski przewijania i inne powszechnie znane elementy, korzysta z bibliotek zwanych Windows API – zestawu funkcji ułatwiających zaprogramowanie określonych komponentów. Dzięki zastosowaniu Windows API możemy umieszczać w naszych aplikacjach typowe dla systemu operacyjnego składniki interfejsu użytkownika i moduły wykorzystujące urządzenia zewnętrzne.

Książka „Visual Studio 2005. Programowanie z Windows API w języku C++” to podręcznik, dzięki któremu poznasz metody tworzenia programów dla systemu operacyjnego Windows. Nauczysz się korzystać z systemowego API w celu zaimplementowania w aplikacji mechanizmów interfejsu użytkownika, wyświetlania elementów graficznych i obsługiwania przetwarzania wielowątkowego. Dowiesz się, jak wykorzystywać zasoby programowe, budować biblioteki statyczne i biblioteki DLL, obsługiwać mysz i klawiaturę oraz mierzyć czas. Przeczytasz o tym, jak działają aplikacje dla systemu Windows, i zdobędziesz wiedzę niezbędną do tego, by tworzyć własne!

- Tworzenie okien i umieszczanie w nich tekstów oraz elementów graficznych
- Korzystanie z GDI
- Wyświetlanie map bitowych
- Tworzenie bibliotek statycznych
- Obsługa klawiatury i myszy
- Zarządzanie zasobami aplikacji
- Budowanie okien dialogowych
- Korzystanie z kontrolek
- Obsługa wielozadaniowości i wielowątkowości

Poznaj techniki i metody tworzenia aplikacji dla systemu operacyjnego Windows

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

Spis treści

Wstęp **9**

Rozdział 1. WinMain — fundament aplikacji Windows **11**
1.1. Pierwszy program 11
1.1.1. Tworzymy projekt aplikacji 12
1.1.2. Dodajemy do projektu plik kodu 16
1.1.3. Piszemy kod programu 17
1.1.4. Funkcja MessageBox 19
1.1.5. Kompilujemy kod i uruchamiamy program 22
1.2. Więcej o Visual Studio 24
1.2.1. Konfiguracja Debug i Release 24
1.2.2. Biblioteki statyczne 26
1.3. Tworzenie okna 27
1.3.1. Klasa okna 28
1.3.2. Rejestracja klasy okna 34
1.3.3. Jak wyrejestrować klasę okna? 35
1.3.4. Funkcja CreateWindowEx 35
1.3.5. Wyświetlanie okna 38
1.3.6. Procedura okna 39
1.3.7. Kolejka wiadomości 40
1.3.8. Pętla wiadomości 40
1.3.9. Definicja procedury okna 43
1.3.10. Przetwarzanie wiadomości 45
1.3.11. Wiadomości WM_CREATE i WM_DESTROY 46
1.3.12. Projekt: Okno 47

Rozdział 2. Rysowanie w oknie **51**
2.1. Wiadomość WM_PAINT 51
2.1.1. Jak unieważnić prostokątny fragment obszaru roboczego? 52
2.1.2. Jak zatwierdzić prostokątny fragment obszaru roboczego? 53
2.1.3. Obsługiwanie wiadomości WM_PAINT 53
2.2. Kontekst urządzenia 57
2.2.1. Czym jest kontekst urządzenia? 58
2.2.2. Właściwości kontekstu urządzenia 58
2.2.3. Pobieranie uchwytu kontekstu urządzenia 64
2.2.4. Inne operacje na kontekście urządzenia 65
2.3. Wyświetlanie tekstu 66
2.3.1. Funkcja DrawText 67
2.3.2. Wiadomości WM_SYSCOMMAND i WM_CLOSE 68
2.3.3. Projekt: Tekst w oknie 01 71
2.3.4. Pobieranie informacji o czcionce 74

2.3.5. Zmiana domyślnej czcionki ... 77
2.3.6. Tworzenie nowej czcionki ... 79
2.3.7. Funkcja TextOut ... 84
2.3.8. Więcej okien? ... 85
2.3.9. Niszczenie okna ... 85
2.3.10. Zmiana właściwości okna ... 86
2.3.11. Wiadomości WM_SETFOCUS, WM_KILLFOCUS i WM_ACTIVATE ... 88
2.3.12. Projekt: Tekst w oknie 02 ... 90
2.4. Rysowanie pikseli, linii, prostokątów, kół, elips i wielokątów ... 99
2.4.1. Rysowanie pikseli ... 100
2.4.2. Rysowanie linii ... 101
2.4.3. Rysowanie prostokątów ... 103
2.4.4. Rysowanie kół i elips ... 105
2.4.5. Rysowanie wielokątów ... 106
2.5. Obiekty GDI ... 107
2.5.1. Pióra ... 108
2.5.2. Pędzle ... 113
2.5.3. Regiony ... 118
2.5.4. Projekt: Malarz ... 129
2.5.5. Pobieranie aktualnego obiektu kontekstu urządzenia ... 140
2.6. Bitmapy ... 141
2.6.1. Czym jest bitmapa? ... 142
2.6.2. Ładowanie bitmapy do pamięci operacyjnej ... 143
2.6.3. Przykład użycia funkcji LoadImage ... 146
2.6.4. Pobieranie informacji o załadowanej bitmapie ... 147
2.6.5. Reprezentacja bitmapy w pamięci ... 148
2.6.6. Wyświetlanie bitmapy ... 149
2.6.7. Rysowanie po powierzchni bitmapy ... 164
2.6.8. Przenoszenie obrazu między bitmapami ... 167
2.6.9. Oczekiwanie na wiadomość ... 171
2.6.10. Wiadomości WM_MOVE i WM_SIZE ... 176
2.6.11. Projekt: Gaz ... 177
2.6.12. Tworzenie własnych bitmap ... 196
2.6.13. Zapisywanie bitmapy do pliku BMP ... 200
2.6.14. Projekt: Spektrum barw ... 217
2.7. Pobieranie informacji o ustawieniach systemu i kontekstu urządzenia ... 228
2.7.1. Informacje o systemie ... 228
2.7.2. Informacje o kontekście urządzenia ... 231
2.7.3. Zamykanie i resetowanie systemu ... 232
2.7.4. Praca ze strukturami RECT ... 234

Rozdział 3. Tworzenie i korzystanie z biblioteki statycznej ... 239
3.1. Czym jest biblioteka statyczna? ... 239
3.2. Po co tworzymy biblioteki statyczne? ... 240
3.3. Przykład utworzenia biblioteki ... 241
3.3.1. Dodajemy do biblioteki funkcje matematyczne ... 243
3.3.2. Dodajemy do biblioteki klasę CKot ... 245
3.3.3. Kompilacja biblioteki ... 251
3.4. Testowanie biblioteki ... 252
3.4.1. Tworzymy aplikację konsolową ... 252
3.4.2. Dodajemy do aplikacji bibliotekę i pliki nagłówkowe ... 256
3.4.3. Piszemy kod programu testowego ... 258
3.5. Operatory new i delete ... 261

Rozdział 4. Klawiatura ... 267
4.1. Jak to działa w Windows? ... 267
4.2. Wirtualne klawisze ... 268
4.3. Wiadomości WM_KEYDOWN i WM_KEYUP ... 270
4.4. Wiadomość WM_CHAR ... 273
4.5. Wiadomości WM_SYSKEYDOWN i WM_SYSKEYUP ... 275
4.6. Niezależne pobieranie informacji o stanie klawiszy ... 276
4.7. Projekt: Klawiatura ... 277

Rozdział 5. Mysz ... 285
5.1. Mysz w Windows ... 285
5.2. Kursor myszy ... 286
5.3. Wiadomości myszy ... 286
5.4. Obsługa podwójnego kliknięcia ... 288
5.5. Niezależne pobieranie i ustalanie pozycji kursora myszy ... 289
5.6. Przechwycenie myszy ... 290
5.7. Sterowanie wyświetlaniem kursora ... 292
5.8. Projekt: Mysz ... 293

Rozdział 6. Odmierzanie czasu ... 301
6.1. Pobieranie systemowego czasu i daty z dokładnością do milisekundy ... 301
6.2. Systemowy licznik milisekund ... 303
6.3. Korzystanie z zegara systemowego ... 306
6.3.1. Tworzenie własnych zegarów ... 306
6.3.2. Techniki tworzenia zegara ... 308
6.3.3. Definiowanie funkcji zegara ... 309
6.3.4. Uśmiercanie zegarów ... 309
6.4. Projekt: Minuta ... 310
6.4.1. Dokładne wyznaczanie rozmiaru okna w oparciu o żądany rozmiar obszaru roboczego ... 318
6.5. Specjalistyczne odmierzanie czasu ... 320
6.6. Klasa CCzas ... 323
6.7. Projekt: Stoper ... 327
6.7.1. Odtwarzanie plików dźwiękowych WAV za pomocą funkcji PlaySound ... 339

Rozdział 7. Zasoby aplikacji ... 341
7.1. Zarządzanie zasobami ... 342
7.2. Projekt: Kursor i ikona ... 342
7.2.1. Kursory ... 343
7.2.2. Ikony ... 350
7.2.3. Odwoływanie się do zasobów ... 352
7.2.4. Piszemy kod dla projektu „Kursor i ikona” ... 353
7.3. Zmiana i pobieranie ustawień klasy okna ... 358
7.4. Projekt: Bitmapa i łańcuchy znaków ... 360
7.4.1. Bitmapy ... 360
7.4.2. Tablice łańcuchów znakowych ... 362
7.4.3. Piszemy kod dla projektu Bitmapa i łańcuchy znaków ... 365
7.5. Wysyłanie wiadomości do okna ... 373
7.6. Projekt: Główne menu okna ... 375
7.6.1. Jak zbudowane jest menu? ... 375
7.6.2. Elementy menu ... 376
7.6.3. Tworzenie menu za pomocą edytora ... 376
7.6.4. Ładowanie menu z zasobów i podłączanie do okna ... 381
7.6.5. Niszczenie menu ... 383
7.6.6. Wiadomości okna związane z menu ... 383

7.6.7. Zaznaczanie elementów menu ... 386
7.6.8. Radiowe elementy menu ... 388
7.6.9. Sterowanie dostępnością elementów menu ... 389
7.6.10. Pobieranie uchwytu głównego menu i uchwytów podmenu ... 390
7.6.11. Pobieranie informacji o stanie elementu menu ... 391
7.6.12. Piszemy kod dla projektu „Główne menu okna” ... 393
7.6.13. Tworzenie menu w trakcie pracy programu ... 401
7.6.14. Modyfikowanie menu w trakcie pracy programu ... 405
7.6.15. Wyświetlanie elementu menu za pomocą bitmapy ... 413
7.6.16. Zmieniamy bitmapę zaznaczenia elementu menu ... 415
7.6.17. Inne operacje przeprowadzane na elementach menu ... 416
7.7. Projekt: Menu kontekstowe i systemowe ... 418
7.7.1. Menu kontekstowe ... 418
7.7.2. Menu systemowe ... 422
7.7.3. Piszemy kod dla projektu „Menu kontekstowe i systemowe” ... 424
7.8. Projekt: Skróty klawiaturowe ... 431
7.8.1. Tworzenie tablicy skrótów klawiaturowych ... 432
7.8.2. Ładowanie tablicy skrótów klawiaturowych ... 434
7.8.3. Tworzenie tablicy skrótów klawiaturowych w trakcie pracy programu ... 435
7.8.4. Tłumaczenie skrótów klawiaturowych na wiadomości WM_COMMAND i WM_SYSCOMMAND ... 437
7.8.5. Piszemy kod dla projektu „Skróty klawiaturowe” ... 439
7.9. Projekt: Niestandardowe zasoby ... 447
7.9.1. Tworzenie niestandardowych zasobów ... 448
7.9.2. Ładowanie niestandardowych zasobów ... 450
7.9.3. Odczytywanie danych z niestandardowych zasobów ... 451
7.9.4. Piszemy kod dla projektu „Niestandardowe zasoby” ... 452
7.10. Biblioteka łączona dynamicznie jako zewnętrzne źródło zasobów i kodu ... 456
7.10.1. Jak działają biblioteki DLL? ... 456
7.10.2. Projekt: Pierwszy DLL ... 457
7.10.3. DllMain — główna funkcja biblioteki DLL ... 459
7.10.4. Dodajemy do biblioteki DLL funkcję ... 461
7.10.5. Dodajemy do biblioteki DLL zmienne ... 466
7.10.6. Kompilacja i konsolidacja biblioteki DLL ... 467
7.10.7. Współużytkowanie zmiennych biblioteki DLL przez wiele aplikacji ... 467
7.10.8. Ładowanie biblioteki DLL w trakcie pracy programu i pobieranie adresów funkcji ... 470
7.10.9. Umieszczanie zasobów w bibliotece DLL i ich wykorzystywanie ... 472
7.10.10. Projekt: Test biblioteki DLL ... 473
7.11. Szerokie znaki — UNICODE ... 478
7.11.1. Typ danych char ... 479
7.11.2. Typ danych wchar_t ... 481
7.11.3. Ogólny znakowy typ danych ... 483
7.11.4. Szerokie znaki i funkcje ... 485

Rozdział 8. Okna dialogowe ... 489
8.1. Modalne okna dialogowe ... 491
8.2. Projekt: Modalne okno dialogowe ... 491
8.2.1. Dodajemy do zasobów programu okno dialogowe ... 491
8.2.2. Procedura okna przetwarzająca wiadomości okna dialogowego ... 495
8.2.3. Wiadomość WM_INITDIALOG ... 497
8.2.4. Przesuwanie okna i pobieranie uchwytu okna nadrzędnego ... 498
8.2.5. Tworzenie, wyświetlanie i zamykanie modalnego okna dialogowego ... 499
8.2.6. Piszemy kod dla projektu „Modalne okno dialogowe” ... 501

8.3. Niemodalne okna dialogowe ... 508
8.4. Projekt: Niemodalne okno dialogowe ... 509
8.4.1. Tworzenie, wyświetlanie i zamykanie niemodalnego okna dialogowego ... 509
8.4.2. Wysyłanie wiadomości do niemodalnego okna dialogowego ... 511
8.4.3. Piszemy kod dla projektu „Niemodalne okno dialogowe” ... 512
8.5. Predefiniowane okna dialogowe systemu ... 521
8.5.1. Otwieranie plików ... 522
8.5.2. Wskazywanie plików do zapisu ... 527
8.5.3. Wybieranie koloru ... 528
8.5.4. Tworzymy klasę CSystemoweDlg ... 530
8.5.5. Projekt: Systemowe okna dialogowe ... 534

Rozdział 9. Kontrolki ... 547
9.1. Jak działają kontrolki? ... 547
9.2. Kontrolki klasy Button ... 549
9.2.1. Projekt: Prosty przycisk ... 549
9.2.2. Typy i style przycisków ... 555
9.2.3. Kody powiadomienia przycisków ... 562
9.2.4. Wiadomości kontrolne przycisków ... 562
9.2.5. Praca w edytorze okien dialogowych ... 564
9.2.6. Projekt: Przyciski ... 566
9.3. Kontrolki klasy Edit ... 583
9.3.1. Style pola edycji ... 583
9.3.2. Kody powiadomienia pola edycji ... 584
9.3.3. Wiadomości kontrolne pola edycji ... 585
9.3.4. Umieszczanie i pobieranie wartości liczbowych z pola edycji ... 587
9.3.5. Projekt: Pole edycji ... 588
9.4. Kontrolki klasy Static ... 596
9.4.1. Style kontrolek statycznych ... 596
9.4.2. Wiadomości kontrolne kontrolek statycznych ... 597
9.5. Kontrolki klasy ComboBox ... 597
9.5.1. Style kontrolki ComboBox ... 598
9.5.2. Kody powiadomienia kontrolki ComboBox ... 599
9.5.3. Wiadomości kontrolne ComboBox ... 599
9.5.4. Projekt: ComboBox ... 603
9.6. Wygląd kontrolek w systemie Windows XP ... 612
9.6.1. Tworzenie manifestu ... 613
9.6.2. Dodawanie manifestu do projektu aplikacji ... 615

Rozdział 10. Paski przewijania ... 617
10.1. Rozmiar pasków przewijania ... 617
10.2. Budowa pasków przewijania ... 618
10.3. Wiadomości WM_HSCROLL i WM_VSCROLL ... 619
10.4. Pobieranie i ustawianie stanu pasków przewijania ... 621
10.5. Wyświetlanie, ukrywanie, włączanie i wyłączanie pasków przewijania ... 624
10.6. Przewijanie obszaru roboczego okna ... 625
10.7. Projekt: Ogromne bitmapy ... 626

Rozdział 11. Procesy i wątki ... 641
11.1. Wielozadaniowość ... 641
11.1.1. Tworzenie nowego procesu ... 643
11.1.2. Projekt: Nowy proces ... 644
11.1.3. Natychmiastowe kończenie pracy aplikacji ... 648

11.2. Wielowątkowość ... 648
11.2.1. Kiedy korzystamy z dodatkowych wątków? ... 649
11.2.2. Tworzenie wątków ... 650
11.2.3. Funkcja wątku ... 652
11.2.4. Kończenie pracy wątków ... 652
11.2.5. Usypianie wątków ... 654
11.2.6. Projekt: Pierwszy wątek ... 654
11.2.7. Projekt: Więcej wątków ... 656
11.2.8. Zmienne globalne programu a zmienne automatyczne i statyczne wątków ... 658
11.2.9. Zawieszanie i przywracanie pracy wątków ... 658
11.2.10. Projekt: Zawieszanie wątku ... 659
11.2.11. Pobieranie uchwytu wątku ... 661
11.2.12. Problemy z wątkami ... 661
11.2.13. Metody synchronizacji pracy wątków ... 662
11.2.14. Oczekiwanie na sygnalizowanie pojedynczego obiektu ... 662
11.2.15. Projekt: Oczekiwanie na koniec wątku ... 663
11.2.16. Oczekiwanie na sygnalizowanie wielu obiektów ... 665
11.2.17. Projekt: Oczekiwanie na koniec wielu wątków ... 666
11.2.18. Zdarzenia ... 668
11.2.19. Projekt: Zdarzenia ... 669
11.2.20. Muteksy ... 674
11.2.21. Projekt: Muteks ... 675
11.2.22. Semafory ... 679
11.2.23. Projekt: Semafor ... 681

Rozdział 12. Nie tylko Windows API ... 685
12.1. Biblioteka DirectX ... 685
12.2. Konfiguracja Visual Studio ... 688
12.3. Obiekt Direct3D ... 689
12.3.1. Tworzenie obiektu Direct3D ... 690
12.3.2. Pobieranie informacji o karcie graficznej ... 692

Dodatek A ... 697
Kody ASCII ... 697
Kody skaningowe klawiatury ... 699

Skorowidz ... 701

Rozdział 4.

Klawiatura

Nadeszła najwyższa pora, aby nauczyć się posługiwania klawiaturą. Oczywiście, mam tu na myśli obsługę związaną z czysto programistycznym punktem widzenia. Zacznę od szybkiego omówienia podstaw działania klawiatury. Następnie dowiesz się, jak system Windows współpracuje z klawiaturą, jak tworzone są wiadomości związane z zdarzeniami klawiatury oraz w jaki sposób są później wstawiane do kolejki wiadomości aplikacji.

Potem wyjaśnię, czym jest wirtualny klawisz, i powiem, jak odbierać wiadomości z nim związane. Tu dowiesz się też, jak bez udziału procedury okna sprawdzać stan poszczególnych klawiszy klawiatury. Poznasz wiadomość systemową związaną z obieraniem wyłącznie znaków tekstowych generowanych za pomocą klawiatury. Na zakończenie rozdziału napiszę przykładowy program, który będzie przechwytywał informacje o naciśniętych klawiszach klawiatury, po czym będzie je wyświetlał w oknie.

4.1. Jak to działa w Windows?

Klawiatura to podstawowe urządzenie, za którego pomocą użytkownik komunikuje się z systemem zainstalowanym na komputerze. Dzisiaj klawiatura tak mocno zakorzeniła się w naszym życiu, że dla sporej grupy ludzi stanowi codzienne, nieodłączne narzędzie pracy. Jednym słowem, trudno wyobrazić sobie dzisiaj świat bez niej, ale jeszcze trudniej wyobrazić sobie alternatywne urządzenie, które mogłoby zastąpić klawiaturę. Wydawać się może, że nic nie zastąpi klawiszy nawet wtedy, gdy klawiatura będzie płaska jak kartka papieru i przeźroczysta jak powietrze.

Spotykane dzisiaj klawiatury składają się najczęściej ze 101 lub 102 klawiszy. Są to tzw. klawiatury rozszerzone, gdyż niektóre klawisze zostały zdublowane, co — jak by nie patrzeć — bardzo ułatwia pracę.

Każda klawiatura posiada wbudowany mikroprocesorowy układ kontrolujący kilkadziesiąt razy w ciągu sekundy stan wszystkich klawiszy. Zebrane informacje są formowane w pakiety danych, wysyłane następnie do jednego z portów wejściowych komputera.

System Windows w odpowiedzi na przyciśnięcie lub zwolnienie klawisza generuje odpowiednią wiadomość. Utworzona wiadomość jest przypisywana zawsze do tego okna, które było aktywne w momencie zaistnienia zdarzenia związanego z klawiaturą. Następnie wiadomość jest wstawiana do systemowej kolejki wiadomości. Skąd dalej trafia do kolejki wiadomości wybranej aplikacji. Najważniejsze jest to, abyś pamiętał, że system nie umieszcza następnej wiadomości związanej ze zdarzeniem klawiatury w kolejce wiadomości aplikacji, dopóki poprzednia wiadomość nie zostanie obsłużona.

4.2. Wirtualne klawisze

Układ mikroprocesorowy klawiatury po wykryciu, że któryś z klawiszy został naciśnięty, generuje i wysyła do komputera tzw. kod skaningowy klawisza (kody skaningowe klawiszy zostały podane w dodatku A). Przykładowo klawiszowi A odpowiada kod o wartości 30. Gdy klawisz jest zwalniany, generowany jest ten sam kod skaningowy, z tą różnicą, że jego wartość zostaje powiększona o 128. Po zwolnieniu klawisza A otrzymujemy kod 158 (128+30).

Pamiętaj, aby nie mylić kodów skaningowych klawiszy (*Original Equipment Manufacturer* — OEM) z kodem ASCII (*American Standards Committee for Information Interchange*) i z kodami ANSI (*American National Standards Institute*). Kody ANSI i ASCII są budowane na podstawie kodu skaningowego, przy uwzględnieniu dodatkowych czynników mających wpływ na stan klawiatury. Te czynniki to stan klawiszy *Caps Lock* (duże, małe litery), *Shift* (górne znaki klawiszy) oraz *Num Lock* (aktywna, nieaktywna klawiatura numeryczna). Możemy rozróżnić 256 kodów ANSI i ASCII (kody ASCII znajdują się w dodatku A na końcu książki).

Jak widzisz, za pomocą pojedynczego kodu skaningowego nie jesteśmy w stanie stwierdzić, czy użytkownik chciał napisać dużą literę, czy małą. Aby poznać tę informację, należy kontrolować klawisze sterujące stanem klawiatury. Nie martw się. Mam dla Ciebie bardzo dobrą wiadomość. Nie będziemy korzystali bezpośrednio z kodów skaningowych. Dlaczego? Kody skaningowe mają tę brzydką cechę, że są zależne od sprzętu. Mimo ogólnie przyjętych na świecie standardów wyznaczających parametry produkowanych klawiatur i ich układów mikroprocesorowych, możemy spotkać takie klawiatury, które dla klawisza A wygenerują odmienny kod skaningowy. Oczywiście, są to sytuacje marginalne. Dzisiaj programiści Windows nie muszą się w ogóle tym przejmować. Było to jednak szczególnie irytujące dla twórców systemu Windows w jego wczesnych latach „życia”.

Aby rozwiązać problem, postanowiono opracować niezależne, oderwane od sprzętu klawisze wirtualne. Kody wirtualnych klawiszy, z wyjątkiem kodów klawiszy literowych i klawiszy numerycznych z podstawowej części klawiatury (nienumerycznej), są odmienne od kodów skaningowych, kodów ASCII i ANSI. Wirtualne klawisze nie istnieją, są one wytworem swoistej interpretacji kodów skaningowych klawiszy przeprowadzonej przez system Windows.

Wszystkim wirtualnym klawiszom, z wyjątkiem klawiszy literowych i numerycznych podstawowej klawiatury, został przypisany identyfikator. Identyfikatory rozpoczynają się od przedrostka VK (ang. *Virtual Key*). Ich definicja znajduje się w pliku nagłówkowym *winuser.h*. Najważniejsze identyfikatory zebrałem w tabeli 4.1. Zwróć uwagę na to, że klawisze literowe i numeryczne posiadają jedynie kod wirtualnego klawisza.

Tabela 4.1. *Kod klawiszy wirtualnych*

Kod dziesiętnie	Kod szesnastkowo	Identyfikator	Uwagi o wirtualnym klawiszu
8	0x08	VK_BACK	*Backspace*
9	0x09	VK_TAB	*Tab*
13	0x0D	VK_RETURN	Jeden z klawiszy *Enter*
16	0x10	VK_SHIFT	Jeden z klawiszy *Shift*
17	0x11	VK_CONTROL	Jeden z klawiszy *Ctrl*
18	0x12	VK_MENU	Jeden z klawiszy *Alt*
20	0x14	VK_CAPITAL	*Caps Lock*
27	0x1B	VK_ESCAPE	*Esc*
32	0x20	VK_SPACE	*Spacja*
33	0x21	VK_PRIOR	*Page Up*
34	0x22	VK_NEXT	*Page Down*
35	0x23	VK_END	*End*
36	0x24	VK_HOME	*Home*
37	0x25	VK_LEFT	Strzałka w lewo
38	0x26	VK_UP	Strzałka w górę
39	0x27	VK_RIGHT	Strzałka w prawo
40	0x28	VK_DOWN	Strzałka w dół
44	0x2C	VK_SNAPSHOT	*Print Screen*
45	0x2D	VK_INSERT	*Insert*
46	0x2E	VK_DELETE	*Delete*
48 – 57	0x30 – 0x39	nie ma	Klawisze numeryczne od 0 do 9 z podstawowej części klawiatury
65 – 90	0x41 – 0x5A	nie ma	Klawisze literowe od A do Z
91	0x5B	VK_LWIN	Lewy klawisz Windows
92	0x5C	VK_RWIN	Prawy klawisz Windows
93	0x5D	VK_APPS	Klawisz aplikacji
145	0x91	VK_SCROLL	*Scroll Lock*
112	0x70	VK_F1	Klawisz *F1*
113	0x71	VK_F2	Klawisz *F2*
114	0x72	VK_F3	Klawisz *F3*
115	0x73	VK_F4	Klawisz *F4*

Tabela 4.1. *Kod klawiszy wirtualnych — ciąg dalszy*

Kod dziesiętnie	Kod szesnastkowo	Identyfikator	Uwagi o wirtualnym klawiszu
116	0x74	VK_F5	Klawisz *F5*
117	0x75	VK_F6	Klawisz *F6*
118	0x76	VK_F7	Klawisz *F7*
119	0x77	VK_F8	Klawisz *F8*
120	0x78	VK_F9	Klawisz *F9*
121	0x79	VK_F10	Klawisz *F10*
122	0x7A	VK_F11	Klawisz *F11*
123	0x7B	VK_F12	Klawisz *F12*
Kody klawiszy z klawiatury numerycznej			
144	0x90	VK_NUMLOCK	*Num Lock*
96	0x60	VK_NUMPAD0	Przy aktywnym *Num Lock*
97	0x61	VK_NUMPAD1	Przy aktywnym *Num Lock*
98	0x62	VK_NUMPAD2	Przy aktywnym *Num Lock*
99	0x63	VK_NUMPAD3	Przy aktywnym *Num Lock*
100	0x64	VK_NUMPAD4	Przy aktywnym *Num Lock*
101	0x65	VK_NUMPAD5	Przy aktywnym *Num Lock*
102	0x66	VK_NUMPAD6	Przy aktywnym *Num Lock*
103	0x67	VK_NUMPAD7	Przy aktywnym *Num Lock*
104	0x68	VK_NUMPAD8	Przy aktywnym *Num Lock*
105	0x69	VK_NUMPAD9	Przy aktywnym *Num Lock*
106	0x6A	VK_MULTIPLY	Klawisz mnożenia (*)
107	0x6B	VK_ADD	Klawisz dodawania (+)
108	0x6C	VK_SEPARATOR	Separator przy nieaktywnym *NL* (-)
109	0x6D	VK_SUBTRACT	Klawisz odejmowania (-)
110	0x6E	VK_DECIMAL	Klawisz kropki (,)
111	0x6F	VK_DIVIDE	Klawisz dzielenia (/)

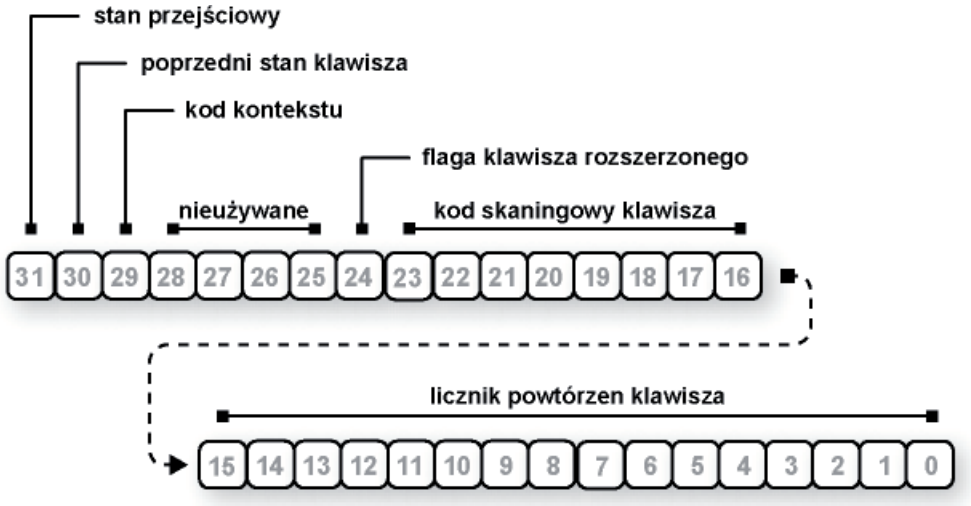
4.3. Wiadomości WM_KEYDOWN i WM_KEYUP

Gdy naciskamy klawisz klawiatury, system generuje wiadomość WM_KEYDOWN, oznaczającą wduszenie jakiegoś klawisza. Parametr wParam wiadomości zawiera kod wciśniętego wirtualnego klawisza. Przykładowy fragment kodu procedury okna obsługujący naciśnięcie klawisza *F1* mógłby wyglądać tak:

```
case WM_KEYDOWN:
            if(wParam == VK_F1)
           {
               // tutaj wstawimy kod wykonujacy jakies dzialanie
               // zwiazane z nacisnieciem klawisza F1
           }
           return 0;
```

Parametr `lParam` dostarcza nieco więcej informacji na temat naciśniętego klawisza. Jest to m.in. licznik powtórzeń klawisza, jego kod skaningowy i kilka mniej przydatnych informacji o stanie klawisza. Spójrz na rysunek 4.1, na którym przedstawiam znaczenie poszczególnych bitów parametru `lParam`.

Rysunek 4.1. *Znaczenie bitów parametru lParam dla wiadomości WM_KEYDOWN*



Bity	Opis

0-15	Młodsze słowo parametru `lParam`, zawiera informacje o liczbie powtórzeń klawisza. Najczęściej jego wartość jest równa 1, gdyż użytkownik szybko uderza klawisz klawiatury, niemal jednocześnie naciskając go i zwalniając. Jednak czasami może się zdarzyć sytuacja, w której użytkownik przytrzyma dany klawisz. Co się wówczas dzieje? System z ustaloną częstotliwością cały czas generuje wiadomości `WM_KEYDOWN`. Do ich parametru `wParam` jest wstawiany wirtualny kod przytrzymanego klawisza. Jeżeli Windows zauważy, że systemowa kolejka wiadomości zapełnia się wiadomościami `WM_KEYDOWN`, dotyczącymi tego samego klawisza, jest to dla niego znak, że któraś z działających aplikacji nie może odpowiednio szybko przetwarzać wiadomości związanych z klawiaturą. Łączy zatem grupę wiadomości `WM_KEYDOWN` w jedną wiadomość, ustawiając jej licznik powtórzeń na odpowiednią wartość.
	Odczytanie liczby powtórzeń klawisza mogłoby być zrealizowane w procedurze okna następująco:
	`case WM_KEYDOWN:`
	`{`
	`unsigned short licznikPowtorzenKlawisza = 0;`
	`licznikPowtorzenKlawisza = LOWORD(lParam);`
	`}`
	`return 0;`
	Dla wiadomości `WM_KEYUP` wartość licznika powtórzeń klawisza zawsze wynosi 1.

Bity	Opis

16 – 23	8-bitowy kod skaningowy klawisza. Możemy go odczytać następująco:
	`case WM_KEYDOWN:`
	`  {`
	`  unsigned char kodSkaningowy = 0;`
	`  kodSkaningowy = (lParam & 0x00FF0000) >> 16;`
	`  }`
	`  return 0;`

24	Jeżeli bit jest ustawiony, klawisz należy do tzw. rozszerzonej grupy klawiszy klawiatury. Rozszerzone klawisze spotykamy przy klawiaturach 101- i 102-klawiszowych. Informację tę odczytujemy następująco:
	`case WM_KEYDOWN:`
	`  {`
	`  bool czyRozszerzony = false;`
	`  czyRozszerzony = lParam & 0x01000000;`
	`  if(czyRozszerzony)`
	`  {`
	`  // dzialanie zwiazane z wykryciem rozszerzonego klawisza`
	`  } else`
	`  {`
	`  // dzialanie zwiazane z wykryciem zwyczajnego klawisza`
	`  }`
	`  }`
	`  return 0;`

25 – 28	Cztery nieużywane bity.

29	Kod kontekstu. Dla wiadomości `WM_KEYDOWN` i `WM_KEYUP` ten bit jest zawsze wyzerowany. Natomiast dla wiadomości `WM_CHAR` jest ustawiony wtedy, gdy podczas naciśnięcia klawisza został przytrzymany klawisz `Alt`, w innym przypadku bit jest wyzerowany. Informację tę możemy odczytać następująco:
	`case WM_KEYDOWN:`
	`  {`
	`  bool czyKodKontekstu = false;`
	`  czyKodKontekstu = lParam & 0x20000000;`
	
	`  if(czyKodKontekstu)`
	`  {`
	`  // dzialanie zwiazane z wykryciem ustawionego bitu kontekstu`
	`  } else`
	`  {`
	`  // dzialanie zwiazane z niewykryciem`
	`  // ustawionego bitu kontekstu`
	`  }`
	`  }`
	`  return 0;`

30	Określa poprzedni stan klawisza. Dla wiadomości `WM_KEYDOWN` i `WM_CHAR` bit jest ustawiony, gdy klawisz był już wciśnięty w chwili wygenerowania wiadomości. Jeżeli klawisz nie był wduszony, bit jest wyzerowany. Dla wiadomości `WM_KEYUP` bit jest zawsze ustawiony. Ustawienie bitu możemy sprawdzić tak:

Bity	Opis

	`case WM_KEYDOWN:`
	`  {`
	`  bool czyBylWduszony = false;`
	`  czyBylWduszony = lParam & 0x40000000;`
	`  if(czyBylWduszony)`
	`  {`
	`  // kod wykonywany, gdy klawisz byl już wcisniety`
	`  } else`
	`  {`
	`  // kod wykonywany, gdy klawisz nie był wcisniety`
	`  }`
	`  }`
	`  return 0;`

31	Bit stanu przejściowego. Dla wiadomości WM_KEYDOWN bit jest zawsze wyzerowany. Dla wiadomości WM_KEYUP jest zawsze ustawiony. Natomiast dla wiadomości WM_CHAR jest wyzerowany, gdy klawisz został wduszony, a ustawiony, kiedy klawisz został zwolniony. Informację o stanie przejściowym klawisza pobieramy następująco:
	`case WM_KEYDOWN:`
	`  {`
	`  bool stanPrzejsciowy = false;`
	`  stanPrzejsciowy = lParam & 0x80000000;`
	`  if(stanPrzejsciowy)`
	`  {`
	`  // kod wykonywany, gdy bit jest ustawiony`
	`  } else`
	`  {`
	`  // kod wykonywany, gdy bit jest wyzerowany`
	`  }`
	`  }`
	`  return 0;`

Jeżeli procedura okna przetworzy wiadomość WM_KEYDOWN, powinna zwrócić do systemu wartość zerową.

Gdy zwalniamy klawisz klawiatury, system generuje wiadomość WM_KEYUP. Parametr wParam wiadomości zawiera kod zwolnionego wirtualnego klawisza, a parametr lParam zawiera takie same dodatkowe informacje o klawiszu, jak wiadomość WM_KEYDOWN. Jeżeli procedura okna przetwarza wiadomość WM_KEYUP, powinna zwrócić do systemu wartość zerową.

4.4. Wiadomość WM_CHAR

Poznałeś już dwie wiadomości związane z klawiaturą, WM_KEYDOWN i WM_KEYUP. Są to tzw. wiadomości klawiszowe. Teraz poznasz jeszcze jedną wiadomość — WM_CHAR — którą często nazywa się wiadomością znakową.

Wiadomości klawiszowe nie pozwalają jednoznacznie określić, czy użytkownik nacisnął dużą, czy małą literę. Aby to stwierdzić, należałoby kontrolować klawisze sterujące stanem klawiatury, czyli takie klawisze jak *Caps Lock*, *Shift* oraz *Alt*. Z uwzględnieniem ich stanu kod skaningowy wciśniętego klawisza powinien zostać przekształcony na odpowiadający mu kod znaku z zestawu znaków aktualnie wybranego w systemie.

System Windows dostarcza funkcję, za której pomocą możemy automatycznie dokonać tego przekształcenia. Jest nią `TranslateMessage`, wywołujemy ją najczęściej wewnątrz pętli wiadomości aplikacji. Funkcja `TranslateMessage` przekształca wiadomość klawiszową (`WM_KEYDWON` i `WM_KEYUP`) na wiadomość znakową (`WM_CHAR`). Niech Ci się nie wydaje, że wiadomość klawiszowa jest zastępowana wiadomością znakową. Funkcja `TranslateMessage` tworzy nową wiadomość i umieszcza ją w kolejce wiadomości aplikacji w taki sposób, że zostanie ona pobrana przez funkcję `GetMessage` lub `PeekMessage` podczas kolejnego przebiegu pętli wiadomość.

Gdy w naturalny sposób uderzamy w klawisz klawiatury oznaczony symbolem *A* (naciskamy go i zaraz puszczamy), system wygeneruje następującą sekwencję wiadomości. Oczywiście zakładamy, że pętla wiadomości aplikacji jest wyposażona w funkcję `TranslateMessage` oraz klawisz *Caps Lock* jest wyłączony, a klawisz *Shift* nie jest przytrzymywany.

- `WM_KEYDOWN` — (kod naciskanego wirtualnego klawisza *A*: 0x41, dziesiętnie 65)
- `WM_CHAR` — (kod ANSI znaku *a*: 0x61, dziesiętnie 97)
- `WM_KEYUP` — (kod zwalnianego wirtualnego klawisza: 0x41, dziesiętnie 65)

Gdy włączymy klawisz *Caps Lock* i naciśniemy klawisz *A* lub przytrzymamy klawisz *Shift* i naciśniemy klawisz *A*, zostanie wygenerowana następująca sekwencja wiadomości. Wybrałem drugą opcję.

- `WM_KEYDOWN` — (kod naciskanego wirtualnego klawisza *Shift*: 0x10, dziesiętnie 16)
- `WM_KEYDOWN` — (kod naciskanego wirtualnego klawisza *A*: 0x41, dziesiętnie 65)
- `WM_CHAR` — (kod ANSI znaku *A*: 0x41, dziesiętnie 65)
- `WM_KEYUP` — (kod zwalnianego wirtualnego klawisza *A*: 0x41, dziesiętnie 65)
- `WM_KEYUP` — (kod zwalnianego wirtualnego klawisza *Shift*: 0x10, dziesiętnie 16)

Zauważ, że klawisz *Shift* nie generuje wiadomości `WM_CHAR`. Dlaczego? To proste, bo nie generuje samodzielnie żadnego tekstowego znaku. Robią to natomiast klawisze *Tab*, *Enter* oraz *Backspace*. Rodzi się więc pytanie, jak identyfikować te znaki? Z pewnością wiesz, że klawiszowi *Tab* odpowiada znak (kod) sterujący `'\t'`, klawiszowi *Enter* — `'\r'`, a klawiszowi *Backspace* — znak `'\b'`. Używamy ich tak samo jak zwykłych klawiszy literowych.

Nie musimy jednak koniecznie przechwytywać zdarzenia naciśnięcia wyżej wymienionych trzech klawiszy za pomocą wiadomości `WM_CHAR`, możemy to równie dobrze zrobić, korzystając z wiadomości `WM_KEYDWON` i identyfikatorów klawiszy wirtualnych `VK_TAB`, `VK_RETURN` i `VK_BACK`.

Parametr wParam wiadomości WM_CHAR nie zawiera kodu wirtualnego klawisza, jest w nim umieszczony kod ANSI znaku, pochodzący z aktualnie wybranego w systemie zestawu znaków.

Oto przykładowy fragment kodu procedury okna przechwytujący wprowadzenie przez użytkownika znaku *A* i znaku tabulacji.

```
case WM_CHAR:
  {
    if(wParam == 'A')
    {
      // dzialanie zwiazane z wprowadzeniem znaku A
    }

    if(wParam == '\t')
    {
      // dzialanie zwiazane z wprowadzeniem znaku tabulacji
    }
  }
  return 0;
```

Parametr lParam zawiera identyczną informację o stanie naciśniętego lub zwolnionego klawisza, jak w przypadku wiadomości WM_KEYDOWN. Jeżeli procedura okna przetwarza wiadomość WM_CHAR, powinna zwrócić do systemu wartość zerową.

4.5. Wiadomości WM_SYSKEYDOWN i WM_SYSKEYUP

Wiadomość WM_SYSKEYDOWN jest wstawiana do kolejki wiadomości, kiedy naciśniemy klawisz *F10* lub przytrzymamy klawisz *Alt* i wciśniemy dodatkowo jakiś inny klawisz. Wiadomości WM_SYSKEYDOWN towarzyszy zawsze pojawienie się wiadomości WM_SYSKEYUP, generowanej w momencie zwolnienia klawisza, który został naciśnięty w czasie, gdy klawisz *Alt* był przytrzymany. Wiadomości WM_SYSKEYDOWN i WM_SYSKEYUP na ogół nie przetwarzamy. Oddajemy je do obsługi przez domyślną procedurę okna — *DefWindowProc*.

Blokowanie tych wiadomości może powodować, że system nie wykona zadań, które wywołujemy za pomocą skrótów klawiaturowych z użyciem klawisza *Alt*. Przykładowo nie zadziała kombinacja *Alt+F4* zamykająca okno.

Z wiadomości WM_SYSKEYDOWN i WM_SYSKEYUP najprawdopodobniej nigdy nie skorzystamy podczas pisania naszych przykładowych programów. Wiedza o nich będzie jednak potrzebna do lepszego zrozumienia treści kolejnych rozdziałów. Parametry wParam i lParam wiadomości WM_SYSKEYDOWN i WM_SYSKEYUP zawierają takie same informacje jak dla wiadomości WM_KEYDOWN i WM_KEYUP. Powinieneś również pamiętać o tym, że gdy je przetwarzasz, procedura okna powinna zwrócić do systemu wartość zerową.

4.6. Niezależne pobieranie informacji o stanie klawiszy

Czy nie wydaje Ci się, że odbieranie informacji o stanie klawiszy wyłącznie przy użyciu wiadomości WM_KEYDOWN, WM_KEYUP i WM_CHAR w procedurze okna jest nieco niewygodne. Czasami może się zdarzyć, że będziemy chcieli niezależnie sprawdzić stan jakiegoś klawisza, np. podczas przetwarzania wiadomości WM_PAINT. Co wtedy? Windows API dostarcza wiele funkcji do obsługi klawiatury, dwie z nich możemy wykorzystać do rozwiązania naszego problemu.

Pierwsza to GetKeyState. Jej nagłówek jest zdefiniowany w pliku *winuser.h* i ma następującą postać:

```
SHORT GetKeyState(int nVirtKey);
```

Jako parametr *nVirtKey* podajemy kod wirtualnego klawisza, którego stan zamierzamy sprawdzić. Jeżeli klawisz jest wciśnięty, funkcja zwraca wartość ujemną (najstarszy bit zwróconej wartości jest ustawiony). Jeżeli zwrócona wartość jest nieujemna, klawisz jest zwolniony (najstarszy bit zwróconej wartości nie jest ustawiony).

Jeżeli najmłodszy bit jest ustawiony, klawisz jest włączony, w przeciwnym razie klawisz jest wyłączony. Stan włączenia i wyłączenia odnosi się do klawiszy sterujących pracą klawiatury, takich jak *Caps Lock*, *Scroll Lock* czy *Num Lock*. Mimo że możemy wyróżnić trzy klawisze tego typu, powinieneś za pomocą funkcji GetKeyState sprawdzać wyłącznie stan klawisza *Caps Lock*.

Znam jeszcze jedną bardzo ważną wskazówkę, dotyczącą pracy z funkcją GetKeyState. Funkcja nie odnosi się bezpośrednio do klawiatury w celu zbadania stanu danego klawisza. Wykorzystuje fakt, że podczas tworzenia każdej wiadomości okna system zapamiętuje aktualny stan klawiatury i na podstawie tego zbioru informacji ocenia stan klawiszy.

Do bezpośredniego badania stanu klawiszy klawiatury służy funkcja GetAsyncKeyState. Podczas wywołania komunikuje się z klawiaturą i zwraca rzeczywisty stan klawisza. Oto jej nagłówek:

```
SHORT GetAsyncKeyState(int nVirtKey);
```

Poprzez parametr *nVirtKey* określamy kod wirtualnego klawisza, którego stanem jesteśmy zainteresowani. Jeżeli wywołanie funkcji się powiedzie, zwracana wartość określa stan klawisza. Gdy jej najbardziej znaczący bit jest ustawiony (wartość jest ujemna), klawisz jest wciśnięty. Kiedy natomiast jest wyzerowany (wartość jest nieujemna), klawisz jest zwolniony.

Jeżeli najmniej znaczący bit jest ustawiony, oznacza to, że klawisz był naciśnięty po poprzednim wywołaniu funkcji GetAsyncKeyState. Jeżeli zwróconą wartością jest zero, znaczy to, że żadne z okien aplikacji (procesu) nie jest aktywne, co automatycznie wiąże się z tym, iż do aplikacji nie docierają żadne wiadomości związane z klawiaturą.

Dodatkowo funkcje GetKeyState i GetAsyncKeyState obsługują trzy kody wirtualnych klawiszy powiązanych z przyciskami myszy (tabela 4.2).

Tabela 4.2. *Kod wirtualnych przycisków myszy*

Kod dziesiętnie	Kod szesnastkowo	Identyfikator	Uwagi o wirtualnym przycisku
1	0x01	VK_LBUTTON	Lewy przycisk myszy
2	0x02	VK_RBUTTON	Prawy przycisk myszy
4	0x04	VK_MBUTTON	Środkowy przycisk myszy

Przykładowe wywołanie funkcji GetAsyncKeyState sprawdzające, czy jest naciśnięty klawisz spacji, mogłoby wyglądać następująco:

```
if(0x8000 & GetAsyncKeyState(VK_SPACE))
{
      // kod wykonywany w odpowiedzi na wykrycie
      // wcisnietego klawisza spacji
}
```

4.7. Projekt: „Klawiatura”

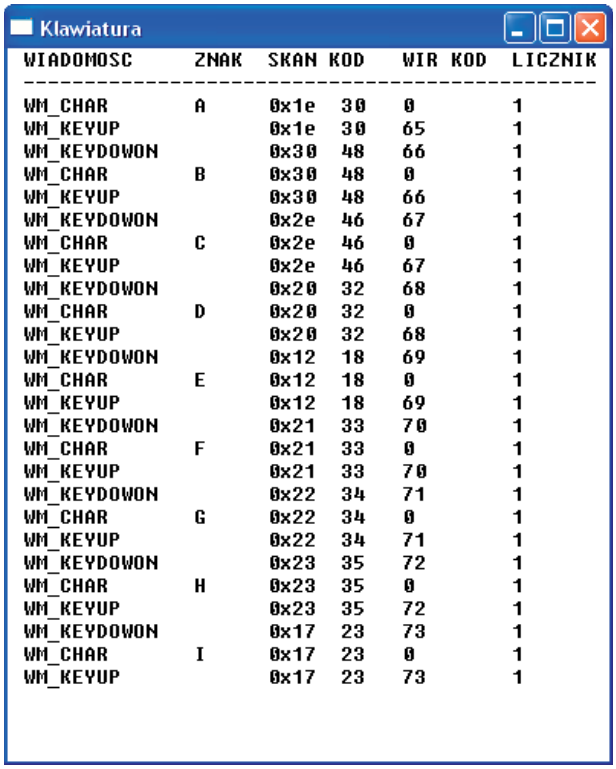
Mówiąc krótko, przykładowy program będzie przechwytywał wiadomości WM_KEYDOWN, WM_KEYUP oraz WM_CHAR, po czym wypisze informacje o naciśniętym klawiszu. Będzie to m.in. jego kod skaningowy, kod wirtualny i licznik powtórzeń.

Budowa programu, w porównaniu do poprzednich przykładowych programów, nie uległa zmianie (listing 4.1). Program wyświetla jedno okno, w którym demonstruje swoje działanie. Poza funkcją WinMain i procedurą okna — *ProceduraOkna* — zdefiniowałem jeszcze tylko jedną funkcję WypiszInformacjeKlawisza. Zanim omówię jej działanie, dokładnie przeanalizuj kod programu. Przypuszczam, że jeśli będziesz miał z nim jakiekolwiek problemy, to będą dotyczyć tylko nowej funkcji. Na rysunku 4.2 przedstawiam okno programu *Klawiatura* po naciśnięciu kilku klawiszy.

Listing 4.1. *Odbieranie informacji o naciśniętych klawiszach (projekt: Klawiatura, plik: WinMain.cpp)*

```
001: // P L I K I   N A G L O W K O W E
002: ////////////////////////////////////////
003: #include <windows.h>
004: #include <stdio.h>
005:
006: // D E K L A R A C J E
007: ////////////////////////////////////////
008: LRESULT CALLBACK ProceduraOkna(
009:   HWND hWnd, UINT message,
010:   WPARAM wParam, LPARAM lParam);
011:
012: void WypiszInformacjeKlawisza(
013:   HWND hwnd, UINT message,
014:   WPARAM wParam, LPARAM lParam);
015:
```

Rysunek 4.2.
Okno programu Klawiatura




```
016: // D E F I N I C J E
017: ////////////////////////////////////////////
018: #define WYSOKOSC_OKNA     500
019: #define SZEROKOSC_OKNA    400
020:
021: char* const g_nazwaKlasyOkna = "StandardoweOkno";
022: BOOL g_czyAktywna = true;
023:
024: // W I N M A I N
025: ////////////////////////////////////////////
026: int WINAPI WinMain(
027:   HINSTANCE hInstance,
028:   HINSTANCE hPrevInstance,
029:   LPSTR lpCmdLine,
030:   int nShowCmd)
031: {
032:   // definiujemy klase okna
033:   WNDCLASSEX wndclassex = {0};
034:
035:   wndclassex.cbSize = sizeof(WNDCLASSEX);
036:   wndclassex.style = CS_VREDRAW | CS_HREDRAW;
037:   wndclassex.lpfnWndProc = ProceduraOkna;
038:   wndclassex.cbClsExtra = 0;
039:   wndclassex.cbWndExtra = 0;
040:   wndclassex.hInstance = hInstance;
041:   wndclassex.hIcon = LoadIcon(0, (LPCTSTR)IDI_APPLICATION);
042:   wndclassex.hCursor = LoadCursor(0, (LPCTSTR)IDC_ARROW);
043:   wndclassex.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);
044:   wndclassex.lpszMenuName = 0;
045:   wndclassex.lpszClassName = g_nazwaKlasyOkna;
046:   wndclassex.hIconSm = LoadIcon(0, (LPCTSTR)IDI_APPLICATION);
047:
048:   // rejestrujemy klase okna
```

```
049:   RegisterClassEx(&wndclassex);
050:
051:   // zmienna na uchwyt okna
052:   HWND uchwytOkna = 0;
053:
054:   // tworzymy okno
055:   uchwytOkna = CreateWindowEx(
056:     0,                                  // styl rozszerzony
057:     g_nazwaKlasyOkna,                   // klasa okna
058:     "Klawiatura",                       // tekst na belce
059:     WS_OVERLAPPEDWINDOW ^ WS_THICKFRAME, // styl okna
060:     100,                                // pozycja X okna
061:     60,                                 // pozycja Y okna
062:     SZEROKOSC_OKNA,                     // szerokosc okna
063:     WYSOKOSC_OKNA,                      // wysokosc okna
064:     0,                                  // okno nadrzedne
065:     0,                                  // uchwyt menu
066:     hInstance,                          // uchwyt aplikacji
067:     0);                                 // wskaznik na dane
068:
069:   // wyswietlamy okno
070:   ShowWindow(uchwytOkna, SW_NORMAL);
071:
072:   // wymuszamy odswiezenie okna
073:   UpdateWindow(uchwytOkna);
074:
075:   // wiadomosc
076:   MessageBox(
077:     uchwytOkna,
078:     "Zamknij okno komunikatu i \n"
079:     "nacisnij dowolny klawisz klawiatury.",
080:     "informacja", MB_OK | MB_ICONINFORMATION);
081:
082:   // petla wiadomosci
083:   MSG msg;
084:   for(;;)
085:   {
086:     if(PeekMessage(&msg, NULL, 0, 0, PM_REMOVE) != 0)
087:     {
088:       if(msg.message == WM_QUIT) break;
089:       TranslateMessage(&msg);
090:       DispatchMessage(&msg);
091:     }
092:
093:     if(g_czyAktywna == false)
094:       WaitMessage();
095:     else ; // po prostu nic nie robimy
096:   }
097:
098:   return (int)msg.wParam;
099: }
100:
101: // D E F I N I C J E   F U N K C J I
102: ////////////////////////////////////////
103: LRESULT CALLBACK ProceduraOkna(
104:   HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
105: {
```

```
106:   switch(message)
107:   {
108:     case WM_PAINT:
109:       ValidateRect(hWnd, 0);
110:     return 0;
111:
112:     // w przypadku przechwycenia jednej z wiadomosci
113:     // klawiatury wywolujemy funkcje WypiszInformacjeKlawisza
114:     case WM_CHAR:
115:     case WM_KEYDOWN:
116:     case WM_KEYUP:
117:       WypiszInformacjeKlawisza(
118:         hWnd, message, wParam, lParam);
119:       return 0;
120:
121:     case WM_CLOSE:
122:       if(IDYES ==
123:         MessageBox(hWnd, "Czy na pewno zamknac okno?",
124:           "Pytanie", MB_YESNO | MB_ICONQUESTION))
125:         return DefWindowProc(hWnd, message, wParam, lParam);
126:       else
127:         return 0;
128:
129:     case WM_ACTIVATE:
130:       if(LOWORD(wParam) == WA_INACTIVE)
131:       {
132:         g_czyAktywna = false;
133:       } else
134:       {
135:         g_czyAktywna = true;
136:       }
137:       return 0;
138:
139:     case WM_DESTROY:
140:       PostQuitMessage(0);
141:       return 0;
142:       } // koniec switch
143:
144:   return DefWindowProc(hWnd, message, wParam, lParam);
145: }
146:
147: void WypiszInformacjeKlawisza(
148:   HWND hwnd, UINT message,
149:   WPARAM wParam, LPARAM lParam)
150: {
151:   // definicja niezbednych zmiennych
152:   // automatycznych i statycznych
153:   int wysokoscWierszaTekstu = 0;
154:   static unsigned short ileWierszy = 2;
155:   char buforWierszaTekstu[80] = {'\0'};
156:
157:   // naglowek tabeli
158:   static char naglowekTabeli[60] =
159:     {"WIADOMOSC     ZNAK  SKAN KOD   WIR KOD  LICZNIK"};
160:   static char podkreslenie[60] =
161:     {"------------------------------------------------"};
162:
```

```
163:   // pobieramy uchwyt kontekstu urzadzenia okna
164:   HDC hdc = 0;
165:   hdc = GetDC(hwnd);
166:
167:   // wybieramy czcionke o stałej szerokosci znaku
168:   HGDIOBJ gdiObj = 0;
169:   gdiObj = SelectObject(hdc, GetStockObject(SYSTEM_FIXED_FONT));
170:
171:   // pobieramy wysokosc pojedynczego wiersza tekstu
172:   // zapisanego za pomoca czcionki aktualnie wybranej w kontekscie
173:   // urzadzenia
174:   TEXTMETRIC tm = {0};
175:
176:   GetTextMetrics(hdc, &tm);
177:   wysokoscWierszaTekstu =
178:     tm.tmHeight + tm.tmExternalLeading;
179:
180:   // sprawdzamy, czy wyswietlany wiersz jest
181:   // pierwszym wierszem tekstu w oknie oraz czy
182:   // okno nie zostalo calkowicie zapisane, gdy jeden z
183:   // warunkow zostanie spelniony, czyscimy obszar roboczy okna,
184:   // malujac go na biało, po czym wyswietlamy naglowek tabeli
185:   if((ileWierszy == 2) ||
186:     ((ileWierszy * wysokoscWierszaTekstu) >
187:       WYSOKOSC_OKNA - wysokoscWierszaTekstu * 3))
188:   {
189:     // pobieramy rozmiar obszaru roboczego okna
190:     RECT rect = {0};
191:     GetClientRect(hwnd, &rect);
192:
193:     // malujemy go na bialo
194:     FillRect(hdc, &rect, (HBRUSH)GetStockObject(WHITE_BRUSH));
195:
196:     // ustalmy pozycje pierwszego wiersza tekstu
197:     ileWierszy = 2;
198:
199:     // wyswietlamy naglowek tabeli
200:     TextOut(
201:       hdc, 10, 0,
202:       naglowekTabeli, (int)strlen(naglowekTabeli));
203:
204:     TextOut(
205:       hdc, 10, wysokoscWierszaTekstu,
206:       podkreslenie, (int)strlen(podkreslenie));
207:
208:     // informacja dzwiekowa
209:     MessageBeep(MB_OK);
210:   }
211:
212:   // przygotowujemy tekst zawierajacy informacje o odebranej
213:   // wiadomosci i stanie klawisza
214:   sprintf_s(
215:     buforWierszaTekstu,
216:     sizeof(buforWierszaTekstu),
217:     "%-14s%-6c%#-6x%-5i%-9i%-7i",
218:
219:     // identyfikator wiadomosci
```

```
220:     (message == WM_CHAR) ? "WM_CHAR" :
221:     (message == WM_KEYDOWN) ? "WM_KEYDOWON" : "WM_KEYUP",
222:
223:     // znak
224:     (message == WM_CHAR) ? (char)wParam : ' ',
225:
226:     // kod skaningowy klawisza, szesnastkowo i dziesietnie
227:     (lParam & 0x00FF0000) >> 16,
228:     (lParam & 0x00FF0000) >> 16,
229:
230:     // kod wirtualnego klawisza
231:     (message != WM_CHAR) ? wParam : 0,
232:
233:     // licznik powtorzen klawisza
234:     (lParam & 0x0000FFFF)
235:
236:   ); // <- koniec wywołania funkcji sprintf_s
237:
238:   // wyswietlamy tekst
239:   TextOut(
240:     hdc, 10, ileWierszy * wysokoscWierszaTekstu,
241:     buforWierszaTekstu, (int)strlen(buforWierszaTekstu));
242:
243:   // ustalamy pozycje nastepnego wiersza
244:   ileWierszy++;
245:
246:   // przywracamy pierwotne ustawienia kontekstu urzadzenia,
247:   // zatwierdzamy caly obszar roboczy okna i zwalniamy uchwyt
248:   // kontekstu urzadzenia
249:   SelectObject(hdc, gdiObj);
250:   ValidateRect(hwnd, 0);
251:   ReleaseDC(hwnd, hdc);
252: }
```

Gdy uruchomisz program i zaczniesz naciskać klawisze klawiatury, zobaczysz wypisane w postaci tabeli informacje o przechwyconej wiadomości klawiszowej lub znakowej. Każdy wiersz tabeli zawiera identyfikator wiadomości, wprowadzony znak (tylko w przypadku wiadomości `WM_CHAR`), kod skaningowy klawisza w postaci szesnastkowej i dziesiętnej, kod wirtualnego klawisza oraz licznik jego powtórzeń.

Pierwszą rzeczą, jaką powinieneś zauważyć, jest to, że program nie odświeża zawartości okna w odpowiedzi na wiadomość `WM_PAINT`. Jest to działanie zamierzone, nie chciałem dodatkowo komplikować kodu tylko ze względów czysto estetycznych. Jeżeli chciałbyś to zmienić, powinieneś gdzieś zapamiętywać informację o ostatnio przechwyconych wiadomościach klawiatury. Lista powinna być na tyle długa, aby podczas odrysowywania okna mogła pokryć jego cały obszar roboczy. Chciałbym również, abyś na tym przykładzie spostrzegł, jak ważne jest odświeżanie zawartości okna dla zwykłego śmiertelnika korzystającego z Twojego programu. Widząc puste lub częściowe odrysowane okno, nie wiedziałby, co z nim zrobić. I na pewno nie winiłby za to siebie, to Ty byłbyś jego wrogiem numer jeden.

Funkcja `WypiszInformacjeKlawisza` jest wywoływana w odpowiedzi na przechwycenie wiadomości klawiatury. Przekazujemy do niej uchwyt okna, identyfikator wiadomości i jej parametry.

Działanie funkcji rozpoczyna się od zdefiniowania zmiennej `wysokoscWierszaTekstu`. Wykorzystamy ją do zapamiętania wysokości pojedynczego wiersza wyświetlanego tekstu. Druga ważna zmienna to `ileWierszy`. Będzie przechowywać liczbę wypisanych wierszy tekstu (pierwsze dwa wiersze zarezerwowane są dla nagłówka tabeli).

Trzecia zmienna to tablica, do której funkcja będzie generować tekst z informacjami o odebranej wiadomości okna. Kolejne dwie zmienne to tablice zawierające tekst tworzący nagłówek tabeli.

Po pobraniu uchwytu kontekstu urządzenia podłączamy do niego systemową czcionkę o stałej szerokości znaku, co pozwoli formować informacje wstawiane do tabeli w czytelne kolumny.

Następnie funkcja pobiera wysokość wiersza tekstu zapisanego za pomocą nowej czcionki. Jeżeli żaden wiersz tekstu nie został wypisany lub cały obszar okna został zapisany, funkcja czyści na biało obszar okna. Następnie wyświetla sam nagłówek tabeli i odtwarza za pomocą funkcji `MessageBeep` prosty dźwięk, identyczny z tym, który służy systemowi do powiadamia o pomyślnym zakończeniu jakiejś operacji.

Nagłówek funkcji `MessageBeep` ma postać:

```
BOOL MessageBeep(UINT uType);
```

Jako parametr *uType* możemy podać jeden z wymienionych poniżej określających rodzaj odtwarzanego dźwięku.

Identyfikator	Opis
`0xFFFFFFFF`	Dźwięk z komputerowego głośniczka
`MB_OK`	Systemowe OK
`MB_ICONHAND`	Systemowa pomoc
`MB_ICONQUESTION`	Systemowe pytanie
`MB_ICONEXCLAMATION`	Systemowy wyjątek (błąd)
`MB_ICONASTERISK`	Systemowy odsyłacz

W wyniku wywołania funkcji zostaje zainicjalizowane asynchroniczne odgrywanie wybranego dźwięku, po czym funkcja natychmiast zwraca sterowanie. Jeżeli wywołanie powiedzie się, zwrócona wartość jest niezerowa (`TRUE`). W przypadku niepowodzenia funkcja zwraca zero (`FALSE`). Najczęstszą przyczyną nieodegrania dźwięku jest zajęte urządzenie audio.

Wracamy do omawiania funkcji `WypiszInformacjeKlawisza`. Jej kolejny krok polega na „zmontowaniu” tekstu z informacją o klawiszu. Jeżeli nie pamiętasz szczegółów działania funkcji `sprintf_s`, powinieneś zajrzeć do podpunktu „Funkcje rand, sprintf_s, Sleep i coś jeszcze” w 2. rozdziale.

Dodatkowo niezrozumiałą rzeczą może być działanie trójargumentowego operatora ?:. Wyjaśnię je, gdyż początkujący programiści C++ bardzo często zapominają o jego istnieniu, przez co tracą bardzo pomocne narzędzie.

Przed znakiem zapytania stawiamy wyrażenie logiczne; jeżeli jest prawdziwe, wykonywany jest fragment kodu między znakiem zapytania a dwukropkiem. Jeżeli wyrażenie jest nieprawdziwe, zostaje wykonany kod znajdujący się za dwukropkiem. Przyjrzyj się poniższemu zapisowi.

```
01:  unsigned short ileWlosow = 0;
02:  (2 > 20) ? ileWlosow = 500 : ileWlosow = 3000;
```

Jaka wartość zostanie przypisana zmiennej `ileWlosow`? Tak! Masz rację będzie to wartość `3000`, bo wyrażenie `2 > 20` jest fałszywe (`FALSE`).

Po wyświetleniu tekstu funkcja ustala pozycję następnego wiersza tekstu, po czym przywraca pierwotne ustawienia kontekstu urządzenia okna, zatwierdza obszar roboczy okna i zwalnia uchwyt kontekst urządzenia.