3

MYSZ I KLAWIATURA

Wiele rzeczy wymyślono po to,
aby nie trzeba było dużo myśleć.
Regedit

Tytułowe dwa **urządzenia wejściowe** (ang. *input devices*) są najintensywniej wykorzystywanymi środkami do komunikacji użytkownika z komputerem. Historycznie starsza jest klawiatura, jednak obecnie oba te sprzęty wzajemnie się uzupełniają, a obsługa większości dobrych programów może być realizowana przy pomocy każdego z nich.
Powstało naturalnie mnóstwo innych urządzeń wejściowych, do których należą choćby joysticki czy tablety graficzne. Coraz więcej mówi się też o sterowaniu aplikacjami za pomocą komend głosowych. Wydaje się jednak, że nawet jeśli ten nowe interfejsy komunikacyjne zostaną w przyszłości udoskonalone, to tradycyjne klawiatury i myszki (albo ich zastępniki, np. *trackballe*) nigdy nie odejdą całkiem do lamusa. Praca z nimi jest po prostu szybka i wygodna, a nadto dyskretna - i chyba nie zmienią tego żadne nadchodzące nowniki. Klawiatury zyskają oczywiście więcej klawiszy, myszki - więcej przycisków i rolek, ale zasadnicze przeznaczenie i wykorzystanie obu tych urządzeń będzie przez cały czas takie same.

Skoro więc są one dzisiaj podstawową metodą porozumienia się użytkownika z komputerem, nowoczesny system operacyjny w rodzaju Windows musi zapewniać właściwą obsługę klawiatury i myszy. I rzeczywiście, środowisko aplikacji rodem z Microsoftu daje bodaj wszystko, co jest potrzebne, by programista mógł zaoferować użytkownikom swych produktów pełną współpracę z możliwościami tych dwóch kluczowych urządzeń. Ta kooperacja jest realizowana w ramach Windows API, którego część za to odpowiedzialną poznamy bliżej w tym oto rozdziale.

Obsługa myszy

Mysz jest **urządzeniem wskazującym** (ang. *pointing device*), którego przeznaczeniem jest współpraca z graficznym interfejsem użytkownika. Nie ma ono większego zastosowania w konsoli tekstowej, gdzie prym cały czas wiedzie (i musi wieść) klasyczna klawiatura.

Każda osoba posługująca się komputerem wie oczywiście, w jaki sposób działa myszka. Nie wszyscy jednak wiedzą, że nie jest ona jedynym możliwym urządzeniem, za pomocą którego można sterować kursorem na ekranie. Do innych należy chociażby *trackball*; jego obsługa polega na umiejętnym poruszaniu kulką, której obroty powodują ruch kursora na ekranie. Urządzenie to ma sporą zaletę w postaci braku konieczności posiadania specjalnej podkładki i dlatego jest szczególnie często wykorzystywane w komputerach przenośnych.

Słowo ‘trackball’ weszło już na dobre do słownika komputerowego i nikt już nawet nie próbuje go tłumaczyć. Ale jeszcze kilka lat można było okazjonalnie spotkać wyjątkowo idiotyczne określenie dla tego urządzenia: otóz nazywano je kotem, chcąc je rzekomo

odróżnić od standardowej myszki. Wyjaśnienie to jest raczej dziwne, bo chociaż komputerowa mysz może faktycznie budzić skojarzenia z pospolitym gryzoniem, to przecież *trackball* nie różni się od niej prawie wcale. Słusznie więc zdaje się, że obecność jednego komputerowego zwierzęcia w zupełności nam wystarczy.

Fotografia 3 i 4. Komputerowe urządzenia wskazujące: myszka oraz *trackball*
(fotografie pochodzą z serwisu internetowego firmy Logitech)

Jako przyjazny system operacyjny Windows zawiera naturalnie odpowiednią obsługę urządzeń wskazujących - niezależnie od tego, czym one są. W WinAPI przyjęło się aczkolwiek nazywać je wszystkie myszami, ponieważ tak jest po prostu wygodniej. My również będziemy tak wobec tego czynić.

W tym podrozdziale zajmiemy się więc tą częścią Windows API, która umożliwia programom okienkowym wykorzystanie obecności myszy. Poznamy wpierw wszystkie najważniejsze komunikaty o zdarzeniach myszy oraz reguły ich otrzymywania przez okna. Później nauczymy się odczytywać stan myszy bezpośrednio, a nawet symulować jego zmianę. Na sam koniec zostawimy sobie odczytywanie różnorakich parametrów myszy.

Zdarzenia myszy

System Windows posługuje się łącznie kilkudziesięcioma (!) komunikatami o zdarzeniach pochodzących od myszy. Spośród tej mnogości najważniejszych jest dla nas kilkanaście, informujących przede wszystkim o wciśnięciu lub puszczeniu któregoś z przycisków myszy, ruchu kursora lub też zmianie pozycji rolki (jeżeli jest obecna). Tymi właśnie komunikatami zajmiemy się w tej sekcji.

One wszystkie posiadają przynajmniej jedną przyjemną cechę, związaną ze swymi parametrami `wParam` i `lParam`. Otóż znaczenie tych parametrów jest dla wymienionych zdarzeń zawsze takie samo: zmienne te zawierają mianowicie aktualną **pozycję kursora myszy** oraz informację o tym, czy **pewne klawisze** są w danej chwili wciśnięte. Pierwsza z tych danych zawarta jest w `lParam`. Pozioma i pionowa współrzędna kursora jest w niej zapisana w dolnym i górnym słowie tej 32-bitowej wartości. Aby je uzyskać, możemy zatem posłużyć się poznanymi makrami `LOWORD()` oraz `HIWORD()`. Windows API deklaruje też dwa bardziej wyspecjalizowane makra:

```
nX = GET_X_LPARAM(lParam);
nY = GET_Y_LPARAM(lParam);
```

Jak wskazują ich nazwy, służą one właśnie do pobrania pozycji kursora z parametru `lParam`. Aby z nich skorzystać, trzeba jeszcze dołaczyć nagłówek *windowsx.h*:

```
#include <windowsx.h>
```

Istnieje również makro `MAKEPOINTS()`, które zmienia `lParam` w strukturę `POINTS` - bardzo podobną do poznanej wcześniej `POINT`, ale z polami typu `SHORT` (16-bitowymi).

Z kolei `wParam` zawiera nieco inną informację[120]. Jest to bowiem kombinacja bitowa pewnych flag, które określają stan kilku ważnych klawiszy na klawiaturze oraz przycisków myszy. Można tam znaleźć wartości stałych wymienionych w tabeli:

stała	***klawisz***
`MK_CONTROL`	*Ctrl*
`MK_SHIFT`	*Shift*
`MK_LBUTTON`	lewy przycisk myszy
`MK_MBUTTON`	środkowy przycisk myszy
`MK_RBUTTON`	prawy przycisk myszy

Tabela 42. Stałe parametru `wParam` komunikatów myszy, określające wciśnięte przy ich okazji klawisze

Jako że są to flagi bitowe, `wParam` może mieć ustawioną więcej niż jedną taką stałą naraz. Sprawdzenia, czy jakaś flaga jest tu zawarta, należy dokonywać za pomocą odpowiedniej operacji bitowej:

```
if ((wParam & stała) /* != 0 */)
{
   // stała jest ustawiona
}
```

Przykładowo, aby dowiedzieć się, czy w momencie zajścia zdarzenia myszy wciśnięty był klawisz *Shift*, trzeba posłużyć się warunkiem:

```
if (wParam & MK_SHIFT)
```

Więcej informacji o flagach bitowych możesz znaleźć w Dodatku B, *Reprezentacja danych w pamięci*.

To wszystko, jeżeli chodzi o parametry komunikatów myszy. Teraz wypadałoby przyjrzeć się bliżej każdemu z tych ważnych zdarzeń.

Kliknięcia przycisków

Ewolucja komputerowych myszek, jaka następowała przez ostatnie dekady, polegała w dużej mierze na dodawania kolejnych przycisków. Pierwsze urządzenia tego typu posiadały tylko jeden taki przycisk, później standardem stały się dwa. Dzisiaj minimalna liczba przycisków, potrzebna dla wygodnej pracy z każdą aplikacją, to trzy; jednak wiele myszek posiada teraz nawet szerszy ich asortyment, z których wszystkie są często konfigurowalne.

Liczbę dostępnych przycisków myszy można pobrać za pomocą wywołania `GetSystemMetrics(SM_CMOUSEBUTTONS)`.

Wszystkie wersje Windows szeroko używane w chwili obecnej zapewniają standardową obsługę dla trzech przycisków myszy:

[120] W przypadku komunikatu `WM_MOUSEWHEEL` informacja ta zajmuje tylko młodsze słowo z `wParam` (`LOWORD(wParam)`), gdyż starsze jest przeznaczone na dane o pozycji rolki. Podobnie jest też z trzema komunikatami `WM_XBUTTON*` w Windows 2000/XP.

- lewego, używanego zdecydowanie najczęściej. Kliknięcia tym przyciskiem są standardową metodą wyboru elementów interfejsu użytkownika, jak na przykład przycisków czy opcji menu
- prawego, służącego głównie do pokazywania menu podręcznego (ang. *context menu*) oraz specjalnych typów przeciągania (ang. *dragging*) obiektów
- środkowego, którego działania jest zwykle zależne od aplikacji. Dla przykładu, w programie 3ds max służy on między innymi do przewijania długich pasków narzędzi; fani gry Saper zapewne znają zastosowanie tego przycisku w ich ulubionej grze

Myszy dwuprzyciskowe symulują środkowy przycisk za pomocą jednoczesnego wciśnięcia swego lewego i prawego przycisku.

Każdy z tych trzech przycisków myszy może z kolei generować trójkę związanych ze sobą zdarzeń:

- wciśnięcie przycisku (ang. *button down*)
- zwolnienie przycisku (ang. *button up*)
- dwukrotne kliknięcie (ang. *double click*)

Zauważmy, że Windows **nie generuje** oddzielnego komunikatu dla **pojedynczego kliknięcia** danym przyciskiem myszy. Takie kliknięcie jest bowiem interpretowane jako dwa zdarzenia: wciśnięcia i zwolnienia przycisku, następujące po sobie.

Nazwy komunikatów

Trzy przyciski i trzy możliwe do wystąpienia akcje… Nie trzeba być specem od matematyki, by wywnioskować, że łącznie daje to nam **9 komunikatów** o zdarzeniach myszy. Każdemu z nich odpowiada oczywiście pewna stała, której nazwę można łatwo zbudować wedle następującego schematu:

```
WM_przyciskBUTTONakcja
```

Etykiety `przycisk` i `akcja` powinny być w nim zastąpione fragmentami nazw, odnoszącymi się do jednego z przycisków oraz do rodzaju występowanego zdarzenia. Możliwe warianty w obu tych kwestiach przedstawiają dwie poniższe tabelki:

przycisk	**przycisk myszy**
L	lewy
M	środkowy
R	prawy

akcja	**zdarzenie przycisku**
DOWN	wciśnięcie przycisku
UP	zwolnienie przycisku
DBLCLK	dwukrotne kliknięcie

Tabela 43 i 44. Fragmenty nazw komunikatów o zdarzeniach myszy

Budując z tych informacji wszystkie możliwe nazwy komunikatów, otrzymamy dziewięć odpowiadających im stałych:

zdarzenie → ***przycisk ↓***	***wciśnięcie przycisku***	***zwolnienie przycisku***	***dwukrotne kliknięcie***
lewy	WM_LBUTTONDOWN	WM_LBUTTONUP	WM_LBUTTONDBLCLK
środkowy	WM_MBUTTONDOWN	WM_MBUTTONUP	WM_MBUTTONDBLCLK
prawy	WM_RBUTTONDOWN	WM_RBUTTONUP	WM_RBUTTONDBLCK

Tabela 45. Nazwy komunikatów o zdarzeniach myszy

Windows 2000 i XP posiada też wbudowaną obsługę ewentualnych dwóch dodatkowych przycisków myszy, oznaczanych jako X1 i X2. Związana jest z nimi aczkolwiek tylko trójka stosownych komunikatów (zamiast sześciu); obydwa przyciski są bowiem

rozróżnianie przez wartość górnego słowa `wParam`.
Wszystkie informacje na temat możesz naturalnie znaleźć w MSDN przy opisach komunikatów WM_XBUTTONDOWN, WM_XBUTTONUP i WM_XBUTTONDBLCLK.

Poznamy obecnie nieco bliżej wszystkie wymienione tu komunikaty.

Pojedyncze kliknięcia

Jak już nadmieniłem, Windows nie wyróżnia żadnego komunikatu do informowania o pojedynczych kliknięciach przycisku myszy. Wysyła za to powiadomienia o wciśnięciu oraz puszczeniu każdego z przycisków.

Szczególnie komunikaty o przyciśnięciach są dla nas interesujące. To właśnie ich używa się, by reagować na kliknięcia w obszarze klienta okna.
Wśród tych zdarzeń zdecydowanie najczęściej jest z kolei wykorzystywane zawiadomienie `WM_LBUTTONDOWN`. Jest to bowiem prosta droga reagowania na kliknięcia myszy dotyczące okna. Na ten komunikat odpowiadaliśmy chociażby w przykładowym programie `TaskbarHider` z poprzedniego rozdziału. Naciśnięcie lewego przycisku myszy powodowało tam pokazywanie lub ukrywanie systemowego paska zadań.

Mówiąc na temat komunikatów o wciśnięciu lub zwolnieniu przycisków myszy trzeba jeszcze zwrócić uwagę na pewien trudno uchwytny fakt. Otóż wystąpienie `WM_?BUTTONDOWN` wcale nie musi pociągać za sobą późniejszego pojawienia się `WM_?BUTTONUP`. Jeżeli bowiem użytkownik, wcisnąwszy przycisk, przeniesie kursor poza obszar klienta okna programu, wówczas komunikat o puszczeniu przycisku nie trafi do tego okna.
Niekiedy bywa to zachowaniem niepożądanym, ale na szczęście Windows oferuje możliwość jego zmiany. Poznamy ją w jednym z następnych paragrafów.

Dwukrotne kliknięcia

Zdarzenie podwójnego kliknięcia występuje wtedy, gdy nastąpi dwukrotne, szybkie wciśnięcie i zwolnienie jednego z przycisków myszy (nie tylko lewego). Musi to nastąpić w odpowiednio krótkim czasie oraz przy stosunkowo niewielkiej lub żadnej zmianie pozycji kursora.

Wiele modeli myszek umożliwia też przypisanie akcji dwukrotnego kliknięcia lewym przyciskiem do jednego z dodatkowych przycisków myszy. Windows traktuje takie emulowane kliknięcia identycznie jak normalne, jednak z wiadomych względów nie stosują się do nich wymienione wyżej ograniczenia.

Restrykcyjność tych ograniczeń można oczywiście regulować i dopasować do swoich potrzeb. Maksymalny interwał czasu jest ustawiany w Panelu Sterowania, zaś tolerowane przesunięcie myszy przy pomocy narzędza Tweak UI.
Oba te parametry systemowe można też zmienić programowo poprzez Windows API - tego również nauczymy się w tym podrozdziale.

Wróćmy jednak do samych komunikatów o dwukrotnych kliknięciach. Od razu trzeba powiedzieć na ich temat dwie ważne kwestie.
Po pierwsze, żadnemu oknu nie jest bezwarunkowo dane odbieranie tych komunikatów. Być może (mam nadzieję :D) pamiętasz, że w grę wchodzą tu style klasy okna. Uściślając to stwierdzenie, trzeba powiedzieć, iż:

Tylko okna, których klasy zawierają **styl** **`CS_DBLCKLS`**, odbierają komunikaty o **dwukrotnych kliknięciach** przyciskami myszy.

Tak więc ażeby reagować na te zdarzenia, należy wpierw ustawić odpowiedni styl klasy okna - na przykład w ten sposób:

```
KlasaOkna.style = CS_HREDRAW | CS_VREDRAW | CS_DBLCLKS;
```

Jeżeli bowiem nie zrobimy tego, nasze okno nie otrzyma żadnego z komunikatów `WM_?BUTTONDBLCLK`.

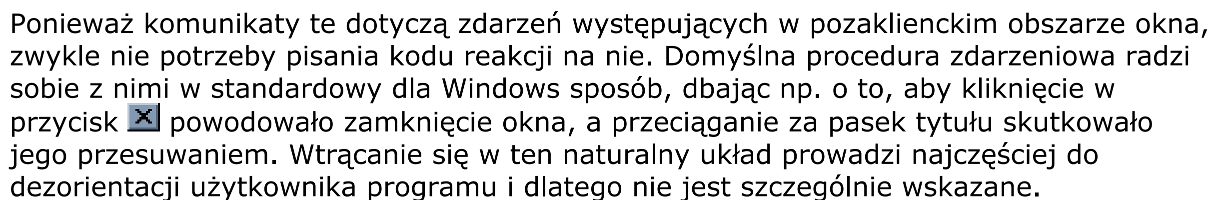
Druga kwestia dotyczy rzeczywistej sekwencji komunikatów, jakie dostaje okno w przypadku wystąpienia dwukrotnego kliknięcia. Nie jest tak, że `WM_?BUTTONDBLCLK` zastępuje informacje o pojedynczych kliknięciach, które składają się w sumie na to podwójne. Prawdziwa kolejność komunikatów wygląda bowiem tak:

```
// nieustawiony styl CS_DBLCLKS        // ustawiony styl CS_DBLCLKS
WM_?BUTTONDOWN                          WM_?BUTTONDOWN
WM_?BUTTONUP                            WM_?BUTTONUP
WM_?BUTTONDOWN                          WM_?BUTTONDBLCLK     // to ten! :)
WM_?BUTTONUP                            WM_?BUTTONUP
```

Widać, że `WM_?BUTTONDBLCLK` zastępuje drugi z komunikatów `WM_?BUTTONDOWN`. Pierwsza notyfikacja o wciśnięciu przycisku myszy trafia jednak do okna i jest przetwarzana tak, jak zwykłe pojedyncze kliknięcie. Dopiero potem do okna dociera również `WM_?BUTTONDBLCLK`, interpretowane jako podwójne naciśnięcie przycisku.
Z tego powodu ważne jest, aby kod obsługi dwukrotnego kliknięcia **nie był całkiem inny** od reakcji na pojedyncze wciśnięcie przycisku myszy. Powinien raczej **uzupełniać** ją; dobrym przykładem jest tu Eksplorator Windows. W programie tym pojedyncze kliknięcie na ikonę pliku powoduje jego zaznaczenie, zaś podwójne poleca otwarcie pliku w domyślnej aplikacji. Akcja otwarcia jest więc uzupełnieniem akcji zaznaczenia.

Komunikaty spoza obszaru klienta

Dziewiątka opisanych tu komunikatów oraz `WM_MOUSEMOVE`, który zostanie omówione za chwilę, powiadamia okno o zdarzeniach myszy, zachodzacych wewnątrz jego obszaru klienta. Takie zdarzenia mogą jednakże zachodzić także poza nim; Windows informuje o nich poprzez dziesięć odmiennych komunikatów[121].
Odpowiadają one dokładnie każdemu ze zdarzeń klienckich i mają nawet podobne nazwy. Dodany jest w nich jedynie przedrostek `NC`, przez co ich stałe to na przykład `WM_NCLBUTTONDOWN` czy `WM_NCMOUSEMOVE`.



Ponieważ komunikaty te dotyczą zdarzeń występujących w pozaklienckim obszarze okna, zwykle nie potrzeby pisania kodu reakcji na nie. Domyślna procedura zdarzeniowa radzi sobie z nimi w standardowy dla Windows sposób, dbając np. o to, aby kliknięcie w przycisk [X] powodowało zamknięcie okna, a przeciąganie za pasek tytułu skutkowało jego przesuwaniem. Wtrącanie się w ten naturalny układ prowadzi najczęściej do dezorientacji użytkownika programu i dlatego nie jest szczególnie wskazane.

Jeden z pozaklienckich komunikatów myszy nie ma swego odpowiednika w zdarzeniach obszaru klienta. Tym komunikatem jest `WM_NCHITTEST`.
Zdarzenie to jest interesujące również z innego powodu. Otóż można je uważać za przyczynek wszystkich pozostałych zdarzeń myszy. Windows poprzedza nim każdy komunikat o zmianie stanu komputerowego gryzonia, wysyłając do okna razem z nim także aktualną pozycję kursora. Procedura zdarzeniowa okna analizuje te dane i na ich podstawie stwierdza, którego miejsca okna dotyczy dane zdarzenie myszy. W ten sposób

[121] Lub raczej poprzez trzynaście komunikatów, jeżeli uwzględnić także powiadomienia o stanie dodatkowych prxycisków myszy (`WM_[NC]XBUTTON*`).

rozróżniana jest potrzeba wysłania komunikatu klienckiego lub pozaklienckiego, zaś system Windows wie, czy kliknięto np. w pasek tytułu czy też wciśnięto przycisk będąc już w obszarze klienta okna.
Zadanie rozróżniania tych wszystkich możliwości przypada najczęściej domyślnej procedurze zdarzeniowej `DefWindowProc()`, jako że zazwyczaj nie zajmujemy się komunikatem `WM_NCHITTEST`. Obsłużenie go może jednak pozwolić na swego rodzaju oszukanie systemu - tak, by „myślał” on, że zainstniałe zdarzenie (np. kliknięcie) dotyczy innego fragmentu okna niż w rzeczywistości.
Typowym zastosowaniem tej techniki jest umożliwienie przesuwania okna poprzez przeciąganie za jego obszar klienta (a nie tylko za pasek tytułu). Prezentuje to przykładowy program `ClientMove`.

Jeżeli jednak chcesz napisać kod obsługi tych zdarzeń i jednocześnie nie przeszkadzać systemowi w normalnej reakcji na nie, możesz samodzielnie wywoływać domyślną procedurę `DefWindowProc()`. Przykładowa reakcja na `WM_NCLBUTTONDOWN` może więc wyglądać tak:

```
case WM_NCLBUTTONDOWN:
{
   // twój kod

   return DefWindowProc(hWnd, uMsg, wParam, lParam);
}
```

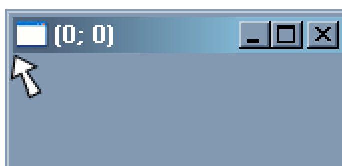
Powinieneś też pamiętać, że w przypadku komunikatów spoza obszaru klienta współrzędne kursora podane w `lParam` są liczone **względem ekranu**, a nie obszaru klienta okna.

Ruch myszy

Następnym z komunikatów myszy, któremu poświęcimy swoją uwagę, jest `WM_MOUSEMOVE`.

System Windows wysyła go do okna, gdy kursor przelatuje nad jego obszarem klienta - także wtedy, kiedy samo okno jest nieaktywne. Otrzymanie tego zdarzenia wskazuje, że pozycja strzałki myszy uległa jakiejś zmianie. Okno jest informowane o każdej takiej zmianie - nawet, jeśli było to tylko przesunięcie kursora o jeden jedyny piksel. `WM_MOUSEMOVE` powiadamia bowiem o **ruchu myszy**; na to też wskazuje nazwa tego komunikatu.

Przy jego przetwarzaniu, bardziej niż w pozostałych zdarzeniach myszy, przydają się dostarczane wraz z nim dane dodatkowe. Szczególnie interesująca jest zmienna `lParam`, zawierająca nową pozycję kursora, liczoną względem lewego górnego rogu obszaru klienta okna. Możemy wyświetlić te współrzędne chociażby na pasku tytułu:



Screen 60. Współrzędne kursora na pasku tytułu okna

```
// CursorPos - pokazywanie pozycji kursora w oknie

// (fragment procedury zdarzeniowej)
case WM_MOUSEMOVE:
{
```

```
        // pobieramy współrzędne kursora i zapisujemy je jako napis
        std::stringstream Strumien;
        Strumien << "(" << GET_X_LPARAM(lParam) << "; " <<
                           GET_Y_LPARAM(lParam) << ")";

        // ustawiamy tytuł okna na ów napis
        SetWindowText (hWnd, Strumien.str().c_str());
        return 0;
    }
```

Oprócz współrzędnych w `lParam`, komunikat `WM_MOUSEMOVE` dostarcza też w `wParam` tych samych informacji o wciśniętych klawiszach, które omawialiśmy na samym początku poznawania zdarzeń myszy.
Po obsłużeniu zdarzenia `WM_MOUSEMOVE` zwracamy do systemu tradycyjną wartość zero.

Pozaklienckim odpowiednikiem przedstawionego komunikatu jest oczywiście `WM_NCMOUSEMOVE`. Okno otrzymuje go, kiedy kursor myszy porusza się ponad paskiem tytułu albo brzegiem okna. Do tego zdarzenia stosują się wszystkie uwagi o pozaklienckich komunikatach myszy, wymienione w poprzednim paragrafie. Szczegółowe wiadomości można jak zwykle znaleźć w MSDN.

Rolka

Od kilku lat wszystkie modele komputerowych myszek są wyposażane w pewien dodatkowy instrument, uzupełniający działanie przycisków. Jest to tak zwana **rolka myszy** (ang. *mouse wheel*), służąca głównie do przewijania dokumentów i stron internetowych. Znajduje się ona zwykle w miejscu środkowego przycisku myszy, zachowując jednak jego pełnią funkcjonalność (można nią klikać tak, jak przyciskiem). Ponadto możliwe jest też obracanie rolką w przód i w tył - powoduje to najczęściej przewinięcie oglądanego tekstu w górę lub w dół.
Przydatność i wygoda rolki jest bardzo duża, zwłaszcza podczas przeglądania serwisów WWW: nie trzeba wówczas kierować kursora, zajętego klikaniem w hiperłącza, do pasków przewijania, aby przejść w inne miejsce na stronie. Podobnie w edytorach tekstu rolka ułatwia i usprawnia pracę.

Komunikat rolki i jego adresaci

Windows zapewnia współpracę z rolką myszy poprzez komunikat `WM_MOUSEWHEEL`. Jak nietrudno się domyślić, jest on wysyłany wtedy, gdy użytkownik zmieni pozycję gryzoniowego pokrętła. Kto jednak otrzyma ten komunikat?…

Sprawa nie jest tak prosta jak w przypadku innych zdarzeń myszy. Kręcenie rolką nie jest bowiem zdarzeniem podobnym choćby do wciśnięcia przycisku. W tamtym przypadku komunikat dostawało zawsze to okno, które znajdowało się „pod kursorem”. Jednocześnie, na co nie zwróciliśmy dotąd uwagi, stawało się ono oknem **aktywnym**. Uaktywnienie okna objawia się zmianą koloru jego paska tytułu, z szarego na (domyślnie) niebieski. Innym objawem, mniej dostrzegalnym dla normalnych okien (ale widocznym doskonale dla pól tekstowych), jest też przejęcie wejścia od klawiatury - czyli uzyskanie **fokusu** (ang. *focus*). Obecnie nie interesujemy się rzecz jasna obsługiwaniem klawiatury, jednak pojęcie fokusu ma znaczenie także dla myszki i jej rolki, ponieważ:

Komunikat rolki `WM_MOUSEWHEEL` otrzymuje **tylko to okno**, którego w danej chwili **posiada fokus**.

Wiedząc o tym, łatwo wyjaśnić, dlaczego możemy przewijać dokumenty i strony WWW za pomocą rolki także wtedy, gdy „wyjedziemy” kursorem poza okna ich programów. Jeśli jednak klikniemy następnie którymś z przycisków myszy, okno straci fokus, a my

możliwość przewijania jego zawartości za pomocą rolki. Możemy ją oczywiście przywrócić poprzez ponowne uaktywnienie okna (np. kliknięciem).

Obsługa rolki

Niektóre kontrolki potomne, jak listy zwykłe i rozwijalne oraz przewijane pola tekstowe, mają standardowo zapewnioną odpowiednią reakcję na komunikat `WM_MOUSEWHEEL`. Warto jednak wiedzieć, jak możemy sami na niego reagować.

Zacznijmy od parametrów tego komunikatu. W dużym stopniu są one zbieżne z parametrami zdarzeń przycisków oraz `WM_MOUSEMOVE`. Istnieją aczkolwiek pewne drobne różnice.
Atoli skoncetrujmy się wpierw na podobnieństwach. Przede wszystkim `lParam` zawiera doskonale znany nam zestaw dwóch wartości, określających pozycję kursora myszki. Możemy je uzyskać za pomocą makr `GET_X_LPARAM()` i `GET_Y_LPARAM()` (dołączywszy wcześniej nagłówek *windowsx.h*).
Odmiennie należy traktować wartość `wParam` - zawiera ona tutaj dwie dane:

- dolne słowo to kombinacja bitowa flag, określających klawisze wciśnięte w chwili zajścia zdarzenia. Została ona przedstawiona na początku tej sekcji, wraz z wielce przydatną tabelką odpowiednich stałych :) Parametr ten możemy uzyskać przy pomocy makra `GET_KEYSTATE_WPARAM()`
- górne słowo specyfikuje dystans, o jaki obróciła się rolka. Pobieramy go poprzez makro `GET_WHEEL_DELTA_WPARAM()`

> Zauważmy, że nie ma czegoś takiego jak „aktualna pozycja rolki”, podobna do bieżącej pozycji kursora myszki. Obrót rolki nie jest bowiem ograniczony żadną skalą i może dokonywać się w obu kierunkach bez żadnych ograniczeń.

`GET_WHEEL_DELTA_WPARAM(wParam)` jest więc miarą obrotu, jakiego dokonał palec użytkownika, poruszający rolką. Wyraża się on **liczbą całkowitą ze znakiem**: dodatnie wartości oznaczają obrót naprzód (w kierunku „od użytkownika”), powodujący zazwyczaj przewijanie ekranu do góry; wartości ujemne odpowiadają obrotowi w tył („do użytkownika”) i przewijaniu tekstów w dół.
Sama wartość jest natomiast wielokrotnością stałej `WHEEL_DELTA`, ustawionej na `120`. Liczba ta odpowiada jednej elementarnej akcji (krokowi), jaką ma powodować obrót rolki - przykładowo, może to być przewinięcie tekstu o określoną liczbę linii (zwykle trzy[122]).
`WHEEL_DELTA` nie jest równe jedności, aby stanowić furtkę dla możliwych przyszłych urządzeń, wyposażonych w bardziej dokładne rolki. Wtedy wartość zapisana w górnym słowiem `wParam` nie będzie musiała być koniecznie całkowitą wielokrotnością delty, lecz mogła wynosić, powiedzmy, `40`. Taka liczba powinna więc spowodować wykonanie „jednej trzeciej akcji” przewidzianej na całą deltę - w opisywanym przypadku będzie to przewinięcie tekstu o jedną linijkę.
Już teraz pojawiają się myszki, umożliwiające w miarę płynne przewijanie, zatem należy być przygotowanym na odbieranie zdarzeń obrotu rolki o mniej niż jedną deltę. W idealnym przypadku powinny one skutkować podjęciem właściwego, „ułamkowego” działania. Jeżeli jednak nie jest to możliwe, wtedy najlepiej dodawać przychodzące dane o obrocie i wykonywać akcję dopiero wtedy, gdy tak powstała suma osiągnie wartość co najmniej `WHEEL_DELTA`:

---

[122] Ilość przewijanych za jednym razem linii jest ustawieniem systemowym i należy je pobierać za pomocą wywołania `SystemParametersInfo(SPI_GETWHEELSCROLLLINES, 0, &nPrzewijaneLinie, 0);`, gdzie `nPrzewijaneLinie` jest zmienną typu całkowitego. Aby zaś obliczyć liczbę wierszy przewijanym w reakcji na `WM_MOUSEWHEEL`, trzeba przemnożyć pobraną wielkość przez liczbę wielokrotności `WHEEL_DELTA` w parametrze zdarzenia, tj.: `float fLinie = (float) GET_WHEEL_DELTA_WPARAM(wParam) / WHEEL_DELTA * nPrzewijanieLinie;`.

```
// zmienna globalna przechowująca obrót rolki
int g_nCalkowityObrot = 0;

// (procedura zdarzeniowa)
case WM_MOUSEWHEEL:
{
   // dodajemy otrzymaną wartość obrotu
   g_nCalkowityObrot += GET_WHEEL_DELTA_WPARAM(wParam);

   // sprawdzamy, czy jest on bezwględnie większy niż WHEEL_DELTA
   if (abs(g_nCalkowityObrot) >= WHEEL_DELTA)
   {
         // dla pewności obliczamy ilość kroków -
         // - wielokrotności WHEEL_DELTA
         int nKroki = g_nCalkowityObrot / WHEEL_DELTA;

         // podejmujemy odpowiednie akcje...

         // odejmujemy wykorzystane obroty od licznika
         // (ustawiając go na resztę z dzielenia przez WHEEL_DELTA)
         g_nCalkowityObrot %= WHEEL_DELTA;
   }

   // tradycyjnie zwracamy zero
   return 0;
}
```

Można się spodziewać, że wraz z upowszechnieniem myszek z płynnie obracającymi się rolkami coraz więcej programów będzie oferowało ciągłe, a nie tylko skwantowane przewijanie dokumentów.

Łapanie myszy

Tyle okien, a tylko jedna myszka… - tak mógłby jęknąć spersonifikowany system Windows, gdy umiał mówić. Programy komputerowe jako twory martwe nie wyrażają jednak swoich opinii i dlatego Windows musi potulnie i sprawnie radzić sobie z problemem współdzielenia jednego urządzenia między wiele aplikacji.

Władza nad myszką

Cały mechanizm odbierania zdarzeń od myszki opiera się na prostej zasadzie. Mówi ona, że dany komunikat (np. o kliknięciu) zostanie wysłany zawsze do tego okna, nad którym aktualnie przebywa kursor myszki. W ten sposób różne okna w systemie dostają informacje tylko o tych zdarzeniach, które bezpośrednio ich dotyczą.

Istnieją jednak sytuacje, w których jedno okno powinno otrzymywać **wszystkie komunikaty** o zdarzeniach myszki. W takim przypadku powinno ono przejąć od systemu **władzę nad myszką**.

Okno posiadające **władzę nad myszką** (ang. *mouse capture*) otrzymuje informacje o **wszystkich zdarzeniach**, pochodzących od urządzenia wskazującego.

W normalnej sytuacji myszka jest „wolna” - żadne okno nie posiada nad nią władzy. Gdy chcemy to zmienić, musimy posłużyć się odpowiednimi funkcjami Windows API.

Przykład przechwycenia myszki

Zobaczmy to na klasycznym już przykładzie okienkowego szkicownika (ang. *scribble*). Jest to prosty program, pozwalający rysować szlaczki i inne zawijasy w swoim oknie:

Screen 61. Okno komputerowego szkicownika

Linie kreślimy w nim poprzez kliknięcie lewym przyciskiem myszy, przytrzymanie go i poruszanie kursorem. Taki programik pomaga początkującym użytkownikom komputera nabrać wprawy w przeciąganiu. My oczywiście nie potrzebujemy żadnych ćwiczeń tego typu i dlatego spojrzymy raczej na kod tej przykładowej aplikacji:

```
// Scribble - okienkowy szkicownik

#include <string>
#define WIN32_LEAN_AND_MEAN
#include <windows.h>
#include <windowsx.h>


// nazwa klasy okna
std::string g_strKlasaOkna = "od0dogk_Window";

// dane okna
HDC g_hdcOkno;       // uchwyt kontekstu urządzenia okna


//-------------------- procedura zdarzeniowa okna ------------------------

LRESULT CALLBACK WindowEventProc(HWND hWnd, UINT uMsg,
                                 WPARAM wParam, LPARAM lParam)
{
   switch (uMsg)
   {
         case WM_LBUTTONDOWN:
               // przejmujemy myszkę
               SetCapture (hWnd);

               // przesuwamy pióro (służące do rysowania po oknie)
               // w punkt kliknięcia
               MoveToEx (g_hdcOkno,
                          GET_X_LPARAM(lParam), GET_Y_LPARAM(lParam),
                          NULL);

               // zwracamy zero
               return 0;

         case WM_MOUSEMOVE:
               // jeżeli nasze okno posiada myszkę
               if (GetCapture() == hWnd)
                     // rysujemy linie od poprzedniego do aktualnego
                     // miejsca kursora myszki
                     LineTo (g_hdcOkno,
                             GET_X_LPARAM(lParam), GET_Y_LPARAM(lParam));

               // zwracamy zero
               return 0;

         case WM_LBUTTONUP:
               // oddajemy władzę nad myszką do systemu
```

```
                  ReleaseCapture();
                  return 0;

            //-------------------------------------------------------------

            case WM_DESTROY:
                  // kończymy program
                  PostQuitMessage (0);
                  return 0;
      }

      return DefWindowProc(hWnd, uMsg, wParam, lParam);
}


//------------------------funkcja WinMain() ----------------------------

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE, LPSTR, int nCmdShow)
{
   /* rejestrujemy klasę okna */

   WNDCLASSEX KlasaOkna;

   // wypełniamy strukturę WNDCLASSEX
   // (pomijamy z tego większość pól)
   KlasaOkna.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);
   KlasaOkna.style = CS_OWNDC;   // własny kontekst urządzenia okna

   // rejestrujemy klasę okna
   RegisterClassEx (&KlasaOkna);


   /* tworzymy okno */

   // tworzymy okno funkcją CreateWindowEx
   // (znana czynność, więc pomijamy (uchwyt trafia do hOkno))

   // pobieramy uchwyt do kontekstu urządzenia obszaru klienta okna
   g_hdcOkno = GetDC(hOkno);

   // pokazujemy nasze okno
   ShowWindow (hOkno, nCmdShow);


   /* pętla komunikatów */

   // (w zwyczajowej formie, darujemy ją sobie)

   // zwracamy kod wyjścia
   return static_cast<int>(msgKomunikat.wParam);
}
```

Ogólna zasada działania tej aplikacji jest prosta. W momencie wciśnięcia lewego przycisku myszy (`WM_LBUTTONDOWN`) przejmuje ona władzę nad myszką, ustawiając ją dla swego okna:

```
SetCapture (hWnd):
```

Odtąd będzie ono otrzymywało informacje o wszystkich zdarzeniach myszki. Zanim jednak zajmiemy się nimi, musimy zapamiętać pozycję kursora w chwili kliknięcia - tak, aby móc potem rysować ślad jego ruchu. Wyręcza nas w tym sam Windows:

```
MoveToEx (g_hdcOkno, GET_X_LPARAM(lParam), GET_Y_LPARAM(lParam), NULL);
```

Funkcja `MoveToEx()` przesuwa tzw. pióro, związane z kontekstem urządzenia naszego okna (`g_hdcOkno`) w miejsce o współrzednych kliknięcia. Koordynaty te pobieramy naturalnie za pomocą makr `GET_X/Y_LPARAM()`. Każda linia, jaką teraz narysujemy w oknie, będzie się zaczynała we wskazanym przed chwilą punkcie.

A kiedyż to rysujemy linie w naszym oknie? Otóż robimy to w reakcji na zdarzenie `WM_MOUSEMOVE`:

```
case WM_MOUSEMOVE:
   if (GetCapture() == hWnd)
         LineTo (g_hdcOkno, GET_X_LPARAM(lParam), GET_Y_LPARAM(lParam));

   return 0;
```

Wcześniej sprawdzamy jeszcze, czy główne (i *notabene* jedyne) okno programu posiada istotnie władzę nad myszką. Dokonujemy tego przy pomocy funkcji `GetCapture()`; jeżeli zwracany przezeń uchwyt jest zgodny z uchwytem docelowego okna zdarzenia `WM_MOUSEMOVE`, wtedy czujemy się zobowiązani do narysowania linii znaczącej drogę kursora.

Zamiast `GetCapture()` moglibyśmy wykorzystać też zmienną logiczną, określającą czy okno programu przechwyciło myszkę. Ustawialibyśmy ją na `true` w reakcji na `WM_LBUTTONDOWN` i na `false` w `WM_LBUTTONUP`, a tutaj dokonywalibyśmy sprawdzenia jej wartości. Wykorzystałem jednak `GetCapture()`, aby pokazać wszystkie funkcje związane z zagadnieniem władzy nad myszką.

Linię te rysujemy poprzez `LineTo()`, podając tej funkcji docelowe współrzędne drugiego końca odcinka. Oprócz kreślenia rzeczonej linii, funkcja ta dokonuje też przesunięcia pióra w owe miejsce, tak więc następne rysowane odcinki będą się łączyły z poprzednimi. Tym sposobem powstanie ciągły ślad drogi kursora myszki, a o to nam przecież chodzi.

O funkcjach `MoveToEx()` oraz `LineTo()` pomówimy sobie dokładnie przy omawianiu geometrycznej części biblioteki Windows GDI w następnym rozdziale.

Wreszcie dochodzimy do komunikatu `WM_LBUTTONUP`, oznaczającego zwolnienie lewego przycisku myszki. W odpowiedzi na niego wykonujemy tylko jedną czynność: oddajemy władzę nad myszką z powrotem do systemu, wywołując funkcję `ReleaseCapture()`. Od tej pory notyfikacje o zdarzeniach myszy będą, jak zwykle, trafiać do okna mieszczącego się pod kursorem myszy, a nie do naszego programu.

Nasuwa się jeszcze pytanie: Co właściwie czyni ta kombinacja funkcja `SetCapture()` i `ReleaseCapture()`? Czy nie można byłoby obejść się bez niej?...
Teoretycznie jest to możliwe[123], jednak niesie pewnie nieprzyjemne konsekwencje praktyczne. Wyobraźmy sobie, że użytkownik wciska lewy przycisk myszki, a następnie przeciąga kursor poza obręb okna i zwalnia przycisk. Kiedy teraz powróci z powrotem w obszar okna programu, kursor będzie kreślił sobą linie - mimo że przecież przycisk myszki nie jest wciśnięty!
Dzieje się tak dlatego, że po przeciągnięciu kursora poza okno, komunikat `WM_LBUTTONUP` nie dociera już do naszego programu. Ten „myśli" więc, że lewy przycisk jest nadal

[123] O ile dodamy jeszcze wspomnianą kilka akapitów wyżej zmienną logiczną, która będzie określała, czy należy rysować ślad kursora.

wciśnięty, a zatem rysuje linie w ślad za strzałką. Dzięki przechwytywaniu władzy nad myszką zapobiegamy podobnej sytuacji.

Zastosowania

Przejmowanie władzy nad myszką ma sporo zastosowań przede wszystkich w różnych programach graficznych, choćby tak prostych jak zaprezentowany przykład. Nie dotyczy to tylko swobodnego rysowania, ale też wyznacza linii prostych, krzywych Beziera; nawet aplikacje do trójwymiarowego modelowania korzystają z tej techniki. Innym zastosowaniem jest też implementowanie specyficznego rodzaju przeciągania jakichś elementów.

Kontrolowanie wejścia od myszy

Przyjmowanie komunikatów o zdarzeniach to nie jedyna forma kooperacji z myszką, dostępna w Windows. W tej sekcji poznamy większość pozostałych, które dają pełen obraz możliwości WinAPI w zakresie obsługi urządzeń wskazujących.

Pozycja kursora

Myszkę na ekranie monitora reprezentuje kursor, mający zwykle postać strzałki. Znajduje się on w określonej pozycji, wyrażonej we współrzędnych ekranowych. Pozycję tę otrzymujemy ze wszystkimi komunikatami o zdarzeniach myszy[124]. Możemy też na nią wpływać w inny sposób niż tylko poprzez bezpośrednie poruszanie gryzoniem.
Spójrzmy więc, jak to się odbywa.

Pobieranie i ustawianie pozycji kursora

Aktualne współrzedne kursora, oprócz tego że dostajemy w `lParam` każdego zdarzenia myszy, możemy pobrać za pomocą funkcji `GetCursorPos()`:

```
BOOL GetCursorPos(LPPOINT lpPoint);
```

Podajemy jej wskaźnik do prostej struktury typu `POINT`, posiadającej dwa pola `x` i `y`. Z nich też odczytujemy żądaną pozycję strzałki.
Jedyną różnicą w stosunku do danych otrzymywanych przy okazji zdarzeń jest to, iż:

`GetCursorPos()` zwraca **ekranowe** współrzędne kursora.

Jak zaś pamiętamy, `lParam` komunikatów myszy zawiera pozycję kursora relatywną do lewego górnego rogu obszaru klienta okna.

A co z ustawianiem pozycji kursora? Służy do tego funkcja `SetCursorPos()`:

```
BOOL SetCursorPos(int X, int Y);
```

Łatwo można się domyślić, że podajemy jej nowe współrzędne dla kursora myszy w obu parametrach. Są to również koordynaty ekranowe, zatem wywołanie w postaci:

```
SetCursorPos (0, 0);
```

Przesunie strzałkę do lewego górnego skraju ekranu (pulpitu) - co było do okazania ;)

[124] Przy czym jest to pozycja liczona względem obszaru klienta okna-adresata komunikatu.

(Bez)względne współrzędne

Dwie metody liczenia współrzędnych kursora (i nie tylko kursora) mogą być trochę kłopotliwe - szczególnie, jeżeli nie byłoby prostego sposobu konwersji między nimi. Taki sposób jednak istnieje i stanowią go niniejsze dwie funkcje:

```
BOOL ClientToScreen(HWND hWnd, LPPOINT lpPoint);
BOOL ScreenToClient(HWND hWnd, LPPOINT lpPoint);
```

Ich przeznaczenie dobrze obrazują nazwy. `ClientToScreen()` zamienia współrzędne liczone względem obszaru klienta okna na koordynaty ekranowe. Należy podać jej uchwyt okna (`hWnd`) oraz rzeczone współrzędne w postaci adresu struktury `POINT`. Stamtąd też odczytamy nowe współrzedne (ekranowe) po wykonaniu funkcji. Odwrotnie działa `ScreenToClient()`. Tutaj podajemy jej liczby odnoszące się do ekranu, a w zamian dostajemy koordynaty tyczące się obszaru klienta okna o uchwycie `hWnd`.

Ogólnie więc zapamiętajmy, że:

`ClientToScreen()` dokonuje konwersji typu *obszar klienta → ekran*.
`ScreenToClient()` zamienia współrzędne wedle schematu *ekran → obszar klienta*.

Te dwie funkcje przydają się w wielu typowych i nietypowych sytuacjach programistycznych.

Ograniczanie swobody w poruszaniu kursorem

Inicjalnie kursor posiada nieograniczoną swobodę w poruszaniu się po całym ekranie. Jeżeli z jakichś względów nie odpowiada to nam, możemy ograniczyć do wybranego prostokąta rejon ekranu, który będzie dla myszy dostępny. Czynimy to za pomoca funkcji `ClipCursor()`:

```
BOOL ClipCursor(const RECT* lpRect);
```

Jako parametru żąda ona wskaźnika do struktury `RECT`, opisującej tenże prostokąt, ograniczający kursor. Skąd go weźmiemy - to już nasza sprawa: może być to np. prostokąt okna naszego programu:

```
RECT rcOkno;
GetWindowRect (hWnd, &rcOkno);
ClipCursor (&rcOkno);
```

Uruchamiając powyższy kod sprawimy, iż użytkownik nie będzie w stanie „wyjechać” kursorem poza obręb okna aplikacji. Takie zachowanie ogranicza więc wygodę korzystania z aplikacji i systemu operacyjnego, zatem powinno być stosowane jedynie w **uzasadnionych przypadkach**. Zawsze też należy pamiętać o uwolnieniu kursora, gdy będzie to już możliwe:

```
ClipCursor (NULL);
```

Przekazanie `NULL` do funkcji `ClipCursor()` spowoduje rozciągnięcie rejonu dostępnego dla myszy na cały ekran. Będzie to więc powrót do stanu początkowego.

Sprawdzanie przycisków myszy

O wciśnięciu i zwolnieniu przycisków myszy informują nas zdarzenia `WM_?BUTTONDOWN/UP`. O aktualnym stanie tychże przycisków możemy też dowiedzieć się podczas przetwarzania

któregokolwiek z klienckim komunikatów myszy - wystarczy odczytać wartość wParam[125] tego komunikatu i porównać ją z odpowiednią flagą bitową (jedną ze stałych MK_).

Jako elastyczny system operacyjny Windows oferuje jednak także inne sposoby na pozyskanie bieżącej kondyncji przycisków myszy - czyli informacji o ich wciśnięciu. Służą do tego na przykład funkcje GetKeyState() i GetAsyncKeyState().

Kody wirtualne przycisków myszy

Nazwy tych dwóch funkcji sugerują, że ich zasadniczym przeznaczeniem jest kontrola stanu klawiszy na klawiaturze. To faktycznie prawda, jednak w Windows API pod pojęciem 'klawisz' (ang. *key*) kryją się także przyciski właściwe innym urządzeniom wejściowym - na przykład myszce. Łącznie nazywa się je **klawiszami wirtualnymi** (ang. *virtual-keys*).
Takie podejście może się wydawać dziwne, ale w praktyce jest bardzo wygodne. Każdemu „klawiszowi" (cokolwiek to słowo chwilowo znaczy…) przyporządkowany jest pewien kod (ang. *virtual-key code*), który go jednoznacznie identyfikuje za pomocą stałej o nazwie zaczynającej się przedrostkiem VK_. Nas oczywiście interesują teraz tylko te kody, za którymi kryją się przyciski myszy. Przedstawia je poniższa tabelka:

stała	wartość	przycisk
VK_LBUTTON	0x0001	lewy
VK_RBUTTON	0x0002	prawy
VK_MBUTTON	0x0004	środkowy
VK_XBUTTON1	0x0005	pierwszy dodatkowy (X1)
VK_XBUTTON2	0x0006	drugi dodatkowy (X2)

Tabela 46. Kody wirtualne przycisków myszy
(dwa ostatnie przyciski są dostępne tylko w Windows 2000/XP lub nowszych)

Zerknijmy teraz na funkcje Get[Async]KeyState() i zobaczmy, jak mogą nam one pomóc w pozyskiwaniu stanu przycisków myszy.

Kontrola stanu przycisków myszy

Omawiane dwie funkcje są na tyle do siebie podobne, że możemy je rozpatrywać łącznie - również pod względem prototypów:

```
SHORT Get[Async]KeyState(int nKey);
```

Widzimy, że funkcje te żądają jednego parametru. Jest nim kod wirtualnego klawisza, który ma być sprawdzany; u nas będzie to rzecz jasna jedna z pięciu stałych właściwych przyciskom myszy.

Co zaś otrzymujemy w zamian? Otóż dostajemy wartość 16-bitową, która łącznie niesie w sobie aż dwie dane. Poznamy je obie przy omawianiu obsługi klawiatury, a teraz skoncetrujemy się na ważniejszej z nich, zawartej w **starszym bajcie** słowa zwracanego przez Get[Async]KeyState().
Jak nietrudno zgadnąć, mam tu na myśli pożądaną przez cały czas informację o tym, czy dany przycisk myszy jest w aktualnej chwili wciśnięty, czy też nie. Sprawdzić można to w prosty sposób: należy ustalić, czy starszy bajt wyniku jest liczbą **różną od zera**. Jeśli tak, znaczy to, iż kontrolowany przycisk jest w danym momencie **wciśnięty**.

Aby więc skontrolować stan lewego przycisku myszy, można użyć wywołania podobnego do poniższego:

[125] W przypadku WM_MOUSEWHEEL jest to dolne słowo wParam, uzyskiwane poprzez GET_KEYSTATE_WPARAM().

```
if (HIBYTE(Get[Async]KeyState(VK_LBUTTON)) /* != 0 */)
{
   // lewy przycisk myszy jest aktualnie wciśnięty
}
```

Sprawa wygląda identycznie dla czterech pozostałych przycisków.

Różnica mała, lecz ważna

Wypadałoby teraz rozróżnić wreszcie funkcje `GetKeyState()` i `GetAsyncKeyState()`. Pełnego rozgraniczenia tych dwóch procedur dokonamy wtedy, gdy poznamy je całkowicie - stanie się to przy okazji poznawania zagadnień związanych z klawiaturą z WinAPI. Obecnie skupimy się na jednym niuansie, dotyczącym przycisków myszy.

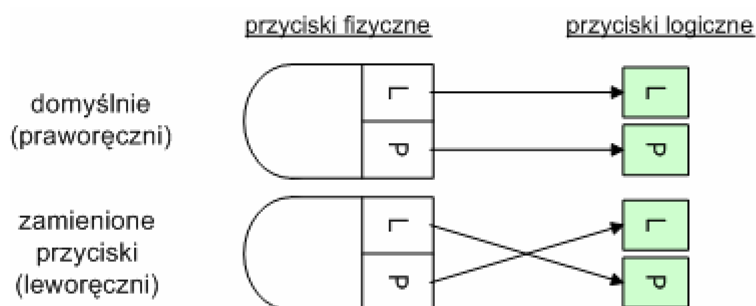
Chodzi o to, iż Windows oferuje pewne przydatne udogodnienie dla osób leworęcznych. Użytkownicy posługujący się odmienną kończyną niż pozostali chcą bowiem trzymać mysz raczej po lewej stronie biurka, w lewej dłoni. Wówczas pod palcem wskazującym znajdzie się nie lewy, lecz **prawy** przycisk myszy; analogicznie palec serdeczny spocznie na **lewym** przycisku myszy, który dla użytkownika-mańkuta **wydaje się** prawym.
Nie jest to przy tym problemem, ponieważ Windows daje możliwość dostosowania się do tej sytuacji. Polega ona na zamianie zwyczajowego znaczenia lewego i prawego przycisku na wzajemnie odwrotne. Opcja taka może być ustawiona na przykład w systemowym Panelu Sterowania.

Jak to jednak czesto bywa, ułatwienia dla użytkownika są utrudnieniami dla programisty. Fakt, że lewy przycisk myszy może w pewnych sytuacjach odpowiadać prawemu i odwrotne, wprowadza trochę zamieszania. Ale przecież nie z takimi rzeczami radziliśmy sobie wcześnie, prawda? :)
Na początek dodajmy do naszego słownika dwa przydatne określenia: fizycznych i logicznych przycisków myszy.

Fizyczne przyciski myszy (ang. *physical mouse buttons*) to przyciski umieszczone na urządzeniu wskazującym (zwykle myszy).

Logiczne przyciski myszy (ang. *logical mouse buttons*) to **systemowa interpretacja** fizycznych przycisków myszy.

Przyciski fizyczne są dosłownie namacalne - możemy ich dotknąć i je wciskać. Poza tym powinniśmy zwrócić uwagę na pozornie oczywisty fakt: fizyczne przyciski **zawsze** „pozostają sobą" - lewy przycisk jest zawsze lewym, a prawy prawym.
Inaczej jest w przypadku przycisków logicznych. W większości przypadków będą one odpowiadały swym fizycznym braciom… z wyjątkiem jednego wyjątku :) Domyślasz się, że tą nietypową sytuacją jest włączona opcja zamiany przycisków. Wtedy też przyciski myszy są interpretowane „na opak":



Schemat 42. Mapowanie fizycznych przycisków myszy na logiczne

No dobrze, ale jak ta sytuacja ma się do odbierania przez system Windows zdarzeń od myszy oraz do bezpośredniego pobierania jej stanu?… Otóż prawie zawsze liczą się tu wyłącznie **logiczne przyciski** myszy.

Niemal wszystkie elementy Windows API przeznaczone do pracy z przyciskami myszy operują na **logicznych przyciskach**.

Nieprzypadkowo zaznaczyłem to drobne słówko - 'niemal'. Istnieje bowiem jedna funkcja, która odczytuje stan wyłącznie fizycznych przycisków myszy - jest nią `GetAsyncKeyState()`.

`GetKeyState()` pobiera stan **logicznych** przycisków myszy.
`GetAsyncKeyState()` pobiera stan **fizycznych** przycisków myszy.

I to jest właśnie ta różnica, na którą chciałem zwrócić uwagę. Wynika z niej, że dwa poniższe wywołania mogą w istocie sprawdzać fizycznie odmienne przyciski:

```
GetKeyState(VK_LBUTTON)
GetAsyncKeyState(VK_LBUTTON)
```

Zależy to od ustawienia systemowego, wprowadzanego w Panelu Sterowania. Programowo możemy je odczytać poprzez `GetSystemMetrics(SM_SWAPBUTTON)`:

```
// sprawdzenie stanu lewego przycisku i opcji zamiany przycisków...
if (HIBYTE(GetAsyncKeyState(VK_LBUTTON))
     && GetSystemMetrics(SM_SWAPBUTTON))
{
   // fizycznie wciśnięto lewy przycisk, ale ze względu na ustawioną
   // należy go zinterpretować jako prawy
}
```

Powyższy kod odpowiada z grubsza (bo nie do końca, o czym powiemy później) prostszej instrukcji z użyciem `GetKeyState()`:

```
if (HIBYTE(GetKeyState(VK_LBUTTON)))
{
   // wciśnięto logicznie lewy przycisk
}
```

Jest ona także bardziej przejrzysta, lecz aby ją właściwie stosować, trzeba dowiedzieć się nieco więcej o kwestiach różniących funkcje `GetKeyState()` i `GetAsyncKeyState()` w odniesieniu do wszystkich klawiszy wirtualnych. Uczynimy to w podrozdziale na temat klawiatury w Windows.
O pobieraniu ustawień myszy, takich jak przytoczona tu zamiana przycisków, powiemy sobie natomiast w jednym z najbliższych paragrafów.

Symulowanie zdarzeń myszy

Normalnie zadaniem programu okienkowego jest reakcja na czynności wykonywane przez użytkownika. Wiążę się to z odbieraniem i obsługą komunikatów systemowych. Komunikaty te generuje pośrednio osoba korzystająca z aplikacji; czyni to za pomocą urządzeń wejściowych.

Także sam program może postawić się w tej roli i symulować występowanie odpowiednich zdarzeń. Najprostszym sposobem zdawałoby się bezpośrednie wysyłanie komunikatów o zdarzeniach poprzez funkcję `SendMessage()` lub `PostMessage()`.

Jednakże tą drogą będziemy emulować jedynie **skutek**, a nie **przyczynę** występowania pewnych zdarzeń. Jest to tylko udawanie **systemowej interpretacji** danych od urządzeń, nie zaś danych jako takich. Nie bez znaczenia jest też fakt, że z wysyłanym komunikatem łączy się wiele pobocznych aspektów, którymi zwykle się nie zajmujemy, lecz które mogą okazać się ważne (np. kwestia wątków). Wreszcie, komunikaty muszą być skierowane do konkretnego okna, mającego je otrzymać, a przecież wiemy, że Windows zwykł sam o tym decydować (przykładem jest kliknięcie myszką: w zależności od pozycji kursora komunikat o tym zdarzeniu mogą dostać zupełnie różne okna).

Samodzielne produkowanie zdarzeń nie jest więc dobrym rozwiązaniem. Byłoby lepiej, gdyby to system oferował jakiś własny sposób „udawania" sygnałów od myszki czy klawiatury. I tak się przypadkowo składa, iż podobny mechanizm faktycznie istnieje :D Poznamy go teraz, zajmując się programowym symulowaniem myszki.

Funkcja `SendInput()`

W starszych wersjach Windows do generowania zdarzeń myszy służyła funkcja `mouse_event()`. Począwszy od Windows 98 zalecane jest jednak użycie innej funkcji[126] - `SendInput()`:

```
UINT SendInput(UINT nInputs,
               LPINPUT pInputs,
               int cbSize);
```

Nie wygląda ona na zbyt złożoną, przyjrzyjmy się więc jej parametrom:

typy	***parametry***	***opis***
`UINT` `LPINPUT`	`nInputs` `pInputs`	Te dwa argumenty określają **tablicę struktur** typu `INPUT`, która zostanie przekazana do funkcji. `nInputs` zawiera liczbę elementów tej tablicy, zaś `pInputs` - wskaźnik do niej. Każdy element jest natomiast oddzielną strukturą, opisującą jedno symulowane zdarzenie myszy lub klawiatury.
`int`	`cbSize`	Musimy podać tutaj **rozmiar typu** `INPUT` w bajtach, czyli po prostu `sizeof(INPUT)`.

Tabela 47. Parametry funkcji `SendInput()`

Widać, że funkcja ta potrafi wygenerować naraz więcej niż jedno zdarzenie od urządzenia wejściowego, ponieważ pobiera ona tablicę struktur `INPUT`. Następnie przetwarza ją element po element, zwracając w wyniku liczbę poprawnie zasymulowanych zdarzeń.

Struktura `INPUT`

Punkt ciężkości zagadnienia przesuwa się nam z funkcji `SendInput()` na strukturę `INPUT`. Spójrzmy zatem na definicję tego typu:

```
struct INPUT
{
   DWORD type;

   union
   {
         MOUSEINPUT mi;
         KEYBDINPUT ki;
         HARDWAREINPUT hi;
```

[126] Funkcja ta zastępuje również `keybd_event()`, służącą do symulowania klawiatury. Jak to robi - o tym napiszę w następnym podrozdziale.

```
    };
};
```

Różni się on zdecydowanie od większości typów strukturalnych, z jakimi mieliśmy dotąd do czynienia. Względną nowością jest bowiem **anonimowa unia** (ang. *anonymous union*), zamykająca trzy pola struktury. Jeżeli pamiętamy, jak funkcjonują unie, to wiemy, iż taka deklaracja powoduje następujący efekt: tylko **jedno z pól** - `mi`, `ki` lub `hi` - może być wykorzystane do zapisywania sensownych informacji. Jest to najlogiczniejsza realizacja założenia, aby jedna struktura `INPUT` opisywała tylko jedno zdarzenie - myszki, klawiatury czy też specjalnego „rodzaju sprzętowego”.
Niemniej jednak system operacyjny (zredukowany chwilowo do funkcji `SendInput()`) musi wiedzieć, jakiego typu symulowane zdarzenie chcemy wygenerować. Informujemy o tym w jedynym „pozaunijnym” polu struktury `INPUT` - `type`. W tym celu może ono przyjmować jedną z następujących wartości, odpowiadających poszczególnym polom unii:

stała	*znaczenie*	*pole unii*
`INPUT_MOUSE`	symulowane wejście od myszy	`mi`
`INPUT_KEYBOARD`	symulowane zdarzenie klawiatury	`ki`
`INPUT_HARDWARE`	symulacja innego urządzenia (tylko Windows 9x/Me)	`hi`

Tabela 48. Stałe pola `type` struktury `INPUT`

Trzecia z nich jest już mocno przestarzała i dlatego nie należy jej używać. Druga nie interesuje nas w tej chwili, gdyż obecnie nie zajmujemy się klawiaturą. Wybieramy zatem bramkę numer 1 - `INPUT_MOUSE` :)

Struktura *MOUSEINPUT*

Jako że chcemy symulować akcje myszy, powinniśmy użyć pola `mi` (oraz wartości `INPUT_MOUSE` w polu `type`). Pole `mi` jest, jak możnaby przypuszczać, również strukturą. Tym razem typem tej struktury jest `MOUSEINPUT`, a opisuje ona wszystkie szczegóły naszego wymuszonego zdarzenia myszy:

```
struct MOUSEINPUT
{
    LONG dx;
    LONG dy;
    DWORD mouseData;
    DWORD dwFlags;
    DWORD time;
    ULONG_PTR dwExtraInfo;
};
```

Całkiem ich sporo, więc nie od rzeczy będzie ujęcie opisów powyższych pól w zgrabnej tabelce:

typ	*pola*	*opis*
`LONG`	`dx` `dy`	Wpisujemy tutaj **współrzędne** opisujące ruch kursora myszy (jeżeli mamy zamiar nim poruszać). Współrzędne te mogą być podane w **liczbach bezwzględnych** - są wówczas liczone w odniesieniu do lewego górnego rogu ekranu; są to więc **koordynaty ekranowe**. Alternatywnie możliwe jest podanie współrzędnych **względnych**, będących raczej określeniem **przesunięcia** kursora; system operacyjny doda je wtedy do aktualnej pozycji strzałki, otrzymując w ten sposób jej nowe położenie.

typ	pola	opis
		O tym, jakiego rodzaju współrzędne podajemy w polach dx i dy informuje obecność lub brak flagi MOUSEEVENTF_ABSOLUTE w polu dwFlags.
DWORD	mouseData	W tym polu podajemy **dodatkowe dane** na temat zdarzenia myszy. Mogą one przyjąć jedną z dwóch postaci, zależnie od rodzaju zdarzenia: ➢ w przypadku symulowanej zmiany położenia rolki myszy pole mouseData zawiera wartość jej obrotu, czyli **deltę**. Jest to taka sama wartość, jaką otrzymujemy w górnym słowie parametru wParam przy przetwarzaniu komunikatu WM_MOUSEWHEEL ➢ gdy mamy na celu emulowanie wciśnięcia jednego z dwóch dodatkowych przypcisków myszy - oznaczonych X1 i X2, a nazywanych wspólnie przyciskami X - pole mouseData powinno zawierać **wskazanie** jednego z tych przycisków: ✓ stała XBUTTON1 wskazuje na przycisk X1 ✓ stała XBUTTON2 odpowiada przyciskowi X2
DWORD	dwFlags	Tutaj dostarczamy **kombinację flag bitowych**, określających m.in. rodzaj zdarzenia myszy (przesunięcie, kliknięcie, itd.), jakie chcemy zasymulować. Dopuszczalnych flag jest całkiem sporo, więc ujmie je za chwile kolejna tabelka :D
DWORD	time	Pole time określa **moment zaistnienia zdarzenia**. Ma on być wyrażony w spotkanej już przez nas formie liczby milisekund od startu systemu. Czas w takiej postaci można uzyskać poprzez GetTickCount() i umieścić w tym polu; można też zostawić w nim zero, wtedy system sam zapisze tutaj chwilę generacji zdarzenia.
ULONG_PTR	dwExtraInfo	To pole może przechowywać jakieś pomocnicze dane dla odbiorcy zdarzenia. Zwykle nie ma potrzeby przekazywania żadnych takich danych, zatem wpisujemy tu najczęściej zero. Owe dodatkowe dane można uzyskać podczas przetwarzania komuniaktów o zdarzeniach - wystarczy wywołać funkcję GetMessageExtraInfo().

Tabela 49. Pola struktury MOUSEINPUT

O rodzaju symulowanego zdarzenia, oraz o kilku innych kwestiach, informujemy funkcję SendInput() za pośrednictwem pola dwFlags. Jest to kombinacja jednej lub kilku flag bitowych spośród poniższych:

flaga	znaczenie
MOUSEEVENTF_MOVE	ruch myszą
MOUSEEVENTF_LEFTDOWN	wciśnięcie lewego przycisku myszy
MOUSEEVENTF_LEFTUP	zwolnienie lewego przycisku myszy
MOUSEEVENTF_MIDDLEDOWN	wciśnięcie środkowego przycisku myszy
MOUSEEVENTF_MIDDLEUP	zwolnienie środkowego przycisku myszy
MOUSEEVENTF_RIGHTDOWN	wciśnięcie prawego przycisku myszy
MOUSEEVENTF_RIGHTUP	zwolnienie prawego przycisku myszy
MOUSEEVENTF_XDOWN	wciśnięcie jednego z dodatkowych przycisków myszy
MOUSEEVENTF_XUP	zwolnienie jednego z dodatkowych przycisków myszy
MOUSEEVENTF_WHEEL	obrót rolką myszy

flaga	znaczenie
MOUSEEVENTF_ABSOLUTE	Obecność tej flagi sprawia, że pola dx oraz dy będą traktowane jako docelowe, **bezwzględne współrzedne ekranowe** kursora. Funkcja SendInput() ustawi więc strzałkę myszy w pozycji wyznaczonej przez te pola Jeżeli zaś flaga nie bdzie obecna w polu dwFlags, wtedy dx i dy zostaną potraktowane jako określenie **przesunięcia** kursora, czyli dystansu poziomowego i pionowego, który zostanie dodany do aktualnego położenia kursora po to, aby otrzymać nowe. Rzeczywiste przesunięcie kursora może się nieco różnić od wartości podanych w dx i dy, gdyż system operacyjny bierze jeszcze pod uwagę kilka innych czynników, jak np. aktualną prędkość ruchu myszki. Jeśli interesują cię szczegóły, zajrzyj do opisu struktury MOUSEINPUT w MSDN.
MOUSEEVENTF_VIRTUALDESK	Flaga ta działa tylko w połączeniu z MOUSEEVENTF_ABSOLUTE. Jej ustawienie powoduje, że absolutne koordynaty kursora podane w dx i dy są traktowane w odniesieniu do całego pulpitu, a nie do ekranu bieżącego monitora. Ma to znaczenie wyłącznie w systemach wielomonitorowych.

Tabela 50. Flagi bitowe pola dwFlags struktury MOUSEINPUT

Ze względu na fakt, iż dwFlags jest kombinacją bitową, możliwe jest ustawienie więcej niż jednej flagi naraz. Tym samym można zasymulować kilka zdarzeń myszy za pomocą jednej struktury [MOUSE]INPUT. Niedozwolone jest jedynie połączenie MOUSEEVENTF_XDOWN/UP z MOUSEEVENTF_WHEEL; powody są czysto techniczne: oba zdarzenia korzystają bowiem z pola mouseData, ale każde na swój własny sposób i nie potrafią się tym polem podzielić.

Stosowalność praktyczna

Uff, sporo tej teorii, w dodatku nie jest ona wcale taka prosta. Najlepiej więc zająć się konkretnymi przypadkami: wtedy wszystko stanie się jasne, a przy okazji zdobędziesz praktyczne umiejętności generowania zdarzeń myszy.
A zatem spójrzmy na sposoby sztucznego wywoływania każdego z możliwych zdarzeń myszy.

Stosunkowo najprościej wytworzyć „oszukane” przyciśnięcia lub zwolnienia trzech przycisków myszy. Ignorujemy wówczas prawie wszystkie pola struktury MOUSEINPUT - wszystkie z wyjątkiem dwFlags, w którym ustawiamy tylko jedną jedyną flagę: którąś z MOUSEEVENTF_*UP/DOWN.
Zobaczmy przykładowy kod, generujący programowo wciśnięcie lewego przycisku myszy:

```
// struktura INPUT, przechowująca nasze zdarzenie
INPUT Klik;
ZeroMemory (&Klik, sizeof(INPUT));     // zerujemy ją

// ustawiamy odpowiednie parametry
Klik.type = INPUT_MOUSE;   // informujemy o tym, że zajmujemy się myszą
Klik.mi.dwFlags = MOUSEEVENTF_LEFTDOWN;      // lewy przycisk "w dół"
```

```
SendInput (1, &Klik, sizeof(INPUT));         // generujemy zdarzenie127
```

Nieco bardziej skomplikowane jest zasymulowanie kliknięcia jednym z dwóch dodatkowych przycisków - wymaga to wykorzystania jeszcze pola mouseData:

```
Klik.mi.mouseData = XBUTTON1;           // przycisk X1
Klik.mi.dwFlags = MOUSEEVENTF_XDOWN;   // dodatkowy przycisk "w dół"
SendInput (1, &Klik, sizeof(INPUT));   // i jazda :D
```

Analogicznie jak w dwóch powyższych kodach możemy również emulować zwolnienie wciśniętych przycisków, zamieniając flagi *DOWN na *UP.

Następnym interesującym wydarzeniem jest ruch myszy. Jak można wnioskować z opisu struktury MOUSEINPUT, może on odbywać się na dwa sposoby. Pierwszym jest natychmiastowa teleportacja kursora w określony rejon ekranu:

```
INPUT Ruch;
ZeroMemory (&Ruch, sizeof(INPUT));

// ustawiamy kursor w środku ekranu
Ruch.type = INPUT_MOUSE;
Ruch.mi.dx = GetSystemMetrics(SM_CXSCREEN) / 2;    // współ. pozioma
Ruch.mi.dy = GetSystemMetrics(SM_CYSCREEN) / 2;    // współ. pionowa
Ruch.mi.dwFlags = MOUSEEVENTF_MOVE | MOUSEEVENTF_ABSOLUTE;     // flagi
SendInput (1, &Ruch, sizeof(INPUT));
```

Można zapytać, czym różni się powyższy kod od wywołania SetCursorPos() (pomijając większą jego długość)?... Odmienność tych dwóch dróg osiągnięcia celu jest żadna - obie powodują dokładnie to samo. Zdaje się, że możliwość bezwględnej zmiany położenia kursora za pomocą SendInput() została raczej gwoli kompletności w symulowaniu myszy - świadczy o tym choćby fakt, iż działanie to wymaga podania dodatkowej flagi. Jedynie dołączenie MOUSEEVENTF_VIRTUALDESK sprawia wyraźną różnicę, która jednak jest widoczna tylko w systemach z kilkoma monitorami[128].
Inaczej jest w przypadku relatywnego przesuwania kursora, gdy SendInput() jest całkowicie niezastąpiona (chyba że przez przestarzałą mouse_event()).

Przesunięcie kursora może też odbywać się w odniesieniu do jego bieżącej pozycji. Jak już kilkakrotnie wspominałem, wartości pól MOUSEINPUT::dx i MOUSEINPUT::dy zostaną wtedy zwyczajnie dodane do aktualnych współrzędnych myszy.
Takie działanie jest w zasadzie domyślne, gdyż nie wymaga podania żadnej dodatkowej flagi (naturalnie poza niezbędną MOUSEEVENTF_MOVE, określającą rodzaj symulowanego zdarzenia myszy):

```
Ruch.mi.dwFlags = MOUSEEVENTF_MOVE;    // bez MOUSEEVENTF_ABSOLUTE
```

Ponieważ użycie SendInput() jest jedynym sposobem na relatywne przesunięcie kursora, zaś przemieszczenie bezwględne ma swój odpowiednik w funkcji SetCursorPos(), flaga MOUSEEVENTF_ABSOLUTE jest używana raczej rzadko. SetCursorPos() jest zwyczajnie prostszą drogą osiągnięcia tego samego celu, czyli ustawienia kursora w ściśle określonym miejscu ekranu.

[127] Korzystamy tu z operatora pobrania adresu, ponieważ mamy pojedynczą zmienną (strukturę), a nie tablicę. Klik możnaby aczkolwiek zadeklarować jako INPUT Klik[1];, lecz wtedy musielibyśmy odwoływać się do jego pól poprzez poprzez Klik[0].. Poza tym tablica składająca się z jednego elementu to raczej dziwny twór, nieprawdaż? :) (podobne uwagi mogą dotyczyć także każdego z następnych kodów w tym akapicie)
[128] SetCursorPos() potrafi przesuwać kursor tylko w obrębie aktualnego monitora, zaś SendInput() ze wspomnianą flagą może działać na całym pulpicie, rozciągniętym nawet na kilka monitorów.

Ostatnią akcją związaną z myszą jest obrót jej rolki. Emulowanie tego zjawiska nie należy do trudnych zadań: wiemy, że wartość żądanego obrotu, wyrażoną jako (pod)wielokrotność `WHEEL_DELTA`, należy wpisać w polu `MOUSEINPUT::mouseData`. Oprócz tego należy jeszcze podać odpowiednią flagę; w całości wygląda to mniej więcej tak:

```
INPUT Obrot;
ZeroMemory (&Obrot, sizeof(INPUT));

// obrót rolki o jeden krok w przód ("od użytkownika")
Obrot.type = INPUT_MOUSE;
Obrot.mi.mouseData = WHEEL_DELTA;      // wartość obrotu rolki
Obrot.mi.dwFlags = MOUSEEVENTF_WHEEL;  // akcja == obrót rolką
SendInput (1, &Obrot, sizeof(INPUT));  // działamy
```

Pamiętajmy, że symulowanie obrotu rolką myszy jest możliwe, tylko wtedy, gdy zainstalowana w komputerze mysz faktycznie taką rolkę posiada. O sprawdzaniu tej i innych cech myszy powiemy sobie w następnej sekcji.

Możliwości i ustawienia myszy

Jeszcze nie tak dawno temu niezwykle popularne wśród użytkowników komputerów były myszy zaledwie dwuprzyciskowe. Szybko dorobiły się jednak kolejnego przycisku, a nawet większej ich liczby; potem zyskały też obrotowe rolki, czasem nawet w liczbie większej niż jedna. Dzisiaj na komputerowym rynku i podkładkach użytkowników istnieje całe mnóstwo modeli urządzeń wskazujących, różniących się swoimi możliwościami.
Co więcej, na potencjał tych urządzeń można w dużym stopniu wpływać programowo, za pośrednictwem różnorodnych opcji, jakie oferuje Windows. Będąc całkiem elastycznym systemem operacyjnym, pozwala on na dostrojenie bardzo wielu ustawień z rejonu myszy i okolic.
Opcje te są ustawiane przede wszystkim przez użytkownika w Panelu Sterowania. Nie znaczy to jednak, że aplikacje działające pdo kontrolą systemu nie mają do nich dostępu. Przeciwnie, mogą one nie tylko odczytywać stan tychże opcji, ale też samodzielnie je zmieniać. W tym celu twórcy programów muszą oczywiście skorzystać z odpowiednich funkcji Windows API - tych, które teraz poznamy.

Są nimi głównie dwa wywołania: znane ci już skądinąd `GetSystemMetrics()` oraz nowe `SystemParametersInfo()`.
Przypomnijmy prototyp pierwszej z tych funkcji:

```
int GetSystemMetrics(int nIndex);
```

Być może pamiętasz, że w jej parametrze podajemy jedną ze stałych `SM_*` (oznaczających globalne ustawienia systemowe), a w zamian otrzymujemy wartość przyporządkowanej jej opcji. Jeśli nie, to właśnie sobie o tym przypomniałeś :D
Druga z ważnych dla nas funkcji to `SystemParametersInfo()`:

```
BOOL SystemParametersInfo(UINT uiAction,
                          UINT uiParam,
                          PVOID pvParam,
                          UINT fWinIni);
```

Ma ona nieco więcej parametrów, gdyż służy nie tylko do pobierania, ale też do zmiany opcji systemowych. Niniejsza tabelka opisuje te parametry:

typy	***parametry***	***opis***
`UINT`	`uiAction`	Tu podajemy stałą identyfikującą **opcję**, której ustawienie chcemy

typy	parametry	opis
		pozyskać lub zmodyfikować. Każdej takiej opcji odpowiada stała o nazwie z przedrostkiem SPI_, a ich liczba oscyluje wokół setki. Nie będziemy oczywiście omawiać ich wszystkich; w tym podrozdziale zajmiemy się tylko tymi, które dotyczą myszy.
UINT PVOID	uiParam pvParam	Są to dwa **parametry specyficznego przeznaczenia**, których użycie zależy od wartości uiAction.
UINT	fWinIni	Ten parametr określa **sposób powiadomienia** działających programów o zainstniałej zmianie ustawienia systemowego. Zwykle nie przejmujemy się tym parametrem i wpisujemy doń zero.

Tabela 51. Parametry funkcji SystemParametersInfo()

Warto zajrzeć do opisu SystemParametersInfo() w MSDN. Jest w nim zawarta m.in. pełna lista wartości, jakie może przyjmować parametr uiAction.

W dalszej części tej sekcji zajmiemy się niektórymi z wyliczeniowych stałych, jakie można przekazać do funkcji GetSystemMetrics() i SystemParametersInfo(), a także poznamy kilka innych, bardziej specyficznych funkcji. Rzecz jasna, wszystkie te elementy Windows API będą dotyczyły wyłącznie ustawień myszy.

Rekonesans możliwości myszy

Najsampierw chcielibyśmy wiedzieć, z jak potężnym urządzeniem mamy do czynienia. Innymi słowy, zrobimy teraz szybki wgląd w arsenał funkcji, w które została wyposażona mysz.

Czy jest na pokładzie…?

Mało kto zdaje sobie sprawę, że myszka **nie jest niezbędnym elementem** zestawu komputerowego, pracującego pod kontrolą systemu Windows. Nasz okienkowy OS radzi sobie całkiem dobrze, mając do dyspozycji wyłącznie klawiaturę. Tego samego nie można zwykle powiedzieć o użytkowniku pozbawionym myszy, co jednak nie znaczy, że takich użytkowników już nie ma. Zobaczmy zatem, jak sprawdzić obecność myszy w komputerze.

Na szczęście jest to bardzo proste i ogranicza się do wywołania funkcji GetSystemMetrics() z parametrem SM_MOUSEPRESENT:

```
BOOL bMyszkaObecna = GetSystemMetrics(SM_MOUSEPRESENT);
```

W wyniku otrzymujemy wartość TRUE lub FALSE o oczywistym znaczeniu; możemy ją wykorzystać chociażby tak:

```
if (!bMyszkaObecna)
{
   MessageBox (NULL, "Ten program nie może działać bez myszki!",
               "Brak myszy", MB_OK | MB_ICONSTOP)
   PostQuitMessage (0);
}
```

Niemniej pamiętajmy, że pomimo powszechności występowania myszek w komputerach użytkowników, dobry program powinien zapewniać również wygodne wsparcie dla klawiatury.

Liczba przycisków

Poszczególne modele myszek różnią się między innymi liczbą dostępnych przycisków. Dzisiejsze minimum zakłada przynajmniej trzy przyciski: lewy, środkowy i prawy, ale rzeczywista ich ilość może być większa lub mniejsza.

Liczbę przycisków myszy też pobieramy za pomocą `GetSystemMetrics()`, lecz tym razem parametrem jest `SM_CMOUSEBUTTONS`:

```
UINT uPrzyciskiMyszy = GetSystemMetrics(SM_CMOUSEBUTTONS);
```

W wyniku otrzymujemy naturalnie liczbę całkowitą, określającą ilość dostępnych przycisków myszy - lub zero, jeśli mysz nie jest obecna w systemie.

Wykrywanie rolki

Prawie niezbędnym elementem myszy stała się rolka, służąca do przewijania długich dokumentów, a okazjonalnie pełniąca honory środkowego przycisku.

Posiadanie przez myszkę rolki możemy ustalić za pomocą… funkcji `GetSystemMetrics()` oczywiście :) Tym razem jej parametrem musi być `SM_MOUSEWHEELPRESENT`:

```
BOOL bRolkaObecna = GetSystemMetrics(SM_MOUSEWHEELPRESENT);
```

Trzeba tu przypomnieć, że brak lub obecność kółka myszy determinuje nie tylko oczywiste tego następstwa, ale też możliwość programowej symulacji obrotu rolki poprzez funkcje `SendInput()` czy `mouse_event()`. Nie można bowiem udawać działania czegoś, czego tak naprawdę nie ma.

Ustawienia podwójnego kliknięcia

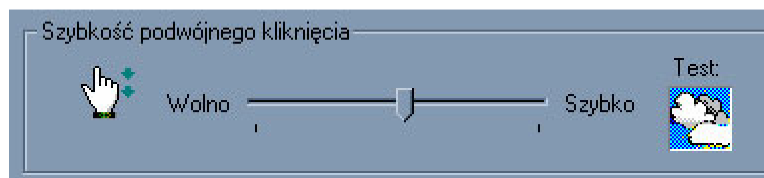
Mechanizm podwójnego kliknięcia wprowadzono do Windows głównie po to, aby skrócić czas wykonywania najczęstszych operacji. Przykładem niech będzie zarządzanie plikami w Eksploratorze Windows: pojedyncze kliknięcie powoduje zaznaczenie pliku, co pozwala na wykonanie na jego rzecz pewnych poleceń, dostepnych na pasku menu programu. Bardzo często takim poleceniem będzie otwarcie pliku, więc wymyślono dla niego łatwiejszy sposób wywoływania - podwójne kliknięcie. Nie wymaga ona długiej wędrówki kursorem do paska menu, a jedynie dwóch szybkich wciśnięc lewego przycisku.

Chociaż podwójne kliknięcie jest z pewnością wygodne, początkującym użytkownikom, nieobytym z myszą, może ono sprawiać problemy. Dlatego też system Windows umożliwia dostrojenie parametrów dwukrotnego kliknięcia tak, aby odpowiadały ony indywidualnym preferencjom.
W tym paragrafie zobaczymy, w jaki sposób nasze programy mogą pobierać i ustawiać opcje podwójnego kliknięcia.

Interwał czasu pomiędzy kliknięciami

Aby system mógł zinterpretować wciśnięcia przycisku myszy jako dwukrotne kliknięcie, muszą one zaistnieć odpowiednio szybko. Jeżeli drugie kliknięcie będzie spóźnione, wówczas Windows zarejestruje dwa pojedyncze przyciśnięcia, a nie jedno podwójne. Ważne jest więc właściwe dopasowanie ustawienia systemowego, regulującego **maksymalny interwał czasu pomiędzy dwoma kliknięciami**, które będą rejestrowane jako jedno podwójne.

Rzeczona opcja znajduje się w aplecie *Właściwości: Mysz* Panelu Sterowania:



Screen 62. Ustawianie szybkości dwukrotnego kliknięcia (podziękowania dla gemGrega za wykonanie tego screena)

Naturalnie, możliwa jest także jej programowa kontrola za pomocą funkcji Windows API. Zauważ, że choć użytkownik może mówić o szybkości dwukrotnego kliknięcia, to my, programiści, będziemy zajmowali się czasem pomiędzy oboma kliknięciami. Nietrudno domyślić się, że obie te wielkości są do siebie odwrotnie proporcjonalne, tj. większa szybkość oznacza mniejszy interwał czasu.
Zobaczmy teraz, jak można pobrać i ustawić tę wielkość systemową.

A więc: w celu pobrania aktualnego interwału czasu dwukrotnego kliknięcia (ang. *double-click time*) należy posłużyć się specjalną funkcją `GetDoubleClickTime()`:

```
UINT uCzasDwukliku = GetDoubleClickTime();
```

Wywołanie jej jest banalnie proste, bo nie potrzebuje żadnych dodatkowych parametrów. W wyniku dostajemy żądany interwał czasu w milisekundach (tysięcznych częściach sekundy).

Ustawienie czasu dwukrotnego kliknięcia jest natomiast możliwe aż na dwa sposoby:

- poprzez wywołanie specjalnej funkcji `SetDoubleClickTime()`. Podajemy jej nowy interwał czasu, oczywiście w milisekundach:

  ```
  SetDoubleClickTime (250);
  ```

- za pomocą funkcji `SystemParametersInfo()`; należy wtedy w jej pierwszym parametrze podać stałą `SPI_SETDOUBLECLICKTIME`, zaś w drugim nową wartość ustawienia (trzeci i czwarty parametr wypełniamy zerami):

  ```
  SystemParametersInfo (SPI_SETDOUBLECLICKTIME, 250, NULL, 0);
  ```

  Z uwagi na fakt, że pierwsza metoda jest znacznie łatwiejsza, nie obrażę się, jeżeli całkiem zapomnisz o drugiej :)

Powiedzmy jeszcze, że w obu przypadkach możemy jako interwał podać zero… Nie, nie spowoduje to wyłączenia dwukrotnego kliknięcia. W takiej sytuacji Windows przyjmie po prostu wartość domyślną dla tego ustawienia, czyli 500 milisekund (pół sekundy).

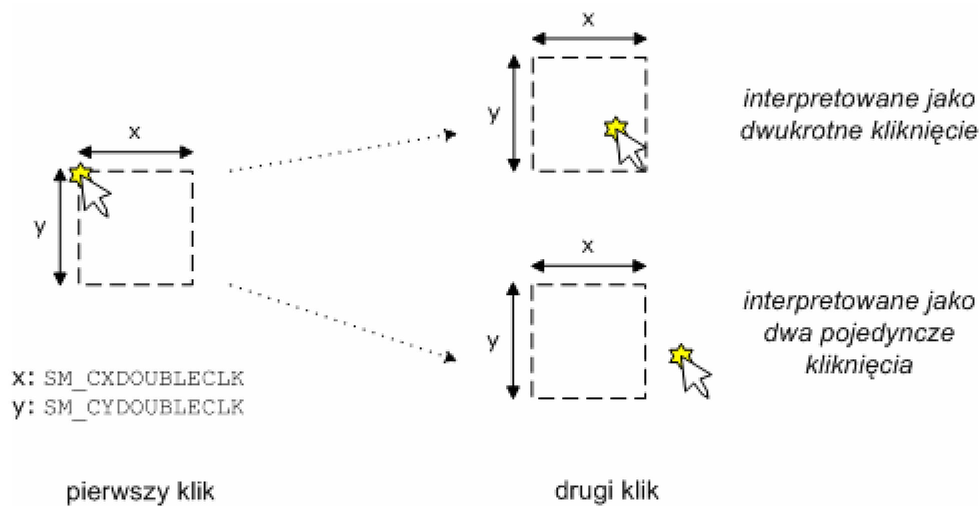
Dopuszczalne przesunięcie kursora

Zmieszczenie się w wąskim przedziale czasu nie jest jedynym wymogiem, jakie stawia system wobec dwukrotnego kliknięcia. Drugim (i na szczęście ostatnim) jest nieruchomość myszy podczas dokonywania operacji.
Nie znaczy to aczkolwiek, że kursor między jednym a drugim przyciśnięciem nie może się przesunąć nawet o piksel. Toleracja w tym względzie jest nieco większa, a co więcej - można ją również ustawić.

Użytkownik może tego dokonać przy pomocy narzędzia Tweak UI, ustalając wartość na zakładce *Mouse*:

Screen 63. Dostrajanie tolerancji przesunięcia myszy podczas dwukrotnego kliknięcia (opcja *Double-click*; druga z opcji, *Drag*, określa najmniejsze przemieszczenie, które inicjuje operację przeciągania - o tej wartości nie będziemy tutaj mówić)
(program Tweak UI możesz ściągnąć ze strony Microsoftu)

Podobnie jak to było w przypadku szybkości dwukrotnego kliknięcia, dla programistów obowiązuje nieco inna miara opisywanej opcji. Tweak UI stosuje odległość pomiędzy miejscami kolejnych kliknięć, zaś Windows API posługuje się okalającym je **prostokątem**. System wymaga, aby oba kliknięcia zawierały się w tym prostokącie (o domyślnych wymiarach kilku pikseli), gdyż w przeciwnym razie nie zostaną potraktowane jako dwukrotne:



Schemat 43. Prostokąt dwukrotnego kliknięcia

Na tym schemacie nieprzypadkowo opatrzyłem wymiary prostokąta (*x* i *y*) nazwami `SM_CXDOUBLECLK` i `SM_CYDOUBLECLK`. Są to bowiem nazwy stałych, jakie należy przekazać do funkcji `GetSystemMetrics()` celem pobrania wymiarów prostokąta dwukrotnego kliknięcia:

```
SIZE cProstokatDwukliku;
cProstokatDwukliku.cx = GetSystemMetrics(SM_CXDOUBLECLK);
cProstokatDwukliku.cy = GetSystemMetrics(SM_CYDOUBLECLK);
```

`SIZE` to predefiniowany typ strukturalny z WinAPI, który zawiera dwa pola typu `LONG` - `cx` i `cy`, przeznaczone do przechowywania wymiarów (szerokości i wysokości) wszelkiego rodzaju obiektów.

Możliwe jest też ustawienie tych wartości przy pomocy funkcji `SystemParametersInfo()` oraz stałych `SPI_SETDOUBLECLKWIDTH/HEIGHT`:

```
SystemParametersInfo (SPI_SETDOUBLECLKWIDTH, 1, NULL, 0);
SystemParametersInfo (SPI_SETDOUBLECLKHEIGHT, 1, NULL, 0);
```

Wykonanie powyższego kodu ustawi prostokąt dwukliku na rozmiar 1×1 piksela, zatem Windows nie będzie teraz tolerował żadnego przesunięcia myszy w czasie dwukrotnego kliknięcia.

Ułatwienia dostępu

Teraz poznamy takie opcje myszy, które ułatwiają korzystanie z komputera także osobom z upośledzoną sprawnością ruchową i niepełnosprawnym. Co ciekawe, niektóre z tych opcji są na tyle przydatne, że bywają pomocne również dla w pełni sprawnych użytkowników.

Zamiana przycisków myszy

Leworęczni użytkownicy Windows chcieliby trzymać mysz w w lewej ręce, po lewej stronie monitora. Aby w takim ustawieniu swobodnie z niej korzystać, należy **zamienić miejscami** (ang. *button swap*) funkcjonalność lewego i prawego przycisku myszy. Można to uczynić w Panelu Sterowania.

Program działający w Windows może pobrać stan ten opcji za pomocą funkcji `GetSystemMetrics()` z parametrem `SM_SWAPBUTTON`:

```
BOOL bZamienionePryciski = GetSystemMetrics(SM_SWAPBUTTON);
```

Pamiętamy, że ustawienie tej opcji ma znaczenie w momencie, gdy sprawdzamy wciśnięcie lewego (`VK_LBUTTON`) lub prawego (`VK_RBUTTON`) przycisku myszy za pomocą funkcji `GetAsyncKeyState()`. Należy wtedy uwzględnić tłumaczenie fizycznych przycisków myszy na logiczne, jak to było mówione w odpowiednim paragrafie poprzedniej sekcji.

Zmiana opisywanej opcji jest natomiast możliwa dwiema drogami:

- poprzez dedykowaną funkcję `SwapMouseButton()`. Należy jej podać wartość logiczną, określającą włączenie (`TRUE`) lub wyłączenie (`FALSE`) opcji:

  ```
  SwapMouseButton (FALSE);   // domyślne znaczenie przycisków
  ```

  Funkcja ta zwraca w wyniku poprzednie ustawienie
- przy pomocy wywołania `SystemParametersInfo()` ze stałą `SPI_SETMOUSEBUTTONSWAP` oraz z nowym stanem opcji w parametrze uiParam:

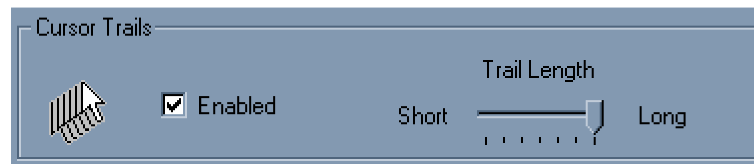
  ```
  SystemParameters (SPI_SETMOUSEBUTTONSWAP, FALSE, NULL, 0);
  ```

  Podobnie jak w przypadku czasu dwukrotnego kliknięcia, specjalnie wydelegowana funkcja jest znacznie protsza w użyciu niż uniwersalne `SystemParametersInfo()`.

Uważajmy, gdyż tak poczyniona zmiana jest znacząca i dotyczy natychmiast **całego systemu**, zatem należy ją stosować tylko w uzasadnionych okolicznościach.

Ślad kursora

Na niektórych monitorach (szczególnie starszych ciekłokrystalicznych, w laptopach) obraz jest na tyle niewyraźny, że dostrzeżenie ruchu jest często trudne. Również wady wzroku (spowodowane np. długim siedzeniem przed komputerem :D) mogą to utrudniać. W takiej sytuacji można włączyć ciekawą *nomen omen* opcję **śladu kursora** (ang. *cursor trails*), podążającego za ruchomą strzałką myszy.



Screen 64. Włączenia śladu kursora dokonujemy we właściwościach myszy w Panelu Sterowania, tam też ustalamy jego długość. Nawet jeśli nie masz kłopotów ze śledzeniem kursora, ta opcja może być interesującym fajerwerkiem graficznym

Pewnie cię nieco zmartwię, ale do programowej kontroli tego ustawienia służy wyłącznie funkcja `SystemParametersInfo()`. By pobrać stan opcji należy wywołać tę funkcję ze stałą `SPI_GETMOUSETRAILS`:

```
int nSladKursora;
SystemParametersInfo (SPI_GETMOUSETRAILS, 0, &nSladKursora, 0);
```

Wynik odczytujemy ze zmiennej liczbowej, do której wskaźnik przekazujemy w trzecim parametrze. Jeśli ma ona wartość większą od 1, wtedy opcja jest włączona; jednocześnie otrzymana wartość wskazuje na długość śladu, jaki zostawia kursor (innymi słowy, ile razy jest on powielany).

Modyfikacja śladu wymaga wywołania naszej wielce szacownej funkcji z parametrem `SPI_SETMOUSETRAILS`. Nową długość śladu podajemy z kolei w drugim parametrze, np.:

```
SystemParametersInfo (SPI_SETMOUSETRAILS, 4, NULL, 0);
```

W powyższy sposób ustawiamy ślad kursora średniej długości (strzałka będzie replikowana 4 razy).

Sonar

Inną metodą poprawy widoczności kursora jest **sonar** (ang. *mouse sonar*). Nie ma on niestety nic wspólnego z łodziami podwodnymi i pełnymi skarbów okrętami na dnie morza. Mechanizm sonaru w systemie Windows polega na wizualnym wyróżnieniu kursora myszy koncentrycznymi kręgami. Są one pokazywane, gdy użytkownik wciśnie klawisz *Ctrl*. Sonar ułatwia więc lokalizowanie kursora na ekranie zaśmieconym oknami, przy wysokiej rozdzielczości lub zepsutym wzroku.

> Sonar jest dostępny tylko Windows Me i XP.

Sprawdzenia, czy sonar jest włączony, dokonujemy poprzez `SystemParametersInfo()` ze stałą `SPI_GETMOUSESONAR`:

```
BOOL bSonar;
SystemParametersInfo (SPI_GETMOUSESONAR, 0, &bSonar, 0);
```

Wynik kontroli ląduje oczywiście w zmiennej `BOOL`owskiej, której adres podajemy w trzecim parametrze.

W(y)łączenia sonaru dokonujemy także za pomocą doskonale już znanej funkcji o długiej nazwie, a także jej parametru `SPI_SETMOUSESONAR`:

```
SystemParametersInfo (SPI_SETMOUSESONAR, TRUE, NULL, 0);// włącz. sonaru
```

Informację o nowym stanie opcji sonaru wysyłamy z drugim argumentem funkcji.

KlawiszeMyszy

Bardzo ciekawą opcją z zakresu ułatwień Windows są KlawiszeMyszy (ang. *MouseKeys*). Pozwala ona na przesuwanie kursora myszy za pomocą klawiszy strzałek klawiatury numerycznej (8, 4, 6 i 2) oraz *Shift* i *Ctrl*. Te dwa ostatnie oferują możliwość zwolnienia lub przyspieszenia ruchu kursora.
Osobiście sądzę, że ciężko zaliczyć tę możliwość do kategorii ułatwień, bo opanowanie mechaniki kursora tą metodą wydaje się o wiele trudniejsze i mniej efektywnie niż poruszanie. KlawiszeMyszy mają jednak pewną niezaprzeczalną zaletę, która może być bardzo przydatna podczas tworzenia grafiki czy konstruowania interfejsu użytkownika. Mam tu na myśli możliwość precyzyjnego poruszania „myszą" - z dokładnością do jednego piksela. Bardzo ułatwia do rozmieszczanie elementów grafiki czy UI.

Programowa kontrola tego ustawienia, posiadającego sporo podopcji, jest dość obszernym zagadnieniem, na który chyba nie ma tu już miejsca. Jeżeli cię to interesuje, poczytaj w MSDN opis struktury `MOUSEKEYS` oraz informacje o parametrach `SPI_GET/SETMOUSEKEYS` funkcji `SystemParametersInfo()`.

BlokadaKliknięcia

Jeżeli masz kłopoty z przeciąganiem kursora myszy przy wciśniętym przycisku, możesz włączyć opcję Windows znaną jako *ClickLock*. Tłumaczyć ją można jako 'BlokadaKliknięcia'. Działa ona w prosty sposób: jeśli użytkownik wciśnie logicznie lewy przycisk myszy i przytrzyma go przez określony czas (który można regulować), system uzna, że rozpoczyna się operacja przeciągania. Można teraz już zwolnić przycisk myszy, a Windows nadal będzie **traktował go jako wciśnięty**. Poruszając myszą w zwykły sposób (przy fizycznie zwolnionym przycisku), można teraz dokonywać przeciągania bez konieczności przytrzymywania przycisku palcem, jak to było dotychczas. Kiedy zaś uznamy, że chcemy zakończyć przeciąganie, kilkamy po prostu jeszcze raz, aby upuścić podniesiony obiekt (np. plik).

BlokadaKliknięcia działa tylko w Windows Me i XP.

Niniejsza opcja jest także domeną funkcji `SystemParametersInfo()`. By dowiedzieć się, czy jest aktualnie włączona, należy zastosować parametr `SPI_GETMOUSECLICKLOCK`:

```
BOOL bBlokadaKlikniecia;
SystemParametersInfo (SPI_GETMOUSECLICKLOCK, 0, &bBlokadaKlikniecia, 0);
```

Ze zmiennej typu `BOOL`, której adres należy podać w pokazany wyżej sposób, odczytamy stan opcji. Jak można się domyślić, `TRUE` oznacza włączone, a `FALSE` - wyłączone ustawienie.
Jego zmiana to z kolei wywołanie z użyciem `SPI_SETMOUSECLICKLOCK`:

```
// włączenie BlokadyKliknięcia
SystemParametersInfo (SPI_SETMOUSECLICKLOCK, TRUE, NULL, 0);
```

Nowy status opcji podajemy w drugim parametrze, `uiParam`.

Kiedy już włączymy Blokadę, możemy też regulować czas (w milisekundach), po którym przycisk myszy zostanie zablokowany. Do jego pobrania służy `SPI_GETMOUSECLICKLOCKTIME`:

```
unsigned uCzasBlokadyKlikniecia;
SystemParametersInfo (SPI_GETMOUSECLICKLOCKTIME, 0,
                      &uCzasBlokadyKlikniecia, 0);
```

Ponownie otrzymujemy pożądaną wartość w zmiennej, do której wskaźnik podajemy. To się już robi całkiem proste, prawda? ;)
Ustawienie nowego interwału czas oznacza konieczność użycia stałej `SPI_SETMOUSECLICKLOCKTIME` - o tak:

```
SystemParametersInfo (SPI_SETMOUSECLICKLOCKTIME, 500, NULL, 0);
```

Nowy czas podajemy w drugim parametrze. Powinien on być nie mniejszy niż kilkaset milisekund, gdyż w przeciwnym razie może być zbyt krótki na wykonanie zwykłego kliknięcia (bez przeciągania).

***

Na tym kończy się nasza opowieść o myszy. Były to długa historia, a mimo nie dotarliśmy do jej definitywnego epilogu. Oprócz kilku nieomówionych ustawień systemowych (o których poczytasz sobie w MSDN), nie zajęliśmy się jeszcze zagadnieniem własnego kształtu kursora myszy - stanie się to w rozdziale o zasobach Windows.

Wykorzystanie klawiatury

Czy można sobie wyobrazić komputer bez klawiatury? „Oczywiście” — odpowiesz pewnie — „Weźmy choćby HALa z *Odysei kosmicznej 2001*!” Faktycznie, masz rację: komputery sterowane głosem (może nie tak inteligentne jak HAL) istnieją już dzisiaj i ta forma komunikacji z maszyną na pewno będzie się rozwijać. Jednak klawiatury nie znikną nigdy, a powodem tego jest zwykła potrzeba poufności i prywatności - nie chcielibyśmy przecież, aby wszyscy dookoła słyszeli, jakie informacje wprowadzamy do komputera. Pośrednictwo klawiatury stwarza więc pewną ochronę danych - o ile nikt nie zagląda nam przez ramię :)

Podobnie jak myszki i wszystkie urządzenia peryferyjne, klawiatury przez lata były udoskonalane. Ich przodkami są zapewne maszyny do pisania, jednak od początku istnienia komputerów osobistych ich klawiatury nie ograniczały się tylko do liter i cyfr. Szybko pojawiły się dodatkowe klawisze strzałek, klawisze funkcyjne, a ostatnio nawet specjalne klawisze ułatwiające serfowanie po Internecie.
Wygląd i kształt klawiatur także ulegał przeobrażeniom. Ostatnimi czasy popularne są tzw. klawiatury ergonomiczne, których część alfanumeryczna jest rozdzielona na dwoje, a klawisz spacji odpowiednio wygięty. W komputerach zadomowiły się też klawiatury bezprzewodowe, tworząc często zestawy z myszkami.

Fotografia 5 i 6. Przykładowe modele dzisiejszych klawiatur
(fotografie pochodzą z serwisu internetowego firmy Logitech)

Ten podrozdział stanowi przegląd możliwości Windows API w zakresie współpracy z klawiaturą. Poznamy w nim zdarzenia, jakie są generowane w reakcji na wciśnięcia klawiszy, sposoby na pobieranie stanu klawiatury oraz symulowania jego zmiany. Popatrzymy też na ustawienia systemowe, związane z tym urządzeniem.

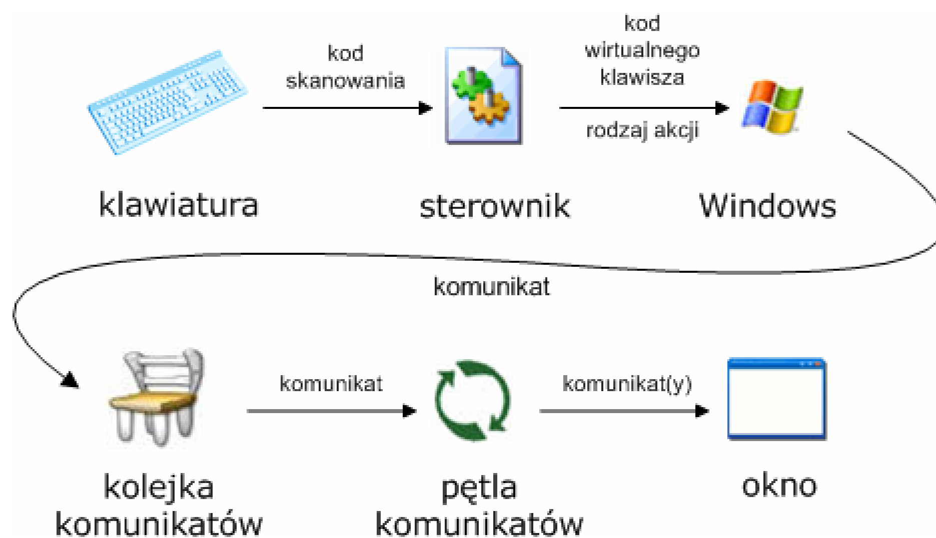
Zdarzenia klawiatury

Najważniejsze są oczywiście komunikaty o zdarzeniach. Reakcja na nie będzie lwią częścią obsługi klawiatury w naszych programach okienkowych. Zajmijmy się zatem ich powstawaniem i rodzajami.

Potok klawiszy

Zanim fakt fizycznego wciśnięcia lub zwolnienia klawisza dotrze do okna w postaci odpowiedniego komunikatu, przechodzi on dosyć długą drogę. Nazywam ją potokiem klawiszy (ang. *keys pipeline*) lub modelem wejścia od klawiatury (ang. *keyboard input model*). Obrazuje on, w jaki sposób system Windows zamienia sygnały pochodzące od urządzenia na komunikaty trafiające do okien.

Potok klawiszy można przedstawić obrazowo na schemacie:



Schemat 44. Potok klawiszy w Windows

Opiszmy każdy z przedstawionych tu etapów, jakie pokonuje informacja o zdarzeniu klawiatury.

Tłumaczenie kodów

Gdy użytkownik wciska lub zwalnia klawisz, wiadomość o tym przedostaje się do systemu poprzez przerwanie klawiatury. Powiadomienie zawiera tzw. **kod skanowania** (ang. *scan code*) lub **kod OEM** (ang. *OEM code*) - po prostu pewną liczbę. Jest ona specyficzna dla każdego klawisza, a dodatkowo różni się w przypadku jego wciśnięcia i zwolnienia.

Kody OEM są ponadto charakterystyczne dla urządzenia i zależne od niego. Znaczy to, że **różne modele** klawiatur mogą generować **różne kody** skanowania. Programując w niskopoziomowym asemblerze, należałoby uwzględniać wszystkie możliwe kody interesujących nas klawiszy. Ponieważ jednak pracujemy w bardziej przyjaznym środowisku Windows, ominie nas ta wątpliwa przyjemność.
Kod skanowania zostaje bowiem przetłumaczony na inny, uniwersalny dla każdej możliwej klawiatury. Takiego tłumaczenia dokonuje **sterownik urządzenia** (ang. *device driver*) - w tym przypadku chodzi oczywiście o sterownik klawiatury, dostarczany wraz ze sprzętem lub systemem Windows.

Kody wirtualnych klawiszy

Produktem tłumaczenia jest **kod wirtualnego klawisza** (ang. *virtual-key code*), zwany dla wygody **kodem wirtualnym**. Jest to 16-bitowy, jednoznaczny identyfikator każdego obsługiwanego przez Windows klawisza na dowolnej klawiaturze. Jak wiemy, swoje kody mają także przyciski myszy.

Większości wirtualnych kodów przyporządkowane są odpowiednie stałe o nazwach zaczynających się od przedrostka `VK_`. Poniższa tabelka przedstawia część najpopularniejszych klawiszy (z wyłączeniem przycisków myszy, których kody były podane wcześniej):

stała	kod	klawisz	stała	kod	klawisz
VK_BACK	0x0008	*Backspace*	VK_NEXT	0x0022	*Page Down*
VK_TAB	0x0009	*Tab*	VK_END	0x0023	*End*
VK_RETURN	0x000D	*Enter*	VK_HOME	0x0024	*Home*
VK_SHIFT	0x0010	*Shift*	VK_LEFT	0x0025	*Strzałka w lewo*
VK_CONTROL	0x0011	*Ctrl*	VK_UP	0x0026	*Strzałka w górę*
VK_MENU	0x0012	*Alt*	VK_RIGHT	0x0027	*Strzałka w prawo*
VK_ESCAPE	0x001B	*Esc*	VK_DOWN	0x0028	*Strzałka w dół*
VK_SPACE	0x0020	*Spacja*	VK_INSERT	0x002D	*Insert*
VK_PRIOR	0x0021	*Page Up*	VK_DELETE	0x002E	*Delete*

Tabela 52. Niektóre kody wirtualnych klawiszy

Pełną listę możesz znaleźć w Tablicach C oraz w MSDN.

Zapakowanie w komunikat

Drugim produktem tłumaczenia, dokonywania przez sterownik klawiatury, jest wydzielona informacja o rodzaju zainstniałego zdarzenia (akcji). Na jej podstawie Windows wybiera komunikat, jaki wygeneruje. A możliwości są generalnie dwie:

- wciśnięcie klawisza (ang. *key down*)
- zwolnienie klawisza (ang. *key up*)

System decyduje się na jedną z nich, tworzy odpowiedni komunikat i wysyła go do kolejki komunikatów.

Produkcja znaków

Stamtąd prędzej czy później wiadomość o zdarzeniu zostanie pobrana przez aplikację i trafi do jej pętli komunikatów. Pętli, która w najprostszej wersji wygląda tak:

```
MSG msgKomunikat;
while (GetMessage(&msgKomunikat, NULL, 0, 0))
{
    TranslateMessage (&msgKomunikat);
    DispatchMessage (&msgKomunikat);
}
```

W każdym z jej możliwych wariantów niezmienne są przywołania dwóch funkcji - `TranslateMessage()` i `DispatchMessage()`. W tym momencie jesteśmy niezwykle zainteresowani pierwszą z nich.

Gdy przedstawiałem pętlę komunikatów na początku kursu WinAPI, wspomniałem, że niewielu programistów wie, co naprawdę robi funkcja `TranslateMessage()`. Zapewne nie jest to do końca prawdą, niemniej faktem jest, że nazwa tej funkcji niezupełnie oddaje wykonywaną przezeń czynność.

Przede wszystkim `TranslateMessage()` nie dokonuje żadnego tłumaczenia komunikatu - cokolwiek miałoby to znaczyć. Jej rolą jest bowiem produkcja dodatkowych zdarzeń, tzw. **komunikatów o znakach** (ang. *character messages*) Generuje je na podstawie następujących po sobie wciśnięć i zwolnień tego samego klawisza na klawiaturze.
O zdarzeniach znaków i odpowiadających im komunikatach powiemy sobie więcej w którymś z nadchodzących paragrafów.

Finisz

Na koniec wszystkie wytworzone komunikaty trafiają do docelowego okna. Dotyczy to zarówno stworzonych w jądrze systemu powiadomień o zdarzeniach klawiszy, jak i komunikatów o znakach, posłanych do kolejki przez `TranslateMessage()`. Wszystkie te zdarzenia są normalnie obsługiwane przez procedurę zdarzeniową okna, które je ostatecznie otrzyma.

Właśnie - którego okna? Skąd Windows wie, komu przekazywać informacje o zdarzeniach klawiatury? Czas odpowiedzieć i na to pytanie.

Fokus

Komputer posiada tylko jedną klawiaturę. To niewiele, jeżeli weźmie się pod uwagę praktycznie nieograniczoną liczbę programów, jakie mogą jednocześnie działać w Windows. Dostęp do tego cennego sprzętu musi być zatem wydzielany, co spoczywa na barkach systemu operacyjnego.

Radzi on sobie z tym zadaniem poprzez kontrolę tzw. **wejściowego fokusu klawiatury** (ang. *keyboard input focus*) - w skrócie **fokusu** (ang. *focus*). Jest to specjalna właściwość, którą w danej chwili może posiadać tylko jedno okno.

Okno posiadające **fokus** otrzymuje wszystkie komunikaty o **zdarzeniach klawiatury**[129].

Dzięki temu, że któreś z okien w systemie posiada fokus, Windows wie, do kogo kierować pojawiające się informacje od klawiatury. Takie okno jest więc ostatecznym celem potoku klawiszy.

Przełączanie fokusu

Fokus jest dla klawiatury mniej więcej tym samym, co dla myszki jest władza nad nią (ang. *mouse capture*). Występują tu jednak pewne różnice.
Przede wszystkim, fokus nie jest tak ulotną własnością jak *capture* - zmienia się on rzadziej i zazwyczaj tylko wtedy, gdy użytkownik sam sobie tego zażyczy. Osoba obsługująca aplikacje jest też znacznie bardziej świadoma istnienia tego zjawiska, gdyż odzwierciedla się ono w wyglądzie okien. Te posiadające fokus odznaczają się np. innym kolorem paska tytułu, migającym kursorem (w polach tekstowych) czy kolorową belką zaznaczenia (w listach).

Jak odbywa się przekazywanie fokusu między oknami?… Jeżeli programy nie ingerują w ten proces, to dzieje się to zwykle poprzez:

- kliknięcie myszą w obszarze innego okna
- posłużenie się odpowiednim klawiszem (kombinacją) przełączania: *Tab* dla kontrolek potomnych, *Alt+Tab* dla nadrzędnych okien programów

Wynika stąd, że potoczne rozumienie „aktywności” okna to nic innego, jak właśnie posiadanie przezeń fokusu. Dla twórcy aplikacji fakt, że okno jest aktywne (ang. *enabled*), oznacza jednak co innego: wcale nie to, że będzie ono przyjmowało dane

[129] Dostaje też komunikat `WM_MOUSEWHEEL`, o czym powiedzieliśmy sobie wcześniej.

wejściowe od urządzeń zewnętrznych (myszy, klawatury), lecz jedynie to, iż **może** ono takie dane przyjmować.
Zatem okno nieaktywne nie może posiadać fokusu, a spośród aktywnych okien fokus ma tylko jedno.

Kontrolując fokus, Windows dba także, aby okna były odpowiednio informowane o jego zmianach. W tym celu wysyłane są dwa komunikaty:

- `WM_KILLFOCUS` otrzymuje okno, które straciło fokus na rzecz innego. Uchwyt tego nowego posiadacza jest zapisany w parametrze `wParam` niniejszego komunikatu (może to być `NULL`)
- `WM_SETFOCUS` dostaje z kolei to okno, które fokus otrzymało. W parametrze `wParam` zapisany jest wtedy uchwyt poprzedniego posiadacza (albo `NULL`)

Obsługa tych komunikatów jest konieczna, jeżeli posługujemy się tzw. karetką (ang. *caret*), służącą do oznaczania miejsca wpisywania tekstu w oknie. Nie będziemy się aczkolwiek zajmować bliżej tym zagadnieniem.
Dla programisty gier i wszelkich innych aplikacji działających na pełnym ekranie dwa powyższe komunikaty są również bardzo ważne. Pozwalają one wykryć czas, gdy program nie jest w kręgu zainteresowania użytkownika i oszczędzić niepotrzebnego wysiłku przeznaczanego np. na rendering kolejnych klatek.

Funkcje pomocnicze

Windows API zawiera dwe funkcje przeznaczone do kontroli fokusu klawiatury. Pierwszą z nich jest `SetFocus()`:

```
HWND SetFocus(HWND hWnd);
```

Po nazwie nietrudno domyślić się, że służy ona do przekazania fokusu innemu oknu; jego uchwyt podajemy w paramatrze funkcji. W wyniku dostajemy uchwyt poprzedniego posiadacza fokusu.

Drugą funkcją jest `GetFocus()`:

```
HWND GetFocus();
```

Tutaj sprawa jest jeszcze prostsza: wywołanie to zwraca po prostu uchwyt aktualnego właściciela fokusu klawiatury.

Parametry komunikatów klawiatury

Teraz moglibyśmy w zasadzie przejść już do omawiania każdego z komunikatów zdarzeniowych klawiatury. Będzie jednak lepiej i wygodniej, jeżeli najpierw skoncetrujemy się na ich parametrach `wParam` i `lParam`, jako że są one wspólne dla wszystkich komunikatów. Podobnego rozpatrzenia parametrów dokonaliśmy zresztą przy okazji zdarzeń myszy, więc tutaj tylko kontynuujemy tę chlubną tradycję ;)
Popatrzmy zatem na dane dodatkowe komunikatów klawiatury.

Parametr `wParam`

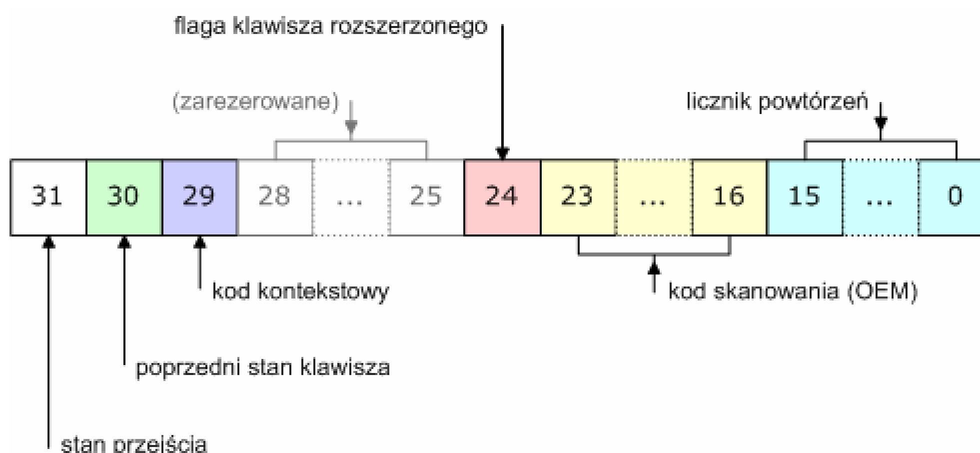
Zawartością tej danej jest specyficzny kod wciśniętego (lub puszczonego) klawisza. Rodzaj owego kodu zależy od typu komunikatu:

- komunikaty o klawiszach zawierają tutaj kod wirtualnego klawisza, czyli wartość, której odpowiada jedna ze stałych `VK_`
- komunikaty o znakach mają tu natomiast wpisany kod znaku, który został wprowadzony (najczęściej jest on w systemie Unicode)

Dokładniejsze wiadomości o obu rodzajach kodów otrzymasz przy omawianiu powiązanych z nimi komunikatów.

Parametr `lParam`

Drugi parametr komunikatu od klawiatury jest chyba najlepszym przykładem gęstego upakowania danych w zmiennej, która wydaje się tylko liczbą. Otóż `lParam` wcale nie jest tutaj liczbą - zawiera w sobie bowiem aż **sześć** (!) różnych informacji. Są one ułożone w odpowiednich bitach tej 32-bitowej wartości, jak to jest ukazane na schemacie:



Schemat 45. Informacje zawarte w parametrze `lParam` komunikatów klawiatury

Poszczególne części oznaczyłem kolorami - tak, aby ułatwić ci ich dopasowanie do poniższych opisów. Część z szarymi konturami nie jest używana przez aktualne wersje Windows i została zarezerwowana dla ewentualnych przyszłych celów.

Wytłumaczmy więc po kolei znaczenie każdej z tych danych - poczynając od najmłodszego bitu:

- **licznik powtórzeń** (ang. *repeat count*) określa nam liczbę przyciśnięć klawiszy, jakie reprezentuje otrzymany komunikat. W przypadku zdarzeń zwolnienia klawisza jest to zawsze 1; dla pozostałych wiadomości najczęściej też tak jest. Czasem jednak Windows nie nadąża z produkcją oddzielnych komunikatów dla każdego powtórzenia klawisza - dzieje się tak wtedy, gdy ustawimy dużą częstotliwość powtarzania w Panelu Sterowania i przytrzymamy wciśnięty klawisz
- **kod skanowania (OEM)** (ang. *OEM scan code*) jest niczym innym, jak tylko sprzetowym kodem, jaki klawiatura wysyła do swojego sterownika. Jak pamiętasz, odbywa się to na samym początku potoku klawiszy. Działanie sterownika klawiatury sprawia, że nie musimy się martwić o interpretację kodu skanowania, zależnego od sprzętu, ponieważ mamy uniwersalne kody wirtualnych klawiszy. Niemniej jednak Windows dołącza oryginalny kod do komunikatu, na wypadek gdyby był on komuś potrzebny
- **flaga klawisza rozszerzonego** (ang. *extended key flag*) mówi nam, czy wciśnięty/zwolniony klawisz należy do tzw. zestawu rozszerzonego klawiatury 101- lub 102-klawiszowej. Obecnie takie modele nie są niczym nadzwyczajnym, więc informacja ta nie służy zbyt wielkim celom i jest najczęściej ignorowana
- **kod kontekstowy** (ang. *context code*) tak naprawdę nie jest żadnym kodem („I całe szczęście" — powiesz zapewne ;D). Jest to tylko przełącznik wskazujący na to, czy zdarzenie klawiatury zaszło w czasie, gdy wciśnięty był klawisz *Alt*. Ten klawisz ma specjalną rolę w Windows (związaną np. z paskami menu) i dlatego jego wciśnięcie jest tak ważne. Faktycznie, oprócz innego kodu kontekstwego, zdarzenia tych samych klawiszy z wciśnięciem i bez wciśnięcia *Alt*a skutują zupełnie innymi komunikatami. Wspomnimy sobie także i o tym zjawisku przy omawianiu komunikatów o klawiszach

- **poprzedni stan klawisza** (ang. *previous key state*) jest bardzo prostą informacją. Określa ona stan klawisza, którego dotyczy zdarzenie, przed wystąpieniem tegoż zdarzenia. Bit ten może być ustawiony na 1 - co odpowiada klawiszowi **poprzednio** wciśniętemu, lub na 0 - w przeciwnym wypadku
- **stan przejścia** (ang. *transition state*) precyzuje rodzaj zdarzenia, czyli co takiego „stało" się z klawiszem. Wartość 0 w tym bicie oznacza wciśnięcie klawisza, a 1 - zwolnienie.

Pozostaje jeszcze pytanie - jak odczytywać te wartości? Wymaga to skonstruowania odpowiedniej maski bitowej i wykonania jej koniunkcji z wartością `lParam`. Później należy jeszcze dosunąć wyizolowane bity do prawej strony za pomocą operacji przesunięcia bitowego w prawo. Jeżeli nie możesz sobie poradzić z tym zadaniem, zajrzyj do Dodatku C, *Manipulacje bitami*.

Najprościej jest uzyskać licznik powtórzeń. Zauważmy, że stanowi on 16 pierwszych bitów dwusłowa `lParam`, czyli jego dolną połówkę. Tę zaś możemy wyekstrahować poprzez makro `LOWORD()`. `LOWORD(lParam)` jest więc wartością licznika powtórzeń w komunikatach klawiatury.

Komunikaty o klawiszach

Najbardziej naturalnymi zdarzeniami klawiatury są **komunikaty o klawiszach** (ang. *keystroke messages*). Windows wysyła je, gdy wykryje wciśnięcie lub puszczenie jednego z wirtualnych klawiszy, przyporządkowanych oczywiście tym realnie istniejącym na klawiaturze.
W tym paragrafie przyjrzymy się tego rodzaju komunikatom.

Przedstawiamy je

Mamy cztery komunikaty o zdarzeniach klawiszy w Windows. Przedstawia je wszystkie poniższa tabelka:

zdarzenie → / rodzaj ↓	wciśnięcie klawisza	zwolnienie klawisza
pozasystemowe	WM_KEYDOWN	WM_KEYUP
systemowe	WM_SYSKEYDOWN	WM_SYSKEYUP

Tabela 53. Komunikaty o zdarzeniach klawiszy

Nie jest żadną niespodzianką, że występują komunikaty o wciśnięciu i zwolnieniu klawisza. Oba te zdarzenia zachodzą zwykle w parach, podobnie jak to się dzieje w przypadku przycisków myszy. Możliwe jest jednak, że Windows wyśle więcej komunikatów `WM_[SYS]KEYDOWN` - będzie tak, jeżeli raz wciśnięty klawisz przytrzymamy na dłużej. Wówczas może zostać wysłana większa liczba komunikatów lub też nadchodzące wiadomości `WM_[SYS]KEYDOWN` będą miały odpowiednio ustawiony licznik powtórzeń w dolnym słowie parametru `lParam`. Zależy to od ustawionej przez użytkownika częstotliwości powtórzeń we *Właściwościach Klawiatury* Panelu Sterowania.

Atoli najczęściej interesuje nas tylko sam fakt, iż dany przycisk jest aktualnie wciśnięty i dlatego na komunikat `WM_[SYS]KEYDOWN` reagujemy podobnie jak na inne, proste zdarzenia notyfikujące. Oto przykład reakcji na wciśnięcie klawisza *Esc*:

```
case WM_KEYDOWN:
{
   if (wParam == VK_ESCAPE)
         if (MessageBox("Czy na pewno chcesz wyjść?", "Wyjście",
                        MB_YESNO | MB_ICONQUESTION) == IDYES))
```

```
                PostQuitMessage (0);

        return 0;
    }
```

Sprawdzenia, czy wciskanym klawiszem jest rzeczywiście klawisz ucieczki, dokonujemy, konfrontując wartość `wParam` ze stałą `VK_ESCAPE`. Wspomnimy jeszcze o takim porównaniu w dalszej części aktualnego paragrafu.

Systemowe i inne

Zauważyłeś pewnie tajemnicze `SYS` w nazwach komunikatów `WM_SYSKEYDOWN` oraz `WM_SYSKEYUP`. Jego obecność znaczy te zdarzenia jako **systemowe** (ang. *system keystrokes*). Należy to rozumieć w ten sposób, iż są one ważniejsze dla samego systemu Windows niż dla pracujących w nim aplikacji.

`WM_SYSKEYDOWN` i `WM_SYSKEYUP` generowane są zwykle dla klawiszy wciśniętych w połączeniu z przytrzymanym klawiszem *Alt*. Takie kombinacje są ważne dla funkcjonowania całego systemu, gdyż nierzadko są im przypisane pewne standardowe lub niestandardowe akcje. Weźmy chociażby *Alt+Tab*, który służy do przełączania się między oknami uruchomionych aplikacji, czy też *Alt+Esc*.
Istnieją też tzw. akceleratory, stanowiące skróty do często używanych poleceń menu aplikacji. Kombinacje klawiszy dla akceleratorów ustala programista tworzący menu, zaś zapewnienie ich prawidłowej obsługi spoczywa potem wyłącznie na barkach Windows. Dlatego też wciśnięcia klawiszy akceleratorów są traktowane jako systemowe.

O akceleratorach będziemy mówić dokładniej przy opisywaniu pasków menu dla okien.

Łatwo zauważyć, że systemowe zdarzenia klawiatury są podobne w swej istocie i przeznaczeniu do pozaklienckich komunikatów myszy. Analogicznie też wygląda ich ewentualna obsługa - jeżeli jest potrzebna. Ze względu na ważny charakter tych zdarzeń powinniśmy **zawsze** przekazywać je do domyślnej procedury zdarzeniowej - czyli obsługiwać w ten oto sposób:

```
case WM_SYSKEYDOWN:          // albo WM_SYSKEYUP, a także WM_SYS[DEAD]CHAR
    kod_obsługi_komunikatu
    return DefWindowProc(hWnd, uMsg, lParam, wParam);
```

W przeciwnym wypadku okno może stać się całkowicie „odporne” na systemowe kombinacje klawiszy, co z kolei skutkować będzie np. niemożnością użycia *Alt+Tab* do zmiany aktywnego programu.

Pozostałe dwa komunikaty - `WM_KEYDOWN` i `WM_KEYUP` - są ignorowane przez domyślną procedurą zdarzeniową. Ich obsługa w programie nie napotyka więc na żadne ograniczenia i może być realizowana dowolnie (albo nijak, jak to się działo dotąd).

Parametry

Omawiane komunikaty, tak samo jak wszystkie powiadomienia od klawiatury, wykorzystują oba parametry `wParam` i `lParam`.

`wParam` zawiera kod identyfikujący klawisz. Jest to tzw. **kod wirtualnego klawisza** (ang. *virtual-key code*). To 16-bitowa, niezależna od sprzętu wartość przyporządkowana każdemu „wirtualnemu” klawiszowi. Ponieważ klawisze są właśnie wirtualne, czyli nieistniejące w rzeczywistości (choć mające rzeczywiste odpowiedniki), ich kody są

uniwersalne. W każdej wersji systemu Windows, obsługującej w miarę normalną klawiaturę[130], kod każdego wirtualnego klawisza jest zawsze taki sam.
Jest to oczywiście bardzo wygodne, gdyż nie musimy przez to martwić się o sprzętowe kody (czyli kody skanowania, ang. *scan code*), jakie wysyłają poszczególne modele klawiatur. Poza tym wiemy, że do zestawu klawiszy wirtualnych włączono też przyciski myszy. Nie oznacza to aczkolwiek, że zdarzenia `WM_KEYDOWN` i `WM_KEYUP` odnoszą się także do kliknięć myszką. „Klawiszowość" przycisków myszy objawia się tylko tym, że ich stan możemy sprawdzać za pomocą funkcji w rodzaju `GetKeyState()`. Mówiliśmy już o tym w podrozdziale o myszce, a niedługo rozciągniemy temat także na klawiaturę.

Natomiast o parametrze `lParam` i zawartym w nim koglomeracie sześciu informacji zdołałem już napisać całkiem sporo wyjaśnień; możesz teraz do nich powrócić. Pamiętaj przy tym, że kombinacja bitowa o opisanym znaczeniu jest treścią `lParam` w przypadku **każdego** z ośmiu komunikatów klawiatury. Nie wiem jak ty, ale ja sądzę, że to dobra wiadomość dla każdego programisty :)

Komunikaty o znakach

Drugim rodzajem zdarzeń klawiatury, z jakim spotykamy się w Windows, są **komunikaty o znakach** (ang. *character messages*). Zamiast informować tylko o przyciśnięciach klawiszy, komunikaty te mówią raczej o rzeczywistych znakach, wprowadzonych do programu.

Obecność tych notyfikacji w Windows API jest niczym innym jak tylko ułatwieniem dla programisty. Komunikaty te są bowiem automatycznie wyprowadzane ze zdarzeń klawiszy. W zasadzie każda aplikacja mogłaby to robić sama, jednak system wyręcza je w tej czynności.
Za generowanie komunikatów o znakach odpowiada funkcja `TranslateMessage()`. Jej działanie sprowadza się w skrócie do:

- wyłowienia z kolejki komunikatu `WM_[SYS]KEYDOWN`
- sprawdzenia kodu wirtualnego klawisza, jaki jest przyporzadkowany takiemu komunikatowi
- przetłumaczenia go na kod odpowiedniego znaku, jeżeli jest to możliwe. W takim tłumaczeniu są uwzględnianie również takie okoliczności jak wciśnięty klawisz *Shift* czy aktywny *Caps Lock*
- posłania do kolejki komunikatu `WM_[SYS][DEAD]CHAR`, zawierającego przetłumaczony kod

Komunikaty o znakach są więc bardziej wysokopoziomową formą obsługi klawiatury. Nie są one tylko prostymi informacjami, mówiącymi o wciśnięciu klawisza, lecz przynoszą ze sobą także dodatkowe dane. Uwzględniając stan kluczowych klawiszy oraz ustawienia językowe klawiatury, komunikaty te powiadamiają o wprowadzonych przez użytkownika **znakach** - nie zaś o klawiszach, które wciska. Potrafią one rozróżnić znak wielkiej litery 'A' od jej małej wersji ('a'), podczas gdy zdarzenia klawiszy mogą jedynie poinformować o wciśnięciu klawisza *A*. Komunikaty o znakach biorą zatem pod uwagę **szerszy kontekst** przychodzących do systemu informacji od klawiatury.

Prosimy na scenę

Jest raczej czystym przypadkiem to, że komunikaty o znakach również występują w liczbie czterech rodzajów. Nie są one jednak żadnymi odpowiednikami zdarzeń klawiszy, lecz zupełnie inaczej zorganizowanymi powiadomieniami.
Wszystkie komunikaty o znakach ujmuje nam poniższa tabelka (jest w niej także odpowiedni podział tychże komunikatów):

[130] Tzn. zawierającą litery z języków europejskich, a nie np. zestawy znaków języka chińskiego czy japońskiego.

rodzaj znaku → / rodzaj zdarzenia ↓	zwykły znak	martwy znak
pozasystemowe	WM_CHAR	WM_DEADCHAR
systemowe	WM_SYSCHAR	WM_SYSDEADCHAR

Tabela 54. Komunikaty o zdarzeniach znaków

Jak widać, także i tu można wyróżnić dwie nakładające się na siebie grupy komunikatów. Pierwszą z nich znamy już dość dobrze, podczas gdy druga wydaje się dość tajemnicza - również z nazwy (martwy znak…?). Dlatego też, aby mieć pełną jasność, wytłumaczymy sobie obie :)

Aby być zupełnie ścisłym muszę wspomnieć, że istnieje jeszcze jeden komunikat o znaku - `WM_UNICHAR`. Różni się on od `WM_CHAR` tym, iż kod znaku, jaki przynosi w parametrze wParam, jest zapisany w 32-bitowej wersji standardu Unicode (UTF-32). `WM_CHAR` używa tylko 16 bitów (UTF-16), jednak wiemy dobrze, że wystarcza to w zupełności na reprezentację wszystkich znaków niemal każdego cywilizowanego i żywego języka. Zawarte w Windows wsparcie dla 4-bajtowych kodów jest więc dalekim wybiegnięciem przed orkiestrę - zwłaszcza, że nawet dwubajtowy unikod nie jest jeszcze powszechnie wykorzystywany. Niemniej jednak należy się spodziewać, że w bliższej lub (raczej) dalszej przyszłości pozostałe komunikaty o znakach zostaną „przestawione” na UTF-32. Wówczas `WM_UNICHAR` zaniknie.
Ale zanim to się stanie, możesz swobodnie przeczytać opis tego komunikatu w MSDN :D

Systemowe - raz jeszcze

Wśród komunikatów o znaków również występuje podział na te systemowe i pozasystemowe. Kryteria owego podziału są też identyczne.

Przypominam, że systemowe zdarzenie klawiatury jest wysyłane w sytuacji, gdy towarzyszy mu wciśnięty klawisz *Alt*. Kombinacje zawierające ten klawisz są bardzo ważne dla systemu jako całości oraz ogólnego sposobu jego funkcjonowania. Systemowe komunikaty o znakach - `WM_SYSCHAR` i `WM_SYSDEADCHAR` - powinny być przetwarzane bez naruszania ich normalnej, domyślnej reakcji, za którą odpowiada `DefWindowProc()`. Komunikaty te muszą więc ostatecznie trafić do tej standardowej procedury - również wtedy, kiedy po drodze „przeszły” przez tą naszą.
Przykład obsługi systemowego komunikatu klawiatury podałem w poprzednich paragrafie o zdarzeniach klawiszy. Zajrzyj tam, jeżeli tego potrzebujesz.

Umarł znak, niech żyje znak

Zdarzenia `WM_DEADCHAR` i `WM_SYSDEADCHAR` noszą intrygującą nazwę **komunikatów o martwych znakach** (ang. *dead characters messages*). Chociaż ich obsłużenie nie jest w większości przypadków konieczne, omówienie tych zdarzeń może być interesujące.

Najprościej mówiąc, martwe znaki nie reprezentują samodzielnie żadnego symbolu, żadnej litery. Ich pojawianie się wiąże się wyłącznie z pewnymi układami klawiatury, dostosowanymi do niektórych języków. Dobrym przykładem jest język niemiecki i występujące w nim litery ‘ä’, ‘ü’ czy ‘ö’. Mają one tak zwany przegłos (niem. *umlaut*), który zmienia wymowę tych głosek w stosunku do zwykłych ‘a’, ‘u’ i ‘o’. Faktycznie są one odrębnymi literami niemieckiego alfabetu.

Wprowadzenie tych znaków do programu może się odbywać na wiele sposobów, gdyż jak wiemy nie występują one na standardowej klawiaturze. Jedną z dróg może być zaprzęgnięcie do pracy określonych kombinacji klawiszy: najczęściej jest to *Prawy Alt* plus odpowiednia „zwykła” litera. Wpisanie znaku ‘ö’ odbywałoby się wówczas w ten sposób, iż użytkownik najpierw **wciska i przytrzymuje** *Prawy Alt*, a następnie uderza w

klawisz *O*. W wyniku tej czynności na ekranie pojawia się znak 'ö' lub 'Ö' (zależnie od stanu klawisza *Shift* i *Caps Lock*).
Istnieje jeszcze inna metoda i to interesuje nas teraz bardziej. Otóż wprowadzenie znaku diakrytycznego może się dokonywać poprzez **oddzielne wciśnięcia** dwóch klawiszy. Pierwszy wysyła do systemu jedynie sam przegłos, natomiast drugi jest dopiero właściwą literą, która ów *umlaut* otrzymuje. Ostatecznie uzyskujemy pożądany symbol.

Gdzie jest więc ten martwy znak?… Otóż jest nim sam przegłos - po wciśnięciu odpowiadającego mu klawisza, do okna z fokusem wysyłany jest komunikat `WM_[SYS]DEADCHAR` (przedtem oczywiście `WM_[SYS]KEYDOWN`) zawierający kod znaku przegłosu. W tej chwili nie wiadomo jeszcze, jaka litera zostanie zaraz wpisana, ale rzeczony komunikat mówi nam, iż będzie posiadała dany ozdobnik (w tym przypadku przegłos).
Nie jest to wielce porywająca informacja i nie ma konieczności jej odczytywania. W następującym dalej komunikacie `WM_[SYS]CHAR` dostajemy ją bowiem niejako ponownie, lecz w bardziej użytecznej formie. Komunikat o „żywym znaku" będzie mianowicie zawierał kod litery z już zaaplikowanym przegłosem (a więc np. 'ö' lub 'Ö'), nie zaś odpowiadającej mu litery bez niego (czyli 'o' lub 'O').

Pomienięciem `WM_[SYS]DEADCHAR` nie czynimy więc żadnej szkody ani sobie, ani systemowi. Dlatego też prawie zawsze możemy sobie na to pozwolić.

W polskim układzie klawiatury martwe znaki generuje klawisz tyldy (~). W połączeniu z klawiszami *A*, *L*, *O*, *Z*, *X* itd. wprowadza on znaki diakrytyczne: 'ą', 'ł', 'ó', 'ż', 'ź' itd.

Parametry

Komunikaty o znakach zachowują podaną na początku sekcji konwencję co do znaczenia parametrów `wParam` i `lParam`. Spójrzmy, co to oznacza w tym przypadku.

Tradycyjnie `wParam` zawiera ważny dla zdarzenia kod. Tutaj jest to kod wprowadzonego znaku - nie klawisza, lecz właśnie **znaku**. Jest to 16-bitowa liczba, która identyfikuje jeden z kilkudziesięciu tysięcy znaków standardu Unicode (UTF-16). Standard ten w zupełności wystarcza na kodyfikację zbioru liter wszystkich języków indoeuropejskich oraz używanych symboli matematycznych, fizycznych i innych.
Nie bedę tu szczegółowo omawiał Unicode, bo jest to materiał na całkiem sporą książkę; jeżeli interesuje cię ten temat, możesz na początek zajrzeć na oficjalną stronę internetową standardu. Muszę jednak wspomnieć, co stało się z tablicą znaków ASCII i ANSI, znaną zapewne większości czytelników.
A zatem - nie stało się nic. Pierwsze 128 liczb (`0x00` do `0x7F`) jest nadal kodami znaków w systemie ASCII. Wraz z kolejnymi 128 wartościami (`0x80` do `0xFF`) tworzą one tabelę ANSI. Ten drugi zestaw kodów jest specyficzny dla Windows i oznaczany nazwą strony kodowej - w Polsce jest to Windows-1250. Różni się ona chociażby od DOSowej strony 852 czy też unormowanego i popularnego w polskim Internecie systemu ISO-8859-2. Jest to nieunikniona konsekwencja stosowania tylko 256 znaków ANSI - Unicode ze swymi 65536 miejscami na znaki rozwiązuje większość tego rodzaju kwestii.

Znaki z ważniejszych stron kodowanych możesz znaleźć w Tablicach C.

A co `lParam`? Nic nadzwyczajnego. Parametr ten zawiera znany już agregat sześciu danych. Raczej jednak nie mają one praktycznego znaczenia, gdyż są dokładną kopią `lParam` z komunikatu `WM_[SYS]KEYDOWN`, poprzedzającego zdarzenie znaku. Zwykle więc informacje odczytuje się z właściwego im komunikatu klawisza, a nie znaku.

Kontrola wejścia od klawiatury

Zabawa z klawiaturą w Windows API nie ogranicza się li tylko do odbierania zdarzeń i reakcji na nie. Tak samo jak dla myszy możliwe jest przejęcie większej kontroli nad współpracą sprzetu z systemem operacyjnym. I tym właśnie zajmiemy się tej sekcji.

Pobieranie stanu klawiszy

Otrzymywanie komunikatów o klawiszach jest biernym sposobem kooperacji z klawiaturą. Istnieją też metody, w których to aplikacja ma większą kontrolę nad tym procesem i sama sprawdza stany poszczególnych klawiszy.
O takim sprawdzaniu opowiemy sobie w tym paragrafie. Obejmie to między innymi dokładne omówienie funkcji `GetKeyState()` i `GetAsyncKeyState()`, z którymi zapoznaliśmy się już w podobnej sytuacji dotyczącej myszy.

Stan pojedynczego klawisza

Jak pamiętamy, sprawdzaniu stanu pojedynczego klawisza służy funkcja `GetKeyState()`:

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

Należy jej podać kod kontrolowanego, wirtualnego klawisza. Jest to jedna ze stałych `VK_`, ewentualnie (w przypadku klawiszy liter i liczb) kod ASCII odpowiedniego znaku.

W zamian dostajemy… wynik :) Jest nim wartość typu `SHORT` (2 bajty) i składa się z dwóch części:

- górny bajt (czytany przez `HIBYTE()`) po zrzutowaniu na typ logiczny[131] informuje o wciśnięciu klawisza. `true`, `TRUE` lub ogólnie wartość różna od zera wskazuje na to, że klawisz jest wciśnięty. Nietrudno się domyślić, że zero znaczy coś przeciwnego :D
- dolny bajt (`LOBYTE()`) daje wiedzę o tym, czy klawisz jest włączony. Większości klawiszy nie dotyczy ta własność, jest ona ważna tylko dla „locków”: *Num Lock*, *Caps Lock* i *Scroll Lock*[132]. Podobnie jak wyżej, logiczna prawda wskazuje na włączenie danego klawisza, fałsz - przeciwnie.

Trzeba jeszcze powiedzieć jedną bardzo ważną rzecz na temat tej funkcji - skąd ona bierze stan klawiszy?… Wcale nie „pyta” o niego samej klawiatury (tak robi `GetAsyncKeyState()`), lecz uzyskuje go na podstawie kolejki komunikatów. Decyduje tu **ostatnio otrzymany komunikat** klawiatury, dotyczący sprawdzanego klawisza. Uzyskiwane informacje są więc zależne od komunikatów, jakie otrzymuje wątek (tzn. cała aplikacja - najczęściej) i nie dotyczą globalnego stanu klawiatury. Nie pochodzą z poziomu przerwań sprzetowych, lecz zdarzeń systemowych.

Czy jest to wada? Nieszczególnie. `GetKeyState()` używamy głównie do sprawdzania stanu takich klawiszy jak *Shift* i *Ctrl* podczas przetwarzania zdarzeń innych klawiszy, np.:

```
case WM_KEYDOWN:
{
   switch (wParam)
   {
         case VK_F1:
               if (HIBYTE(GetKeyState(VK_SHIFT))
                     // kombinacja Shift+F1
```

[131] Jak wiesz, takie rzutowanie można zastąpić porównaniem z zerem - także tym niejawnym, stosowanym w warunkach `if` czy pętli.
[132] Klawiszom tym odpowiadają stałe `VK_NUMLOCK`, `VK_CAPITAL` i `VK_SCROLL`.

```
                    else
                          // samo F1

                    break;

              // itd.
        }
    }
```

Stan tych innych klawiszy otrzymujemy w komunikatach i to właśnie ich powinniśmy używać do reakcji na wciśnięcia i zwolnienia klawiszy w normalnych aplikacjach.

Stan całej klawiatury

W Windows API znajdziemy też funkcję pobierająca stan wszystkich klawiszy - `GetKeyboardState()`:

```
BOOL GetKeyboardState(PBYTE lpKeyState);
```

Działa ona mniej więcej tak, jak zastosowanie `GetKeyState()` dla parametrów z przedziału od zera do 256. `GetKeyboardState()` przyjmuje mianowicie tablicę 256 bajtów, której indeksami są kody kolejnych wirtualnych klawiszy.
Wynikiem funkcji jest `TRUE` dla operacji zakończonej powodzeniem i zero (`FALSE`) w innym przypadku.

> Jest jeszcze funkcja `SetKeyboardState()`, pozwalająca ustawić chwilowy stan klawiatury dla danego wątku. Możesz o niej poczytać w MSDN.
> Lepszą formą zmiany stanu klawiatury jest aczkolwiek użycie symulowane wejścia, czyli funkcji `SendInput()`.

Asynchroniczne pobieranie stanu klawisza

Drugą z funkcji stworzonych do uzyskiwania stanu pojedynczego klawisza jest `GetAsyncKeyState()`:

```
SHORT GetAsyncKeyState(int vKey);
```

Tak samo przynosimy jej kod wirtualnego klawisza, który chcemy sprawdzać. A co z wynikiem? Również jest podzielony na dwie części po jednym bajcie każda:

- starszy bajt (`HIBYTE()`) znaczy to samo, co w `GetKeyState()`: po konwersji na wartość logiczną informuje o wciśniętym klawiszu (`true`) lub zostawionym w spokoju (`false`)
- młodszy bajt (`LOBYTE()`) wskazuje, czy klawisz był wciskany (logiczna prawda) od czasu ostatniego wywołania `GetAsyncKeyState()`. Trzeba jednak wiedzieć, że to ostatnie wywołanie wcale nie musi pochodzić z naszej aplikacji i dlatego omawiana tu wartość nie ma praktycznego sensu

Co różni tę funkcję od `GetKeyState()`? Pewne wskazówki co do tego mogłeś wyczytać między wierszami powyższego opisu i w akapicie o tamtej pokrewnej funkcji. Powiedzmy jednak wprost, o co chodzi.
Otóż `GetAsyncKeyState()`, czyniąc zadość swej nazwie, pobiera tzw. asynchroniczny stan klawisza. Asynchroniczy to znaczy niezależny od wątku, a mówiąc po ludzku - **globalny** dla całego systemu oraz **niezależny od kolejki komunikatów**. Funkcja ta pobiera dane bezpośrednio od sprzętu, niejako z pominięciem mechanizmu zdarzeń Windows. Przejawia się to chociażby w tym, że kontroluje stan fizycznych, a nie logicznych przycisków myszki, o czym wspomniałem przy pierwszym spotkaniu z tą funkcją.

Ze względu na ten sposób działania `GetAsyncKeyState()` jest przydatna w programach czasu rzeczywistego. Będziemy więc używać tej funkcji do pobierania stanu klawiszy w naszych grach - przynajmniej na początku. Przygotuj się zatem na wiele długich i owocnych spotkań z funkcją `GetAsyncKeyState()` ;)

Symulowanie klawiatury

W poprzednim podrozdziale nauczyliśmy się udawać myszkę. Nie ma więc powodu, abyśmy tego samego nie mogli czynić z klawiaturą. Jest to o tyle proste, iż odbywa się za pomocą niemal tych samych narzędzi. Są nimi: funkcja `SendInput()` i struktura `INPUT`, które sobie przypomnimy, oraz struktura `KEYBDINPUT`, którą teraz poznamy.
A zatem do dzieła!

Funkcja `SendInput()` i struktura `INPUT` - powtórzenie

Jak pamiętamy, do generowania sztucznych zdarzeń od urządzeń wejściowych służy funkcja `SendInput()`:

```
UINT SendInput(UINT nInputs,
               LPINPUT pInputs,
               int cbSize);
```

Przypomnijmy, że w pierwszym parametrze `nInputs` należy jej przekazać tablicę struktur `INPUT` o liczbie elementów określonej drugim parametrem, `pInputs`. Trzeci argument trzeba natomiast ustawić na rozmiar struktury `INPUT`, czyli po prostu `sizeof(INPUT)`.

Pojedynczy element przekazywanej do funkcji tablicy opisuje jedno symulowane zdarzenie od urządzenia wejściowego. Czyni to za pomocą struktury `INPUT`:

```
struct INPUT
{
   DWORD type;

   union
   {
         MOUSEINPUT mi;
         KEYBDINPUT ki;
         HARDWAREINPUT hi;
   };
};
```

W niej też pole `type` określa nam źródło zdarzenia, czyli rodzaj urządzenia. Niedawno, zajmując się myszką, ustawialiśmy je na `INPUT_MOUSE`. Obecnie, gdy chcemy emulować klawiaturę, posłużymy się raczej stałą `INPUT_KEYBOARD`.
Wiąże się to także z porzuceniem pola `mi`, używanego dotąd. Zdarzenie klawiatury musimy bowiem zapisać w polu `ki`, należącym do innego typu - `KEYBDINPUT`.

Struktura `KEYBDINPUT`

Struktura opisująca zdarzenie klawiatury przedstawia się w ten oto sposób:

```
struct KEYBDINPUT
{
   WORD wVk;
   WORD wScan;
   DWORD dwFlags;
   DWORD dwTime;
   ULONG_PTR dwExtraInfo;
};
```

Co z szczęście - tylko pięć pól ;D Ich znaczenie opisuje niniejsza tabelka:

typ	***parametry***	***opis***
WORD WORD	wVk wScan	W którymś z tych parametrów należy podać **kod klawisza**, którego ma dotyczyć zdarzenie. Może to być kod wirtualnego klawisza - wtedy wprowadzamy go w wVk - lub kod skanowania (OEM) - wówczas wykorzystujemy pole wScan. Nieużywane pole wypełniamy zwykle zerem lub ignorujemy. Znaczenie obu tych pól zmienia aczkolwiek flaga KEYEVENTF_UNICODE, jeżeli jest ustawiona w polu dwFlags. Za chwilę powiemy nieco więcej na ten temat.
DWORD	dwFlags	Są to **flagi** kontrolujące produkowane zdarzenie. Ich lista jest podana poniżej.
DWORD ULONG_PTR	dwTime dwExtraInfo	Te dwa pola mają identyczne przeznaczenie, jak time i dwExtraInfo w strukturze MOUSEINPUT. Przypomnijmy tylko, że pierwsze z nich określa moment wystąpienia symulowanego zdarzenia w formie liczby milisekund od startu systemu (czyli rezultatu GetTickCount()). Drugie pole to natomiast jakieś dodatkowe informacje związane ze zdarzeniem, zwykle niewykorzystywane.

Tabela 55. Pola struktury KEYBDINPUT

Z tabelki dowiedzieliśmy się, że możliwe jest podanie kodu klawisza, który bierze udział w generowanej akcji klawiatury. To jednak niewystarczająca informacja i dlatego jest jeszcze pole dwFlags, będące kombinacją bitową odpowiednich flag. Flagi te podsumowuje następna tabela:

flaga	***znaczenie***
KEYEVENTF_SCANCODE	Obecność tej flagi informuje funkcję SendInput(), że ma brać pod uwagę pole wScan, a więc sprzętowy kod skanowania klawisza. Analogicznie, jej brak sprawia, że ważne staje się pole wVk, czyli że klawisz jest rozpoznawany na podstawie swego uniwersalnego kodu wirtualnego.
KEYEVENTF_EXTENDEDKEY	Tę flagę ustawiamy, gdy za pomocą kodu skanowania (wScan) generujemy zdarzenie klawisza rozszerzonego. Naturalnie, musi ona wystąpić razem z KEYEVENTF_SCANCODE.
KEYEVENTF_KEYUP	Kiedy flaga ta jest ustawiona, symulowanym zdarzeniem będzie zwolnienie klawisza. W przeciwnym wypadku klawisz zostanie programowo wciśnięty.
KEYEVENTF_UNICODE	Pozwala na zasymulowanie wprowadzania znaku Unicode - jego kod powinien być w polu wScan. Z oczywistych względów wszystkie znaki Unicode nie są dostępne na klawiaturze, więc system radzi sobie tutaj w inny sposób: jako wciśnięty wirtualny klawisz przyjmuje specjalną stałą VK_PACKET - można ją potem znaleźć w parametrze wParam komunikatów WM_[SYS]KEYDOWN/UP. Natomiast kod znaku w WM_[SYS]CHAR jest już podanym w wScan 16-bitowym kodem Unicode. W sumie więc aplikacjom „wydaje się”, że użytkownik nabrał magicznej mocy wprowadzania kilkudziesięciu tysięcy znaków bezpośrednio ze swojej skromnej, nieco ponadstuklawiszowej klawiatury. Flaga KEYEVENTF_UNICODE musi wystapić z KEYEVENTF_KEYUP,

flaga	znaczenie
	lecz bez KEYEVENTF_SCANCODE.

Tabela 56. Flagi bitowe pola dwFlags struktury KEYBDINPUT

Jaką wiedzę nabyliśmy stąd? Przede wszystkim taką, że domyślnie generowanym zdarzeniem jest zawsze wciśnięcie klawisza; jeżeli chcemy symulować jego zwolnienie, musimy posłużyć się flagą KEYEVENTF_KEYUP. Poza tym wiemy też, że standardowo SendInput() bierze pod uwagę kod wirtualnego klawisza, czyli wartość pola wVk; jeśli pragniemy oprzeć się na kodzie skanowania (polu wScan), powinniśmy podać flagę KEYEVENTF_SCANCODE. Wreszcie poznaliśmy ciekawą możliwość symulowania zdarzeń fizycznie niemożliwych, czyli bezpośredniego wprowadzania znaków z całego zestawu Unicode - dzieje się to dzięki fladze KEYEVENTF_UNICODE.

Przykłady

Gdy mamy już za sobą formalny opis narzędzia, czas przyjrzeć się przykładom jego wykorzystania.

Najpierw więc programowo przyciśniemy klawisz *Enter*. Do wykonania tego zadania można posłużyć się takim kodem:

```
// deklaracja i wyzerowanie struktury INPUT
INPUT Klawisz;
ZeroMemory (&Klawisz, sizeof(INPUT));

// ustawienie pól struktury i wygenerowanie zdarzenia
Klawisz.type = INPUT_KEYBOARD;   // generujemy zdarzenie klawiatury...
Klawisz.ki.wVk = VK_RETURN;      // a dokładniej klawisza Enter
SendInput (1, &Klawisz, sizeof(INPUT));      // i voilà :)
```

Zauważmy, że nie musieliśmy w nim w ogóle zajmować się polem dwFlags. Jest tak, gdyż domyślne jego opcje (odczytanie kodu wirtualnego klawisza i jego wciśnięcie) całkowicie nam odpowiadają.

Po wykonaniu powyższych wierszy przycisk *Enter* pozostaje wciśnięty - pamiętajmy o tym. Konsekwencją tego jest ciągłe wysyłanie komunikatów WM_[SYS]KEYDOWN, zgodnie z ustawioną częstotliwością powtarzania. Aby przerwać tę serię, musimy zwolnić wciśnięty klawisz:

```
INPUT Klawisz;
ZeroMemory (&Klawisz, sizeof(INPUT));

// zwolnienie klawisza
Klawisz.type = INPUT_KEYBOARD;         // wskazujemy na klawiaturę
Klawisz.ki.wVk = VK_RETURN;            // kod klawisza Enter
Klawisz.ki.dwFlags = KEYEVENTF_KEYUP;  // flaga zwolnienia klawisza
SendInput (1, &Klawisz, sizeof(INPUT));      // it's showtime! ;)
```

W ten sposób klawisz *Enter* wróci do stanu wyjściowego, ale jego wciśnięcie i puszczenie zostanie zarejestrowane.

Pora na ostatni przykład, znacznie bardziej skomplikowany. Napiszemy ciekawą funkcję, która zasymuluje wprowadzenie **całego tekstu**, podanego jej w parametrze - klawisz po klawiszu. Funkcja ta mogłaby wyglądać tak[133]:

[133] Intensywnie używam tu Biblioteki Standardowej, więc jeśli nie znasz jej choć trochę, możesz mieć problemy ze zrozumieniem kodu. Komentarze powinny jednak sporo wyjaśniać.

```
#include <string>
#include <vector>
#include <windows.h>

//----------------------------------------------------------------------

bool SymulujTekst(const std::string& strTekst)
{
   // sprawdzamy, czy napis nie jest pusty
   if (strTekst.empty()) return false;

   // zapisujemy długość napisu w pomocniczej zmiennej
   UINT uDlugosc = (UINT) strTekst.length();


   /* generujemy tablicę zdarzeń */

   // deklarujemy zmienne
   std::vector<INPUT> aZdarzenia;      // rzeczona tablica
   INPUT Zdarzenie;                    // jedno zdarzenie

   // w tablicy potrzebne są dwa elementy dla każdego
   // znaku napisu (wciśnięcie i zwolnienie odpowiedniego klawisza)
   // i tyleż rezerwujemy
   aZdarzenie.reserve (uDlugosc * 2);

   // iterujemy po napisie i dla każdego znaku tworzymy dwa zdarzenia
   for (std::string::const_iterator i = strTekst.begin();
        i != strTekst.end(); ++i)
   {
         // kontrolujemy, czy znak nalezy do zestawu ASCII
         if ((*i) > 0x7F) return false;

         // ustawiamy strukturę na parametry wspólne obu zdarzeniom
         ZeroMemory (&Zdarzenie, sizeof(INPUT));
         Zdarzenie.type = INPUT_KEYBOARD;
         Zdarzenie.ki.wVk = (*i); // kod ASCII znaku == kod wirt. klaw.

         // dodajemy pierwsze zdarzenie - wciśnięcie klawisza
         aZdarzenia.push_back (Zdarzenie);

         // dodajemy drugie zdarzenie - zwolnienie klawisza
         Zdarzenie.ki.dwFlags = KEYEVENTF_KEYUP;
         aZdarzenia.push_back (Zdarzenie);
   }


   /* symulujemy zdarzenia */

   // wywołujemy SendInput(), sprawdzając liczbę poprawnych zdarzeń
   // rzutowanie const_cast w drugim parametrze jest konieczne ze
   // względu na ewidentny burak w deklaracji SendInput(), gdzie
   // ten parametr jest zwkłym wskaźnikiem, zmiast stałym do stałej
   if (SendInput(aTablica.size(), const_cast<LPINPUT>(aTablica.data()),
                 sizeof(INPUT)) < (UINT) aTablica.size())
         // gdy wygenerowane mniej zdarzeń niż trzeba, zwracamy false
         return false;

   // w końcu, zwracamy true
   return true;
```

```
}
```

Wadą tej funkcji jest nieumiejętność generowania zdarzeń znaków spoza zestawu ASCII. Ten problem można jednak obejść, jeżeli zastosuje się flagę `KEYEVENTF_UNICODE`. Spróbuj samodzielnie napisać poprawioną wersję funkcji - teraz lub później, bo będzie to częścią pracy domowej na koniec rozdziału :D

Ustawienia klawiatury

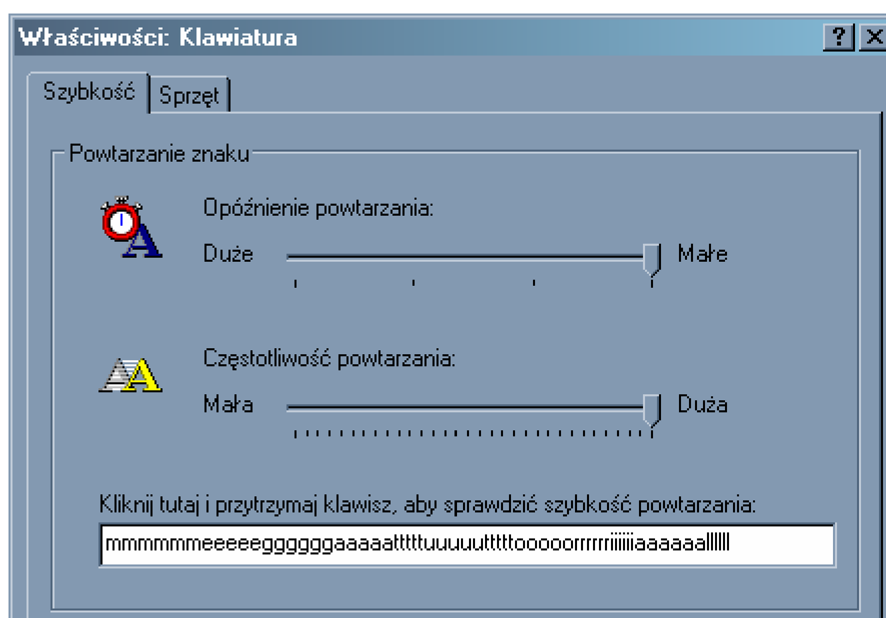
Klawiatura to pospolite urządzenie, które jest w dużym stopniu konfigurowalne. Windows posiada kilka opcji, umożliwiających zmianę jego parametrów - będą one treścią tej sekcji.

Użyjemy tutaj kilka razy funkcji `SystemParametersInfo()`, zatem dobrze byłoby, gdybyś przypomniał ją sobie - z poprzedniego podrozdziału o myszce lub bezpośrednio z MSDN.

Powtarzanie znaku

Chyba najważniejszymi ustawieniami personalizacyjnymi klawiatury (albo jednymi z najważniejszych) są opcje powtarzania znaku. Mam tu na myśli regulację czasu przytrzymywania klawisza, po którym następuje powtarzanie, oraz szybkości duplikacji. Ustawienie nieodpowiednich dla ciebie parametrów może prowadzić albo do powstawania 'tttaakkkiiiicchhh bbłęędóóww' w pisaniu, albo do frustracji spowodowanej długim czekaniem na wyprodukowanie np. sekwencji myślników (-) imitujących poziomą linię.

Z punktu widzenia użytkownika opcje powtarzania można ustawić w aplecie Panelu Sterowania *Właściwości: Klawiatura*. Jego interesujący fragment wygląda tak:



Screen 65. Opcje powtarzania znaku

Widzimy tu dwa ustawienia, dostrajane za pomocą suwaków:

- *Opóźnienie powtarzania* ma wpływ na czas przytrzywania klawisza, po upłynięciu którego znak jest powtarzany
- *Częstotliwość powtarzania* reguluje szybkość produkcji kolejnych znaków przy wciśniętym i przytrzymanym klawiszu

Nas, jako programistów, będzie naturalnie interesować sposób manipulowania tymi opcjami za pośrednictwem funkcji Windows API. Tym więc zajmiemy się w aktualnym paragrafie - przyjrzymy się obu ustawieniom powtarzania znaku.

Opóźnienie powtarzania

Interwał czasu, po jakim rozpocznie się powtarzanie, możemy kontrolować za pomocą funkcji `SystemParametersInfo()`.

Pobranie wartości tego ustawienia wiąże się z wykorzystaniem stałej `SPI_GETKEYBOARDDELAY` i wygląda tak:

```
UINT uOpoznienie;
SystemParametersInfo (SPI_GETKEYBOARDDELAY, 0, &uOpoznienie, 0);
```

W zmiennej, której adres należy podać w trzecim parametrze (`pvParam`) odnajdziemy teraz liczbę z przedziału od `0` do `3`, mówiącą jak długi jest omawiany okres czasu. Faktyczna jego rozciągłość zależy od sprzętu i wynosi mniej więcej 250 milisekund dla ustawienia `0`, a następnie o tyleż przyrasta z każdym krokiem (osiąga więc ok. 1 sekundę dla ustawienia `3`).

Taką samą jednostkę dla opóźnienia musimy przyjąć, gdy chcemy je zmodyfikować. Służy do tego stała `SPI_SETKEYBOARDDELAY` użyta na przykład tak:

```
SystemParametersInfo (SPI_SETKEYBOARDDELAY, 3, NULL, 0);
```

W drugim parametrze `SystemParametersInfo()` należy podać nową wartość opcji. W powyższym kodzie będzie więc ona ustawiona na maksimum, a powtarzanie znaku rozpocznie się dopiero po około sekundzie przytrzymywania klawisza.

Częstotliwość powtarzania

Gdy repetycja już się rozpocznie, za szybkość jej wykonywania odpowiada druga z opcji powtarzania, czyli częstotliwość. Jej programistyczna obsługa także wymaga użycia funkcji `SystemParametersInfo()`.

Oczywiście zaczniemy od pobierania. Aby uzyskać częstotliwość powtarzania znaku posługujemy się identyfikatorem `SPI_GETKEYBOARDSPEED`:

```
UINT uCzestotliwosc;
SystemParametresInfo (SPI_GETKEYBOARSPEED, 0, &uCzestotliwosc, 0);
```

Ponownie otrzymana wielkość nie jest bezwględna i oscyluje w granicach od `0` (co odpowiada ok. 2-3 powtórzeniom znaku na sekundę) do `31` (to znaczy przeciętnie 30 powtórzeń na sekundę). Dokładna częstotliwość jest, podobnie jak opóźnienie, zależna od posiadanego modelu klawiatury.

Teraz zajmijmy się ustawianiem tego ustawienia ;) By je zmodyfikować, należałoby podeprzeć się stałą `SPI_SETKEYBOARDSPEED` w niniejszy sposób:

```
SystemParametersInfo (SPI_SETKEYBOARDSPEED, 31, NULL, 0);
```

Tak też ustawiamy największą możliwą prędkość powtarzania znaków (`31`).

Ułatwienia dostępu

Na koniec zapoznamy się opcjami klawiatury, które ułatwiają pracę z komputerem osobom niepełnosprawnym. Wiele z tych ustawień może być aczkolwiek wygodna także dla zupełnie zdrowych użytkowników.

Ułatwienia klawiatury są dość złożonymi zagadnieniami; każde z nich posiada na swój użytek pewną strukturę, której pola należałoby omówić. Nie ma na to już miejsca ani czasu, dlatego w tym paragrafie opiszę jedynie poszczególne ułatwienia i wskażę źródła, z których możesz się dowiedzić więcej na ich temat.

KlawiszeFiltru

KlawiszeFiltru (ang. *FilterKeys*) są opcją, której zadaniem jest przeciwdziałanie skutkom nieumyślnych wciśnięć klawiszy. Odbywa się to poprzez ignorowanie takich przyciśnięć, które nie są przytrzymane przez odpowiednio długi czas (długi znaczy tu raczej ułamek sekundy). Możliwe jest także drastyczne zmniejszenie szybkości powtórzeń znaków.

Programowa kontrola KlawiszyFiltru może być przeprowadzana funkcją `SystemParametersInfo()` oraz stałymi `SPI_GETFILTERKEYS` i `SPI_SETFILTERKEYS`. Z opcją jest też związana struktura `FILTERKEYS`.

KlawiszeTrwałe

KlawiszeTrwałe (ang. *StickyKeys*) zmieniają sposób działania klawiszy *Ctrl*, *Shift* i *Alt*, ułatwiając wykonywanie zawierających je kombinacji. Zamiast jednoczesnego wciskania wszystkich klawiszy lub przytrzymywania wspomnianych trzech, wystarczy ich jednokrotne dociśnięcie i zwolnienie. Przy włączonych KlawiszachTrwałych wykonanie kombinacje *Alt+Tab* sprowadza się do wciśnięcia i puszczenia klawisza *Alt*, a następnie wciśnięcia *Tab* - nie trzeba przytrzymywać pierwszego z klawiszy.

Za KlawiszeTrwałe odpowiadają stałe `SPI_GETSTICKYKEYS` i `SPI_SETSTICKYKEYS` funkcji `SystemParametersInfo()` oraz struktura `STICKYKEYS`.

KlawiszePrzełączające

Po uaktywnieniu KlawiszyPrzełączających (ang. *ToggleKeys*) komputer będzie generował dźwięk w momencie wciśnięcia jednego z klawiszy *Lock*: *Num Lock*, *Caps Lock* i *Scroll Lock*. Powinno to na przykład zapobiec błędom polegającym na wpisywaniu 'tEKSTU pODOBNEGO dO tEGO' :)

Modyfikacja ustawień KlawiszyPrzełączających odbywa się stałymi `SPI_GETTOGGLEKEYS` i `SPI_SETTOGGLEKEYS` oraz strukturą `TOGGLEKEYS`.

***

Zaprezentowaniem powyższej trójcy ułatwień dostępu kończymy nasze spotkanie z klawiaturą. Poznaliśmy tutaj większość aspektów jej wykorzystania przy pomocy Windows API, co powinno nam pomóc przy tworzeniu aplikacji okienkowych.

Z ważniejszych, a nieomówionych kwestii należy wymienić układy klawiatury oraz karetkę. Jeżeli chcesz, możesz poczytać na ich temat w MSDN.

Podsumowanie

Dobrnęliśmy wreszcie do końca tego rozdziału. Teraz wiesz już wszystko, co niezbędne do poprawnego wykorzystania klawiatury i myszy w twoich programach dla środowiska

Windows. Znasz już odpowiednie komunkaty oraz pomocnicze funkcje WinAPI, które będą ci w tym pomocne.

W następnym rozdziale zajmiemy się wreszcie rysowaniem i grafiką. Wprawdzie nie będzie to jeszcze DirectX, ale i tak powinieneś być zadowolony. Zapoznamy się bowiem dokładnie z bogatą biblioteką graficzną Windows GDI.

Pytania i zadania

Oto niezbędny zestaw pytań kontrolnych i zadań do wykonania. Miłej pracy ;)

Pytania

1. Czym jest urządzenie wejściowe? Jakie znasz rodzaje takich urządzeń?
2. Co w Windows API rozumiemy pod pojęciem myszy?
3. Jakie rodzaje komunikatów myszy może otrzymać okno w Windows?
4. Jakie informacje są dostarczane w parametrach `wParam` i `lParam` każdego komunikatu myszy?
5. Który komunikat przycisku myszki należy obsługiwać, aby zapewnić reakcję na pojedyncze kliknięcie?
6. Jaki wymóg musi spełnić okno, aby otrzymywać informacje o dwukrotnych kliknięciach?
7. Które okno otrzymuje komunikat `WM_MOUSEWHEEL` o obrocie rolki myszy?
8. Co to znaczy, że okno ma władzę nad myszką? Jak można taką władzę uzyskać?
9. Jak można pobrać pozycję kursora w dowolnym momencie?
10. W jaki sposób sprawdzamy stan wciśnięcia przycisków myszy? O czym należy pamiętać, jeżeli używamy do tego funkcji `GetAsyncKeyState()`?
11. Jak można programowo symulować ruch myszy, wciśnięcia przycisków oraz obrót rolką?
12. Jak sprawdzamy obecność w komputerze i możliwości myszki?
13. Jakie ułatwienia dostępu są związane z myszką?
14. Czym jest potok klawiszy i jakie są jego kolejne etapy?
15. Czym różni się kod skanowania od kodu wirtualnego klawisza?
16. Które okno otrzymuje komunikatu o zdarzeniach klawiatury?
17. Jakie informacje można odczytać z parametru `lParam` komunikatów klawiatury?
18. Jakie komunikaty o klawiszach generuje system Windows?
19. Czym się różni komunikat systemowy od pozasystemowego?
20. Skąd pochodzą komunikaty o znakach i jaka jest ich rola?
21. Co zawiera parametr `wParam` komunikatów o znakach?
22. Jakimi dwoma funkcjami pobieramy stan pojedynczego klawisza wirtualnego i czym różnią się one między sobą?
23. Jak wygląda programowe symulowanie klawiatury?
24. Jakie dwa ustawienia kontrolują powtarzanie znaku przy wciśniętym klawiszu?
25. Podaj trzy ułatwienia dostępu związane z klawiaturą.

Ćwiczenia

1. Napisz program, który wyświetli komunikat po kliknięciu lewym przyciskiem myszy w obszarze klienta swojego okna.
2. (**Trudne**) Stwórz aplikację, która będzie reagowała pokazaniem menu sterującego okna w odpowiedzi na kliknięcie jego wnętrza.
   Wskazówka: wykorzystaj komunikat `WM_NCHITTEST`.
3. Zmodyfikuj przykład `CursorPos` tak, ażeby wyświetlał on współrzędne ekranowe kursora. Najlepiej, jeżeli nie wykorzystasz do tego funkcji `GetCursorPos()`.
4. Zmień nasz przykładowy szkicownik `Scribble` - niech okno nie traci swej zawartości po odrysowywaniu.

Wskazówka: przypomnij sobie omówienie procesu tworzenia okna z poprzedniego rozdziału.

5. Utwórz program pokazujący w swym oknie kod wirtualnego klawisza, który wciska użytkownik.
   (**Trudne**) Dodaj do tego jeszcze nazwę klawisza w postaci tekstu, np. `"Enter"` czy `"Strzałka w dół"`.
6. (**Ekstremalne**) Stwórz aplikację zliczającą wciśnięte przez użytkownika klawisze w całym systemie i pokazującą ją w małym okienku w trybie „zawsze na wierzchu"
   Wskazówka: zainteresuj się filtrami (ang. *hooks*), a szczególnie jednym rodzajem - `WH_JOURNALRECORD`. Potrzebne infomacje znajdziesz w opisie funkcji `SetWindowsHookEx()`.
7. Napisz program, który pozwalałby na zmianę tytułu swego okna. Niech będzie on początkowo pusty, a wciśnięcia klawiszy alfanumerycznych niech powoduje dodanie do niego odpowiednich znaków.
   (**Trudniejsze**) Spraw jeszcze, aby klawisz Backspace usuwał już wprowadzone znaki.
8. (**Trudniejsze**) Napisz lepszą wersję funkcji `SymulujTekst()`. Powinna ona przyjmować dowolny tekst, najlepiej w formacie Unicode.