Contrôle du clavier et de la souris sous Windows

Par gRRosminet

Rédigé en C/C++ par gRRosminet, traduit en Delphi par Hachesse

10 Novembre 2002

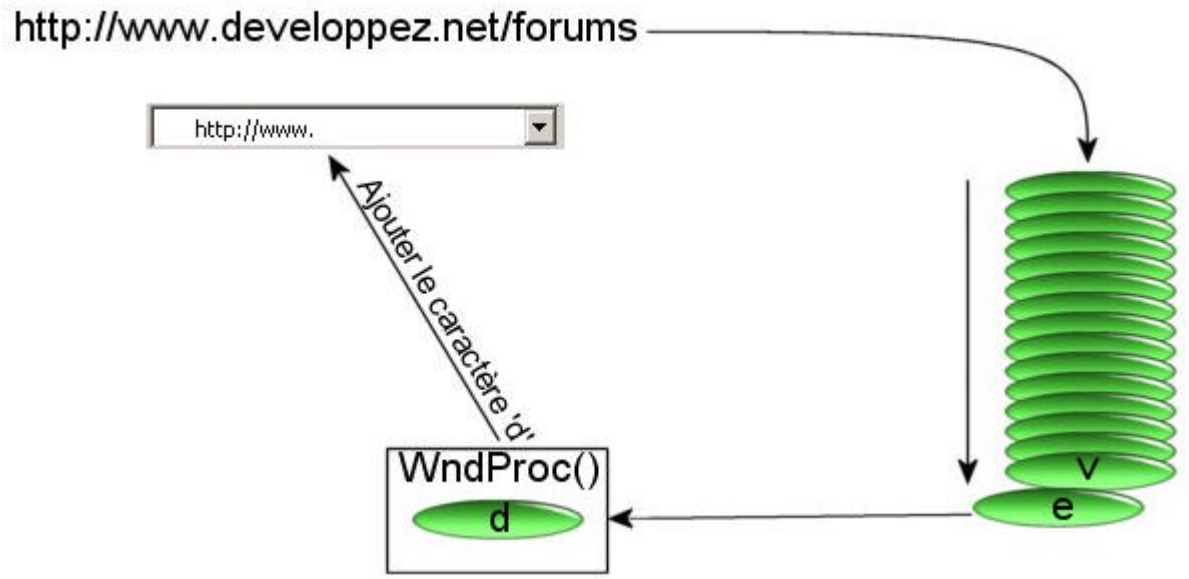
Introduction

Depuis ces dernières années, les applications multimédias fleurissent à foison : lecteurs MP3, DVD, DivX et autres logiciels de TV sont désormais monnaie courrante sur les ordinateurs personnels qui autrefois étaient destinnés à installer Office et le dernier jeu à la mode. Si les interfaces sont de plus en plus conviviales, elles nécessitent toujours que l'utilisateur vienne interagir soit avec le clavier, soit avec la souris pour réaliser les opérations usuelles telles que lecture, pause, rembobinage, changement de chaines, etc ... Nous avons donc vu apparaître des claviers multimédia aux merveilleuses touches complémentaires pour s'occuper de ces fonctions (comme si 105 touches c'était pas assez ! ;-)). Cependant, si on regarde bien, la prise en charge de ces touches par les applications n'est pas toujours implémentée et quand elle l'est, elle ne l'est pas toujours correctement. En effet, il arrive que certaines applications réagissent ou non suivant le média lu ou suivant le temps écoulé depuis la dernière "vraie" action du clavier ou de la souris. Les développeurs ont donc eu l'idée de simuler les racourcis claviers des applications multimédia pour mieux se faire comprendre des applications. Cependant, il n'est pas toujours évident de réaliser cette simulation et c'est pourquoi nous allons diviser cette étude en deux parties : *simulation par la file de message de Windows* et *prendre le contrôle du clavier et de la souris*

Rappels sur la file de message Windows

Le principal moyen de communication utilisé par Windows et les applications écrites pour ce système d'exploitation est la file de message associée à chaque fenêtre. En effet, chaque fois qu'un évènement se produit, Windows le traite en envoyant un message, soit à lui-même, soit à un autre programme. Le programme procède alors au traitement de ce message si celui-ci a un interêt.

Nous allons étudier l'exemple suivant : vous êtes dans votre navigateur favoris et vous tapez l'adresse : "http://www.developpez.net/forums" puis vous validez avec la touche ENTREE



A chaque pression d'une touche, Windows va envoyer trois messages au navigateur : WM_KEYDOWN qui va indiquer qu'une touche a été enfoncée, WM_CHAR qui va préciser qu'un caractère à été donné à l'application et finalement WM_KEYUP qui va indiquer que la touche a été relachée. Chacun des ces messages est placé dans la file de messages du navigateur qui va les analyser un à un. Imaginons qu'il ne se préoccupe que du message WM_CHAR et qu'il ignore les autres. A chacun de ces messages, il va analyser les informations fournies par windows en complément du message et ajouter le caractère indiqué à la chaine de caractères représentant l'URL de la ressource désirée. Au moment ou il va recevoir le caractère "ENTREE", il va lancer la recherche.

L'interêt d'une telle étude est de comprendre par quel moyen on va pouvoir dire a une application multimédia (ou une autre, mais l'interêt serait plus limité) *"l'utilisateur a pressé la touche X : lance la lecture"*. En effet, vous l'avez surement remarqué, la majorité des applications met à notre disposition des racourcis clavier pour accéder rapidement aux principales fonctionnalités ; nous allons les simuler afin de contrôler les applications.

Démarche de la simulation par la file de messages Windows

Simuler l'interaction de l'utilisateur par la file de messages nécessite deux étapes que nous allons détailler ci-après : déterminer la cible et envoyer le message.

Tout d'abord, il faut déterminer le handle de l'application à laquelle on veut envoyer un message. Un handle est une sorte d'identifiant utilisé par Windows pour identifier chaque composant manipulé par l'interface. Deux possibilités s'offrent à nous : envoyer le message à une fenêtre précise ou envoyer le message à la fenêtre au premier plan.

Récupérer le handle de la fenêtre au premier plan

C'est la chose la plus simple. Cela se fait par un appel à la fonction *GetForegroundWindow()*. Cette fonction ne prend pas de paramètre et renvoie une valeur de type `HWND` (handle de fenêtre). Cette valeur vaut `NULL` en C ou **`nil`** en Pascal si la fonction a échoué. Voici un exemple d'utilisation en langage C :

```c
#include <windows.h>
#include <stdio.h>
int main(int argc, char * argv[])
{
     HWND hFore = GetForegroundWindow();
     printf("le handle de la fenêtre au premier plan est : 0x%X",hFore);
     return 0;
}
```

Récupérer le handle d'une fenêtre précise

Cela nécessite de connaître le nom de la "classe" de composant à laquelle la fenêtre appartient. Le nom de classe ou *class name* correspond à un identifiant qu'on associe à une fenêtre pour la distinguer d'une autre. Par exemple, WinAMP dans ses version 1.x et 2.x avait pour nom de classe *"Winamp v1.x"* ; le nom de classe du lecteur Windows Media est *"Media Player 2".*

Pour récupérer le handle d'une fenêtre dont on connait le class name, on utilise la fonction FindWindow(). Cette fonction prend deux paramètres : un pointeur vers une chaine de caractères contenant le class name de la fenêtre recherchée (**`char *`** en C, **`character^`** en Pascal) et un pointeur vers une chaine de caractères contenant le titre de la fenêtre recherchée (optionel, mettre `NULL` en C ou **`nil`** en Pascal). Elle renvoie une valeur de type `HWND` (handle de fenêtre). Cette valeur vaut `NULL` en C ou **`nil`** en Pascal si la fonction a échoué. Voici un exemple d'utilisation en langage C :

```c
#include <windows.h>
#include <stdio.h>
int main(int argc, char * argv[])
{
     HWND hMediaplayer = FindWindow("Media Player 2",NULL);
     printf("le handle de la fenêtre de media player 2 est : 0x%X",hMediaplayer);
     return 0;
}
```

Nous allons maintenant voir quel message envoyer et comment l'envoyer.

Les différents messages pour le clavier et la souris

Il existe trois messages pour répertorier les actions du clavier, trois messages pour chacun des trois boutons de la souris et un message pour la molette.

Le clavier :

WM_KEYDOWN : c'est le premier message qu'envoie Windows à une application lorsqu'on enfonce une touche du clavier.
WM_CHAR : il est envoyé au moment ou WM_KEYDOWN est récupéré par l'application.
WM_KEYUP : c'est le dernier message qu'envoie Windows. Il est envoyé lorsque la touche est relâchée.

Nous avons donc trois messages à notre disposition pour simuler des actions au clavier. En général, l'utilisation de WM_KEYDOWN est suffisante, mais il arrive avec certaines applications (WinAMP 3, Lecteur Windows Media, ...) que ce ne soit pas suffisant. En effet, certaines applications prennent en compte l'envoi du premier message puis ignorent les suivants ; pour éviter cela on peut, quand cela est nécessaire, simuler également le relâchement de la touche avec WM_KEYUP. Il est déconseillé de le faire dans tous les cas car il arrive régulièrement que cela ait pour effet de dédoubler l'action et le résultat ne sera donc pas celui attendu.

La souris :

WM_LBUTTONDOWN : ce message correspond à l'enfoncement du bouton gauche de la souris.
WM_LBUTTONUP : ce message intervient lors du relâchement du bouton gauche de la souris.

WM_LBUTTONDBLCLK : ce message simule un double clique du bouton gauche. Il est précédé des messages WM_LBUTTONDOWN, WM_LBUTTONUP et suivi de WM_LBUTTONUP.
WM_MOUSEWHEEL : ce message simule une rotation de la molette et indique l'amplitude de la rotation.

Chacun des trois premiers messages a son équivalement pour les bouton du milieu (mollette) et de droite en remplaçant respectivement L par M ou R. L'interêt de ces trois messages est assez limité car pour simuler le clique sur un bouton, il faut connaître son identificateur (unique et propre à l'application à laquelle il appartient) et c'est un message du type WM_COMMAND qu'il faut générer. Par contre, WM_MOUSEWHELL peut s'avérer utile car il est fréquent que la molette de la souris permette de contrôler le volume ou de faire défiler des images.

Comment envoyer un message

L'objectif de cette partie est plus d'expliciter les paramètres qu'on va passer à la fonction *PostMessage()* que d'expliquer son fonctionnement. Si j'ai choisi d'utiliser PostMessage() plutôt que SendMessage(), c'est parceque j'ai considéré qu'on n'attendait pas de résultat de la part de l'application qu'on cherche à commander.

Le clavier :

Les messages utilisés pour manipuler le clavier utilisent tous les même paramètres : le code de la touche d'une part et des informations de répétition et / ou d'état des touches systèmes qui ne sont généralement pas utilisées par les applications d'autre part. Nous n'allons détailler que le premier puisque le second ne nous avancera pas dans la résolution de notre problème et que toutes les informations sont disponnibles dans le SDK Microsoft (Win32 SDK).

Le premier paramêtre est donc le code "virtuel" de la touche pressée. Les codes sont des valeurs prédéfinies dans les entêtes windows et commencent par VK. Voici un tableau récapitulatif :

Symbolic constant name	Value (hexadecimal)	Mouse or keyboard equivalent
VK_LBUTTON	01	Left mouse button
VK_RBUTTON	02	Right mouse button
VK_CANCEL	03	Control-break processing
VK_MBUTTON	04	Middle mouse button (three-button mouse)
-	05-07	Undefined
VK_BACK	08	BACKSPACE key
VK_TAB	09	TAB key
-	0A-0B	Undefined

VK_CLEAR	0C	CLEAR key
VK_RETURN	0D	ENTER key
-	0E-0F	Undefined
VK_SHIFT	10	SHIFT key
VK_CONTROL	11	CTRL key
VK_MENU	12	ALT key
VK_PAUSE	13	PAUSE key
VK_CAPITAL	14	CAPS LOCK key
-	15-19	Reserved for Kanji systems
-	1A	Undefined
VK_ESCAPE	1B	ESC key
-	1C-1F	Reserved for Kanji systems
VK_SPACE	20	SPACEBAR
VK_PRIOR	21	PAGE UP key
VK_NEXT	22	PAGE DOWN key
VK_END	23	END key
VK_HOME	24	HOME key
VK_LEFT	25	LEFT ARROW key
VK_UP	26	UP ARROW key
VK_RIGHT	27	RIGHT ARROW key
VK_DOWN	28	DOWN ARROW key
VK_SELECT	29	SELECT key
-	2A	Original equipment manufacturer (OEM) specific
VK_EXECUTE	2B	EXECUTE key
VK_SNAPSHOT	2C	PRINT SCREEN key for Windows 3.0 and later
VK_INSERT	2D	INS key
VK_DELETE	2E	DEL key
VK_HELP	2F	HELP key
VK_0	30	0 key
VK_1	31	1 key
VK_2	32	2 key
VK_3	33	3 key
VK_4	34	4 key

VK_5	35	5 key
VK_6	36	6 key
VK_7	37	7 key
VK_8	38	8 key
VK_9	39	9 key
-	3A-40	Undefined
VK_A	41	A key
VK_B	42	B key
VK_C	43	C key
VK_D	44	D key
VK_E	45	E key
VK_F	46	F key
VK_G	47	G key
VK_H	48	H key
VK_I	49	I key
VK_J	4A	J key
VK_K	4B	K key
VK_L	4C	L key
VK_M	4D	M key
VK_N	4E	N key
VK_O	4F	O key
VK_P	50	P key
VK_Q	51	Q key
VK_R	52	R key
VK_S	53	S key
VK_T	54	T key
VK_U	55	U key
VK_V	56	V key
VK_W	57	W key
VK_X	58	X key
VK_Y	59	Y key
VK_Z	5A	Z key
VK_LWIN	5B	Left Windows key (Microsoft Natural Keyboard)

VK_RWIN	5C	Right Windows key (Microsoft Natural Keyboard)
VK_APPS	5D	Applications key (Microsoft Natural Keyboard)
-	5E-5F	Undefined
VK_NUMPAD0	60	Numeric keypad 0 key
VK_NUMPAD1	61	Numeric keypad 1 key
VK_NUMPAD2	62	Numeric keypad 2 key
VK_NUMPAD3	63	Numeric keypad 3 key
VK_NUMPAD4	64	Numeric keypad 4 key
VK_NUMPAD5	65	Numeric keypad 5 key
VK_NUMPAD6	66	Numeric keypad 6 key
VK_NUMPAD7	67	Numeric keypad 7 key
VK_NUMPAD8	68	Numeric keypad 8 key
VK_NUMPAD9	69	Numeric keypad 9 key
VK_MULTIPLY	6A	Multiply key
VK_ADD	6B	Add key
VK_SEPARATOR	6C	Separator key
VK_SUBTRACT	6D	Subtract key
VK_DECIMAL	6E	Decimal key
VK_DIVIDE	6F	Divide key
VK_F1	70	F1 key
VK_F2	71	F2 key
VK_F3	72	F3 key
VK_F4	73	F4 key
VK_F5	74	F5 key
VK_F6	75	F6 key
VK_F7	76	F7 key
VK_F8	77	F8 key
VK_F9	78	F9 key
VK_F10	79	F10 key
VK_F11	7A	F11 key
VK_F12	7B	F12 key
VK_F13	7C	F13 key
VK_F14	7D	F14 key

VK_F15	7E	F15 key
VK_F16	7F	F16 key
VK_F17	80H	F17 key
VK_F18	81H	F18 key
VK_F19	82H	F19 key
VK_F20	83H	F20 key
VK_F21	84H	F21 key
VK_F22	85H	F22 key
VK_F23	86H	F23 key
VK_F24	87H	F24 key
-	88-8F	Unassigned
VK_NUMLOCK	90	NUM LOCK key
VK_SCROLL	91	SCROLL LOCK key
-	92-B9	Unassigned
-	BA-C0	OEM specific
-	C1-DA	Unassigned
-	DB-E4	OEM specific
-	E5	Unassigned
-	E6	OEM specific
-	E7-E8	Unassigned
-	E9-F5	OEM specific
VK_ATTN	F6	Attn key
VK_CRSEL	F7	CrSel key
VK_EXSEL	F8	ExSel key
VK_EREOF	F9	Erase EOF key
VK_PLAY	FA	Play key
VK_ZOOM	FB	Zoom key
VK_NONAME	FC	Reserved for future use.
VK_PA1	FD	PA1 key
VK_OEM_CLEAR	FE	Clear key

Ce tableau à été extrait de l'aide du SDK Win32 de Microsoft

Il arrive que les éditeurs de compilateurs ne définissent pas les constantes VK_A à VK_Z et cela pour la simple et bonne raison que c'est en réalité la valeur du caractère majuscule correspondant. Dans ce cas, il suffit de remplacer VK_A par 'A' et ainsi de suite pour les atres lettres de l'alphabet.

Voyons un exemple d'utilisation de la fonction PostMessage(). Ce programme va rechercher une instance du lecteur Windows Media et lancer la lecture en simulant la pression de la barre d'espace puis le mettre en pause:

```
#include <windows.h>
 int main(int argc, char * argv[])
{
     HWND hMPlayer = FindWindow("Media Player 2",NULL);
     SetForegroundWindow(hMPlayer);
     Sleep(100);
     PostMessage(hMPlayer,WM_KEYDOWN,VK_SPACE,0);
     Sleep(5000);
     PostMessage(hMPlayer,WM_KEYUP,VK_SPACE,0);
     return 0;
}
```

Nous savons maintenant simuler des actions du clavier, mais il n'est cependant toujours pas possible de simuler la pression d'un raccourcis clavier plus complexe qu'une touche tel que ALT + ENTREE.

La souris :

Comme je l'ai dit dans la partie précédente, l'interêt des évênements WM_LBUTTONDOWN, WM_LBUTTONUP et WM_LBUTTONDBLCLK est minime dans notre démarche, nous n'allons donc étudier que l'utilisation du message WM_MOUSEWHEEL.

Le message WM_MOUSEWHEEL fournit quatre informations de 16 bits chacunes réparties entre les mots hauts et bas des deux arguments (32 bits chacuns) de PostMessage() restants à notre disposition. Voyons quelles sont les informations à fournir :

Tout d'abord, il y a les états des touches systèmes et des trois boutons de la souris. Ceux-ci sont représentés à l'aide de masques prédéfinis dans le SDK Microsoft par les constantes : MK_CONTROL (Contrôle), MK_SHIFT (Majuscule), MK_LBUTTON, MK_MBUTTON et MK_RBUTTON (boutons gauche, milieu et droit de la souris). Il n'y a aucune garantie que l'application cible s'en serve

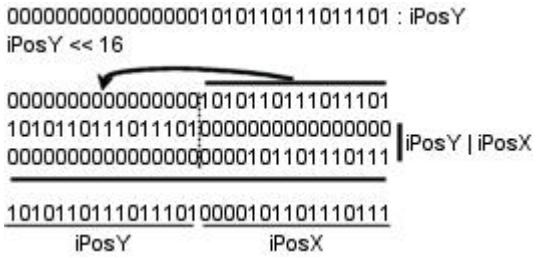
Ensuite, il y a la valeur de la rotation de la molette. Une valeur positive représente une rotation vers le haut (vers le fil de la souris) et une valeur négative représente une rotation vers le bas (vers l'utilisateur). La rotation équivalent à un "cran" de la molette est définie par la constante WHEEL_DELTA (120). Il est possible d'utiliser des valeurs plus précises, mais toutes les applications n'en tiennent pas compte.

Finalement, il y a la positon verticale et horizontale du curseur au moment de l'évènement. Là encore, il n'y a aucune garantie que

l'application cible s'en serve.

Voyons maintenant comment construire les arguments de PostMessage() à partir de ces informations. La position du curseur doit être placée dans le second paramètre et les autres informations dans le premier. L'exemple ci-dessous montre comment le faire en C :

```
int iPosX,iPosY;
int iMasqueTouches;
int iDeplMolette;
DWORD dwWParam,dwLParam;
HWND hWindow;
// initialisation des variables avec les valeurs souhaitées
dwLParam = iPosY << 16 | iPosX; // on effectue un décallage de 16 bits vers la gauche
dwWParam = iDeplMolette << 16 | iMasqueTouches;
PostMessage(hWindow,WM_MOUSEWHEEL, dwWParam, dwLParam);
```



Une autre solution peut consister à utiliser une structure :

```
struct _dword {
     short w1;
     short w2;
} dwWParam, dwLParam;
int iPosX,iPosY;
int iMasqueTouches;
int iDeplMolette;
HWND hWindow;
// initialisation des variables avec les valeurs souhaitées
dwLParam.w1 = iPosY;
dwLParam.w2 = iPosX;
```

```
dwWParam.w1 = iDeplMolette;
dwWParam.w2 = iMasqueTouches;
PostMessage(hWindow,WM_MOUSEWHEEL, (DWORD)(dwWParam), (DWORD)(dwLParam));
```

Voilà, nous avons maintenant tous les éléments dont nous avions besoin pour simuler des actions simples à l'attention d'une application déterminée. Cependant, cela n'est pas suffisant car on ne peut toujours pas faire de simulation globale, ni déplacer la souris : il n'est donc pas possible d'interfacer un périphérique multimédia tel qu'un joystick avec la sourris et le clavier. C'est pourquoi nous allons voir dans une deuxième partie comment simuler le clavier et la souris à un niveau beaucoup plus proche de la machine.

Prendre le contrôle du clavier et de la souris

Comme nous l'avons vu dans la partie précédente, il n'est pas toujours possible d'obtenir le résultat attendu avec la file de messages de Windows. On va donc devoir intervenir à un niveau beaucoup plus proche du matériel. En effet, nous allons agir comme le ferait un pilote de périphérique pour la souris ou le clavier. Vous devez être en train de vous dire houlala ... il va nous sortir des commandes en assembleur de derrière les fagots. Rassurez-vous, il n'en est rien : nous n'utiliserons que des fonctions de l'API Win32. Nous allons comme à l'habitude commencer par traiter le sujet du clavier, puis nous verrons que le fonctionnement de la souris y ressemble beaucoup.

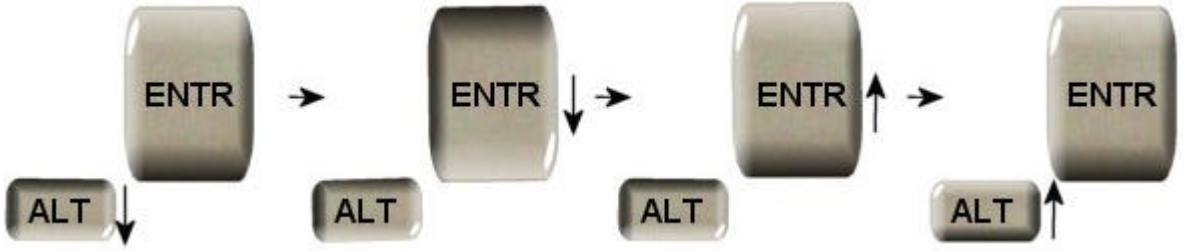
Le clavier dans tous ses états

Dans cette partie, nous allons voir comment faire croire au système qu'une touche est pressée ou non. Si je parle ici de système et non pas d'application, c'est parceque la fonction que nous allons utiliser n'a pas conscience d'une telle chose. Pour elle, il n'y à qu'un seul logiciel en cours : Windows ; et elle se contente de lui indiquer les modification d'état du clavier. Il faudra donc prendre en compte lors des futurs développements qu'il n'est pas possible de dire "je veux simuler telle action pour telle fenêtre".

Pour comprendre le fonctionnement du principe que nous allons voir, il faut considérer que le programme qu'on est en train de réaliser est en réalité le pilote d'un périphérique réel et agir en tant que tel. Les pilotes de périphérique des claviers utilisent la fonction *keybd_event()* pour indiquer au système l'état des différentes touches. Comme son nom l'indique, cette fonction ne signalera au système que les changements d'état des touches. Il n'y aurra donc pas à intercepter les significations d'état que pourraient être amenés à faire d'autres pilotes de périphériques si le système avait voulu qu'on l'informe toutes les n millisecondes de l'état de chacune des touches. Voyons maintenant quels sont les paramètres que la fonction keybd_event() attend.

La fonction keybd() prend quatre paramètres : le code touche virtuel (ENTREE, A, F1, ...), le code matériel de la touche (touche n° 1, n°2, ...), un masque indiquant si la touche est une touche étendue et si la touche est relâchée et finalement un double-mot contenant des information supplémentaires. Les codes de touches virtuels ont été présentés dans la partie précédente. Les codes matériels de touche ne sont pas utiles pour nous car ils correspondent à une étape antérieure des pilotes de périphérique ; nous le laisserons donc à 0. Le masque est composé à partir de KEYEVENTF_EXTENDEDKEY qui indique que la touche est soit AltGr soit Ctrl droite, et KEYEVENTF_KEYUP qui indique que la touche est relâchée. Finalement le dernier paramètre est une donnée complémentaire qui ne nous sert à rien non plus, on le laissera donc à 0. Voici un exemple dans lequel on considère que par exemple le lecteur Windows Media est en train de lire une video. Notre programme exemple va simuler la pression de Alt + Entree pour faire passer le lecteur en mode plein écran.

```
#include <windows.h>
int main(int argc, char *argv[])
{
    SetForegroundWindow(FindWindow("Media Player 2",NULL));
    Sleep(100);
    keybd_event(VK_LMENU,0,0,0);
    keybd_event(VK_RETURN,0,0,0);
    keybd_event(VK_RETURN,0,KEYEVENTF_KEYUP,0);
    keybd_event(VK_LMENU,0,KEYEVENTF_KEYUP,0);
    return 0;
}
```



Les poules picorent, la souris cliquette

De la même façon que précédemment, nous allons faire croire au système qu'il y a eu un click ou un mouvement de souris. Contrairement au cas du clavier, nous allons avoir besoins de plusieurs fonctions. Nous allons donc partager cette partie en deux thèmes : le déplacement du curseur d'une part et les clicks et autres tours de molettes d'autre part.

Déplacer le curseur :

Pour déplacer le curseur, nous allons utiliser la fonction *SetCursorPos()*. Cependant, si on veut déplacer le curseur, il va d'abord falloir déterminer sa position actuelle puis indiquer à Windows quelle est sa nouvelle position ; c'est ce que nous allons faire avec la fonction *GetCursorPos ()*.

La fonction GetCursorPos() prends en paramètre un pointeur sur une structure POINT qui contient deux entiers longs (type LONG de l'API) : la position x et y du curseur. La fonction SetCursorPos() prends en paramètre deux entiers les coordonnées x et y ou placer le curseur. Le point d'origine est le coin supérieur gauche de l'écran. Voici un exemple qui va récupérer la position du curseur et le déplacer en fonction de la touche flèchée enfoncée :

```
#include <windows.h>
int main(int argc, char *argv[])
{
     POINT pt;
     BOOL bContinue = TRUE;
     const SHORT Mask = 32768;
     while (bContinue)
     {
          if (GetKeyState(VK_ESCAPE) & Mask)
               bContinue = FALSE;
          GetCursorPos(&pt);
          if (GetKeyState(VK_UP) & Mask)
               pt.y -= 1;
          if (GetKeyState(VK_DOWN) & Mask)
               pt.y += 1;
          if (GetKeyState(VK_LEFT) & Mask)
               pt.x -= 1;
          if (GetKeyState(VK_RIGHT) & Mask)
               pt.x += 1;
          SetCursorPos(pt.x,pt.y);
          Sleep(10);
     }
     return 0;
}
```

Simuler les clicks et la mollette

Pour simuler les clicks et la molette, nous allons utiliser une fonction très semblable à celle utilisée pour le clavier : *mouse_event()*. Cette fonction prend cinq paramètres. Le premier est un double-mot qui contient des bits drapeaux pour indiquer l'évènement qui a eu lieu. Le second et le troisième paramètre sont les coordonnées x et y du curseur au moment ou l'évènement a eu lieu. Le quatrième paramètre est la valeur de la rotation de la mollette. Le dernier paramètre est une information complémentaire sur 32 bits dont nous ne nous servirons pas.

Revenons un instant sur les drapeaux qui décrivent l'évènement.

MOUSEEVENTF_ABSOLUTE	Indique que les coordonées x et y données sont exprimées a partir du coin supérieur gauche de l'écran. Si ce drapeau n'est pas activé, les coordonnées sont relatives au dernier rapport de position. Il est fortement conseillé de donner des coordonnées absolues, car on n'a aucun moyen de contrôle pour affirmer quelle était la position précédente.
MOUSEEVENTF_MOVE	un mouvement vient d'avoir lieu
MOUSEEVENTF_LEFTDOWN	Le bouton gauche est baissé
MOUSEEVENTF_LEFTUP	Le bouton gauche est remonté
MOUSEEVENTF_RIGHTDOWN	Le bouton droit est baissé
MOUSEEVENTF_RIGHTUP	Le bouton droit est remonté
MOUSEEVENTF_MIDDLEDOWN	Le bouton du milieu est baissé
MOUSEEVENTF_MIDDLEUP	Le bouton du milieu est remonté
MOUSEEVENTF_WHEEL	La molette a effectué une rotation dont la quantité est indiquée dans l'avant-dernier paramètre

Voyons maintenant un exemple. Nous allons complêter l'exemple précédent en ajoutant les fonctionnalités suivantes : les touches 1, 2 et 3 du pavé numérique seront les trois boutons de la souris, le - du pavé numérique sera la mollette vers le haut et le + la mollette vers le bas.

```
#include <windows.h>
int main(int argc, char *argv[])
{
     POINT pt;
     BOOL bContinue = TRUE;
     SHORT Mask = 32768;
     DWORD dwEventFlags;
     DWORD dwData;

     while (bContinue)
     {
          dwData = 0;
          dwEventFlags = MOUSEEVENTF_ABSOLUTE;
```

```
if (GetKeyState(VK_ESCAPE) & Mask)
     bContinue = FALSE;
GetCursorPos(&pt);
if (GetKeyState(VK_NUMPAD1) & Mask)
{
     if (!(GetKeyState(VK_LBUTTON) & Mask))
         dwEventFlags |= MOUSEEVENTF_LEFTDOWN;
}
else if (GetKeyState(VK_LBUTTON) & Mask)
     // Ce test est nécessaire pour s'assurer qu'on n'envoie pas plus
     // de relevés de touche que d'abaissements
     dwEventFlags |= MOUSEEVENTF_LEFTUP;
if (GetKeyState(VK_NUMPAD2) & Mask)
{
     if (!(GetKeyState(VK_MBUTTON) & Mask))
         dwEventFlags |= MOUSEEVENTF_MIDDLEDOWN;
}
else if (GetKeyState(VK_MBUTTON) & Mask)
     dwEventFlags |= MOUSEEVENTF_MIDDLEUP;
if (GetKeyState(VK_NUMPAD3) & Mask)
{
     if (!(GetKeyState(VK_RBUTTON) & Mask))
         dwEventFlags |= MOUSEEVENTF_RIGHTDOWN;
}
else if GetKeyState(VK_R3BUTTON) & Mask)
     dwEventFlags |= MOUSEEVENTF_RIGHTUP;
if (GetKeyState(VK_SUBTRACT) & Mask)
{
     dwEventFlags |= MOUSEEVENTF_WHEEL;
     dwData += WHEEL_DELTA;
}
else if (GetKeyState(VK_ADD) & Mask)
{
     dwEventFlags |= MOUSEEVENTF_WHEEL;
     dwData -= WHEEL_DELTA;
}
if (GetKeyState(VK_UP) & Mask)
     pt.y -= 1;
if (GetKeyState(VK_DOWN) & Mask)
     pt.y += 1;
if (GetKeyState(VK_LEFT) & Mask)
     pt.x -= 1;
if (GetKeyState(VK_RIGHT) & Mask)
     pt.x += 1;
```

```
            SetCursorPos(pt.x,pt.y);
            mouse_event(dwEventFlags,pt.x,pt.y,dwData,0);
            Sleep(10);
        }
        return 0;
}
```

Maintenant que nous avons vu touts les moyebs mis à notre disposition pour résoudre les problèmes que nous pose le contrôle d'applications tierces, il ne nous reste plus qu'à résoudre certains problèmes qui peuvent apparaître dans leur mise en oeuvre.

Divers problèmes et leur voie de résolution

Dans certains cas, il serait nécessaire d'avoir un hybride des deux méthodes présentées dans cet exposé. En effet, il est souvent nécessaire de simuler un raccourci clavier complexe utilisant une touche modificateur telle que Alt ou Ctrl, ce qui nous oriente vers une solution de type "contrôle du clavier", mais on voudrait en plus pouvoir indiquer à quelle fenêtre envoyer le message. Il arrive également qu'on ne connaisse pas la fenêtre à laquelle envoyer le message. Je vous propose ici diverses solutions pour résoudre ce genre de problèmes.

Ciblage d'une fenêtre spécifique avec la fonction de contrôle du clavier

Comme vous avez pu le remarquer, on aurrait bien souvent besoin d'une fonction hybride entre PostMessage() et keybd_event(). En effet, il n'est pas possible de simuler le raccourci clavier ALT + ENTREE avec la fonction PostMessage() et l'utilisation de keybd_event() ne permet pas de limiter l'évènement à une application cible. Par ailleurs, une combinaison de ces deux fonction ne résoud pasle problème à coup sûr. Le problème vient du fait que la fonction PostMessage() n'attends pas que le message ai été traité par l'application cible avant de rendre la main à notre programme de contrôle. Pour parer à ce problème, on est obligé d'utiliser la fonction SendMessage() qui a le même prototype que PostMessage() mais qui attend que le message ai été traité pour rendre la main. Cependant, si l'application cible ne rend pas la main très rapidement il y a de grande chances pour que l'utilisateur appuye sur touche du clavier pendant ce laps de temps et on ne peut pas mesurer les conséquences de cette interférence. La seule solution qui nous reste est donc d'espèrer que l'utilisateur sera loin de son clavier au moment où on effectuera la simulation.

Voyons un bref exemple de simulation pseudo-ciblée d'un raccourci clavier. Nous allons faire passer l'affichage de la video du lecteur Windows Media en plein écran (ou inverse)

```
#include <windows.h>
#include <stdio.h>
int main(int argc, char * argv[])
{
     HWND hMediaplayer = FindWindow("Media Player 2",NULL);
     if (hMediaplayer)
     {
          keybd_event(VK_LMENU,0,0,0);
          SendMessage(hMediaPlayer,WM_CHAR,VK_RETURN,0);
          keybd_event(VK_LMENU,0,KEYEVENTF_KEYUP,0);
     }
     return 0;
}
```

Trouver le class name d'une fenêtre quelconque.

Voici un exemple de programme permettant de retrouver le class name d'une fenêtre quelconque. Celui-ci est facilement adaptable à une interface graphique pour que n'importe quel utilisateur puisse fournir le class name au programme qui en a besoin.

```
#include <windows.h>
#include <stdio.h>
#include <conio.h>
int main(int argc, char * argv[])
{
     HWND hFore;
     char szBuffer[128];
     int iNbChar;

     printf("Vous avez 5 secondes pour activer la fenêtre dont vous voulez connaître le class name\r\n");
     printf("Appuyez sur une touche pour déclancher le compteur ...");
     getch();
     Sleep(5000);

     hFore = GetForegroundWindow();
```

```
	iNbChar = GetClassName(hFore,szBuffer,128);
	if (iNbChar && (iNbChar < 128))
	{
		printf("\r\nLe class name de cette fenêtre est :\r\n|%s|",szBuffer);
		printf("\r\nLes barres verticales (|) de gauche et droite n'en font pas partie\r\n");
	}
	else
	{
		printf("Impossible de récupérer le class name ou celui-ci est trop long\r\n");
	}
	return 0;
}
```

Les hooks

(non non, ici on ne parle pas du capitaine crochet ;-))

L'objectif d'une application qui contrôle le clavier ou la souris, n'est pas toujours de simuler un comportement, mais peut également être de réagir à un comportement de l'utilisateur. Par ailleurs, il peut être utile d'étudier le comportement d'une application pour mieux la contrôler par la suite. Nous allons voir comment faire cela au travers des hooks. Nous verrons tout d'abord ce qu'est un hook, les différents types de hooks que propose Windows, les prérequis pour la mise en place d'un hook, puis comment les mettre en place.

Qu'est-ce qu'un hook ?

Un hook est un moyen de contrôler ce qui transite par le système de gestion de messages de Windows : c'est comme si on dénudait un fil électrique et qu'on y branchait un appareil de mesure pour voir ce qui s'y passe. La mise en place d'un hook va permettre de demander au système d'exploitation de faire un appel à une fonction de notre cru à chaque fois qu'il rencontrera un certain type de message et avant de le transmettre à l'application cible.

Les différents types de hooks

L'API Win32 propose actuellement 11 types de hooks dont deux sont spécialisés en deux sous types chacun soit un total de 13 hooks possibles. Chaque type de hook permet à une application de surveiller différents aspects du mécanisme de gestion des messages de Windows.

Voici un bref descriptif de chacun d'entre eux :

Constante	Action hookée	Portée
WH_CALLWNDPROC	voir les messages envoyés par SendMessage avant l'application destinataire.	Thread ou Système
WH_CALLWNDPROCRET	idem mais après traitement.	Thread ou Système
WH_CBT	contrôler et autoriser les actions essentielles (activation, création, destruction, réduction, agrandissement, déplacement, redimensionnement d'une fenêtre, avant de terminer une commande système, avant de retirer un événement de la souris ou du clavier de la file de message, avant d'appliquer le focus, avant de synchroniser l'application avec la file de messages)	Thread ou Système
WH_DEBUG	autoriser ou non l'appel des fonctions associées aux autres hooks car il est appelé avant n'importe quel autre type de hook.	Thread ou Système
WH_FOREGROUNDIDLE	effectuer des applications de basse priorité quand le thread principal de l'application est inactif.	Thread ou Système
WH_GETMESSAGE	contrôler tout type de message envoyé à une application.	Thread ou Système
WH_JOURNALPLAYBACK	insérer des messages enregistrés avec WH_JOURNALRECORD dans la file système.	Système
WH_JOURNALRECORD	enregistrer une séquence d'événement pour les rejouer plus tard.	Système
WH_KEYBOARD	intercepter les messages de type WM_KEYDOWN et WM_KEYUP.	Thread ou Système
WH_MOUSE	intercepter les messages de la souris.	Thread ou Système
WH_MSGFILTER	intercepter les messages destinés aux menus, ascenseurs, boites de dialogue et de détecter si une fenêtre va être active par ALT+TAB ou ALT+ESC de l'application.	Thread ou Système
WH_SYSMSGFILTER	idem mais pour toutes les applications	Système
WH_SHELL	recevoir les notifications importantes (une fenêtre au premier plan est sur le point d'être créée ou détruite)	Thread ou Système

Ce tutoriel étant destiné à la gestion du clavier et de la souris, nous ne verrons que les hooks WH_MOUSE et WH_KEYBOARD, mais les autres hooks s'utilisent exactement de la même façon.

Prérequis à la mise en place d'un hook

Deux problèmes se posent lorsqu'on veut créer un hook. Premièrement, si celui-ci est destiné à observer les événement du système complet, on ne peut pas implémenter la fonction associée à ce crochet dans un programme séparé mais on est obligé de le faire dans une DLL séparée. Exactement comme un programme normal qui ne peut se servir d'une fonction développée par un tiers que si elle se trouve dans une DLL (et autres ActiveX...) et non dans un programme normal. Deuxièmement, la fonction qui gère le hook doit transmettre le message qu'elle a reçu dans la chaîne et pour cela elle a besoin du handle du hook ; hors, si on pourrait être tenté de stocker cette information de manière globale, cela ne résoud en rien le problème car il existe en réalité une instance de la DLL par processus recevant des messages ce qui implique que le handle ne serait connu que du processus appelant et que cela provoquerait des erreurs dans les autres applications. On va donc devoir créer une zone de mémoire nommée afin que toutes les instances de la DLL puissent accéder à ce handle de hook.

Dans l'exemple exposé dans la partie suivante, on considérera qu'on évolue sous Borland C++ Builder et qu'on a utilisé l'expert DLL. Pour des informations complémentaires concernant la création de DLL dans BC++ Builder, voyez ce tutoriel de LFE

Mise en place d'un hook

Pour mettre en place un hook, il suffit d'un appel à la fonction SetWindowsHookEx() et pour l'arrêter nous ferrons appel à UnhookWindowsHookEx(). SetWindowsHookEx() prend quatre arguments : le type de hook à mettre en place, le pointeur sur la fonction à utiliser pour le hook, l'instance de la DLL contenant la fonction de hook et finalement l'identifiant du thread à hooker. Elle renvoie le handle du hook créé qui a pour la valeur NULL en cas d'échec. UnhookWindowsHookEx() prends comme seul paramètre le handle du hook à défaire et renvoie la constante FALSE si elle échoue.

Voyons le prototype de la fonction de hook :

```
LRESULT CALLBACK HookProc(int nCode, WPARAM wParam, LPARAM lParam);
```

Pour chaque type de hook, ces trois paramètres ont une signification particulière : nCode est un indicateur pour l'action a effectuer et wParam et lParam contiennent les informations du message qui a été envoyé. La valeur renvoyée par cette fonction doit être celle renvoyée par la fonction de propagation du message, CallNextHookEx(). Cette fonction prends en paramètres le handle du hook qu'on vient de gérer, suivi des arguments reçus par HookProc(), éventuellement modifiés. Si le message est intercepté par le hook (pas d'appel à CallNextHookEx()), la valeur renvoyée par HookProc() devrait être nulle (0L en C).

Détaillons maintenant le contenu des paramètres pour les fonctions associées à WH_KEYBOARD et à WH_MOUSE.

- WH_KEYBOARD :
  Le paramètre nCode prends sa valeur parmis HC_ACTION et HC_NOREMOVE. Dans le cas (rare) de la seconde valeur, cela

veut dire qu'une application a lu le message sans le retirer de la file par un appel à PeekMessage() spécifiant le drapeau PM_NOREMOVE. Dans tous les cas, wParam et lParam contiennent les informations a propos de l'événement clavier. Il peut arriver que ce paramètre ait une valeur négative, la fonction doit alors immédiatement se terminer dans un appel à CallNextHookEx().

Les paramètres wParam et lParam contiennent les mêmes informations que dans le cas de WM_KEYDOWN ou WM_KEYUP : le code virtuel de la touche actionnée et les informations de répétition, d'état ...

- WH_MOUSE :

Le paramètre nCode prends sa valeur parmis HC_ACTION et HC_NOREMOVE. Dans le cas (rare) de la seconde valeur, cela veut dire qu'une application a lu le message sans le retirer de la file par un appel à PeekMessage() spécifiant le drapeau PM_NOREMOVE. Dans tous les cas, wParam et lParam contiennent les informations a propos de l'événement souris. Il peut arriver que ce paramètre ait une valeur négative, la fonction doit alors immédiatement se terminer dans un appel à CallNextHookEx().

Le paramètre wParam contient le type du message (WM_LBUTTONDOWN à WM_MOUSEWHEEL)

Le paramètre lParam contient un pointeur sur une structure MOUSEHOOKSTRUCT qui, elle, contient une structure POINT indiquant la position de la souris ; le handle de la fenêtre qui va recevoir le message ; le type de cible qui se trouve sous la souris et finalement des informations complémentaires pour le message. Voici un tableau exhaustif des types de cibles.

Valeur	Position du curseur
HTBORDER	Sur le bord d'une fenêtre qui ne peut pas être redimensionnée
HTBOTTOM	Sur le bord horizontal inférieur d'une fenêtre
HTBOTTOMLEFT	Sur le coin inférieur gauche d'une fenêtre
HTBOTTOMRIGHT	Sur le coin inférieur droit d'une fenêtre
HTCAPTION	Sur une barre de titre
HTCLIENT	Sur une zone cliente
HTERROR	Sur le fond d'écran ou sur une ligne de séparation entre deux fenêtres (pareil que HTNOWHERE, à l'exception que la fonction DefWindowProc() produit un beep système pour indiquer une erreur)
HTGROWBOX	Sur une boite de redimensionnement (pareil que HTSIZE)
HTHSCROLL	Sur un ascenceur horizontal
HTLEFT	Sur le bord gauche d'une fenêtre
HTMENU	Sur un menu
HTNOWHERE	Sur le fond d'écran ou sur une ligne de séparation entre deux fenêtres
HTREDUCE	Sur un bouton de minimisation
HTRIGHT	Sur le bord droit d'une fenêtre
HTSIZE	Sur une boite de redimensionnement (pareil que HTGROWBOX)
HTSYSMENU	Sur un menu system ou sur un bouton Fermer d'une fenêtre enfant.
HTTOP	Sur le bord horizontal supérieur d'une fenêtre
HTTOPLEFT	Sur le coin supérieur gauche d'une fenêtre

HTTOPRIGHT	Sur le coin supérieur droit d'une fenêtre
HTTRANSPARENT	Sur une fenêtre recouverte par une autre
HTVSCROLL	Sur un ascenceur vertical
HTZOOM	Sur un bouton de maximisation

Voyons maintenant un exemple de DLL permettant le hook du clavier et de la souris. Celle-ci sera composé de cinq fonctions : la fonction d'inistialisation (fermeture) de la DLL qui attachera (détachera) la zone de mémoire partagée, la fonction d'initialisation des hooks, la fonction de destruction des hooks et les deux fonctions de crochetage (clavier et souris).

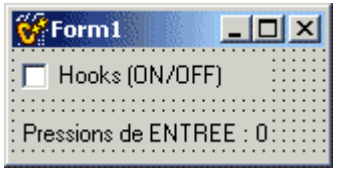
Voici le code de la DLL. Le fichier compilé sera nommé "hooksClSo.dll".

```
#include <windows.h>
// Définition de la structure de la zone mémoire partagée

typedef struct _TDonnees
{
     HHOOK MouseHookHandle; // Handle du hook de la souris
     HHOOK KeybdHookHandle; // Handle du hook du clavier
     HWND hDestWindow; // Handle de la fenêtre à laquelle le hook du clavier doit les données
     // Mettez ici toutes les données que vous voulez partager
} TDonnees;

// Déclaration des variables globales de la DLL
HANDLE MemPartagee; // Handle de la zone de mémoire partagée
TDonnees * VueDonnees; // Pointeur vers la zone de mémoire
HINSTANCE HInst; // Handle d'intance de la DLL

// Déclaration des fonctions de la DLL
void _export InitHook(HWND hDest);
void _export EndHook();
LRESULT CALLBACK MouseProc(int nCode,WPARAM wParam,LPARAM lParam);
LRESULT CALLBACK KeybdProc(int nCode,WPARAM wParam,LPARAM lParam);

// fonction d'initialisation de la DLL
int WINAPI DllEntryPoint(HINSTANCE hinst, unsigned long reason, void* lpReserved)
{
     HInst = hinst;
     switch (reason)
     {
          case DLL_PROCESS_ATTACH : // à la création de l'instance de la DLL
```

```
                // Attachement d'une zone de mémoire partagée (notez le cast)
                MemPartagee = CreateFileMapping((HANDLE)0xFFFFFFFF, // On map un espace mémoire
                                        NULL, // Pas d'informations de sécurité
                                        PAGE_READWRITE, // Partage en lecture/écriture
                                        0, // Taille de la zone réservée sur 64 bits (32 bits de poid fort)
                                        sizeof(TDonnees), // 32 bits de poids faible
                                        "Tutoriel Hooks par gRRosminet"); // Nom de la zone réservée

                // Création d'une vue pour pouvoir accéder à la zone de mémoire partagée (notez le cast)
                VueDonnees = (TDonnees *)(MapViewOfFile((HANDLE)MemPartagee, // Zone sur laquelle créer la vue
                                        FILE_MAP_WRITE, // Mode d'accès en lecture/écriture
                                        0, 0, // Offset de début de la vue sur 64 bits
                                        0)); // Taille de la vue (0 = tout)
                                        break;

        case DLL_PROCESS_DETACH : // au détachement de la DLL
                // Destruction de la vue sur la zone de mémoire partagée
                UnmapViewOfFile((LPVOID)VueDonnees);
                // Détachement de la zone de mémoire partagée
                CloseHandle(MemPartagee);
                break;

        default :
                // DLL_THREAD_ATTACH, DLL_THREAD_DETACH
                break;
    }
    return 1;
}

void _export InitHook(HWND hDest) // _export est spécifique à BCB
{
    // Installation du hook sur la souris
    VueDonnees->MouseHookHandle = SetWindowsHookEx(WH_MOUSE, // Créer un hook sur la souris
                                        (HOOKPROC)MouseProc, // Utiliser la fonction MouseProc
                                        HInst, // Dans la DLL d'instance HInst
                                        0); // Pour tous les threads

    // Installation du hook pour le clavier
    VueDonnees->KeybdHookHandle = SetWindowsHookEx(WH_KEYBOARD, // Créer un hook sur le clavier
                                        (HOOKPROC)KeybdProc, // Utiliser la fonction KeybdProc
                                        HInst, // Dans la DLL d'instance HInst
                                        0); // Pour tous les threads
```

```
     // Partage de la fenêtre destinatrice des données du clavier
     VueDonnees->hDestWindow = hDest;
}

void _export EndHook()
{
     // Supression des hooks
     UnhookWindowsHookEx(VueDonnees->MouseHookHandle);
     UnhookWindowsHookEx(VueDonnees->KeybdHookHandle);
}

LRESULT CALLBACK _export MouseProc(int nCode,WPARAM wParam,LPARAM lParam)
{
     // On joue un son à chaque fois que l'utilisateur enfonce le bouton gauche
     if (wParam == WM_LBUTTONDOWN)
           MessageBeep(MB_OK);
      return CallNextHookEx(VueDonnees->MouseHookHandle,nCode,wParam,lParam);
}

#define WMAP_KEYBDHOOKMSG WM_APP + 1

LRESULT CALLBACK _export KeybdProc(int nCode,WPARAM wParam,LPARAM lParam)
{
     // On envoie un message WMAP_KEYBDHOOKMSG à chaque fois que
      // l'utilisateur presse la touche ENTREE
     if (wParam == VK_RETURN)

       PostMessage(VueDonnees->hDestWindow, WMAP_KEYBDHOOKMSG, 0, 0);

     return CallNextHookEx(VueDonnees->KeybdHookHandle,nCode,wParam,lParam);
}
```

Voyons maintenant l'application. Tout d'abord créons une fiche qui contient une case à cocher et un label comme le montre l'image ci-contre. Ensuite, il va faloir définir un gestionnaire de message pour WMAP_KEYBDHOOKMSG ainsi que quelques variables nécessaires à l'application. Ci-après, l'entête de la fiche.




```
#ifndef appliH
```

```
#define appliH

#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>

typedef void (*TInitFunc)(HWND);
typedef void (*TEndFunc)();

#define WMAP_KEYBDHOOKMSG WM_APP + 1

class TForm1 : public TForm
{
     __published: // Composants gérés par l'EDI
          TCheckBox *CheckBox1;
          TLabel *Label1;
          void          __fastcall CheckBox1Click(TObject *Sender);
          void          __fastcall FormClose(TObject *Sender, TCloseAction &Action);
     private: // Déclarations utilisateur
          TInitFunc InitHooks; // fonction d'initialisation des hooks
          TEndFunc EndHooks; // fonction de suppression des hooks
          HINSTANCE hinstDLL; // instance de la DLL
          bool bHook; // Les hooks ont-ils été initialisés ?
          int nbEntree; // compteur de pressions pour le gestionnaire de messages
          public: // Déclarations utilisateur
          void __fastcall PressionEntree(TWMNoParams &p);
          __fastcall TForm1(TComponent* Owner);
     protected: // déclaration du gestionaire de messages
          BEGIN_MESSAGE_MAP
               VCL_MESSAGE_HANDLER(WMAP_KEYBDHOOKMSG, TWMNoParams, PressionEntree)
          END_MESSAGE_MAP(TForm)
};

extern PACKAGE TForm1 *Form1;

#endif
```

Voyons maintenant du côté du source :

```
#include <vcl.h>
```

```
#include "appli.h"

#pragma package(smart_init)
#pragma resource "*.dfm"

TForm1 *Form1;

__fastcall TForm1::TForm1(TComponent* Owner)
          : TForm(Owner)
{
     bHook = false; // Les hooks ne sont pas initialisés
     hinstDLL = LoadLibrary("hooksClSo.dll"); // Chargement de la librairie
     if (!hinstDLL) // Erreur lors du chargement de la librairie ?
          Application->MessageBox("Impossible de charger la librairie.","gloups",MB_OK);
     else
         {
          // On récupère les adresses des fonctions
          InitHooks = (TInitFunc)GetProcAddress(hinstDLL, "_InitHook");
          EndHooks = (TEndFunc)GetProcAddress(hinstDLL, "_EndHook");
     }
}

void __fastcall TForm1::CheckBox1Click(TObject *Sender)
{
     if (!hinstDLL) // La librairie n'est pas chargée, inutile de continuer
          Close();
     else if (CheckBox1->Checked) // Activer les hooks
     {
          nbEntree = 0; // On réinitialise le compteur
          InitHooks(Form1->Handle); // On initialise les hooks
     }
     else // Désactiver les hooks
          EndHooks(); // On supprime les hooks
     }

 void __fastcall TForm1::FormClose(TObject *Sender, TCloseAction &Action)
{
     if (bHook) // Si les hooks sont actifs, on les supprime avant de quitter
          EndHooks();

     Action = caFree;
}

void __fastcall TForm1::PressionEntree(TWMNoParams &p)
```

```
{
     nbEntree++; // On a reçu un message du hook : on incrémente et on affiche le compteur
     Label1->Caption = AnsiString("Pressions de ENTREE : ") + nbEntree;
}
```

Une fois compilé, cet exemple permettra de compter les pressions et relâchements de la touche ENTREE, et jouera le son "par défaut" de Windows à chaque click de souris.