Подсекание Клавиатура Нажатие клавиши Общее программирование Применение
Дополнительно Промежуточный Дев Визуальная студия Windows C++

Сочетание Raw Input и keyboard Hook для выборочной блокировки ввода с нескольких клавиатур

Vít Blecha

4,96/5 (24 голосаза)

27 января 2014 г. CPOL 22 минут чтения 116301 5132

Как объединить API Raw Input и keyboard Hook и использовать их для выборочной блокировки ввода с некоторых клавиатур.

Загрузите HookingRawInputDemo.zip — 29,9 КБ

Содержание

Введение

Если вы подключите к компьютеру несколько клавиатур, у вас могут возникнуть интересные идеи, что с ними делать. Возможно, вы могли бы использовать одну клавиатуру для обычного набора текста, а другую — для каких-то специальных задач. Углубившись в изучение настройки нескольких клавиатур, вы можете задаться вопросом: «Можно ли заблокировать ввод с клавиатуры для запускаемого приложения в зависимости от того, с какой клавиатуры он был выполнен?» Допустим, у вас открыт блокнот. Когда вы нажимаете клавишу **a** на клавиатуре 1, вы хотите, чтобы в документ была записана буква **a**, но когда вы нажимаете клавишу **a** на клавиатуре 2, вы хотите, чтобы в фоновом режиме выполнялось какое-то действие, а блокнот даже не заметил нажатия клавиши. Когда я столкнулся с этой проблемой, мне было непросто найти ответ и решение. Поэтому я решил написать эту статью, чтобы вам было проще, если вы когда-нибудь

зададитесь тем же вопросом. В статье предполагается, что вы немного разбираетесь в программировании для Windows, но я постараюсь указать вам на соответствующие ресурсы, чтобы вы могли изучить всё необходимое.

Предыстория

Мне пришлось решать эту проблему, когда я захотел создать собственный менеджер горячих клавиш для нескольких клавиатур. Основная идея заключалась в том, чтобы использовать стандартную клавиатуру для стандартного ввода и одну небольшую цифровую клавиатуру для горячих клавиш. Горячие клавиши должны быть настроены для различных приложений, чтобы я мог использовать одну клавишу для выполнения действия в одном приложении и ту же клавишу для выполнения другого действия в другом приложении. Сначала я поискал существующие решения, но не смог найти ни одного, которое соответствовало бы всем моим требованиям. "AutoHotkey" не может самостоятельно различать несколько клавиатур. "Макросы HID" не позволяют настраивать горячие клавиши для конкретных приложений. Можно объединить эти два инструмента и использовать их одновременно, но мне это решение не очень понравилось. Я наткнулся на несколько менеджеров, которые, казалось, могли справиться с этой задачей, но все они были лицензионным программным обеспечением. Поэтому я решил написать свой собственный.

Проще говоря, задача состоит в том, чтобы решить, следует ли блокировать ввод с клавиатуры в зависимости от устройства, с которого поступил ввод, в управляющем приложении, а затем успешно заблокировать его в работающем приложении, на котором сосредоточено внимание пользователя. Если вы хотите работать с вводом с клавиатуры в Windows, у вас есть два основных варианта: использовать собственный драйвер для устройства или один из API-интерфейсов Windows для работы с вводом с клавиатуры. Следует рассмотреть два API-интерфейса: Raw Input и keyboard Hooks.

- **Пользовательский драйвер устройства**: до сих пор у меня не было опыта написания собственного драйвера для устройства. Я изучил документацию и примеры кода, и мне показалось, что это слишком сложно для моего проекта. Кроме того, это может привести к негативным последствиям. Для установки драйвера диспетчеру потребуются права администратора, что ограничит переносимость. А пользователям может быть некомфортно устанавливать такой драйвер. Возможно, драйвер также должен быть подписан уполномоченным лицом (я не совсем уверен, что это так, поскольку не изучал этот вопрос глубже).
  Другим вариантом может быть использование пользовательского драйвера, который предоставляет мне программный интерфейс, чтобы мне не пришлось писать собственный драйвер. «Библиотека перехвата», похоже, предлагает такую возможность. Но некоторые негативные аспекты пользовательского драйвера сохранятся (требование прав администратора, неудобство для пользователя), плюс появятся новые. Исходный код самой библиотеки доступен, а вот исходный код драйвера — нет, поэтому я не могу полностью контролировать продукт, который предоставляю потенциальным пользователям.
- **Необработанный ввод**: этот интерфейс обеспечивает более сложный способ ввода данных с различных устройств, включая клавиатуру, по сравнению со стандартными сообщениями о вводе с клавиатуры Windows. В частности, можно определить, на какой клавиатуре была нажата клавиша, что невозможно при использовании стандартных сообщений о вводе. Raw Input может отслеживать события на клавиатуре в масштабах всей системы, поэтому с его помощью можно решить первую часть задачи. К сожалению, вторую часть решить не получится. По крайней мере, мне не удалось найти способ заблокировать ввод данных в целевом окне с помощью Raw Input API.
- **Перехватчики событий клавиатуры**: Перехватчики событий Windows — это механизмы, которые позволяют приложению перехватывать системные сообщения, например события клавиатуры. При правильной настройке они работают во всей системе. Они даже позволяют приложению изменять или останавливать распространение соответствующего сообщения. Это значит, что перехватчики можно использовать для блокировки нажатия клавиш. Проблема в том, что невозможно определить, на какой клавиатуре была нажата клавиша.

Как видите, если только вы не хотите использовать какой-то специальный драйвер устройства, не существует единого API, который позволил бы вам решить проблему целиком. «Можно ли объединить два API, чтобы это сработало?» — спросите вы. И, как оказалось, да, можно. К сожалению, документация по взаимодействию практически отсутствует, и это очень нелогично. Я надеюсь, что эта статья прольёт свет на проблему для тех, кто с ней столкнулся.

Использование кода

Как я уже упоминал ранее, я предполагаю, что вы в некоторой степени разбираетесь в программировании для Windows. Я не хочу вдаваться в подробности настройки каждого отдельного API, потому что, на мой взгляд, об этом уже достаточно сказано. Если вы не знакомы ни с одним из этих API, я бы порекомендовал вам изучить документацию. Для системы сообщений Windows в целом: «О сообщениях и очередях сообщений» [1]; для Raw Input: «О Raw Input» [2]; для хуков: «Обзор хуков» [3].

В своём проекте я использую C++, но считаю, что код можно адаптировать для .NET, приложив некоторые усилия. Для простоты я не буду проверять наличие ошибок при системных вызовах, распределении памяти и т. д., если только они не являются частью самой проблемы. Демонстрационный проект использует простой вывод отладки только в демонстрационных целях и призван показать основную идею. Его определённо можно структурировать лучше. Кроме того, примеры кода содержат только самое важное. Здесь не будет готовых функций или модулей, которые можно просто скопировать и вставить (для этого вы можете скачать прикреплённый демонстрационный проект), но я всегда буду указывать функцию и файл, чтобы вы могли понять, к чему относится тот или иной код. Эти примеры должны помочь вам разобраться в проблеме.

На данный момент я тестировал код только в Windows 7 (64-разрядная версия). Возможно, в других системах поведение API будет отличаться.

Основная Концепция

Я узнал об этой концепции объединения Raw Input с Hooks благодаря комментарию **Петра Медека**, автора макросов HID [4], здесь, на CodeProject. Он подсказал мне основную идею этого решения. Так что я должен отдать должное тому, кто этого заслуживает; огромное спасибо Петру! Я построил свой проект на этой основной идее и преодолел несколько дополнительных препятствий. С разрешения Петра я опишу вам всю концепцию со всеми сложностями, с которыми я столкнулся.

Напомню, что наша цель — перехватывать нажатия клавиш, решать, блокировать их или передавать активному приложению, в зависимости от того, какая клавиатура была использована. Для принятия решения мы будем использовать Raw Input, а для блокировки ввода — Hook. Итак, приступим!

Базовая настройка

Чтобы использовать их, нам сначала нужно настроить оба API в нашем приложении. Для Raw Input нет никаких скрытых уловок, поэтому мы просто регистрируемся для получения данных с клавиатуры глобально (указывая флаг `RIDEV_INPUTSINK`).

C++

```
// --- InitInstance (HookingRawInputDemo.cpp) ---
// Register for receiving Raw Input for keyboards
RAWINPUTDEVICE rawInputDevice[1];
rawInputDevice[0].usUsagePage = 1;
rawInputDevice[0].usUsage = 6;
rawInputDevice[0].dwFlags = RIDEV_INPUTSINK;
rawInputDevice[0].hwndTarget = hWnd;
RegisterRawInputDevices (rawInputDevice, 1, sizeof (rawInputDevice[0]));
```

Что касается обработки `Raw Input messages`, давайте просто проверим, правильно ли мы их получаем. Мы будем отслеживать виртуальный код клавиши и то, нажата она или отпущена.

C++

```
// --- WndProc (HookingRawInputDemo.cpp) ---
// Raw Input Message
case WM_INPUT:
{
    // ...
    // Get the virtual key code of the key and report it
    USHORT virtualKeyCode = raw->data.keyboard.VKey;
    USHORT keyPressed = raw->data.keyboard.Flags & RI_KEY_BREAK ? 0 : 1;
    WCHAR text[128];
    swprintf_s (text, 128, L"Raw Input: %X (%d)\n", virtualKeyCode, keyPressed);
    OutputDebugString (text);
    // ...
}
```

Что касается хука, то здесь уже начинаются сложности. Когда я впервые экспериментировал с комбинацией API, я пытался использовать глобальный низкоуровневый хук клавиатуры (`WH_KEYBOARD_LL`). Проблема в том, что когда мы используем низкоуровневый хук клавиатуры для

блокировки ввода (мы останавливаем передачу сообщения), Windows не генерирует событие Raw Input, то есть ни одно приложение не получит соответствующий `Raw Input message` (`WM_INPUT`). Из-за этого мы не можем использовать низкоуровневый клавиатурный хук, но нам нужно использовать стандартный клавиатурный хук (`WH_KEYBOARD`), который немного сложнее настроить. Если мы хотим использовать этот хук глобально, то есть для любого запущенного приложения, его процедура должна находиться в отдельном модуле DLL. Если вы не знаете, как настроить хук в модуле DLL, вам поможет одна из этих статей: «Хуки и библиотеки DLL» [5], «Утилита для перехвата мыши и клавиатуры с помощью VC++» [6]. Хук следует регистрировать следующим образом:

C++

```
// --- InstallHook (HookingRawInputDemoDLL.cpp) ---
// Register keyboard Hook
hookHandle = SetWindowsHookEx (WH_KEYBOARD, (HOOKPROC)KeyboardProc, instanceHandle, 0);
```

Давайте пока не будем ничего делать с входными данными в процедуре Hook, а просто проверим их, как и в случае с необработанными входными данными, и передадим их по цепочке Hook.

C++

```
// --- KeyboardProc (HookingRawInputDemoDLL.cpp) ---
// Get the virtual key code of the key and report it
USHORT virtualKeyCode = (USHORT)wParam;
USHORT keyPressed = lParam & 0x80000000 ? 0 : 1;
WCHAR text[128];
swprintf_s (text, 128, L"Hook: %X (%d)\n", virtualKeyCode, keyPressed);
OutputDebugString (text);

return CallNextHookEx (hookHandle, code, wParam, lParam);
```

Теперь, когда вы запустите приложение, вы должны получать корректные уведомления как для Raw Input, так и для Hook, когда вы что-то вводите в другом окне.

Информационные материалы

Мы планируем использовать Hook для блокировки ввода, а Raw Input — для принятия решений. Но они находятся в разных модулях, поэтому для связи между ними мы будем использовать систему обмена сообщениями Windows. Связь будет довольно простой. Мы отправим сообщение из процедуры Hook в главное окно вместе с исходными `Hook message` параметрами. Главное окно решит, что делать с вводом. Если Hook должен заблокировать ввод, возвращаемое значение вызова сообщения будет равно 1; в противном случае — 0. Мы можем попробовать, не принимая пока никаких взвешенных решений.

C++

```
// --- KeyboardProc (HookingRawInputDemoDLL.cpp) ---
// Report the event to the main window. If the return value is 1, block the input; otherwise p
// along the Hook chain
if (SendMessage (hwndServer, WM_HOOK, wParam, lParam))
{
    return 1;
}

// --- WndProc (HookingRawInputDemo.cpp) ---
// Message from Hooking DLL
case WM_HOOK:
{
    return 0;
}
```

Обратите внимание, что важно использовать `SendMessage`, а не `PostMessage`, потому что мы хотим дождаться решения.

Принятие решений

Теперь, когда мы наладили связь, мы наконец-то можем что-то сделать. Давайте посмотрим, с чем мы имеем дело. Самая простая ситуация, когда нужно нажать всего несколько клавиш, должна выглядеть так (первое число — это шестнадцатеричный код виртуальной клавиши, а число в скобках указывает, нажата ли клавиша):

Обычный текст

```
Raw Input: 67 (1)
Hook: 67 (1)
Raw Input: 67 (0)
Hook: 67 (0)
Raw Input: 63 (1)
Hook: 63 (1)
Raw Input: 63 (0)
Hook: 63 (0)
```

При каждом нажатии клавиши сначала будет появляться `Raw Input message`, а затем его `Hook message`. Я могу заранее предупредить вас, что при быстром наборе или использовании специальных клавиш могут возникнуть сложности, но мы разберёмся с этим позже, а пока рассмотрим только этот простой случай. Поэтому всякий раз, когда мы получаем `Raw Input message`, мы можем решить, что делать с вводом (в зависимости от используемой клавиатуры), запомнить это решение и, когда мы получим `Hook message`, просто ответить в соответствии с принятым решением.

Давайте сначала улучшим обработку необработанного ввода (именно там принимаются решения), чтобы добавить распознавание клавиатуры. Наше решение будет довольно простым: если нажата клавиша «**7**» на цифровой клавиатуре, мы заблокируем ее. Все остальные нажатия клавиш будут просто пропускаться.

C++

```
// --- WndProc (HookingRawInputDemo.cpp) ---
// Raw Input Message
case WM_INPUT:
{
    // ...
    // Prepare string buffer for the device name
    GetRawInputDeviceInfo (raw->header.hDevice, RIDI_DEVICENAME, NULL, &bufferSize);
    WCHAR* stringBuffer = new WCHAR[bufferSize];

    // Load the device name into the buffer
    GetRawInputDeviceInfo (raw->header.hDevice, RIDI_DEVICENAME, stringBuffer, &bufferSize);

    // Check whether the key struck was a "7" on a numeric keyboard
    if (virtualKeyCode == 0x67 && wcscmp (stringBuffer, numericKeyboardDeviceName) == 0)
    {
        blockNextHook = TRUE;
    }
    else
    {
        blockNextHook = FALSE;
    }
    // ...
}
```

Осталось только использовать это решение при обработке событий Hook.

C++

```
// --- WndProc (HookingRawInputDemo.cpp) ---
// Message from Hooking DLL
case WM_HOOK:
{
    // ...
    // If next Hook message is supposed to be blocked, return 1
    if (blockNextHook)
    {
        swprintf_s (text, 128, L"Keyboard event: %X (%d) is being blocked!\n",
                    virtualKeyCode, keyPressed);
        OutputDebugString (text);
        return 1;
    }
    // ...
}
```

И вот оно! Наше приложение, которое блокирует ввод с клавиатуры в зависимости от того, какая клавиатура используется, работает. Основная идея комбинации Raw Input и keyboard Hook должна быть вам понятна.

Наиболее важные меры противодействия

К сожалению, в реальном мире всё немного сложнее. Во многих случаях мы не получим упорядоченную последовательность `Raw Input` и `Hook messages`, как предполагали ранее. И нам придётся иметь дело с такими ситуациями. Давайте рассмотрим эти сложности по очереди. Читая о различных ситуациях, вы можете задаться вопросом: «А что, если это...? А может ли это...?» И

вы, вероятно, правы: всё может оказаться гораздо сложнее, чем можно предположить, прочитав один раздел ниже. Я стараюсь излагать всё как можно проще, хотя из-за этого нам придётся пересмотреть некоторые решения. Я думаю, что это самый простой способ разобраться во всём. И я верю, что в конце концов вам всё станет ясно.

Буферизация необработанных входных сообщений

Когда вы начинаете печатать очень быстро (или просто стучите по клавиатуре пальцами), довольно часто бывает так, что вы получаете несколько `Raw Input messages` подряд, и только после этого появляется подходящее `Hook messages` . Например, так:

Обычный текст

```
Raw Input: 44 (1)
Hook: 44 (1)
Raw Input: 4B (1)
Raw Input: 53 (1)
Hook: 4B (1)
Hook: 53 (1)
```

Если бы мы остановились на простом решении, которое у нас есть на данный момент, наше приложение работало бы некорректно, поскольку оно блокировало бы ввод клавиш **k** (**4B**) и **s** (**53**) на основе одного решения, принятого для ввода клавиши **s** (**53**), поскольку мы помним только самое последнее принятое решение.

Поэтому нам нужно запомнить несколько решений и использовать их по порядку. Для этого мы воспользуемся контейнером FIFO (первым пришёл — первым вышел). Я решил использовать простой `Deque`. Когда мы получаем `Raw Input messages`, мы решаем, что делать с входными данными, и помещаем решение в `Deque`. Когда мы получаем `Hook message`, мы просто извлекаем решение.

C++

```
// --- WndProc (HookingRawInputDemo.cpp) ---
// Raw Input Message
case WM_INPUT:
{
    // ...
    // Check whether the key struck was a "7" on a numeric keyboard, and remember the decision
    // whether to block the input
    if (virtualKeyCode == 0x67 && wcscmp (stringBuffer, numericKeyboardDeviceName) == 0)
    {
        decisionBuffer.push_back (TRUE);
    }
    else
    {
        decisionBuffer.push_back (FALSE);
    }
    // ...
}
```

C++

```
// --- WndProc (HookingRawInputDemo.cpp) ---
// Message from Hooking DLL
case WM_HOOK:
{
    // ...
    // Check the buffer if this Hook message is supposed to be blocked; return 1 if it is
    BOOL blockThisHook = FALSE;
    if (!decisionBuffer.empty ())
    {
        blockThisHook = decisionBuffer.front ();
        decisionBuffer.pop_front ();
    }
    if (blockThisHook)
    {
        // ...
        return 1;
    }
    // ...
}
```

Ожидание сообщения Raw Input

До сих пор мы предполагали, что `Raw Input messages` всегда появляется раньше `Hook messages`. Обычно так и происходит, но бывают случаи, когда это не так и `Hook message` появляется раньше, например:

Обычный текст

```
Raw Input: 4A (0)
Hook: 4A (0)
Hook: 48 (0)
Raw Input: 48 (0)
```

Это означает, что мы хотим удалить решение из нашего буфера, когда в нём ничего не останется. Когда это происходит, нам нужно дождаться отложенного `Raw Input message`. После получения сообщения мы можем сразу решить, блокировать ввод или нет.

C++ Сжиматься ▲

```
// --- WndProc (HookingRawInputDemo.cpp) ---
// Message from Hooking DLL
case WM_HOOK:
{
    // ...
    // Check the buffer if this Hook message is supposed to be blocked; return 1 if it is
    BOOL blockThisHook = FALSE;
    if (!decisionBuffer.empty ())
    {
        // ...
    }
    // The decision buffer is empty
    else
    {
        MSG rawMessage;

        // Waiting for the next Raw Input message
        while (!PeekMessage (&rawMessage, mainHwnd, WM_INPUT, WM_INPUT, PM_REMOVE))
        {
        }

        // The Raw Input message has arrived; decide whether to block the input
        // ...
    }
    // ...
}
```

Сообщение о крючке отсутствует

Мы каким-то образом справились с перепутанной последовательностью сообщений, но вы, возможно, уже задаётесь вопросом: «Может ли какое-то сообщение вообще не прийти? И если да, то что произойдёт?» Действительно, такое возможно. Давайте посмотрим, что происходит, когда мы теряем некоторые `Hook message`, то есть получаем `Raw Input message` для какого-то события, но не получаем `Hook message` для него. Со мной такое происходит, например, когда я использую сочетание клавиш "**Ctrl**" + "**Esc**". Сообщения для этой последовательности выглядят так:

Обычный текст

```
Raw Input: 11 (1)
Hook: 11 (1)
Raw Input: 1B (0)
Raw Input: 11 (0)
```

Как видите, Windows не отправляет нам несколько сообщений. В частности, `Raw Input` и `Hook events` для нажатия клавиши **Esc** (**1B**), а `Hook events` для отпускания клавиш **Esc** (**1B**) и **Ctrl** (**11**). Это может вызвать у нас некоторые затруднения, если мы просто поместим решения для этих нажатий в буфер и будем использовать их неправильно. Эту проблему можно решить (частично) за счёт расширения нашего буфера решений. Мы будем запоминать не только само решение, но и виртуальный код клавиши, к которой оно относится (для более надёжного решения можно даже указать, нажата ли клавиша). Таким образом, когда приходит `Hook message` и мы ищем решение, что с ним делать, мы не просто извлекаем первое решение, а проверяем, совпадает ли его виртуальный код клавиши. Если нет, мы просматриваем весь буфер в поисках правильного решения. Я упомянул, что это решает проблему лишь «частично». Даже при такой проверке виртуального кода клавиши существует потенциальная опасность того, что какая-то кнопка будет нажата дважды, причём на разных клавиатурах. Если `Hook message` для первого нажатия клавиши не поступит, мы неправильно оценим второе нажатие (исходя из решения, принятого для первого). К сожалению, я не смог придумать никакого разумного решения этой проблемы. Если вы знаете, как это сделать, буду очень признателен, если вы поделитесь информацией с остальными в комментариях или отправите мне электронное письмо. Это определённо одна из проблем, на которые следует обратить внимание при использовании этой комбинации API.

C++

```
// --- (HookingRawInputDemo.h) ---
struct DecisionRecord
{
    USHORT virtualKeyCode;
    BOOL decision;

    DecisionRecord (USHORT _virtualKeyCode, BOOL _decision) : virtualKeyCode (_virtualKeyCode)
                    decision (_decision) {}
};
```

C++ Сжиматься ▲

```
// --- WndProc (HookingRawInputDemo.cpp) ---
// Message from Hooking DLL
case WM_HOOK:
{
    // ...
    // Check the buffer if this Hook message is supposed to be blocked; return 1 if it is
    BOOL blockThisHook = FALSE;
    BOOL recordFound = FALSE;
    UINT index = 1;
    if (!decisionBuffer.empty ())
    {
        // Search the buffer for the matching record
        std::deque::итератор итератор = decisionBuffer.begin ();  while (итератор != decisionB
```

Обратите внимание, что при удалении решения мы удаляем не только найденное подходящее решение, но и все решения, которые старше текущего. Если больше ничего не пошло не так, то это решения, у которых нет подходящих `Hook message`. Это полезно для того, чтобы буфер не разрастался без необходимости. Но, конечно, есть и другие способы держать его под контролем по вашему усмотрению.

Исходное входное сообщение отсутствует

Когда мы разбирались с ситуацией, когда `Hook message` стоит перед своим `Raw Input message`, вы, возможно, немного забеспокоились из-за потенциально бесконечного цикла while, который мы ввели. На самом деле вам стоит побеспокоиться. Мы уже заметили, что не все сообщения доходят до адресата. Мы рассмотрели, что может произойти, если `Hook message` потеряется. Теперь давайте посмотрим, что происходит, если `Raw Input message` не доставляется должным образом. Потому что в реальной жизни такое случается. Например, клавиша **AltGr** выдаёт мне такие сообщения (возможно, вы с этим не столкнётесь, потому что такое поведение зависит от языкового стандарта):

Обычный текст

```
Raw Input: 12 (1)
Hook: 11 (1)
Hook: 12 (1)
Raw Input: 12 (0)
Hook: 11 (0)
Hook: 12 (0)
```

В Windows комбинация `Hook messages` "**Ctrl**" (**11**) + "**Alt**" (**12**) заменяет одно нажатие "**AltGr**" ("**Ctrl**" + "**Alt**" часто используется вместо "**AltGr**"). Проблема в том, что он выдаёт `Raw Input message` только для самого "**AltGr**" (**12**) ("**AltGr**" в данном случае идентичен "**Alt**"; насколько мне известно, у них даже одинаковый скан-код). Использование нашего цикла ожидания Raw Input может привести к серьёзным проблемам, потому что он может никогда не получить `Raw Input message` который ожидает. Нам нужно ввести некоторые ограничения в цикл.

C++ Сжиматься ▲

```
// --- WndProc (HookingRawInputDemo.cpp) ---
// Message from Hooking DLL
case WM_HOOK:
{
    // ...
    // Wait for the matching Raw Input message if the decision buffer was empty or the matchin
    // record wasn't there
    DWORD currentTime, startTime;
    startTime = GetTickCount ();
    while (!recordFound)
    {
        MSG rawMessage;
        while (!PeekMessage (&rawMessage, mainHwnd, WM_INPUT, WM_INPUT, PM_REMOVE))
        {
            // Test for the maxWaitingTime
            currentTime = GetTickCount ();
```

```
            // If current time is less than start, the time rolled over to 0
            if ((currentTime < startTime ? ULONG_MAX - startTime + currentTime :
                 currentTime - startTime) > maxWaitingTime)
            {
                // Ignore the Hook message, if it exceeded the limit
                WCHAR text[128];
                swprintf_s (text, 128, L"Hook TIMED OUT: %X (%d)\n", virtualKeyCode,
                            keyPressed);
                OutputDebugString (text);
                return 0;
            }
        }
        // ...
    }
    // ...
}
```

Несмотря на то, что цикл теперь не может быть бесконечным, я считаю, что это (отсутствие или задержка `Raw Input message`) является одной из самых больших проблем всей концепции. Потому что любое ожидание `Raw Input message` приводит к задержке ввода с клавиатуры. Следовательно, лимит ожидания должен быть действительно низким, чтобы пользователи не заметили никаких задержек. Возможно, даже стоит подумать об использовании более тонкого таймера, я использовал самый простой для демонстрационных целей.

Еще больше странностей

Я думаю, что то, что у нас есть сейчас, — это разумная основа для приложения, сочетающего в себе Raw Input и перехват клавиатуры. Демонстрационный проект соответствует тому, что мы рассмотрели до сих пор. Теперь я хотел бы рассказать вам о некоторых особенностях поведения, с которыми вы можете столкнуться при работе с этими API. В основном это будет связано с одной из проблем, которые мы рассмотрели ранее, а именно с отсутствием `Raw Input messages`. Я считаю, что это заслуживает внимания, потому что, как я уже упоминал, отсутствие `Raw Input message` может привести к задержке ввода с клавиатуры, что крайне нежелательно. Я покажу вам все известные мне проблемные ситуации и предложу несколько способов избежать или хотя бы минимизировать опасность, которую они могут представлять. Но я не буду включать эти расширенные настройки в демонстрационный проект, потому что хотел соблюсти баланс между ясностью и полнотой и решил, что на этом остановлюсь. Я также считаю, что раз вы дочитали до этого места, то способны самостоятельно реализовать описанные подходы или даже придумать более эффективные, о которых мы с радостью прочитаем в комментариях. Но если вас интересует какой-либо из этих методов и вы не можете разобраться в нём самостоятельно, пожалуйста, не стесняйтесь задавать вопросы в комментариях о деталях или примерах кода, и я постараюсь вам помочь. Когда я закончу, я добавлю ссылку на своё приложение для управления ярлыками (с исходным кодом), в котором реализованы все эти дополнительные меры.

Проблема с AltGr

Давайте начнём с того, с чем мы уже сталкивались. Неудобство, когда Windows выдаёт «**Ctrl**" + «**Alt**" `Hook messages`», но на самом деле нажимается только «**Alt**" `Raw Input message`». Вы можете вернуться и посмотреть, какие сообщения мы получаем в этом случае.

Обычно "**Alt**" `Raw Input message` приходит в правильном виде, поэтому мы можем изучить `Raw Input message` и получить дополнительную информацию из его флагов. Эта информация заключается в том, была ли клавиша "**Alt**" правильной версией клавиши, то есть той, которая на самом деле может быть "**AltGr**". Если это так, то мы добавим в буфер решений не только решение для клавиши **Alt**, но и сначала ложное решение для клавиши **Ctrl**. Таким образом, при нажатии клавиши **Ctrl** `Hook message` в буфере будет найдена соответствующая запись, и ввод не будет задерживаться. Как я уже упоминал ранее, такое поведение характерно только для некоторых языковых стандартов, поэтому вы можете использовать этот обходной путь только для определённых языковых стандартов с помощью функции `GetKeyboardLayout` .

Бессмысленные сообщения- крючки

Если говорить о проблемах, связанных с локализацией, то в некоторых локалях может наблюдаться действительно странное поведение. Например, при нажатии одной клавиши **Win** (на чешской раскладке клавиатуры QWERTZ) может появиться такая последовательность сообщений:

Обычный текст

```
Raw Input: 5C (1)
Hook: 5C (1)
```

```
Raw Input: 5C (0)
Hook: 5C (0)
Hook TIMED OUT: 11 (0)
```

Возможно, вам интересно, что там делает этот "**Ctrl**" `Hook message` — по крайней мере, мне интересно. Тем более если вы изучите детали сообщения. Оказывается, для "**Ctrl**" `Hook message` установлены флаги, указывающие на то, что клавиша отпускается и что клавиша была нажата до события (согласно `Previous Key-State Flag`). Другими словами, клавиша "**Ctrl**" отпускается, когда она уже была отпущена. Это может происходить с различными клавишами (не только с клавишей **Win**) на клавиатуре, но, насколько мне известно, неправильным `Hook message` всегда является клавиша **Ctrl**, которая отпускается, когда уже нажата. Поскольку я никогда не сталкивался с ситуацией, когда эти флаги устанавливались при правильном нажатии клавиши **Ctrl** (я замечал их при обычном нажатии некоторых специальных клавиш), я полагаю, что вы можете отслеживать `Hook messages` на предмет этого шаблона и не ждать `Raw Input message`, если такое сообщение придёт.

Умноженные сообщения- крючки

Бывают ситуации, когда Windows генерирует несколько `Hook messages` для одного события клавиатуры и только один `Raw Input message`. Каждый умноженный `Hook message` будет пытаться дождаться своего `Raw Input message` (который так и не придёт), что приведёт к задержке. Это часто происходит, когда приложение переходит в меню (будь то строка меню или контекстное меню), или некоторые приложения могут вызывать такое поведение более непредсказуемым образом. Например, Firefox делает это, когда я печатаю слишком быстро. Последовательность сообщений может выглядеть следующим образом:

Обычный текст

```
Hook: 27 (1)
Raw Input WAITING: 27 (1)
Hook: 27 (1)
Raw Input WAITING: 27 (0)
Hook: 27 (0)
Hook TIMED OUT: 27 (0)
Hook: 27 (0)
Hook TIMED OUT: 27 (0)
```

Как видите, Windows добавляет два `Hook messages` к каждому `Raw Input message`, но это не правило: к каждому `Hook messages` может добавляться три или даже больше `Raw Input message` . В настоящее время я подхожу к решению этой проблемы следующим образом.

Суть в том, чтобы запомнить, какой `Hook message` был последним, полученным приложением. Когда приходит новый `Hook message`, мы можем проверить, идентично ли это сообщение предыдущему (совпадает ли код виртуальной клавиши и совпадает ли нажатие клавиши). Если да, то мы не будем ждать `Raw Input message`. Но нам всё равно нужно проверить буфер решений на наличие совпадающего `Raw Input message`. Потому что, например, при удержании какой-либо клавиши будет значимая последовательность одинаковых `Hook messages` с совпадающими `Raw Input messages`, и мы не должны их игнорировать.

При таком подходе следует помнить, что одно и то же решение (блокировать ли ввод с помощью хука) нужно применять к каждому повторяющемуся `Hook message`. Допустим, вы пропускаете первое событие хука и блокируете все последующие идентичные события, полагая, что первого события достаточно для обработки ввода в активном окне (например, нажатия клавиши в строке меню). Как оказалось, это не так. Приложение не выполнит нажатие клавиши, если вы не выполните следующие действия, а также первое из них.

Если вы хотите более избирательно подходить к использованию этого подхода, вы можете проверить, находится ли приложение в данный момент в режиме меню, с помощью функции `GetGUIThreadInfo` . Хотя я бы рекомендовал использовать её всегда, просто потому что есть приложения, которые могут вести себя подобным образом даже вне режима меню.

Клавиши для системных сочетаний клавиш

До сих пор мы использовали стандартный хук клавиатуры (`WH_KEYBOARD`), чтобы блокировать исходный ввод, когда нам это было нужно. У этой настройки есть одно ограничение. Некоторые клавиши обрабатываются системой независимо от того, блокирует ли их стандартный хук. Например, сочетания клавиш «**Win**» + «**d**», «**Alt**» + «**Tab**» или даже отдельные специальные клавиши, такие как кнопка «**Калькулятор**» (**B7** на моей клавиатуре). Даже если вы отключите распространение символа `Hook message`, при его нажатии Windows запустит Калькулятор.

Если вы хотите иметь возможность блокировать такие ключевые события, вам нужно будет использовать низкоуровневый хук клавиатуры (`WH_KEYBOARD_LL`). Когда мы всё настраивали, мы

поняли, что невозможно объединить низкоуровневый хук и Raw Input API, поэтому нет способа (я думаю, что нет, но если вы знаете, как это сделать, пожалуйста, сообщите мне в комментариях или по электронной почте), как блокировать эти специальные клавиши и при этом определять, какая клавиатура использовалась. Но, по крайней мере, их можно использовать параллельно, так что у вас может быть стандартный хук с необработанным вводом, обрабатывающий большинство событий, и низкоуровневый хук, обрабатывающий эти специальные клавиши, которые невозможно отличить по идентификации клавиатуры.

И последнее, на что я наткнулся, — это клавиша **Num Lock** (я не заметил её среди других клавиш блокировки). Даже если я заблокирую эту клавишу с помощью Low Level Hook, она всё равно будет влиять на состояние клавиатуры, как и обычное нажатие клавиши **Num Lock** (только светодиодный индикатор на клавиатуре не будет меняться). Эту проблему можно обойти, проверив состояние клавиатуры при отпускании клавиши **Num Lock** (в стандартном хуке клавиатуры) и при необходимости синтезировав нажатие клавиши **Num Lock**, которое вернет клавиатуру в предыдущее состояние.

Заключение

Надеюсь, эта статья помогла вам понять, как сочетать Raw Input и API для перехвата клавиатуры.

Если вкратце изложить мои мысли по поводу всего подхода, то я считаю его весьма полезным. Он способен решить проблему, даже несмотря на то, что есть несколько нерешённых вопросов. Поскольку Windows API не предлагает более простого способа решения проблемы, я думаю, что с этими вопросами можно справиться. Но если вам нужно реализовать на 100 % надёжное решение, вам, вероятно, придётся использовать собственный драйвер.

Если у вас есть какие-то мысли, вопросы или что-то ещё, что вы хотели бы добавить, пожалуйста, поделитесь этим в комментариях.

История

- 27 января 2014 года: опубликована первая версия статьи.

Ссылки

- [1] «О сообщениях и очередях сообщений». Microsoft. http://msdn.microsoft.com/en-us/library/windows/desktop/ms644927.aspx (по состоянию на 26 января 2014 года).
- [2] «О необработанном вводе». Microsoft. http://msdn.microsoft.com/en-us/library/windows/desktop/ms645543.aspx (по состоянию на 26 января 2014 года).
- [3] «Обзор хуков». Microsoft. http://msdn.microsoft.com/en-us/library/windows/desktop/ms644959.aspx (по состоянию на 26 января 2014 года).
- [4] Медек, Петр. «Макросы HID». http://www.hidmacros.eu/ (по состоянию на 26 января 2014 года).
- [5] Ньюкомер, Джозеф М. «Хуки и библиотеки DLL» http://www.codeproject.com/Articles/1037/Hooks-and-DLLs (по состоянию на 27 января 2014 года).
- [6] Родерик, Адам Дж. «Утилита для перехвата мышиных и клавиатурных событий с помощью VC++». http://www.codeproject.com/Articles/67091/Mouse-and-KeyBoard-Hooking-utility-with-VC (по состоянию на 27 января 2014 года).