Tutorial über Windows-Hooks

Inhaltsverzeichnis

1. Vorwort

Dieses Tutorial wird sich mit Windows-Hooks befassen.

Dabei wird erklärt was ein Hook ist, wie ein Hook funktioniert, wie er eingerichtet und wie er verwendet wird.

Anschließend wird eine Beispielanwendung erstellt, die einen Hook „setzt“ und diesen verwendet. Diese Beispielanwendung wird in C/C++ geschrieben und greift auf die Windows-API zu. Erstellt wird diese Anwendung mit der frei erhältlichen IDE Dev-C++ von Bloodshed Software.

Im Rahmen der Beispielanwendung wird auch kurz auf die Einrichtung und Bedienung von Dev-C++ eingegangen.

Die Anwendung wird als Hauptfunktion eine sog. Farbpipette haben, mit der man systemweit eine Farbe abgreifen kann. Diese Anwendung ist vor allem im Grafikbereich bzw. in der Webentwicklung sehr nützlich.

Da es unter Windows verschiedenartige Hooks gibt, werde ich hier nur die Funktionalität eines „Maus-Hooks“ beschreiben, da dieses Tutorial sonst zu umfangreich werden würde.

2. Was ist ein Hook ?

Ein Hook (zu deutsch: Haken) ist eine Einrichtung im Windows Betriebssystem, deren Aufgabe darin besteht Systemnachrichten abzufangen.

Im „Normalzustand“ kann eine Anwendung (bzw. nur ihr aktives Fenster) nur Nachrichten empfangen, welche auch für diese Anwendung (bzw. das aktive Fenster) bestimmt sind.

Am Beispiel der Maus würde das in folgender Reihenfolge funktionieren:

1. Benutzer betätigt die Maus (Bewegung, Klick oder Mausrad drehen)
2. Windows bekommt dies über die Hardware und das BIOS mit.
3. Windows überprüft, welche Anwendung, bzw. auf welches Fenster z.B. geklickt wurde.
4. Windows entscheidet an welche Anwendung bzw. welches Fenseter die Nachrichten (Mausaktivitäten) geschickt werden sollen.
5. Windows schickt die entsprechende Mausaktivität an die aktive Anwendung bzw. das aktive Fenster der aktiven Anwendung.

Ein Hook setzt nun am Punkt 3 an, um praktisch als Zwischensystem zu fungieren. Somit bekommt nun dieses Zwischensystem die Nachrichten der Mausaktivitäten zuerst und entscheidet darüber, was mit diesen passiert.

Das Zwischensystem hat somit die Möglichkeit über alle Mausaktivitäten zu verfügen, bevor sie an andere Anwendungen weitergeleitet werden.

Dabei gibt es 2 Arten von Hooks.

a) Lokale Hooks:
Hier werden die Nachrichten, nur einer Anwendung behandelt

b) Systemweite bzw. globale Hooks
Hier werden alle Nachrichten verarbeitet

Des weiteren kann man Hooks noch in verschiedene Bereiche gliedern:

Keyboard-, Mouse-, Shell-Hooks, usw...

In unserem Fall werden wir aber nur den Maushook behandeln.

Da diese Art von System nicht ganz ungefährlich ist, sollten sich mit Hooks, nur geübte Programmierer befassen, da laufende Anwendungen bzw. das Betriebssystem gestört werden kann.

3. Wozu werden Hooks benötigt?

Einige werden sich fragen: „Wenn das schon so gefährlich ist, wozu benötigt man einen Hook?“

Die Antwort darauf ist ganz einfach:

Immer dann, wenn wir in unserer Anwendung (bzw. in unserem aktiven Fenster) Nachrichten haben wollen, die eigentlich nicht für „uns“ bestimmt sind. In diesem Fall nehmen wir

unseren Spion (Hook) der sich dann *„festhookt“* äh fest hakt.

Um dies etwas verständlicher zu machen, gehe ich gleich auf die Beispielanwendung ein, die wir später erstellen werden.

Diese Anwendung möchte die Farbe eines Pixels haben, das irgendwo auf dem Bildschirm zu sehen ist, egal ob es im Fensterbereich der Anwendung liegt, oder nicht und egal ob die Anwendung aktiv ist (den sog. Fokus besitzt), oder nicht.

Wie die Farbe des jeweiligen Pixels „geholt“ wird, betrachten wir später noch eingehender.

Legen wir uns hier mal darauf fest, dass wir einfach ein Klickereignis mitbekommen wollen, wenn der Benutzer über irgendeinem Pixel auf dem Bildschirm mit der Maus klickt.

4. Wie funktioniert ein Hook genau?

In Kapitel 2 habe ich erklärt was ein Hook ist und in Kapitel 3 wofür wir einen Hook benötigen, nun werde ich tiefer ins Detail gehen und aufzeigen wie wir einen Hook „installieren“, diesen verwenden und wieder „deinstallieren“.

Ich werde dabei aber nicht auf sprachbezogene Teile bzw. Funktionen der Windows API eingehen, denn das wird dann der Teil sein, in dem wir unsere Anwendung erstellen.

4.1. Einrichten/Installieren eines systemweiten Hooks

Laut Microsoft und Windows, müssen globale Hooks in einem separatem Modul gesetzt werden, deshalb brauchen wir unabhängig von einer Anwendung ein separates Modul in Form einer DLL.

In dieser DLL wird nun der Hook installiert.

Diese DLL muss zusätzlich auch noch etwas besonderes aufweisen indem sie globale Variablen systemweit global zur Verfügung stellen.

Da die DLL später nicht nur von unserer Anwendung geladen wird sondern auch vom Betriebssystem, müssen wir einen Teil der DLL dazu bringen, dass er „geshared“ wird.

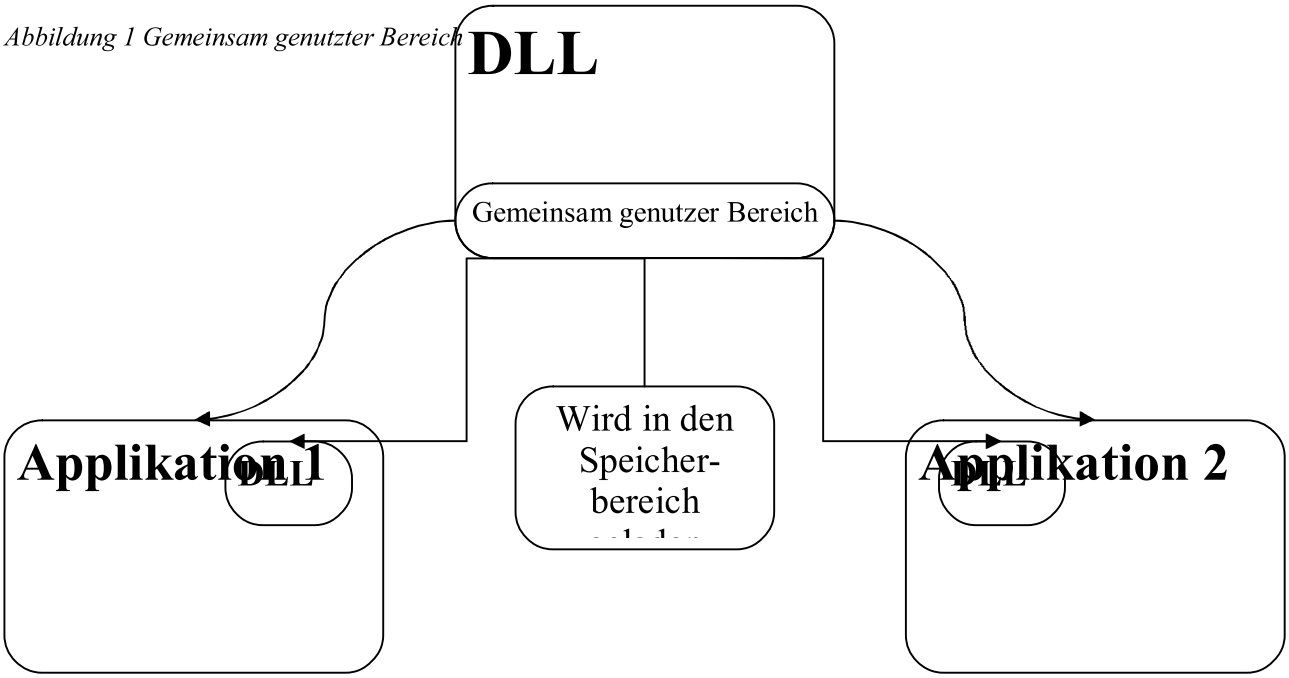
Denn im Normalfall läuft eine DLL unter Windows im Speicherraum der Applikation, die sie lädt.

Das heißt jede Applikation die unsere DLL laden würde, hätte eine eigene Instanz dieser, was zur Folge hätte, dass die Globalen Variablen, die in der DLL definiert werden nur innerhalb der DLL, bzw. Applikation global wären.

Was wir aber benötigen, ist zumindest einen Teil, der von mehreren Applikationen gleichzeitig benutzt werden kann.

Folgende Zeichnung verdeutlicht das:

Abbildung 1 Gemeinsam genutzter Bereich



Um dies zu erreichen gibt es mehrere Möglichkeiten, eine davon (wir werden diese Möglichkeit auch benutzen) ist das Einrichten eines „shared data segment“ innerhalb der DLL.

Das eigentliche Einrichten des Hooks, kann von einer Applikation aus durchgeführt werden. Die DLL wird nur für die Rückruffunktion (Callback-Funktion) des Hooks benötigt.

In unserer Beispielanwendung werden wir allerdings den Hook innerhalb der DLL installieren und deinstallieren.

4.2 Verwendung des Hooks

Die Verwendung des Hooks ist eigentlich ziemlich schnell erklärt.

Nachdem wir Windows „gesagt“ haben was wir wollen (einen systemweiten Hook, in diesem Fall ein Maus-Hook) und „gesagt“ haben welche unsere Rückruffunktion ist und vor allem wo diese sich befindet, ist der Hook auch schon aktiv.

Jedesmal wenn eine Mausaktivität erfolgt, wird vom System unsere Rückruffunktion aufgerufen, in der wir dann die Aktivität auswerten und weiterverarbeiten können.

4.3 Deinstallieren des Hooks

Nachdem wir den Hook nicht mehr benötigen sollten wir diesen unbedingt wieder deinstallieren, da systemweite Hooks teilweise einiges an Rechenzeit kosten.

Der Hook wir durch einen einfachen Funktionsaufruf deinstalliert, mehr dazu in der Beispielanwendung.

5. Die Beispielanwendung (Pipette)

Wir erstellen uns eine Beispielanwendung welche uns aufzeigt, wie man unter Windows mit einem systemweiten Hook umgeht.

Dazu benötigen wir zunächst mal Dev-C++ als IDE. (Wer möchte kann natürlich auch Visual C++ oder andere IDEs für C++ unter Windows verwenden). Es werden keine speziellen Bibliotheken verwendet.

5.1 Dev-C++ Einrichten

Zunächst laden wir uns die aktuelle Version von Dev-C++[1] herunter, diese Software unterliegt der GNU General Public License[2] und ist somit frei verwendbar. Für dieses Tutorial wurde die Version 4.9.8.9 verwendet.

Dev-C++ ist nur eine Entwicklungsumgebung (ohne Compiler/Linker), verwendet aber den ebenfalls freien GNU-GCC[3] Kompiler.

Aber keine Angst, Dev-C++ kommt mit einer komfortablen Installation die den GCC Compiler bereits enthält, desweiteren befindet sich auch der ebenfalls freie Debugger GNU-GDB[4] in der Installation.

Bevor wir anfangen, benötigen wir **unbedingt** ein Update, der in Dev-C++ enthaltenen ***MinGW[5]-binutils***. Da in der enthaltenen Version ein Fehler ist, der sich leider auf unser Projekt auswirkt. Wenn wir uns zurückerinnern, benötigen wir ein „shared data segment“ und das Einrichten dieses ist kompilerspezifisch und genau an dieser Stelle befindet sich das Problem. Für weitere Informationen verweise ich, dazu auf die Seite von MinGW. Das Update der „binutils“ kann man auch bei MinGW -Download[6], bzw. bei SourceForge[7] erhalten. Ich habe mir folgende Version heruntergeladen:

binutils-2.13.90-20030111-1.tar.gz

das heruntergeladene Paket entpacken wir einfach in unser Installationsverzeichnis von Dev-C++.

Nachdem wir Dev-C++ installiert haben, können wir die IDE zum ersten mal starten, da durch die Installation alles benötigte eingerichtet wird, können wir auch gleich Anfangen.

5.2 Das DLL-Projekt

Wir fangen mit der DLL an und erstellen uns hierzu ein Verzeichnis auf der Festplatte:

„C:\HookPipette\“

In dieses Verzeichnis kommen sämtliche Quelldateien und auch Projektdateien rein.

5.2.1 Das Dev-C++ Projekt der DLL

Wir starten Dev-C++ und erstellen ein neues DLL-Projekt über „Datei->Neu->Projekt“. Hier wählen wir DLL und **C-Projekt**, da wir in unserem Fall „nur“ eine C-DLL erstellen wollen. Nun geben wir unserem Projekt noch einen Namen (ich habe hier **„mousehook“** gewählt).

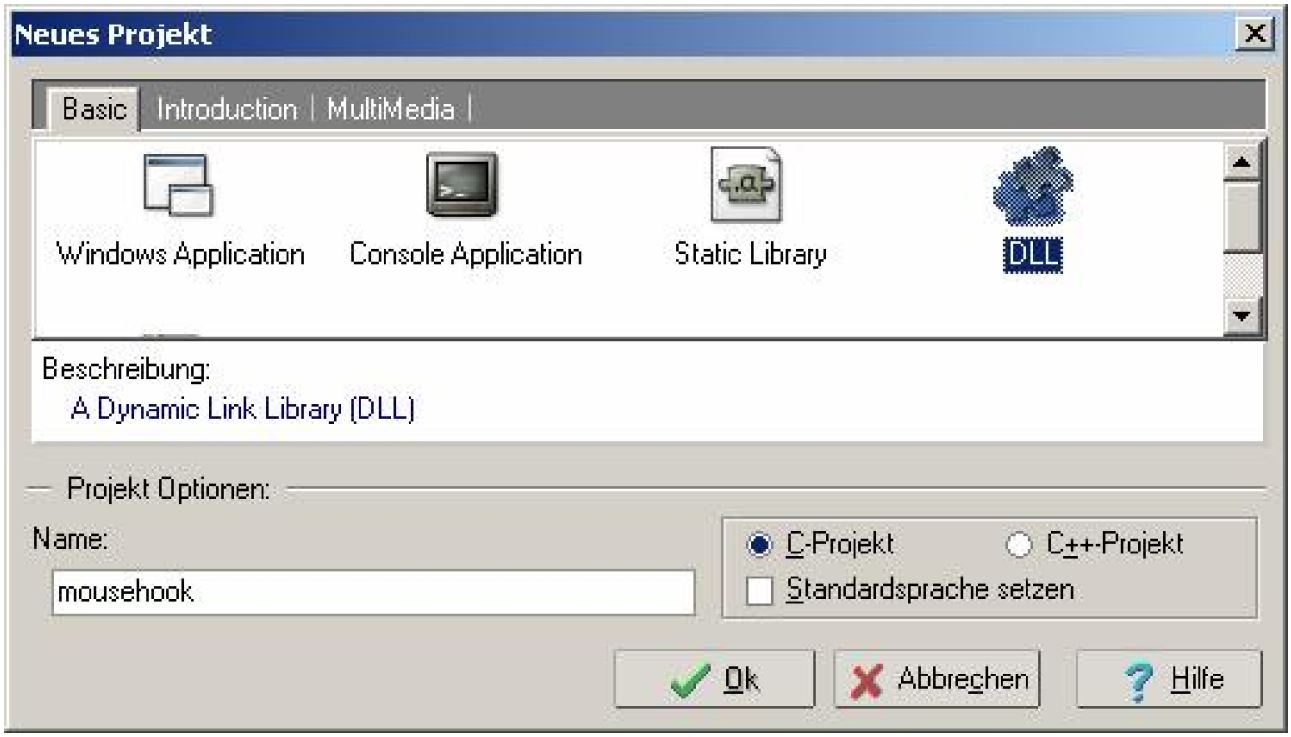


Abbildung 2 Dll Projekt Assistent

Nach einem Klick auf Ok werden wir nach einem Speicherort für unser Projekt gefragt, hier geben wir das zuvor erstellte Verzeichnis an. Dort befindet sich nun eine .dev Datei, die das Dev-C++ Projekt darstellt.

Innerhalb des offenen Projektes sehen wir 2 Dateien die geöffnet sind:

- dllmain.c
- dll.h

Um nun alles erst einmal zu speichern, klicken wir auf „Datei->Alles Speichern“ und bestätigen die Vorgaben.

Jetzt drücken wir STRG-F9 um unser DLL-Projekt zu kompilieren und zu linken. Wenn der Vorgang erfolgreich war, dann haben wir in unserem Verzeichnis, ausser ein paar zusätzlichen Dateien auch eine .dll liegen (Namen je nach Projektbezeichnung). Ist dies nicht der Fall, bzw. war das Erstellen nicht erfolgreich, dann liegt das an der Installation von Dev-C++.

5.2.2 Der erstellte Code

Nun widmen wir uns dem Code.

Zunächst betrachten wir, was der Projektassistent in die beiden o.g. Dateien geschrieben hat:

```
#ifndef _DLL_H_
#define _DLL_H_

#if BUILDING_DLL
# define DLLIMPORT __declspec (dllexport)
#else /* Not BUILDING_DLL */
# define DLLIMPORT __declspec (dllimport)
#endif /* Not BUILDING_DLL */


DLLIMPORT void HelloWorld (void);
```

Listing 1 Originale dll.h

In der Header-Datei des DLL-Projektes stehen diverse Defines und eine Funktion HelloWorld()

```
/* Replace "dll.h" with the name of your header */
#include "dll.h"
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>

DLLIMPORT void HelloWorld ()
{
    MessageBox (0, "Hello World from DLL!\n", "Hi", MB_ICONINFORMATION);
}


BOOL APIENTRY DllMain (HINSTANCE hInst     /* Library instance handle. */ ,
                       DWORD reason        /* Reason this function is being called. */ ,
                       LPVOID reserved     /* Not used. */ )
{
    switch (reason)
    {
      case DLL_PROCESS_ATTACH:
        break;

      case DLL_PROCESS_DETACH:
        break;

      case DLL_THREAD_ATTACH:
        break;

      case DLL_THREAD_DETACH:
        break;
    }

    /* Returns TRUE on success, FALSE on failure */
```

Listing 2 Originale dllmain.c

In der dllmain.c befinden sich die benötigten Includes, die Implementation der Funktion HelloWorld() und die DLLMain (der Einsprungspunkt der DLL selbst).

Das heißt wenn wir diese DLL so wie sie jetzt ist verwenden, dann könnten wir mit ihr eine MessageBox mit „Hello World from DLL!“ erzeugen, aber das wollen wir ja nicht, denn wir möchten ja einen Maus-Hook einrichten und verwenden.

5.2.3 Modifikationen im Code

Also schreiben wir den entsprechenden Code für unsere DLL.

Die DLL wird im Gesamten „nur“ vier Funktionen enthalten:

- DllMain() // Einsprungsfunktion in die DLL
- InstallHook() // exportierte Funktion zum Einrichten des Hooks
- UninstallHook() // exportierte Funktion zum Entfernen des Hooks
- MouseProc() // Callbackfunktion (Rückruffunktion) beim Auslösen einer Mausaktivität

Nicht benötigten Code, der vom Dev-C++ Assistenten eingefügt worden ist, können wir löschen.

```
#ifndef _DLL_H_
#define _DLL_H_

#include <windows.h>

#if BUILDING_DLL
# define DLLIMPORT __declspec (dllexport)
#else /* Not BUILDING_DLL */
# define DLLIMPORT __declspec (dllimport)
#endif /* Not BUILDING_DLL */

// zur Deklaration von Variablen innerhalb eines "shared data segment"
#define SHARED __attribute__((section(".shr"), shared))

DLLIMPORT BOOL InstallHook();
DLLIMPORT BOOL UninstallHook();

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

Listing 3 Geänderte dll.h

Zuerst inkludieren wir die benötigten Headerdateien.

Wir deklarieren uns zunächst mal ein Macro mit dem Namen „SHARED“ damit „markieren“ wir später Variablen, die wir innerhalb eines „shared data segment“ einrichten wollen. Der darauffolgende Code in dieser Zeile ist compilerspezifisch und richtet sich an den GCC-Compiler aus der MinGW-Umgebung. Die Erklärung des `__attribute__` Schlüsselwortes kann man der Dokumentation des GCC-Compilers[8] entnehmen.

Anm:
Bei Visual C++ 6.0 würde man die Variablen für „shared data segment“ folgendermaßen deklarieren:

```
#pragma data_seg ("shared")
int i_MyGlobalSharedVariable = 0;
#pragma data_seg ()
#pragma comment(linker,"/SECTION:shared,RWS")
```

Danach deklarieren wir uns die beiden Funktionen, die exportiert werden. `InstallHook()` und `UninstallHook()` beide haben keine Übergabeparamter und haben als Rückgabewert `BOOL`. `DLLIMPORT` bedeutet in unserem Falle die sog. „Calling convention“. Zum Schluss benötigen wir noch unsere Callbackfunktion (`MouseProc()`).

Als nächstes schauen wir uns den Code der dllmain.c an, diesen Code werde ich stückchenweise aufzeigen und erklären:

```
/* Replace "dll.h" with the name of your header */
#include "dll.h"


// Globale Variablen
HHOOK g_hMouseHook SHARED = NULL;    // Handle unseres Hooks (als "shared" Deklariert)
HINSTANCE g_hInst SHARED = NULL;     // Handle der DLL selbst

BOOL APIENTRY DllMain (HINSTANCE hInst     /* Library instance handle. */ ,
                       DWORD reason        /* Reason this function is being called. */ ,
                       LPVOID reserved     /* Not used. */ )
{
    g_hInst = hInst;
    return TRUE;
}
```

Listing 4 Hauptteil von dllmain.c

Wir deklarieren uns hier 2 globale Variablen:

- `g_hMousHook` das ist ein globales Handle auf unseren Hook, das zusätzlich als „shared“ deklariert wird
- `g_hInst` das ist ein globales Handle auf die DLL-Instanz, auch als „shared“ deklariert

Außerdem nehmen wir den nicht benötigten `switch` aus der DllMain raus und weisen unserem globalen Instanz-Handle, die aktuelle DLL-Instanz zu.

```
DLLIMPORT BOOL InstallHook()
{
    if(g_hMouseHook != NULL)
                return TRUE;

        g_hMouseHook = SetWindowsHookEx(WH_MOUSE, MouseProc, g_hInst, 0);
        if(g_hMouseHook == NULL)
            return FALSE;

    return TRUE;
}

DLLIMPORT BOOL UninstallHook()
{
        if(g_hMouseHook != NULL)
        {
                UnhookWindowsHookEx(g_hMouseHook);
                g_hMouseHook = NULL;
        }
        return TRUE;
```

Listing 5 InstallHook und UninstallHook von dllmain.c

Hier sehen wir die beiden Implementationen von `InstallHook()` und `UninstallHook()`.

Beim Einsprung in `InstallHook()` prüfen wir ab, ob unser globales Hook-Handle gültig ist (`!= NULL`), ist dies der Fall, dann geben wir `TRUE` zurück, denn dann verwenden wir das bestehende Handle, ansonsten rufen wir die Windows-API-Funktion `SetWindowsHookEx()` auf und übergeben ihr folgende vier Parameter:

- `WH_MOUSE` -> Konstante, für einen Mousehook
- `MouseProc` -> ein Zeiger auf unsere Callbackfunktion
- `g_hInst` -> unser globales Handle der DLL
- `0` -> ist die ThreadId des zugeordneten Threads, wir (wollen einen systemweiten Hook, darum 0)

Die genauere Beschreibung der Funktion und die genaue Bedeutung ihrer Parameter kann man in

der MSDN[9] nachlesen. Die Rückgabe der Funktion `SetWindowsHookEx()` ist bei Erfolg ein gültiges Handle auf einen Hook, sonst erhalten wir `NULL`, in diesem Fall geben wir `FALSE` zurück, um der Applikation (später unser Pipettenprogramm) mitzuteilen, dass InstallHook() fehlgeschlagen ist.

Beim Einsprung in `UninstallHook()` prüfen wir ab, ob unser globales Hook-Handle gültig ist (`!= NULL`), ist dies der Fall, dann rufen wir `UnhookWindowsHookEx()` auf und übergeben das Handle des Hooks, dann geben wir noch `TRUE` zurück.

```
LRESULT CALLBACK MouseProc(int nCode, WPARAM wParam, LPARAM lParam)
{
        if (nCode < 0)
                return CallNextHookEx(g_hMouseHook, nCode, wParam, lParam);

        if(nCode == HC_ACTION)
        {
                if ( (wParam == WM_LBUTTONDOWN)||(wParam == WM_NCLBUTTONDOWN) )
                {
                     MOUSEHOOKSTRUCT *mhs = (MOUSEHOOKSTRUCT*)lParam;
                     HWND caller = FindWindow("Pipette", NULL);
                     if(caller != NULL)
                         PostMessage(caller, WM_USER+123, 0, MAKELPARAM(mhs->pt.x, mhs->pt.y));
                }
        }
        return CallNextHookEx(g_hMouseHook, nCode, wParam, lParam);
```

Listing 6 Die Callbackfunktion von dllmain.c

Hier werden wir unsere Mausaktivitäten verarbeiten, d.h. unsere Funktion `MouseProc()` wird vom Betriebssystem aufgerufen, sobald die Maus bewegt wird, bzw. geklickt wird.

Nach dem Einsprung in unsere `MouseProc()` fragen wir zunächst den Wert von `nCode` ab, er enthält die Art der Aktivität, ist der Wert `< 0`, dann müssen wir laut MSDN die Nachrichten mit `CallNextHookEx()` „weiterschicken“.

Ansonsten können wir die Mausaktivität verarbeiten. Wir fragen ab, ob `nCode` gleich `HC_ACTION` entspricht was bedeutet, dass `wParam` und `lParam` Informationen über die Mausaktivität enthält. Dann prüfen wir ab, ob die linke Maustaste entweder in einer Client-Area (`WM_LBUTTONDOWN`) oder in einer Nicht-Client-Area (`WM_NCLBUTTONDOWN`) gedrückt wurde. Ist dies der Fall, dann zeigt `lParam` auf die Daten einer `MOUSEHOOKSTRUCT`-Struktur. Also deklarieren wir uns einen lokalen Zeiger dieser `MOUSEHOOKSTRUCT` (`*mhs`) und weisen ihm die Adresse von `lParam` über eine Typenkonvertierung (type cast) zu. Anschließend „suchen wir unsere Anwendung“, wir versuchen mit `FindWindow()` unser Pipettenprogramm zu finden, indem wir dieser Funktion den registrierten Klassennamen übergeben („`Pipette`“), als Rückgabe erhalten wir ein Fensterhandle (`HWND caller`). Wenn caller gültig ist (`!= NULL`) dann können wir an dieses Fenster eine Nachricht schicken. Wir schicken eine Nachricht vom Typ `WM_USER+123` (wir addieren `123`, als frei gewählten Wert zu der Konstante `WM_USER` dazu) an unsere Anwendung und übergeben in dieser Nachricht die X und Y Position der Mauskoordinaten. Zum Schluss rufen wir **immer** `CallNextHookEx()` auf, sonst würde die Mausaktivität von der dazugehörigen Anwendung nicht mehr verarbeitet werden können, was fatale Folgen hätte.

Weitere Informationen zu der Maushook Callbackfunktion `MousProc` finden wir wie immer in der MSDN[10].

Die Möglichkeiten der Kommunikation zwischen unserer DLL und der Pipettenhautpanwendung sind vielseitig, ich habe hier nur einen Weg gewählt:

- Fenster suchen
- Nachricht an das Fenster schicken

Weitere Möglichkeiten wären das über eine andere Art von Kommunikation zu lösen, z.B. Named-Pipes oder Shared Memory.

5.2.4 Die fertige DLL

Der Code unserer DLL wäre jetzt fertig und wir können die DLL mit CTRL+F9 erstellen. Es sollte alles ohne Fehler compiliert werden und dann in unserem Verzeichnis (C:\HookPipette\) sollte sich die fertige „mousehook.dll“ befinden.

Nun geht es daran ein Programm zu erstellen, dass auch die Funktionalität unserer DLL benutzen kann.

5.3 Das Pipettenprogramm

Das Pipettenprogramm soll kein hochmodernes und innovatives Softwareprojekt darstellen, vielmehr soll es ausschließlich die Verwendung der vorher erstellten Hook-DLL aufzeigen. Aus diesem Grund wird die Oberflächenfunktionalität auf das Nötigste begrenzt sein, außerdem wird im Rahmen dieses Tutorials auch nur ansatzweise auf die Oberflächenelemente eingegangen.

5.3.1 Das Dev-C++ Projekt des Pipettenprogramms

Wir starten eine neue Instanz von Dev-C++ und erstellen ein neues Projekt über „Datei->Neu->Projekt“. Hier wählen wir Windows Application und **C++-Projekt.** Nun geben wir unserem Projekt noch einen Namen (ich habe hier **„Pipette“** gewählt).

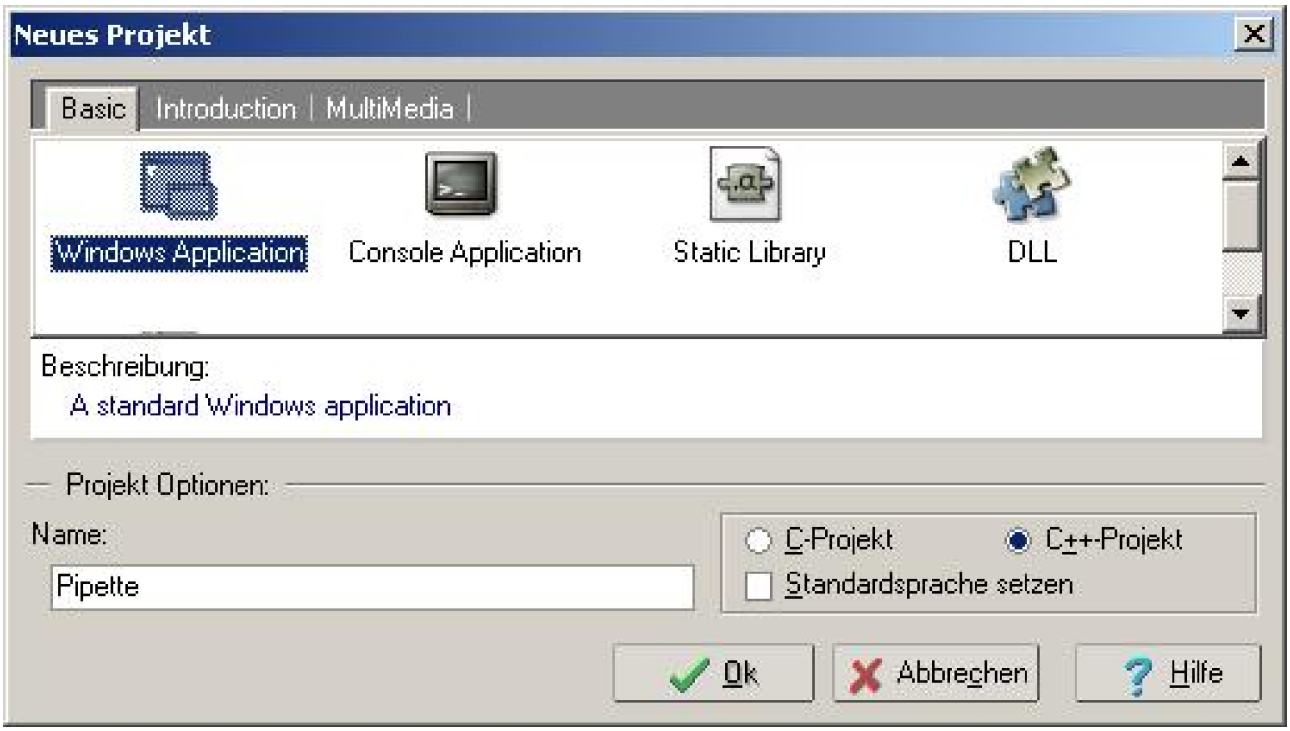


Abbildung 3 Pipettenprojekt

Nach einem Klick auf Ok werden wir nach einem Speicherort für unser Projekt gefragt, hier geben wir wieder unser Arbeitsverzeichnis (C:\HookPipette) an. Dort befindet sich nun eine .dev Datei, die das Dev-C++ Projekt darstellt.

Innerhalb des offenen Projektes sehen wir 1 Datei die geöffnet ist:

- main.cpp

Um nun alles erst einmal zu speichern, klicken wir auf „Datei->Alles Speichern“ und bestätigen die Vorgaben.

Jetzt drücken wir STRG-F9 um unser Pipetten-Projekt zu kompilieren und zu linken. Wenn der Vorgang erfolgreich war, dann haben wir in unserem Verzeichnis ausser ein paar zusätzlichen Dateien auch eine .exe liegen (Namen je nach Projektbezeichnung). Ist dies nicht der Fall, bzw. war das Erstellen nicht erfolgreich, dann liegt das an der Installation von Dev-C++.

Nun starten wir das Programm zunächst mal, damit wir sehen, was wir bis jetzt haben, das Fenster sollte in etwa so aussehen:

Abbildung 4Pinetterprojekt 1. Start

Nicht sehr spektakulär oder? Naja immerhin haben wir ein Fenster, das ist doch schon mal was. Um das Weitere werden wir uns jetzt kümmern.

5.3.2 Der erstellte Code

Wir betrachten jetzt zunächst einmal was uns der Projektassistent erstellt hat.

```
#include <windows.h>

/*  Declare Windows procedure  */
LRESULT CALLBACK WindowProcedure (HWND, UINT, WPARAM, LPARAM);

/*  Make the class name into a global variable  */
char szClassName[ ] = "WindowsApp";

int WINAPI WinMain (HINSTANCE hThisInstance,
                    HINSTANCE hPrevInstance,
                    LPSTR lpszArgument,
                    int nFunsterStil)

{
    HWND hwnd;               /* This is the handle for our window */
    MSG messages;            /* Here messages to the application are saved */
    WNDCLASSEX wincl;        /* Data structure for the windowclass */

    /* The Window structure */
    wincl.hInstance = hThisInstance;
    wincl.lpszClassName = szClassName;
    wincl.lpfnWndProc = WindowProcedure;      /* This function is called by windows */
    wincl.style = CS DBLCLKS;                 /* Catch double-clicks */
    wincl.cbSize = sizeof (WNDCLASSEX);

    /* Use default icon and mouse-pointer */
```

```
    wincl.hIcon = LoadIcon (NULL, IDI_APPLICATION);
    wincl.hIconSm = LoadIcon (NULL, IDI_APPLICATION);
    wincl.hCursor = LoadCursor (NULL, IDC_ARROW);
    wincl.lpszMenuName = NULL;                 /* No menu */
    wincl.cbClsExtra = 0;                      /* No extra bytes after the window class */
    wincl.cbWndExtra = 0;                      /* structure or the window instance */
    /* Use Windows's default color as the background of the window */
    wincl.hbrBackground = (HBRUSH) COLOR_BACKGROUND;

    /* Register the window class, and if it fails quit the program */
    if (!RegisterClassEx (&wincl))
        return 0;

    /* The class is registered, let's create the program*/
    hwnd = CreateWindowEx (
           0,                   /* Extended possibilites for variation */
           szClassName,         /* Classname */
           "Windows App",       /* Title Text */
           WS_OVERLAPPEDWINDOW, /* default window */
           CW_USEDEFAULT,       /* Windows decides the position */
           CW_USEDEFAULT,       /* where the window ends up on the screen */
           544,                 /* The programs width */
           375,                 /* and height in pixels */
           HWND_DESKTOP,        /* The window is a child-window to desktop */
           NULL,                /* No menu */
           hThisInstance,       /* Program Instance handler */
           NULL                 /* No Window Creation data */
           );

    /* Make the window visible on the screen */
    ShowWindow (hwnd, nFunsterStil);

    /* Run the message loop. It will run until GetMessage() returns 0 */
    while (GetMessage (&messages, NULL, 0, 0))
    {
        /* Translate virtual-key messages into character messages */
        TranslateMessage(&messages);
        /* Send message to WindowProcedure */
        DispatchMessage(&messages);
    }

    /* The program return-value is 0 - The value that PostQuitMessage() gave */
    return messages.wParam;
}


/*  This function is called by the Windows function DispatchMessage()  */

LRESULT CALLBACK WindowProcedure (HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    switch (message)                  /* handle the messages */
    {
        case WM_DESTROY:
            PostQuitMessage (0);       /* send a WM_QUIT to the message queue */
            break;
        default:                      /* for messages that we don't deal with */
            return DefWindowProc (hwnd, message, wParam, lParam);
    }

    return 0;
}
```

Listing 7 Die originale main.cpp

Wenn wir den Quellcode überfliegen sehen wir, dass hier ein Fenster erstellt wird `CreateWindowEx` und in der Callbackfunktion `WindowProcedure` ausser der `WM_DESTROY` Nachricht bisher noch keine Nachricht abgefangen wird.

5.3.3 Modifikationen im Code

Nun machen wir uns daran unser Pipettenprogramm zu schreiben.

Zunächst benötigen wir eine eigene Headerdatei, die wir später dann von unserer main.cpp inkludieren. Dazu gehen wir auf „DATEI->Neu->Quelldatei“ und beantworten die Frage, ob wir die neue Datei zum Projekt hinzufügen wollen mit „Ja“. Nun haben wir eine neue Datei mit dem Namen „Unbenannt1“ geöffnet und in unser Projekt eingefügt. Nun klicken wir wieder auf „Datei->Alles Speichern“, um diese Datei, unter einem sinnigen Namen in unserem Pojektordner abzuspeichern. Wir benennen die Datei „main.h“.

Anschließend schreiben wir den Code für die Headerdatei:

```
#include <windows.h>
#include <stdio.h>

/* DEFINES */

#define WINDOW_WIDTH 310
#define WINDOW_HEIGHT 200

#ifndef GET_X_LPARAM
#define GET_X_LPARAM(lp)    ((int)(short)LOWORD(lp))
#endif
#ifndef GET_Y_LPARAM
#define GET_Y_LPARAM(lp)    ((int)(short)HIWORD(lp))
#endif

typedef bool (CALLBACK *MYFUNC)(void);

/* Prototypen */

void Init();
void DrawColor(POINT &p, HWND &mainWindow);
void DrawDialog(HWND &parent, HINSTANCE &inst);
void ToHTML();
void ToRGB();
bool IsPointInApp(POINT &p, HWND &mainWindow);
LRESULT CALLBACK WindowProcedure (HWND, UINT, WPARAM, LPARAM);


/* Globale Variablen */

MYFUNC InstallHook;
MYFUNC UninstallHook;

HWND hInstallHookBtn;
HWND hUninstallHookBtn;
HWND hEditRGB;
HWND hEditHTML;
HWND hStaticFrame;

COLORREF col;
RECT ColorRect;

char szClassName[ ] = "Pipette";
```

Listing 8 Die neue main.h

Zunächst müssen wir die `windows.h` und die `stdio.h` inkludieren. Dann deffinieren wir uns 2 Konstanten für die Fensterbreite und Fensterhöhe (`WINDOW_WIDTH` und `WINDOWS_HEIGHT`).

Anschließend benötigen wir noch 2 Makros, die uns später unsere Mauskoordinaten aus der Nachricht, die von unserer DLL an die Pipettenanwendung gesendet wird herausholen.

Danach benötigen wir noch eine Typendeffinition (`typdef`) die unseren Funktionszeiger für unsere beiden DLL-Funktionen deffiniert: `typedef bool (CALLBACK *MYFUNC)(void);`

Nun erstellen wir uns einige Prototypen für Funktionen die wir benötigen:

`void Init();`
Diese Funktion benötigen wir um einige Initialisierungen durchzuführen.

`void DrawColor(POINT &p, HWND &mainWindow);`
Hier werden wir dann die Mauskoordinaten verarbeiten.

`void DrawDialog(HWND &parent, HINSTANCE &inst);`
Hier werden unsere Steuerelemente eingefügt.

`void ToHTML();`
Diese Funktion benutzen wir um den HTML-Code der Farbe in unser Fenster reinzuschreiben

`void ToRGB();`
Diese Funktion benutzen wir um den RGB-Code der Farbe in unser Fenster reinzuschreiben

`bool IsPointInApp(POINT &p, HWND &mainWindow);`
Hier überprüfen wir, ob unsere Mauskoordinaten innerhalb der eigenen Anwendung liegen.

`LRESULT CALLBACK WindowProcedure (HWND, UINT, WPARAM, LPARAM);`
Das wird unsere Callbackfunktion, die die Nachrichten von Windows verarbeiten soll.

Nun brauchen wir einige globale Variablen:

```
MYFUNC InstallHook;
MYFUNC UninstallHook;
```
Das werden unsere Funktionszeiger der beiden DLL-Funktionen sein.

```
HWND hInstallHookBtn;
HWND hUninstallHookBtn;
HWND hEditRGB;
HWND hEditHTML;
HWND hStaticFrame;
```
Diese Fensterhandles benötigen wir für die Steuerelemente.

`COLORREF col;`
Eine Farbe.

`RECT ColorRect;`
Das Rechteck in dem wir dann die gefundene Farbe darstellen.

`char szClassName[ ] = "Pipette";`
Das ist der Klassennamen unserer Anwendung.

Nun widmen wir uns der main.cpp, diese werden wir uns Schritt für Schritt betrachten. Wir müssen einiges am Code, der vom Assistenten erstellt wurde verändern und hinzufügen.

Fangen wir mit dem 1. Teil der WinMain an:

```
#include "main.h"

int WINAPI WinMain (HINSTANCE hThisInstance,
                    HINSTANCE hPrevInstance,
                    LPSTR lpszArgument,
                    int nFunsterStil)

{
    HWND hwnd;               /* This is the handle for our window */
    MSG messages;            /* Here messages to the application are saved */
    WNDCLASSEX wincl;        /* Data structure for the windowclass */
```

```
    Init();

    /* The Window structure */
    wincl.hInstance = hThisInstance;
    wincl.lpszClassName = szClassName;
    wincl.lpfnWndProc = WindowProcedure;      /* This function is called by windows */
    wincl.style = CS_DBLCLKS;                 /* Catch double-clicks */
    wincl.cbSize = sizeof (WNDCLASSEX);

    /* Use default icon and mouse-pointer */
    wincl.hIcon = LoadIcon (NULL, IDI_APPLICATION);
    wincl.hIconSm = LoadIcon (NULL, IDI_APPLICATION);
    wincl.hCursor = LoadCursor (NULL, IDC_ARROW);
    wincl.lpszMenuName = NULL;                 /* No menu */
    wincl.cbClsExtra = 0;                      /* No extra bytes after the window class */
    wincl.cbWndExtra = 0;                      /* structure or the window instance */
    /* Use Windows's default color as the background of the window */
    wincl.hbrBackground = (HBRUSH) COLOR_BACKGROUND;

    /* Register the window class, and if it fails quit the program */
    if (!RegisterClassEx (&wincl))
        return 0;

    /* The class is registered, let's create the program*/
    hwnd = CreateWindowEx (
           0,                   /* Extended possibilites for variation */
           szClassName,         /* Classname */
           "Pipette",           /* Title Text */
           WS_SYSMENU|WS_MINIMIZEBOX, /* default window */
           CW_USEDEFAULT,       /* Windows decides the position */
           CW_USEDEFAULT,       /* where the window ends up on the screen */
           WINDOW_WIDTH,                 /* The programs width */              /*
           WINDOW_HEIGHT,                 /* and height in pixels */           /*
           HWND_DESKTOP,        /* The window is a child-window to desktop */
           NULL,                /* No menu */
           hThisInstance,       /* Program Instance handler */
           NULL                 /* No Window Creation data */
           );
```

Listing 9 Teil 1 der geänderten main.cpp

Zuerst inkludieren wir unsere `main.h`. Wie wir sehen, haben wir den Prototyp der `WindowProcedure` raus gelöscht, denn diese haben wir ja bereits in unserer Headerdatei deffiniert, das gleiche gilt für unseren `szClassName`.

Dann rufen wir ziemlich früh die Funktion `Init()` auf, um einige Initialisierungen durchzuführen.

Als weitere Anpassung ändern wir die Aufrufparameter der Funktion `CreateWindowEx()`. Zum einen den Fenstertitel (wird auf „`Pipette`“ gesetzt), dann stellen wir die Style-Flags um, damit wir das Fensterverhalten anpassen, zuletzt übergeben wir noch die Breite und Höhe des Fensters.

Weiter geht es mit dem 2. Teil:

```
DrawDialog(hwnd, hThisInstance);

    /* Make the window visible on the screen */
    ShowWindow (hwnd, nFunsterStil);


    HINSTANCE hDLL = LoadLibrary("mousehook.dll");
    if(hDLL == NULL)
    {
        MessageBox(NULL, "Fehler beim Laden der DLL: mousehook.dll", "Fehler", MB_OK);
        return FALSE;
    }

    InstallHook = (MYFUNC) GetProcAddress(hDLL, "InstallHook");
    if(InstallHook == NULL)
    {
        MessageBox(NULL, "Fehler: InstallHook nicht gefunden in: mousehook.dll", "Fehler", MB_OK);
```

```
        return FALSE;
    }

    UninstallHook = (MYFUNC) GetProcAddress(hDLL, "UninstallHook");
    if(UninstallHook == NULL)
    {
        MessageBox(NULL, "Fehler: UninstallHook nicht gefunden in: mousehook.dll", "Fehler", MB_OK);
        return FALSE;
    }


    /* Run the message loop. It will run until GetMessage() returns 0 */
    while (GetMessage (&messages, NULL, 0, 0))
    {
        /* Translate virtual-key messages into character messages */
        TranslateMessage(&messages);
        /* Send message to WindowProcedure */
        DispatchMessage(&messages);
    }

    /* The program return-value is 0 - The value that PostQuitMessage() gave */
    FreeLibrary(hDLL);
    return messages.wParam;
}
```

Listing 10 Teil 2 der geänderten main.cpp

Nachdem wir das Fenster erstellt haben, rufen wir unsere Funktion `DrawDialog()` auf und übergeben das Handle auf das Hauptfenster (`hwnd`) und die Instanz der Anwendung (`hThisInstance`).

Anschließend laden wir unsere DLL mit `LoadLibrary()` und belegen die beiden Funktionszeiger mit der richtigen Adresse mit `GetProcAddress()`.

Danach gehen wir in die Dialog-Nachrichtenschleife, die solange läuft, bis unser Programm beendet wird.

Zuletzt entladen wir unsere DLL mit `FreeLibrary()`.

Weiter geht es mit der Init() Funktion:

```
void Init()
{
    InstallHook = NULL;
    UninstallHook = NULL;

    hInstallHookBtn = NULL;
    hUninstallHookBtn = NULL;
    hEditRGB = NULL;
    hEditHTML = NULL;
    hStaticFrame = NULL;

    col = RGB(0,0,0);
    ColorRect.left = 10;
    ColorRect.top = 120;
    ColorRect.right = 290;
    ColorRect.bottom = 160;

}
```

Listing 11 Init-Funktion der geänderten main.cpp

Hier werden eigentlich nur einige globale Variablen initialisiert.

Weiter geht es mit der DrawDialog() Funktion:

```
void DrawDialog(HWND &parent, HINSTANCE &inst)
{
    hInstallHookBtn = CreateWindow("BUTTON", 0, WS_CHILD|WS_VISIBLE, 10, 10, 120, 30, parent, 0,
inst, 0);
        SetWindowText(hInstallHookBtn, "Install Hook");

        hUninstallHookBtn = CreateWindow("BUTTON", 0, WS_CHILD|WS_VISIBLE, 170, 10, 120, 30, parent,
0, inst, 0);
        SetWindowText(hUninstallHookBtn, "Uninstall Hook");

        HWND tmp;
        tmp = CreateWindow("STATIC", 0, WS_CHILD|WS_VISIBLE, 10, 60, 100, 18, parent, 0, inst, 0);
        SetWindowText(tmp, "RGB Farbe:");
        tmp = CreateWindow("STATIC", 0, WS_CHILD|WS_VISIBLE, 10, 90, 100, 18, parent, 0, inst, 0);
        SetWindowText(tmp, "HTML Farbe:");

        hEditRGB = CreateWindow("EDIT", 0, WS_BORDER|WS_CHILD|WS_VISIBLE|ES_READONLY, 120, 59, 100,
20, parent, 0, inst, 0);
        hEditHTML = CreateWindow("EDIT", 0, WS_BORDER|WS_CHILD|WS_VISIBLE|ES_READONLY, 120, 89, 100,
20, parent, 0, inst, 0);
        //hStaticFrame = CreateWindow("STATIC", 0, WS_BORDER|WS_CHILD|WS_VISIBLE, 10, 120, 280, 40,
parent, 0, inst, 0);
}
```

Listing 12 DrawDialog-Funktion der geänderten main.cpp

Hier erstellen wir unsere Steuerelemente für unser Dialogfenster.

Weiter geht es mit der WindowProcedure() Funktion, unsere Nachrichtencallbackfunktion:

```
LRESULT CALLBACK WindowProcedure (HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    HWND hBut;
    static HBRUSH brush;

    switch (message)                  /* handle the messages */
    {
        case WM_COMMAND:
            if(HIWORD(wParam) == BN_CLICKED)
            {
                hBut = (HWND)lParam;
                if(hBut == hInstallHookBtn)
                {
                    BOOL ret = InstallHook();
                    if(ret == FALSE)
                        MessageBox(NULL, "InstallHook fehlgeschlagen", "Fehler", MB_OK);
                    else
                        SetWindowPos(hwnd,  HWND_TOPMOST , 0, 0, 0, 0, SWP_NOSIZE |SWP_SHOWWINDOW);
                }
                else if(hBut == hUninstallHookBtn)
                {
                    if(UninstallHook() == FALSE)
                        MessageBox(NULL, "UninstallHook fehlgeschlagen", "Fehler", MB_OK);
                }
            }
            break;

        case WM_USER+123:
           POINT p;
           p.x = GET_X_LPARAM(lParam);
           p.y = GET_Y_LPARAM(lParam);
           DrawColor(p, hwnd);
           break;

       case WM_PAINT:
          PAINTSTRUCT ps;
          brush = CreateSolidBrush(col);
          HDC dc;
          dc = NULL;
          dc = BeginPaint(hwnd, &ps);
            SelectObject(dc, brush);
            FillRect(dc, &ColorRect, brush);
```

```
          EndPaint(hwnd, &ps);
          return 0;

        case WM_DESTROY:
            DeleteObject(brush);
            if(!UninstallHook())
                MessageBox(NULL, "UninstallHook fehlgeschlagen", "Fehler", MB_OK);

            PostQuitMessage (0);       /* send a WM_QUIT to the message queue */
            break;

        default:                      /* for messages that we don't deal with */
            return DefWindowProc (hwnd, message, wParam, lParam);
    }

    return 0;
}
```

Listing 13 WindowProcedure-Funktion der geänderten main.cpp

Diese Funktion wird von Windows aufgerufen, wenn eine Nachricht für unser Programm bestimmt ist.

Am Anfang deklarieren wir uns 2 Variablen, einmal ein Fensterhandle zum Abfragen der Buttons und einen Brush (Pinsel), mit dem wir dann die gefundene Farbe „malen“.

Zunächst setzen wir uns mit den Nachrichten der Art `WM_COMMAND` auseinander. Diese Nachrichten werden geschickt sobald auf einen Button geklickt wird. Trifft die Nachricht zu, dann müssen wir prüfen welcher der beiden Buttons unseres Dialoges gedrückt worden ist. Dann installieren bzw. deinstallieren wir unseren Hook, indem wir die Funktionen aus unserer DLL aufrufen.

Anschließend haben wir noch eine Nachrichtenbehandlung für Nachrichten vom Typ `WM_USER+123`, wir erinnern uns zurück – unsere DLL schickt ja eine Nachricht vom Typ `WM_USER+123`. Hier erstellen wir uns eine `POINT` struct um die Mauskoordinaten zu speichern, dann holen wir uns über die beiden Makros (`GET_X_LPARAM` und `GET_Y_LPARAM`) die Koordinaten und speichern sie in der POINT struct. Danach rufen wir unsere DrawColor()-Funktion auf, die die weitere Behandlung vornimmt.

Dann fangen wir noch die Nachricht `WM_PAINT` ab, damit wir unsere gefundene Farbe auch in einem Rechteck darstellen können. Hier erstellen wir uns eine `PAINTSTRUCT` struct und erstellen unseren Brush mit der gesetzten Farbe (`col`). Anschließend legen wir uns einen Devicecontext (`HDC`) an, worauf wir dann „malen“ werden. Wir teilen Windows mit, dass wir zum „malen“ (`BeginPaint()`) anfangen wollen und übergeben hier unser Hauptfenster und die `PAINTSTRUCT ps`. Jetzt selektieren wir unseren Pinsel in den Devicecontext und „malen“ unser Rechteck mit dem selektiertem Pinsel (`FillRect()`). Zum Schluss teilen wir Windows mit, dass wir mit dem Malen fertig sind.

Zuletzt fangen wir noch `WM_DESTROY` Nachrichten ab, damit wir vor dem Beenden der Anwendung noch unseren Pinsel löschen und den Hook deinstallieren können.

Weiter geht es mit der DrawColor()-Funktion

```
void DrawColor(POINT &p, HWND &mainWindow)
{
    HWND owner = WindowFromPoint(p);
    if(owner)
    {
        HDC ownerDC = GetDC(owner);
        if(ownerDC && !IsPointInApp(p, mainWindow))
        {
```

```
            ScreenToClient(owner, &p);
            col = GetPixel(ownerDC, p.x, p.y);
            ToHTML();
            ToRGB();
            InvalidateRect(mainWindow, &ColorRect, TRUE);
        }
    }
}
```

Listing 14 DrawColor-Funktion der geänderten main.cpp

Hier holen wir uns zu den ermittelten Mauskoordinaten die darunterliegende Farbe. Als erstes benötigen wir das dazugehörige Fenster (`owner`) auf das mit der Maus geklickt wurde, dann holen wir uns den zugehörigen Devicecontext des Fensters (`ownderDC`). Anschließend prüfen wir ob dieser Devicecontext gültig ist und ob der Klick nicht innerhalb unserer Anwendung passiert ist. Nun müssen wir die Koordinaten noch von den globalen Screen-Koordinaten in die lokalen Client-Koordinaten (Client-Koordinaten sind Fenster-Lokale Koordinaten -> 0,0 ist dabei die Ecke links-oben des Fensters) umrechnen. Nun können wir uns die Farbe des Pixels mit `GetPixel()` holen. Jetzt rufen wir unsere beiden Funktionen auf um die Farbwerte anzuzeigen (`ToHTML()` und `ToRGB()`), dann rufen wir noch `InvalidateRect()` auf um Windows mitzuteilen, dass ein Teil des Fensters neu gezeichnet werden soll.

Weiter geht es mit den beiden Funktionen ToHTML() und ToRGB():

```
void ToHTML()
{
    char sHtml[8]; sHtml[0] = 0;
    sprintf(sHtml, "#%.2X%.2X%.2X", GetRValue(col),
                                    GetGValue(col),
                                    GetBValue(col));
    SetWindowText(hEditHTML, sHtml);
}

void ToRGB()
{
    char sRGB[12]; sRGB[0] = 0;
    sprintf(sRGB, "%d,%d,%d", GetRValue(col),
                              GetGValue(col),
                              GetBValue(col));
    SetWindowText(hEditRGB, sRGB);
}
```

Listing 15 ToHTML und ToRGB-Funktion der geänderten main.cpp

In diesen beiden Funktionen verarbeiten wir den Farbe und zeigen sie entsprechend an.

Zum Schluss haben wir noch die Funktion IsPointInApp():

```
bool IsPointInApp(POINT &p, HWND &mainWindow)
{
  RECT rect;

  GetWindowRect(mainWindow, &rect);

  if(p.x >= rect.left &&
     p.x <= rect.right &&
     p.y >= rect.top &&
     p.y <= rect.bottom)
  {
    return true;
  }
  else
    return false;
}
```

Listing 16 IsPointInApp-Funktion der geänderten main.cpp

Hier prüfen wir ob, ob der übergebene Punkt innerhalb unserer Anwendung liegt.

5.3.4 Das fertig Programm

Nun können wir unser Pipettenprogramm übersetzen (CTRL-F9) und anschließend starten (CTRL-F10), der Dialog sollte in etwa so aussehen:

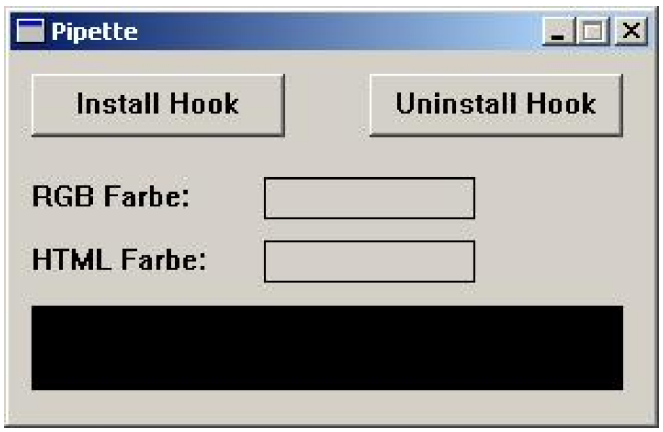


Abbildung 5Das fertige Pipettenprogramm

Das große schwarze Feld an der Unterseite ist unser Rechteck in dem wir die gefundene Farbe darstellen, die wird initial auf RGB(0,0,0) gesetzt, was dem Wert von schwarz entspricht. Mit den Buttons „Install Hook“ und „Uninstall Hook“ können wir den Hook einrichten und wieder entfernen.

6. Nachwort

Ich hoffe das ich mit diesem Tutorial einen kleinen Einblick zur Verwendung von Hooks auf Windows Betriebssystemen geben konnte.

Am Ende dieses Tutorials befindet sich eine Liste der benötigten Internet-Links.

Anhang A Dateianhänge

Zu diesem Tutorial gehören folgende Dateien:

Dateianhang	*Beschreibung*
Mouse-Systemhook.sxw	OpenOffice-Writer-Dokument (dieses Tutorial)
Mouse-Systemhook.pdf	PDF-Format dieses Tutorials

Dateianhang	*Beschreibung*
DLL-Shared.sxd	OpenOffice-Zeichnung (Schaubild über DLL-SharedMemory)
dll.h	Headerdatei für die DLL
dllmain.c	Quelltext für die DLL
mousehook.dev	DevC++ Projekt für die DLL
main.h	Headerdatei für die Pipettenanwendung
main.cpp	Quelltext für die Pipettenanwendung
pipette.dev	DevC++ Projekt für die Pipettenanwendung

1 http://www.bloodshed.net/devcpp.html
2 http://www.gnu.de/gpl-ger.html
3 http://gcc.gnu.org/
4 http://sources.redhat.com/gdb/
5 http://www.mingw.org/
6 http://prdownloads.sf.net/mingw/binutils-2.13.90-20030111-1.tar.gz?download
7 http://sourceforge.net/projects/mingw/
8 http://gcc.gnu.org/onlinedocs/gcc-3.3.2/gcc/Variable-Attributes.html#Variable Attributes
9 http://msdn.microsoft.com/library/default.asp?url=/library/en-us/winui/winui/windowsuserinterface/windowing/hooks/hookreference/hookfunctions/setwindowshookex.asp
10 http://msdn.microsoft.com/library/default.asp?url=/library/en-us/winui/WinUI/WindowsUserInterface/Windowing/Hooks/HookReference/HookFunctions/MouseProc.asp