Hook (+IPC) Tutorial

zu den Themen:

Kommunikation mit einer DLL;
Maus- und Tastaturhooks;
weitere Hookanwendungen;

IPC auf dem lokalen Rechner

geschrieben von:

-=ΔssarbΔd=-

Kontaktmöglichkeiten:

http://assarbad.net

hooktut@assarbad.net

Alle Ausführungen beziehen sich auf Win32-Systeme ab Windows 95, NT4 respektive – anderenfalls sind entsprechende Stellen im Text extra für eine entsprechende Windows-Version ausgewiesen.
Die Beispiele können mit Delphi ab Version 3 und teils erst ab Version 4 nachvollzogen werden.

May the source be with you, stranger ...
Снижок это не только кефир, снижок это стиль жизни.

Version 1.00 beta2 [2003-07-06]

Dank an Mathias Simmack, der als Korrekturleser in verschiedenen Stadien wertvolle Hinweise gab. Sowie an Xandros, der als Linuxer half den Text dummy-proof zu machen ... Delphianer, selbst ohne viel Hintergrundwissen, müssen den Text jetzt einfach verstehen.

Lizenzbedingungen

Das komplette Tutorial inklusive des Quellcodes unterliegt folgender, der BSD-Lizenz abgewandelter, Lizenzvereinbarung (Software und Sourcecode bezieht sich dabei auch auf die Textform des Tutorials):

Keine der hier zur Verfügung gestellten Informationen darf zu illegalen Zwecken eingesetzt werden, dies schließt sowohl den Text als auch den Quellcode ein.

Für die Links zu externen Ressourcen übernehme ich keinerlei Verantwortung. Zum Zeitpunkt der Erstellung dieses Schriftstücks erschienen sie mir nützlich und sinnvoll und enthielten keinerlei erkennbare illegale Inhalte oder Tendenzen.

Die Lizenz ist nur deshalb nicht auf deutsch verfügbar, weil der englische Text als rechtlich verbindlich und korrekt gilt und ich mich nicht in der Lage fühle den Text in dieser Form zu übersetzen. Zumal unsere deutschen Advokaten und Paragraphenakrobaten ja im Sinne des Rechts und der Gleichheit ihre eigene Sprache erfunden haben, die kein anderer mehr zu verstehen hat – aber: „Alle Menschen sind vor dem Gesetz gleich.“ (Artikel 3 Abs. 1), nur die die sich einen Advokaten leisten können, sind gleicher – Danke für Euer Verständnis!

Vorwort

Der Leser sollte mit der Pascal / ObjectPascal[1]-Syntax vertraut sein und als Entwicklungswerkzeug vorzugsweise Delphi 4 oder später zur Verfügung haben. Außerdem wäre ein Ressourcen-Editor (WEDITRES, Visual C Standard)[2] äußerst nützlich und sinnvoll.

Wer die Vorgänger dieses Tutorials kennt, der weiß: es hat sich stark verändert. Die alten Versionen waren zeitweise als CHM und hernach als HTML verfügbar. Im Sinne der Druckbarkeit des Dokuments habe ich mich jedoch für das PDF-Format entschieden. Dies sollte auch denjenigen, die eigentlich nicht überall Internet verfügbar haben, die Möglichkeit geben, das Tutorial auch offline jederzeit lesen und drucken zu können. Für mich persönlich ist dies eine wichtige Eigenschaft eines jeden Dokumentes ;)

Die Vorgängerversion dieses Tutorials hatte einige Fehler im Quelltext, sowie scheinbar offen gelassene Fragen - zumindest schien das so, wenn man sich in den deutschsprachigen Delphi-Foren umschaute.

Voraussetzungen seitens des Lesers
Ich setze voraus, dass der Leser mit den Prinzipien von DLLs, deren Programmierung, Einbindung und so weiter vertraut ist. Sollte dies nicht der Fall sein, verweise ich auf mein DLL-Tutorial, zu finden unter:

http://assarbad.net/stuff/tutorials

Des Weiteren ist mindestens Grundlagenwissen in Sachen Fensternachrichten und virtuelle Tastencodes erforderlich. Wer auch hier Defizite hat, sei auf die Delphi-Bücher von Andreas Kosch und diversen anderen Autoren hingewiesen. Außerdem sei ihm das Platform SDK von Microsoft ans Herz gelegt, welches allerdings nur in Englisch verfügbar ist. Links zu genannten Ressourcen finden sich am Ende dieses Dokumentes. nonVCL-Kenntnisse erwünscht!
Will man meine Kommentare im Quelltext verstehen, sollte man außerdem Englisch beherrschen.

Format
Zur Erstellung habe ich OpenOffice.org 1.0.3.1 verwendet. Die Konvertierung zu PDF erfolgte mittels pdf-Factory Pro. Beide Programme kann ich nur empfehlen, auch wenn letzteres nur als Shareware und damit nicht-kostenlos verfügbar ist.

Danksagung
Dank möchte ich zumal denen sagen, die mich bei der Entwicklung meiner Programme und auch meiner Tutorials durch Feedback und zum Teil auch Korrekturen so freizügig unterstützt und zu deren Verbesserung beigetragen haben.

Mein **Spezieller Dank** geht dabei an folgende Personen:

Eugen Honeker,
Mathias Simmack,
Michael Puff,
Nico Bendlin,
Ronny Purmann,
„Xandros“ (für die Ausdruckshilfen),
Сергий "Полёт" Польетаев

Dank auch an all jene Künstler die einen bei solch kreativen Prozessen wie dem Programmieren durch ihre Musik immer wieder von Neuem inspirieren:

Wolfgang Amadeus Mozart, Antonio Vivaldi, Ludwig van Beethoven, Poeta Magica, Kurtzweyl, Krless, Sarbande, Vogelfrey, Manowar, Blind Guardian, Weltenbrand, In Extremo, Wolfsheim, Carl Orff, Veljanov, Lacrimosa, Finisterra, Enigma, Beautiful World, Adiemus, Земфира, Би-2

1 ObjectPascal ist der Oberbegriff für OOP-fähiges Pascal. Es gibt neben Delphi noch weitere.
2 Siehe Referenzen.

Was sind Hooks, wozu braucht man sie, was können sie leisten, was nicht ...

Hook ist Englisch für Haken. Dabei geht es nicht um Haken als Angel- oder Mordwerkzeug, sondern vielmehr um das „Einhaken“ im besten Sinne des Wortes. Wir haken uns mit Hooks in die Kommunikation eines Fensters ein.
Wiederholen wir dazu einmal kurz die Grundlagen der Nachrichtenbearbeitung des Systems. Fensternachrichten sind ein ziemlich cleveres Prinzip um ein Fenster zu benachrichtigen, wenn sich etwas ändert, oder wenn das Fenster auf ein Ereignis reagieren soll, bspw. eine Maus- oder Tastatureingabe. Die Benachrichtigung ist immer das bessere Prinzip, da ansonsten nur noch Polling, also die ständige Abfrage eines Status, in Frage käme. Beim Polling ist der größte Nachteil, dass die CPU stark ausgelastet ist. Selbst wenn ein separater Thread das Polling übernimmt und alle 2 Mikrosekunden den Status eines Ereignisses überprüft, so ist es dennoch besser, sich stattdessen über das Ereignis benachrichtigen zu lassen. Dieses Prinzip ist an vielen Stellen zu finden, beispielsweise lässt sich der Prozessor von Interrupts unterbrechen, wenn ein I/O-Ereignis ansteht.

Ein Hook ist nun im Grunde eine Callbackfunktion. Dem System kann man mehrere dieser Callbackfunktionen bekannt machen. Das System ruft sie der Reihe nach auf und übergibt diverse Parameter - welche im Falle von Hooks Informationen über Ereignisse liefern, die man dann weiter auswerten oder verarbeiten kann.

Callbacks

Callbacks sind Funktionen, welche meist vom System aufgerufen werden und entsprechend des Funktionsrumpfes, der von der Anwendung vorgegeben ist, eine Aufgabe für die Anwendung verrichten. Oftmals geschieht dies im Kontext des Prozesses, von dem Informationen geholt werden sollen - dies gilt unter anderem für `EnumWindows()` ... für Hooks gilt Gleiches.

Wenn von Hooks in diesem Tutorial die Rede ist, sind Fensterhooks gemeint, insofern nicht extra ausgezeichnet!

Was Hooks nicht leisten:

Um es vorauszuschicken: Hooks können nicht zum Abfangen von Strg-Alt-Entf benutzt werden. Zumindest geht dies nicht auf der NT-Plattform. Wer dies beabsichtigt, sollte einen sehr guten Grund vorzuweisen haben und muss sich andererseits mit Windows-Treiberprogrammierung auseinandersetzen. Ja, richtig gehört: wer beabsichtigt eine Systemtasten-Kombination zu unterdrücken, der muss echte Kopfstände machen. Informationen finden sich auf der Seite von Sysinternals – es gibt dort einen Beispielsource der mit Caps-Lock zu tun hat.
Es sollte auch gehen den SAS-Dialog durch Modifikation der `MSGINA.DLL` zu umgehen, aber es kommt eben immer darauf an, was man nun mit der Unterdrückung von Strg-Alt-Entf zu erreichen gedenkt.
Um ehrlich zu sein sehe ich aber kaum Gründe für normale Programmierer, eine so integrale und für jedermann wichtige Tastenkombination zu unterdrücken. Warum wichtig? Nun, sie ist dazu da um globale Hooks zu umgehen, sollten diese beispielsweise das System zum Hängen bringen oder Tastatur und Maus ansonsten unterdrücken. Außerdem startet man unter NT so den Taskmanager, sperrt die Workstation und ändert das Passwort. Sollten einige dieser Funktionen unerwünscht sein, sollte man sich mit den alten Policies unter NT 4.0 und der Group Policy[3] ab Windows 2000 auseinandersetzen.
Ein Beispiel für eine sinnvolle Anwendung habe ich in iFix, einem Produkt der Firma Intellution, gesehen. Dort geht es u.a. um Prozessvisualisierung, wobei danach vor den PCs Operatoren sitzen, die von PCs nicht notwendigerweise Ahnung haben müssen. Entsprechend wird Strg-Alt-Entf unterdrückt, um ihnen die Möglichkeit zu nehmen, am System herumzuspielen.

3 In deutschen Versionen auch „Sicherheitsrichtlinie“ genannt.

Hooks gibt es in verschiedenen Geschmacksrichtungen, hier eine Übersicht über mögliche Varianten (i.e. Hook-Typen) [*thread-lokal schließt system-global ein*]:

System-global:	**Thread-lokal:**
WH_JOURNALPLAYBACK = 1	WH_CALLWNDPROC = 4
WH_JOURNALRECORD = 0	WH_CALLWNDPROCRET = 12
WH_KEYBOARD_LL = 13 *(NT)*	WH_CBT = 5
WH_MOUSE_LL = 14 *(NT)*	WH_DEBUG = 9 *(NT)*
WH_SYSMSGFILTER = 6	WH_FOREGROUNDIDLE = 11
WH_HARDWARE = 8 (??)	WH_GETMESSAGE = 3
	WH_KEYBOARD = 2
	WH_MOUSE = 7
	WH_MSGFILTER = -1
	WH_SHELL = 10

In der Übersicht sind alle Hooktypen dargestellt und danach unterteilt, ob sie global oder nur lokal (also nur innerhalb eines Prozesses) benutzbar sind. Die *hervorgehobenen* (i.e. kursiv gefassten) Typen sind nur unter NT nutzbar!
`WH_HARDWARE` (= 8) ist der einzige undokumentierte Hooktyp von den oben genannten. Angeblich wird er eigentlich von noch keinem Betriebssystem unterstützt. Auf einer russischen Seite[4] habe ich aber eine Beschreibung dazu gefunden: „Функция-фильтр, обрабатывающая сообщения оборудования. Функция-фильтр ловушки вызывается, когда из очереди приложения считывается сообщение оборудования.“ - was soviel bedeutet wie: „Filterfunktion, welche Hardware-Nachrichten verarbeitet. Die (Filter-)Funktion wird aufgerufen, wenn eine Hardware-Nachricht von der Nachrichtenschleife einer Anwendung empfangen wurde.“ Möglicherweise handelt es sich gar um einen lokalen Hooktyp - ... nix genaues weiß man nich ... ;-)

Übrigens, die Namen der oben angegebenen Konstanten sind natürlich, wie eigentlich üblich, selbsterklärend. In diesem Tutorial werden wir uns aber aufgrund der großen Nachfrage fast ausschließlich mit den beiden Hooktypen `WH_KEYBOARD` und `WH_MOUSE` beschäftigen.

Da wir IPC erst später innerhalb dieses Tutorials besprechen wollen, beschäftigen wir uns zuerst mit lokalen Hooks und allgemein dem Prinzip, das hinter den Hooks steht.

Lokale Hooks am Beispiel von Maus- und Tastaturhooks

Lokal bedeutet bei Hooks ersteinmal, dass der entsprechend gesetzte Hook *nur innerhalb eines Prozesses* wirkt - und zwar innerhalb jenes Prozesses, der ihn auch gesetzt hat. Aber nicht nur das; lokale Hooks beschränken sich zusätzlich noch auf den Thread für den sie gesetzt wurden. Dazu übergibt man beim Erstellen des Hooks die ThreadID des Zielthreads als Parameter. Üblicherweise wird sich das auf den einen Hauptthread einer Anwendung beschränken. Aber für Multithreading-Anwendungen ist es schon gut zu wissen, dass diese Beschränkung existiert.

Vorbereitendes
Um nun Hooks zu setzen, brauchen wir natürlich ein paar APIs, die Windows schon anbietet. Diese Funktionen werden sowohl von lokalen als auch von globalen Hooks benötigt.

Funktionen: `SetWindowsHookEx(), UnhookWindowsHookEx(), CallNextHookEx(), HookProc()`

`SetWindowsHookEx()` ist da um den Hook zu setzen. Dazu wird als erster Parameter (`idHook`) der Typ des zu setzenden Hooks übergeben (siehe oben: Übersicht der Hooktypen). Die oben angegebenen Werte sind Konstanten welche in der `Windows.pas` vordeklariert sind. Als zweiter Parameter wird die Adresse der Hook-Callbackfunktion übergeben. Dritter Parameter (`hMod`) ist das Instanzenhandle zu dem Modul in welchem die Callbackfunktion liegt. Im Falle eines lokalen Hooks ist hier 0 (null) als Konstante ausreichend. Der vierte Parameter (`dwThreadId`) ist die TID des Threads, mit welchem der Hook assoziiert ist. Bei globalen Hooks gibt man hier 0 (null) an, um den

4 „Hooks - аспекты реализации“ (Алексей Павлов) [http://www.delphimaster.ru/articles/hooks/index.html]

Hook mit allen Threads des Systems / Desktops zu assoziieren. Rückgabewert ist ein Hookhandle vom Typ `HHOOK`. Allerdings tut es hier auch ein beliebiger anderer Handletyp.
`UnhookWindowsHookEx()` ist die Gegenfunktion zu `SetWindowsHookEx()`. Damit wird der Hook wieder aus der internen Hookliste des Systems entfernt. Als einziger Parameter (`hhk`) dient das Hookhandle, welches von `SetWindowsHookEx()` zurückgegeben wurde. Rückgabewert ist ein `BOOL` (32bit Boolean), welcher Erfolg (`True`) oder Misserfolg (`False`) anzeigt.
`CallNextHookEx()` wird bei allen Hooks aufgerufen um die erhaltenen Werte (die Parameter der `HookProc()`) weiterzugeben.
`HookProc()` ist der symbolische Name für die Gruppe von Callbackfunktionen, bei denen die Parameter zwar verschiedene Bedeutungen haben, deren Deklaration aber für alle Hooktypen identisch ist. Der Name der Funktion ist wie bei allen Callbackfunktionen Sache des Programmierers. Der Compiler wandelt den Namen sowieso maximal in eine Adresse und ein Debug-Symbol um.

Später weiteres zu den *einzelnen Parametern* der letzten beiden Funktionen.

Schauen wir uns aber zuerst einmal die delphigerechten Deklarationen der Funktionen an:

```
{ --- 1. --- }
function SetWindowsHookEx(idHook: Integer; lpfn: TFNHookProc;
  hmod: HINST; dwThreadId: DWORD): HHOOK; stdcall;
type
  TFNHookProc = function(nCode: Integer; wParam: WPARAM;
    lParam: LPARAM): LRESULT; stdcall;

{ --- 2. --- }
function SetWindowsHookEx(idHook: Integer; lpfn: Pointer; hmod: HINST;
  dwThreadId: DWORD): HHOOK; stdcall;
function HookProc(nCode: Integer; wParam: WPARAM;
  lParam: LPARAM): LRESULT; stdcall;

function UnhookWindowsHookEx(hhk: HHOOK): BOOL; stdcall;
function CallNextHookEx(hhk: HHOOK; nCode: Integer; wParam: WPARAM;
  lParam: LPARAM): LRESULT; stdcall;
```

Wie man sehen kann gibt es verschiedene Möglichkeiten, `SetWindowsHookEx()` und auch `HookProc()` zu deklarieren. In Variante 1 wird für `HookProc()` ein echter Typ vereinbart, und `SetWindowsHookEx()` bekommt eine Variable dieses Typs zugewiesen. In Variante 2 ist es so, dass die `HookProc()` als feste Funktion im Quelltext, oder importiert aus DLL oder OBJ-Datei, eingebunden wird. `SetWindowsHookEx()` bekommt dann einen Pointer zu `HookProc()` in der Form `@HookProc` zugewiesen. Ich persönlich benutze eher die letzte Variante, wobei man sagen muss, dass sich erstere hervorragend eignet, um eine `HookProc()` aus einer DLL dynamisch einzubinden! Da Funktionen natürlich auch in Delphi nur Zeiger auf Codebereiche sind, sind beide oben genannten Varianten auch beliebig miteinander kombinierbar[5].

> Als Beispiel für die Unterschiede der Bedeutung der Parameter bei `HookProc()` kann man `wParam` und `lParam` bei Maus- und bei Tastaturhooks heranziehen. Während bei den Tastaturhooks `wParam` der virtuelle Tastencode und `lParam` der Scancode mit einigen weiteren Flags ist, ist `wParam` beim Maushook der Wert der Mausnachricht (z.B. `WM_MOUSEMOVE`) und `lParam` ein als Integer verpackter Zeiger auf eine `MOUSEHOOKSTRUCT`.

Zielsetzung für unseren lokalen Hook
Ziel soll es sein, einen lokalen Hook so in unser Programm einzubinden, dass abhängig von der ID des Controls, über dem die Maus gerade ist, ein Hinweis (Hint) in der „Statusleiste“ (bei mir ein Button) erscheint! Mit einer echten Statusleiste ist dies natürlich auch möglich, der Einfachheit halber bleiben wir aber bei einem „Status-Button“.

Ich benutze diese Technik für viele meiner Programme, um dem Benutzer eine leichte und komfortable Benutzerführung zu bieten.

5 Zumindest für Delphi 4 konnte ich dies verifizieren.

Erarbeitung des Quelltextes

Da ich den lokalen Hook in das Beispielprogramm zum Tutorial einbinde, und dieses etwas komplexer ist, als *nur* ein lokaler Hook, werde ich nur die entsprechenden Quelltextsegmente hier besprechen.

Wir schreiben zuerst eine Funktion, die anhand der Control ID (`CtrlID`) entscheidet, welcher Text angezeigt werden soll. Dabei beschränken wir uns auf statischen[6] Text - dynamischen Text einzubinden ist nicht viel schwerer, nur eben weiterer unübersichtlicher Code, deshalb lasse ich dies als kleine Übung für den Leser :-).

```
function MouseProc(nCode: Integer; wParam: WPARAM; lParam: LPARAM): LRESULT; stdcall;
var
  CtrlID: Word;
begin
  case nCode < HC_ACTION of
    True:
      Result := CallNextHookEx(MouseHintHook, nCode, wParam, lParam);
  else
    CtrlID := GetDlgCtrlId(PMOUSEHOOKSTRUCT(lParam)^.HWND);
    if lastID <> CtrlID then
    begin
    { Hier kommt der Code rein, welcher die CtrlID auswertet und den Text anzeigt }
    end;
    Result := CallNextHookEx(MouseHintHook, nCode, wParam, lParam);
  end;
end;
```

Dies ist nun die Hookfunktion, also unsere Variante von `HookProc()`. Wie man sehen kann überprüfen wir zuerst `nCode` auf negative Werte (kleiner als `HC_ACTION`). Dies ist notwendig, damit Windows noch die Kontrolle über die Hooks in der Hookchain[7] behalten kann - aus diesem Grunde *müssen alle Hooks* sich an die vorgegebenen Konventionen halten! Ist `nCode` gleich `HC_ACTION`, so muss der Hook seine Bearbeitung der erhaltenen Daten durchführen - wirklich wichtig ist das nur für Hooks, welche Informationen abfangen wollen, bevor sie beim gehookten Fenster ankommen! Für unsere Zwecke (einfache Überwachung) ist das nicht zwingend notwendig. Außerdem bearbeiten wir die Daten sowieso nur dann, wenn uns das das System nicht per Konvention verbietet (also wenn `nCode` kleiner als `HC_ACTION = 0` ist). Für `nCode` gibt es weitere vordeklarierte Werte, wie folgt:

```
const
  HC_ACTION = 0;
  HC_GETNEXT = 1;
  HC_SKIP = 2;
  HC_NOREMOVE = 3;
  HC_NOREM = HC_NOREMOVE;
  HC_SYSMODALON = 4;
  HC_SYSMODALOFF = 5;
```

Deren Bedeutung ist grundsätzlich abhängig vom Hooktyp, so dass ich hier nicht näher darauf eingehen werde. Der wichtigste Wert ist, wie oben erwähnt, `HC_ACTION`.

`CallNextHookEx()`[8] ist dafür verantwortlich, dass die Daten, welche wir in unserer `HookProc()` bekommen, an die anderen Hooks in der Hookchain weitergegeben werden! Wird es nicht aufgerufen, so blockieren wir damit die Weitergabe der Daten an die anderen Hooks. Das kann natürlich gewollt sein, normalerweise *sollte dies aber nicht* der Fall sein! Dadurch werden nämlich eventuell andere Anwendungen oder gar das System[9] (bei globalen Hooks) so beeinträchtigt, dass keinerlei Eingaben mehr möglich sind und unter Umständen bei diversen Hooktypen die Nachrichten soweit abgeblockt werden, dass kein Fenster mehr reagiert.
Die erhaltenen Parameterwerte der Callbackfunktion werden übrigens in gleicher Reihenfolge mit `CallNextHookEx()` weitergereicht. Der erste Parameter ist das Handle des aktuellen Hooks.

---

6 Soll heißen, es gibt keine Veränderlichen, wie z.B. ein aktueller Wert aus dem Anwendungsfenster.
7 Hookchain = Hook-Kette, es handelt sich dabei um alle Hooks die im System, bzw. im Thread verfügbar sind. Es ist nicht bestimmbar (außer für `WH_DEBUG`) in welcher Position der eigene Hook in der Kette ist.
8 Es gab auch einmal `CallNextHook()`, was aber wie auch `SetWindowsHook()` inzwischen veraltet ist!
9 Dies gilt insbesondere für Windows 9x/Me, da dort Hooks eine stärkere Durchschlagskraft entwickeln!

Zitat aus der Beschreibung von `CallNextHookEx()`

Calling **`CallNextHookEx`** is optional, but it is highly recommended; otherwise, other applications that have installed hooks will not receive hook notifications and may behave incorrectly as a result. You should call **`CallNextHookEx`** unless you absolutely need to prevent the notification from being seen by other applications.

[Platform SDK February 2003][10]

Da ich mich entschlossen habe, die Maus-Hints als Klasse zu verpacken, kommen wir nun zur Klassendeklaration. Hinweisen sollte ich vielleicht noch auf den Fakt, dass Callbackfunktionen (und `HookProc()` gehört dazu) nicht ohne weiteres als Methoden deklariert werden können. Dies ist der Fall, weil sich die Adresse der Funktion (Methode) innerhalb einer Klasse jederzeit dynamisch verändern kann! Aus diesem Grunde bleibt unsere HookProc() außerhalb der Klasse als globale Funktion innerhalb der Unit deklariert. Hier nun die Klassendeklaration:

```
type
// Responsible for showing the hints in the status button :)
  TMouseHints = class
  private
  public
    constructor Create(hwnd: HWND);
    destructor Destroy;
  end;
```

Und hier nun Constructor und Destructor der Klasse:

```
constructor TMouseHints.Create(hwnd: HWND);
begin
  inherited Create;
  MouseHintHook := SetWindowsHookEx(WH_MOUSE, @MouseProc, 0, GetCurrentThreadId());
end;

destructor TMouseHints.Destroy;
begin
  if MouseHintHook <> 0 then
    UnHookWindowsHookEx(MouseHintHook);
  inherited Destroy;
end;
```

Als Parameter des Constructors wird das Handle des Fensters (`hwnd`) übergeben, für dessen Controls die Hints angezeigt werden sollen. Die Instanz der Klasse sollte im Idealfall während `WM_CREATE` beziehungsweise `WM_INITDIALOG` erzeugt werden.

Danach wird das Fenster, beziehungsweise die Nachrichtenschleife seines Threads, gehookt. Der Rückgabewert ist ein Handle, wobei man unter Delphi statt des vordeklarierten Typs `HHOOK` auch `THandle` oder ein `DWORD` oder `LongWord` nehmen kann. Die Parameter von `SetWindowsHookEx` sind, der Reihenfolge nach:

- `WH_MOUSE`: Hooktyp des zu erzeugenden Hooks.
- `@MouseProc`: Pointer zu unserer eigenen `HookProc()`, die hier `MouseProc` heißt.
- `0`: ist das Instanzenhandle des Moduls, in welchem die Callbackfunktion liegt. Null bedeutet dabei ganz simpel „aktuelles Modul“ (i.e. aufrufendes Modul)
- `4. Parameter`: ID des zu hookenden Threads. Ist dieser Parameter Null und liegt die Callbackfunktion in einer DLL, deren Modulhandle als dritter Parameter übergeben wurde, so erzeugen wir einen *globalen Hook*. Dazu aber gleich näheres ...

Wer mehr zur Implementation der Klasse `TMouseHints` erfahren möchte, möge sich den beiliegenden Quelltext anschauen. Das Thema lokale Hooks ist hiermit aber erst einmal abgeschlossen.

10 Link siehe Referenzen

Globale Hooks ...

Kommen wir also zu den *globalen Hooks*. Globale Hooks heißen so, weil sie innerhalb mehrerer Threads in verschiedenen Prozessen des gleichen Desktops aktiv sein können. Wie wir schon bei lokalen Hooks gesehen haben, sind, obwohl die behandelten Hooks auch als Fensterhooks bezeichnet werden, nicht Fenster, sondern vielmehr Threads die Bezugsgröße.
Vielleicht sollte man an dieser Stelle nochmals daran erinnern, dass auch Threads eine Nachrichtenschleife haben können, selbst dann wenn sie keine zugehöriges Fenster haben.

Globale Hooks funktionieren also systemweit, aber wie?

Funktionsweise globaler Hooks

Globale Hooks bedienen sich, wie auch lokale Hooks, des von Windows angebotenen Callback-Mechanismus[11]. Da ich nicht davon ausgehen kann, dass jedem hier Callbacks vertraut sind, nochmals eine kurze Auffrischung:
Callbacks sind Funktionen, die vom meist System aufgerufen werden, wobei der Funktionsrumpf Code enthält, welcher eine Arbeit für die Anwendung verrichtet.
Im Falle von globalen Hooks ist es nicht anders. Da aber die Callbackfunktion in anderen Prozessen aufgerufen werden soll (global!), muss diese in einer DLL, also quasi einer Shared-Resource („verteilte Ressource“) liegen.
Man kann also sagen, dass wir Informationen *aus verschiedenen Prozessen* zugeschickt bekommen wollen, wenn wir einen Hook installieren.

Dem einen oder anderen Leser mag jetzt schon ein Licht aufgegangen sein, warum dieses Tutorial im Titel auch die Bezeichnung IPC[12] trägt. Für globale Hooks die aus der DLL heraus mit einer Anwendung kommunizieren wollen, brauchen wir IPC, da wir die Callbackfunktion eben nicht nur innerhalb unseres Prozesses aufrufen, sondern auch innerhalb der anderen gehookten Prozesse (und Threads). Würden wir also in unserem eigenen Prozess einen Speicherbereich bei Adresse $1000000 reservieren und dem Hook „auftragen“, er solle an diese Adresse die Daten schreiben, welche er gerade von unserem globalen Maus- oder Tastaturhook bekommt, so könnte dies rein theoretisch zum dutzendweisen Absturz der gehookten Prozesse führen, da sie ja an Adresse $1000000 in ihrem Speicherbereich schreiben, nicht aber in unserem! An dieser Adresse kann sich alles mögliche befinden - seien es nun wichtige Daten oder gar Code, den wir zu überschreiben versuchen. Aus diesem Grunde muss man eine andere Möglichkeit finden um die Daten von den fremden Prozessen in unseren eigenen Prozess zu bekommen. Dies ist (wenn man so will) ein „Nachteil“ des Betriebssystems zugunsten von Sicherheit durch Isolierung der Prozesse.

(Wer meint er kenne sich hinreichend mit dem Thema Memory Mapped Files aus, und/oder wer kein Interesse an IPC hat, kann diese Sektion überspringen.)

Schweifen wir also erst einmal ab zu den interessanten Möglichkeiten von ...

IPC - Inter Process Communication

Der Möglichkeiten gibt es viele, und z.B. die Kommunikation zwischen zwei Rechnern wird auch als IPC bezeichnet - logischerweise. Wir beschränken uns aber hier auf IPC innerhalb eines Desktops[13].

Bei IPC unterscheidet man generell nach dem Server und dem Client. Der Server stellt, leger ausgedrückt, den Mechanismus bereit, zu dem sich der Client verbindet.

Der Vollständigkeit halber seien einmal alle Arten von IPC genannt, die in Windows implementiert sind: Clipboard, COM, Data Copy, DDE, File Mapping, Mailslots, Pipes, RPC, Windows Sockets.

11 Unsere `HookProc()` ist nichts anderes als eine Callbackfunktion.
12 IPC steht für Inter Process Communication - also Kommunikation zwischen Prozessen.
13 Desktops sind in diesem Kontext Container für Prozesse. Nicht zu verwechseln mit einer Shell!

Jene Möglichkeiten, die unterstrichen sind, werden wir besprechen.

COM (Component Object Model) kann komplette eigene Bücher füllen und bietet unter anderem ein komplettes Set an Sicherheitsfunktionen, deshalb soll es genau wie sein Vorgänger OLE hier außen vor bleiben. Gleiches gilt für RPC (Remote Procedure Call), sowie DDE (Dynamic Data Exchange) und Windows Sockets (Stichwort: TCP, UDP).

Wie schon erwähnt, geht es hier nicht um die Besprechung aller Aspekte von IPC, sondern vielmehr darum, desktop-lokale Methoden anzuschneiden, kurz zu besprechen und anhand von Beispielen, einen Wegweiser zu diesen verschiedenen Methoden zu bieten. Für unsere globalen Hooks, das sei hier schon einmal vorausgeschickt, werden wir uns (fast) ausschließlich mit Memory Mapped Files[14] (MMF) beschäftigen. Aus diesem Grunde werden sie auch am ausführlichsten besprochen.
Sollte weitergehendes Interesse an IPC bestehen, kann ich nur das Platform SDK als weitergehende Wissensquelle empfehlen ;)

Beispiele im Sourcecode finden sich zu jeder der hier besprochenen Methoden bei den Quelltexten unter .\IPC. Die Quelltexte sind stark kommentiert - allerdings wie schon erwähnt in Englisch.

Es gibt dazu aber noch ein wenig zu sagen. Ich habe versucht alle Funktionen, welche wichtig für IPC (bzw. Client/Server-Kommunikation) sind, in eine Include-Datei zu packen, die sich je nach Methode mehr oder weniger unterscheidet. Dabei habe ich anschauliche Namen für die Funktionen benutzt. Hier die Erklärungen:

- `ReadServerMessage()`
  liest eine Nachricht nach der entsprechenden Methode aus
- `WriteServerMessage()`
  sendet eine Nachricht nach entsprechender Methode zum Server
- `OpenServer()`
  Erzeugt serverseitig die nötigen Strukturen um als Server zu arbeiten
- `CloseServer()`
  konträr zu `OpenServer()`
- `FindAndOpenServerConnection()`
  Sucht clientseitig nach dem Server und gibt Statuswerte aus
- `CloseServerConnection()`
  schließt eine offene Serververbindung

Das nonVCL-Skelett braucht uns also gar nicht weiter interessieren. Allen Beispielen ist gemein, dass der Server vor dem Client gestartet werden sollte - es sind eben sehr simple Beispiele.

Clipboard

Funktionen: `SetClipboardData()`, `GetClipboardData()`, `SetClipBoardViewer()`, `ChangeClipboardChain()`

Hierbei ist zu beachten, dass das Clipboard[15] bestimmte Datenformate bereits kennt und andere erst bekannt gemacht bekommen muss.
Es gibt verschiedene Varianten, wie das Clipboard als Datenaustausch-Medium benutzt werden kann. Eine Variante wäre es, die Daten einfach in das Clipboard zu kopieren und dem designierten Empfänger eine Nachricht zu schicken, dass neue Daten verfügbar sind. Dies könnte man mit `SendMessage()` oder Events oder anderswie realisieren.
Die wohl eleganteste Methode ist es aber, den Empfänger als Clipboard-Viewer zu registrieren, so dass er quasi auf Veränderungen und neue Daten im Clipboard „lauscht“ - ganz dem Server-Prinzip folgend. Da wir uns als Clipboard-Viewer an bestimmte Konventionen zu halten haben, tanzt es ein wenig aus der Reihe. Hier die entsprechende Stelle:

14 Andere Bezeichnung für File Mapping, MMF.
15 Zu deutsch: Zwischenablage

```
WM_CHANGECBCHAIN:
  try
  finally
    if Windows.HWND(wParam) = NextCBViewer then
      NextCBViewer := Windows.HWND(lParam);
    if NextCBViewer <> 0 then
      SendMessage(NextCBViewer, uMsg, wParam, lParam);
  end;
```

Wie man sehen kann (oder auch nicht) wird getestet, ob das Fenster, welches uns als nächster Clipboard-Viewer bekannt ist, aus der Kette von Viewern entfernt wurde. Sollte dies der Fall sein, aktualisieren wir unser Handle des Fensters für den nächsten Clipboard-Viewer.

Data Copy

Message: `WM_COPYDATA`

Dies ist die vielleicht leichteste der hier beschriebenen Methoden. Hierbei wird über `SendMessage()` die Nachricht `WM_COPYDATA` an die Anwendung geschickt, welche die Daten erhalten soll. Zuvor wird `wParam` mit dem Handle des sendenden Fensters bestückt, und `lParam` erhält einen Pointer zu einem Record vom Typ `COPYSTRUCT`. Diese Struktur enthält einen DWORD-Wert, welcher allein kopiert werden kann, sowie einen Pointer zu Daten, die geschickt werden sollen, sowie eine Größenangabe zu den Daten, auf die der Pointer zeigt. Im Prinzip sind alle diese Mitglieder des Records optional zu befüllen.
Um den Rest, also das Kopieren eines Speicherbereiches von einem Prozess in den anderen, kümmert sich Windows intern. Davon sieht der Programmierer nichts, weshalb diese Methode auch so einfach ist.
Es gibt ein paar Kleinigkeiten zu den Daten zu beachten:

- Die Daten selbst dürfen keine Pointer enthalten, auf die der Empfängerprozess nicht zugreifen kann (das schließt eigentlich beliebige Pointer ein, außer Views eines File Mappings - s.u.)
- Während des Sendens der Daten darf kein anderer Thread des Senderprozesses die Daten verändern. Das Senden ist vorbei, sobald `SendMessage()` zurückkehrt!
- Der Empfängerprozess sollte die Daten als nur-lesbar behandeln. Der Pointer ist nur innerhalb der Nachrichtenbearbeitung gültig. Will der Empfänger danach auf die Daten zugreifen, so muss er sie in einen eigenen Puffer kopieren!

File Mapping

Funktionen: `CreateFile(), CreateFileMapping(), OpenFileMapping(), CloseHandle(), MapViewOfFile(), UnmapViewOfFile()`

File Mapping, auch Memory Mapped Files (MMF) genannt, hat etwas mit Paging zu tun. Das ist der Mechanismus, der durch Festplattenspeicher und Auslagerung auf diesen versucht, die Benutzung des RAM „effizienter“ zu gestalten. Wenn man so will, sind MMFs das Komplement von Paging, da Dateien in den Speicher abgebildet werden, statt wie beim Paging umgekehrt der Speicher in eine Datei. MMFs gibt es in zwei Geschmacksrichtungen: a.) eine Variante, die eine echte darunter liegende Datei als Grundlage hat, und b.) eine Variante, die sich auf den Paging-Mechanismus stützt, und quasi die Auslagerungsdatei als Grundlage hat.
Ersteres bietet Vorteile für Datenbanken und andere Anwendungen, die riesige Datenbestände haben, deren Größe aber die des RAM übersteigt. Es ist nämlich möglich, eine Datei stückchenweise in den Speicher abzubilden!
Letzteres wird oft als Möglichkeit für IPC benutzt, da MMFs Kernelobjekte[16] sind und damit wie auch viele andere Kernelobjekte (z.B. Event, Mutex) aus verschiedenen Prozessen heraus angesprochen werden können. Da ansonsten die Speicherbereiche von Prozessen strikt getrennt werden, ist dies eine Möglichkeit, einen „verteilten“ Speicherbereich einzurichten. MMFs der zweiten Sorte haben immer Namen innerhalb des Objekt-Namensbereiches des Kernels, *es kann daher keine zwei MMFs mit dem gleichen Namen geben*!
Wird eine EXE-Datei oder DLL geladen, so geschieht dies über Sections (siehe [14]). Werden DLLs, die schon von einem anderen Prozess geladen wurden, in einen weiteren Prozess geladen, so teilen sich beide Prozesse in Wirklichkeit das selbe Abbild der gleichen DLL! Dies geht solange, bis ein Prozess beispielsweise versucht, seinen Teil der DLL zu patchen - erst dann wird die Speicherseite, welche gepatcht wurde, vom System für den patchenden Prozess anders dargestellt als für alle

16 Im Kernelkontext nennt man sie Sections.

anderen. Diesen Modus nennt man Copy-On-Write. Das aber nur nebenbei ;-)

Da wir diese Methode für unsere Zwecke verwenden wollen, hier ein kleines Beispiel mit Erklärung. Alle wichtigen Stellen aus der Erklärung sind im Quellcode gelb hinterlegt.

Serverseitig erstellen wir die MMF, wie man sich schon denken kann, mit einem Aufruf von `CreateFileMapping()`. Dies muss nur einmal geschehen - bei zukünftigen Zugriffen kann man die MMF simpel per `OpenFileMapping()` öffnen. `CreateFileMapping()` schadet aber auch nicht; sollte die MMF bereits vorhanden sein, so wird sie einfach geöffnet und das Handle zurückgegeben:

```
procedure OpenServer(hwnd: HWND);
begin
  SendMessage(hwnd, WM_SETTEXT, 0, Integer(@servername[1]));
  SetProp(hwnd, propname, 123);
  hMap := CreateFileMapping(INVALID_HANDLE_VALUE, nil, PAGE_READWRITE, 0, bufsize + 1,
    mmfname);
  if hMap <> 0 then
    MapView := MapViewOfFile(hMap, FILE_MAP_READ, 0, 0, 0);
  if Assigned(MapView) then
    SetDlgItemText(hwnd, IDC_EDIT1, 'Here the text will be received')
  else
    SetDlgItemText(hwnd, IDC_EDIT1, 'Failed to create MMF');
end;
```

Clientseitig kann die MMF wahlweise mit einer der beiden o.g. Funktionen geöffnet werden.
In beiden Fällen, also auf Server und auf Client, muss nun ein so genannter View[17] des Mapping-Objektes erstellt werden. Dazu bemühen wir `MapViewOfFile()`, und zum Schließen entsprechend `UnmapViewOfFile()`:

```
procedure FindAndOpenServerConnection(hwnd: HWND);
var
  s: string;
begin
  SendMessage(hwnd, WM_SETTEXT, 0, Integer(@clientname[1]));
  EnumWindows(@FindServer_EnumFunc, 0);
  if IsWindow(TargetWnd) then
    s := Format('Text you enter here will be sent.'#13#10 +
      'Found server with window handle: %8.8X', [TargetWnd])
  else
    s := 'Found no server window. Start the server part first.';
  hMap := OpenFileMapping(FILE_MAP_WRITE, True, mmfname);
  if hMap <> 0 then
    MapView := MapViewOfFile(hMap, FILE_MAP_WRITE, 0, 0, 0);
  if Assigned(MapView) then
    s := s + #13#10 + 'Successfully opened MMF'
  else
    s := s + #13#10 + 'Failed to open MMF';
  SetDlgItemText(hwnd, IDC_EDIT1, @s[1]);
end;
```

Nun haben wir einen Pointer auf die MMF innerhalb unseres Speicherbereiches und können darauf zugreifen wie auf andere Speicherbereiche, die mit `GlobalAlloc()`, `GetMem()` oder `GetMemory()` alloziert wurden:

```
function ReadServerMessage(hwnd: HWND; umsg: UINT; wParam: WPARAM; lParam: LPARAM): BOOL;
begin
  result := False;
  if Assigned(MapView) then
  begin
// Show the result
    SetDlgItemText(hwnd, IDC_EDIT1, MapView);
    result := True;
  end;
end;
```

Es ist nicht nötig, innerhalb eines Prozesses den View oder gar die Referenz zur MMF komplett zu

17 View ist Englisch für „Ansicht“, „Blick“

schließen. Stattdessen kann man bequem die ganze Zeit auf den einen zurückgegebenen Pointer zurückgreifen. In unserem Falle schreibt der Client dorthin und der Server liest daraus.

Mailslots

Funktionen: `CreateFile(), CloseHandle(), CreateMailslot(), GetMailslotInfo(), SetMailslotInfo(), ReadFile(), WriteFile()`

Mailslots[18] sind eine weitere Form von Einweg-Kommunikation. Dabei legt der Sender im Mailslot des Empfängers die Daten ab, die er senden möchte. Der Empfänger holt sie dann dort ab. Dazu sollte er natürlich idealerweise informiert werden, dass neue Daten da sind.
Da Mailslots über UDP, also ein Netzwerkprotokoll, implementiert sind, können sie erstens zwischen zwei Rechnern verwendet werden und zweitens kann die Übertragung schiefgehen! Es gibt für den Sender keine Möglichkeit festzustellen, ob die Nachricht beim Empfänger ankam.
Es hat etwas von Postwurfsendungen: der Absender (z.B. eine Firma) gibt die Blättchen weiter an einen Austräger und hofft, dass dieser sie einwirft. Wenn er sie nicht einwirft, sondern sie stattdessen in einem Papiercontainer landen, so entzieht sich das üblicherweise dem Wissen des Absenders.
Wenn man wissen will, ob die Übertragung geklappt hat, sollte man Named Pipes benutzen. Es kommt aber immer auf die Idee an, die hinter der Datenübertragung steht: will man beispielsweise Daten eines Hooks auf einen anderen Rechner oder mehrere Rechner übertragen, so bieten sich Mailslots an, da sonst (aufgrund des Netzwerk-Timeouts) lange Wartezeiten innerhalb des Hooks entstehen können, und da es „nur" um Informationsweiterleitung geht. Will man hingegen „Befehle" weiterleiten (z.B. Start eines Daemons auf dem Zielrechner), sollte man schon über das Ergebnis informiert sein. Aus diesem Grunde bieten sich Named Pipes an, da sie bidirektional und nur mit einem Zielrechner arbeiten! Außerdem kann eine Pipe verschiedene Sicherheits-APIs verwenden, die stark in Richtung RPC gehen - so zum Beispiel die API um eine Pipe zu „personifizieren"[19]: `ImpersonateNamedPipeClient()`. (siehe Pipes)

Pipes

Funktionen: `CreatePipe(), CreateNamedPipe(), CreateFile(), ReadFile(), WriteFile()`

Zuallererst, es gibt zwei Typen von Pipes[20]: Named Pipes[21] und Anonymous Pipes[22]. Erstere bieten die Möglichkeit, über das Netzwerk und auch zwischen zwei Prozessen auf einem lokalen Rechner zu kommunizieren. Leider gibt es sie nur auf der NT-Plattform[23]. Anonymous Pipes werden oft im Zusammenhang mit dem Umleiten der Ausgabe einer Konsolenanwendung benutzt. Sie gibt es auf allen Windows-Systemen ab Windows 95. Da wir über die Prozessgrenze hinweg kommunizieren wollen, müssen wir in Kauf nehmen, dass wir uns auf Named Pipes zu beschränken haben. *Damit ist das anhängige Beispiel auch nur auf der NT-Plattform nachvollziehbar*!
Im Grunde kann man schon ein Beispiel wie unseres über das Netzwerk mit einem Windows 9x/Me Rechner nachvollziehen, denn clientseitig muss man die Pipe nur mit `CreateFile()` öffnen. Jedoch beschränke ich mich bei dem Beispiel explizit auf NT, da das Beispiel ja nur auf dem lokalen Desktop zwischen zwei Prozessen funktionieren soll.
Named Pipes funktionieren wie alle Pipes als bidirektionales Kommunikationsmedium, im Gegensatz zu Mailslots also nicht nur als Einbahnstraße. Dadurch kann der Senderprozess auch vom Empfängerprozess Nachricht erhalten, ob alles angekommen ist. Innerhalb von NT-Domänen wird starker Gebrauch von Pipes gemacht! Ein Beispiel ist der Nachrichtendienst (Messenger Service).

18 Mailslot bedeutet etwa „Postfach"
19 Damit kann der Serverprozess der Pipe den Sicherheitskontext des Clients annehmen. Beispielsweise könnte am Clientende ein Administrator eingeloggt sein, am Serverende aber nur ein Gast. Dann wäre es für den Serverprozess möglich, die Rechte des Administrators zu erhalten. Meist wird dieses Beispiel aber eher umgekehrt zutreffen.
20 Pipe bedeutet „Rohr", „Leitung"
21 Benannte Pipes - `CreateNamedPipe()`
22 Anonyme Pipes - `CreatePipe()`
23 Dabei meine ich Windows NT 4, Windows 2000, Windows XP, Windows 2003 (und zukünftige)

Globale Hooks ... zum Zweiten

Im letzten Kapitel haben wir die Grundlagen von Memory Mapped Files sowie einige andere Methoden für IPC kennen gelernt. Kommen wir nun zu einer Praxis-Anwendung. Ich habe dazu schon bei der ersten Ausgabe dieses Tutorials eine kleine nonVCL-Beispielanwendung geschrieben. Ich habe sie sinnigerweise „Captain Hook“ genannt.

Captain Hook ist eine Anwendung, die je einen globalen Maus- und Tastaturhook anschaulich darstellt. Da die (globalen) Hooks sowieso, wie schon gelernt, in einer DLL implementiert werden müssen, braucht es auch nicht zu stören, dass die Anwendung eine nonVCL-Anwendung ist. Wer sich den Quelltext dennoch näher ansehen will, ist herzlichst dazu eingeladen. Allerdings wird sich schnell Ernüchterung einstellen, da er weitestgehend aus Konvertierung von Werten (des Hooks) in Strings und deren Anzeige besteht. Also einfach Visualisierung der Vorgänge bei Hooks.

Um das Programm kompakt zu gestalten, ist die DLL als Ressource in die EXE eingebunden - wer mehr zu diesem Thema erfahren möchte, möge bitte in meinem DLL-Tutorial im Kapitel „*DLLs Huckepack – DLLs als binäre Ressource*“ nachschauen.

Kommen wir nun zum eigentlichen Quelltext. Ich habe ihn so modular wie möglich gestaltet, um Wiederverwertbarkeit und auch bessere Lesbarkeit zu gewährleisten. Es gibt eine besondere Text-Datei, die per `$INCLUDE`-Direktive eingebunden wird und sowohl für die DLL als auch für die Anwendung an sich wichtig ist. Diese Datei enthält ein paar Variablen, sowie eine Deklaration der Datenstruktur `TDLLData`, die in der MMF abgelegt wird:

```
const
  MMFName = 'HookingSampleDLL_Data';

type
  TDLLData = packed record
    wnd:HWND;
    ProcessID,
    ThreadID:DWORD;
    nCode: Integer;
    WM_MOUSEHOOKMSG,
      WM_KEYBHOOKMSG: UINT;
    mouse: TMOUSEHOOKSTRUCT;
  end;
  PDLLData = ^TDLLData;

var
  lpView: PDLLData;
  hMap: THandle;
```

Die meisten Namen sollten selbsterklärend sein. Die Konstante ist der Name der MMF, mit Hilfe derer beide (Anwendung und auch DLL) auf die Daten zugreifen - die DLL schreibend und die Anwendung lesend. Dies ist einer der Gründe, warum ich keine Schutzmechanismen wie Mutexe et cetera eingebaut habe. Will man von mehreren Stellen aus schreibend auf eine Ressource zugreifen, so muss man, genauso wie beim Multithreading, den Zugriff synchronisieren[24]. So einen Aufwand brauchen wir hier nicht zu betreiben. Ein zweiter Grund den Aufwand nicht zu betreiben ist, dass der Server (serverseitig = „Captain Hook“) ja von der DLL immer „gesagt“ bekommt, wann er die Daten zu lesen hat. Die DLL, welche wir schreiben, bietet drei exportierte Funktionen an:

- `InstallHooks()`
  installiert beide Hooks - also Maus- und Tastaturhook.
- `UninstallHooks()`
  entfernt die Hooks wieder.
- `ReturnView()`
  gibt den Pointer auf den View unserer MMF zurück.

Sie wird dynamisch von Captain Hook geladen. Da die DLL Huckepack als Ressource in der EXE

24 Grund könnte z.B. eine Datenbank sein, die von zwei Anwendungen beschrieben wird. Nehmen wir an, gegeben sei ein Datensatz, und Anwendung 1 liest gerade die einzelnen Felder. Dann könnte es sein, dass Anwendung 2 die Daten zur gleichen Zeit überschreibt, oder gar den Datensatz löscht. Dies gilt es zu vermeiden, deshalb Synchronisation. Dazu gibt es so genannte Exclusion Objects (Ausschlussobjekte).

sitzt, wird sie zuallererst in ein temporäres Verzeichnis geschoben und von dort geladen. An dieser Stelle holen wir uns auch die Adressen der exportierten Funktionen. Serverseitig ist der Ladevorgang damit auch schon quasi abgeschlossen - es wird dann der Hauptdialog angezeigt. Clientseitig wird (= Hook-DLL) schon während des Ladens der DLL einiges getan. Schauen wir uns also erst einmal an, was die DLLMain() unserer DLL zu tun hat:

```
procedure DLLMain(dwReason: DWord);
begin
  case dwReason of
    DLL_PROCESS_ATTACH:
// Create the MMF to share data
      if CreateMMF then
      begin
// Register our private window messages
        lpView^.WM_MOUSEHOOKMSG := RegisterWindowMessage(mousmsg);
        lpView^.WM_KEYBHOOKMSG := RegisterWindowMessage(keybmsg);
      end
      else
        FreeLibrary(hInstance);
    DLL_PROCESS_DETACH:
// Free the MMF
      FreeMMF;
  end;
  if Assigned(DLLProcNext) then DLLProcNext(dwReason);
end;
```

Lädt ein Prozess die DLL, wird zuallererst die MMF erzeugt und der Pointer zum View der globalen Variablen `lpView` zugewiesen. `lpView` wird dann sogleich mit den Werten für die von uns registrierten Fensternachrichten[25] gefüllt. Die Nachrichten werden von der DLL verwendet, um die Daten zu schicken, und deshalb muss die Anwendung natürlich auch deren Werte kennen.
Wird die DLL aus dem Prozess entladen, gibt er die MMF frei.

Klickt man nun im Hauptdialog den Button zum Installieren der Hooks, wird von der Anwendung folgende exportierte Routine aus der DLL aufgerufen:

```
function InstallHooks(Hwnd: Cardinal): Boolean; stdcall;
var mouseh, keybh: boolean;
begin
  Result := False;
  if CreateMMF then
  try
    keybh := false;
    mouseh := false;
// First install the hooks
    case Mouse_HookHandle of
      0: begin
// Install the mouse hook
          Mouse_HookHandle := SetWindowsHookEx(WH_MOUSE, @MouseHookProc, hInstance, 0);
// Check for success
          mouseh := Mouse_HookHandle <> 0;
        end;
    end;
    case Keyboard_HookHandle of
      0: begin
// Install the keyboard hook
          Keyboard_HookHandle := SetWindowsHookEx(WH_KEYBOARD, @KbdHookProc, hInstance,
0);
// Check for success
          keybh := Keyboard_HookHandle <> 0;
        end;
    end;
// Save the given window handle
    lpView^.wnd := Hwnd;
  finally
    Result := keybh and mouseh;
  end;
end;
```

25 Man kann Fensternachrichten systemweit registrieren. Diese sind garantiert einzigartig während einer Session. Man könnte auch konstante Werte nehmen, allerdings birgt dies die Gefahr, dass ein anderes Fenster unsere Nachricht missversteht. Das soll so vermieden werden.

Schauen wir uns der Reihe nach die markierten Codezeilen an. `CreateMMF` stellt sicher, dass die MMF existiert und ein View geöffnet ist. Als erstes installieren wir dann den Maushook, danach den Tastaturhook - mithilfe eines Aufrufes von `SetWindowsHookEx()`. Als letztes wird das Fenster gesetzt, an welches unsere Nachrichten (die während der Initialisierung registrierten) geschickt werden sollen. Dies dürfte wohl meist das Anwendungsfenster sein - in unserem Fall zumindest ist es das. Als Resultat wird zurückgegeben, ob die Hooks installiert werden konnten.
Schauen wir uns der Vollständigkeit halber noch den Code zum Entfernen der Hooks an:

```
function UninstallHooks: Boolean; stdcall;
var mouseh, keybh: Boolean;
begin
//Uninstall hook from hooks chain
  mouseh := UnhookWindowsHookEx(Mouse_HookHandle);
  keybh := UnhookWindowsHookEx(Keyboard_HookHandle);
  Mouse_HookHandle := 0;
  Keyboard_HookHandle := 0;
  Result := keybh and mouseh;
  if Result then
    lpView^.wnd := 0;
end;
```

Dabei werden die Hooks einfach durch den Aufruf von `UnhookWindowsHookEx()` sowie Übergabe der vorher erhaltenen `HHOOK`-Handles entfernt. Als letztes wird dann noch das Zielfenster auf den Wert Null zurückgesetzt.

Wie wir weiter oben schon gelernt haben, müssen wir eine Callbackfunktion angeben, wenn wir einen Hook installieren. Wie die beiden für unseren Maus- und Tastaturhook heißen, wissen wir ja schon: `KbdHookProc()` und `MouseHookProc()`. Diese müssen als **`stdcall`** deklariert sein und machen die eigentliche Arbeit, beispielsweise bei einem Keylogger das Speichern der Daten oder das Senden der Daten an ein Serverprogramm - bei uns trifft letzteres zu. Außerdem müssen sie (das ist ja nun schon gegeben) in einer DLL liegen, also in einem Bereich, der innerhalb verschiedener Prozesse existieren kann.

Um es nochmals zu sagen: bei globalen Hooks werden diese Callbacks aus fremden Prozessen aufgerufen, deshalb der ganze Kopfstand!!!

```
function KbdHookProc(nCode: Integer; wParam: WPARAM; lParam: LPARAM): LRESULT; stdcall;
var
  lpView: PDLLData;
begin
// It's possible to call CallNextHookEx conditional only to block information
// from the target window.
  Result := CallNextHookEx(Keyboard_HookHandle, nCode, wParam, lParam);
  case nCode < 0 of
    True: exit; // If the code is smaller than 0 nothing _has_ to be done
    False:
      begin
        lpView := OpenAndMapMMF;
        if Assigned(lpView) then
        try
// Here one can work with the parameters.
          lpView^.nCode := nCode;
          lpView^.ProcessID := GetCurrentProcessID;
          lpView^.ThreadID := GetCurrentThreadID;
// We use HWND_BROADCAST to send the message to any window which handles this WM
          SendMessage(HWND_BROADCAST, lpView^.WM_KEYBHOOKMSG, wParam, lParam);
        finally
          CloseAndUnmapMMF(lpView);
        end;
      end;
  end;
end;
```

Wie wir sehen wird bei uns sofort der Status an den nächsten Hook in der Hookchain weitergereicht: `CallNextHookEx()`. Dabei müssen das Handle zum aktuellen Hook, sowie die

empfangenen Statuswerte weitergereicht werden. Will man anderen Hooks und vor allem dem Fenster die vom Hook abgefangene Eingabe vorenthalten, so darf man `CallNextHookEx()` nicht aufrufen. So wird die Hookchain von uns quasi unterbrochen.

Wir wollen dies nicht. Nachdem wir die Daten weitergereicht haben, können wir uns an die Auswertung machen. Da die Verarbeitung und das Weitersenden selbst Zeit kosten, ist es notwendig, sich an die von Windows vorgegebenen Konventionen zu halten[26]. Die besagen, dass man wenn der Wert von `nCode` kleiner als Null ist keinerlei Bearbeitung der Daten vornehmen darf.

Ist der Wert von `nCode` größer als Null, so beginnen wir die Datenverarbeitung. Mit einem Aufruf von `OpenAndMapMMF()` wird dazu zuerst die MMF geöffnet, welche die Daten enthalten wird (sie wurde ja bekanntlich während der Initialisierung schon vorbereitet). Sollte dieser Aufruf erfolgreich sein, also der View ungleich `Nil`, dann wird ein Exception-Handler eingesetzt, der als Abschlusshandlung die MMF wieder schließt. Da wir innerhalb verschiedener Prozesse sicher auch verschiedene Werte für den View zurückbekommen, sollte man davon ausgehen, dass wir dieses Prozedere immer und immer wieder (eben bei jedem Aufruf der Callbackfunktion) machen müssen ... sollte man! Jedoch habe ich bei Tests herausbekommen, dass, verwendet man die DLL-globale Variable lpView aus verschiedenen Prozessen heraus, der gleiche Speicher mit der gleichen Adresse angesprochen werden kann[27] (insofern es sich um den View einer MMF handelt).

Ich hielt es aber für robuster, den Code so zu gestalten wie er oben steht, da meiner Meinung nach die Gefahr besteht, dass die DLL in einem anderen Prozess vielleicht nicht an der gleichen Stelle geladen wurde - was konsequenterweise bedeuten müsste, dass die MMF ebenfalls an einer anderen Stelle in einen anderen Prozess geladen werden kann. Um beide Möglichkeiten dennoch einfach zur Verfügung zu stellen, habe ich ein paar Compiler-Bedingungen an den entsprechenden Stellen eingefügt.

Die folgenden Zuweisungen vor dem Aufruf von `SendMessage()` kopieren die Daten in die MMF - hier unter anderem die PID und die TID. Die Orginalwerte von `lParam` und `wParam` des Hooks werden als `lParam` und `wParam` an `SendMessage()` übergeben. Deshalb ist unsere Callbackfunktion hier auch so einfach. Beim Maushook wird es etwas mehr Arbeit:

```
function MouseHookProc(nCode: Integer; wParam: WPARAM; lParam: LPARAM): LRESULT; stdcall;
{$IFNDEF USEONLYLPVIEW}
var
  lpView: PDLLData;
{$ENDIF}
begin
// It's possible to call CallNextHookEx conditional only to block information
// from the target window.
  Result := CallNextHookEx(Mouse_HookHandle, nCode, wParam, lParam);
  case nCode < 0 of
    True: exit; // If the code is smaller than 0 nothing _has_ to be done
    False:
      begin
{$IFNDEF USEONLYLPVIEW}
        lpView := OpenAndMapMMF;
        if Assigned(lpView) then
        try
{$ENDIF}
// Here one can work with the parameters.
          lpView^.nCode := nCode;
          lpView^.ProcessID := GetCurrentProcessID;
          lpView^.ThreadID := GetCurrentThreadID;
// Here we copy the data into our MMF to provide the data to our application.
          CopyMemory(@lpView^.mouse, PMOUSEHOOKSTRUCT(lParam), sizeof(TMOUSEHOOKSTRUCT));
// We use HWND_BROADCAST to send the message to any window which handles this WM.
          SendMessage(HWND_BROADCAST, lpView^.WM_MOUSEHOOKMSG, wParam, lParam);
{$IFNDEF USEONLYLPVIEW}
        finally
          CloseAndUnmapMMF(lpView);
        end;
{$ENDIF}
      end;
  end;
end;
```

26 Als „böser“ Programmierer, der die Eingabe blockieren will, kann man (nach reiflicher Überlegung) durchaus auch mal eine Ausnahme machen und die Auswertung von `nCode` stecken lassen.

27 Wurde auf Windows 95B und Windows 2000 Pro SP3 getestet und funktionierte.

Hier sieht man auch sehr schön die von mir oben bereits erwähnte Compiler-Bedingung `USEONLYLPVIEW`. Für die Callbackfunktion des Tastaturhooks sieht es analog aus. Außer nCode und PID und TID zu setzen, wird hier noch Speicher kopiert. Da bei Maushooks `lParam` auf eine etwas komplexere Struktur (i.e. Record) vom Typ `TMOUSEHOOKSTRUCT` zeigt, können wir nun nicht einfach `lParam` übergeben, da im Empfängerprozess (i.e. Captain Hook) diese Adresse sonstwas sein könnte. Stattdessen kopieren wir den Inhalt der `TMOUSEHOOKSTRUCT` in unsere MMF an die richtige Stelle, so dass der Empfängerprozess damit auch etwas anfangen kann. Dazu wird einfach `CopyMemory()` benutzt.

Nun da wir uns beide Callbacks angeschaut haben, kommen wir zu der Gemeinsamkeit: der Aufruf von `SendMessage()`. Schauen wir uns beide Aufrufe nochmal genauer an:

```
SendMessage(HWND_BROADCAST, lpView^.WM_KEYBHOOKMSG, wParam, lParam);

SendMessage(HWND_BROADCAST, lpView^.WM_MOUSEHOOKMSG, wParam, lParam);
```

Uns fällt sofort ein Wert namens `HWND_BROADCAST` ins Auge. Dies ist ein in den Headern, beziehungsweise der `Windows.pas`, vordeklarierter Wert, der für `SendMessage()` eine ganz besondere Bedeutung hat. `SendMessage()` wird damit angewiesen, die angegebene Nachricht an alle Top-Level-Fenster des Systems[28] zu schicken. Da unsere Fensternachrichten ja innerhalb der aktuellen Session einzigartig sind und nur Captain Hook deren Bedeutung kennt, wird auch nur Captain Hook die Nachricht verarbeiten.
Rein prinzipiell sollte auch `PostThreadMessage()` and Stelle von `SendMessage()` mit `HWND_BROADCAST` funktionieren.

Anekdote
Ich habe übrigens auch versucht, den Code für globalen Hook mit in die EXE zu packen und die entsprechenden gleichnamigen Funktionen zu exportieren[29]. Danach habe ich die EXE sich einfach in ein temporäres Verzeichnis kopieren lassen, um sie von dort zu laden. Um dem Loader die richtige Spur zu geben, habe ich vorher sogar den PE-Header so angepasst, dass die Datei als DLL durchgehen würde. Leider aber sind es die Restriktionen von Delphi, die solche Spielereien ohne Modifikation der System-Units unmöglich machen. Das missglückte Experiment liegt nun aber als RAR-Archiv bei den Quelltexten. Wer möchte, darf sich gern dran versuchen. Über Erfolgsmeldungen würde ich mich sehr freuen.

Keylogger
Wollte man beispielsweise „nur“ die Tastendrücke in einer Datei speichern, so könnte man dies direkt in der Callbackfunktion tun! Es wäre also nicht mehr nötig, Daten an eine Serveranwendung zu schicken, sondern man könnte ohne Umweg und ohne Prozessgrenzen zu überschreiten die Daten speichern. Ein kleines Beispielprojekt dazu gibt es bei den Quelltexten unter Keylogger.

Weitere Anwendungen für Hooks

Der Möglichkeiten gibt es, wie immer, viele. Ich möchte mich hier auf zwei populäre Verwendungszwecke beschränken. API-Hooking durch Code-Injektion mit Fenster-Hooks und die alte Kiste mit dem „Wie kann ich auf Tastendruck in einer anderen Anwendung was auslösen“ ...

[Diese Sektion ist in Bearbeitung. Aufgrund der starken Nachfrage nach Informationen zu Hooks, habe ich mich jedoch entschlossen die Beta2 erst einmal ohne sie zu veröffentlichen!]

28 So steht es im Platform PSDK. Ich vermute aber, dass es nur an alle Fenster in der gleichen WindowStation oder sogar nur im gleichen Desktop geht.
29 Das ist auch bei EXE-Dateien wie bei DLLs erlaubt.

Referenzen

Name	Kontaktadresse, ISBN oder URL
Coder-Area	http://www.coder-area.de
Delphi Forum (AUQ)	http://www.delphi-forum.de
Delphi-Groups-Forum	http://www.delphi-groups.de/YaBBSe
Delphi Praxis	http://www.delphipraxis.net
Delphi-Source.de	http://www.delphi-source.de
Delphi-Treff.de	http://www.delphi-treff.de
Entwickler-Forum	http://www.entwickler-forum.de
LCC & WEDITRES	http://www.cs.virginia.edu/~lcc-win32 Enthalten in LCC für Win32
OpenOffice.org	http://www.openoffice.org
Pdf-Factory (& Pro)	http://www.fineprint.com
Platform SDK	http://www.microsoft.com/msdownload/platformsdk/sdkupdate
Spotlight Delphi Forum	http://spotlight.de/nzforen/dlp/t/forum_dlp_1.html

Viel Spaß mit
der Programmierung in Delphi
und viel Erfolg,

wünscht Dir

Oliver aka Assarbad