Labor Embedded Control

Einführung in die Treiberprogrammierung mit dem

Kernel-Mode Driver Framework

Autor : Tom Schreiber | tom@tomhost.de
Datum : 26. April 2007 | Version 1 Revision 4

Inhaltsübersicht

Einleitung.... 3
1. Voraussetzungen.... 4
2. Das Kernel-Mode Driver Framework.... 4
  2.1 Überblick.... 4
  2.2 Aufbau eines KMDF Treibers.... 4
  2.3 Die KMDF Objekte.... 5
  2.4 Synchronisation.... 6
  2.5 Kompilieren eines KMDF Treibers.... 7
3. Signieren und Ausliefern von KMDF Treibern.... 7
  3.1 Benötige Dateien.... 7
  3.2 Ein Testzertifikat erstellen und installieren.... 8
  3.3 Den Treiber signieren.... 9
  3.4 Besonderheiten unter der 64bit Version von Windows Vista.... 9
4. Debuggen von Kernel-Mode Treibern.... 10
  4.1 Einrichten einer Debugumgebung.... 10
  4.2 Verwenden von WinDbg zum Debuggen des Kernels.... 10
  4.3 Meldungen an den Debugger senden (KdPrintEx).... 12
  4.4 Weitere Debugeinstellungen und -möglichkeiten.... 13
5. Software Tracing.... 14
  5.1 Einführung.... 14
  5.2 Tracing Unterstützung hinzufügen.... 14
  5.3 Unterstützung für Tracing-Level hinzufügen.... 16
  5.4 Nachrichten aufzeichnen, anzeigen und filtern.... 17
6. Windows Management Instrumentation.... 18
  6.1 Einführung.... 18
  6.2 Vorbereitung zur WMI Unterstützung.... 18
  6.3 Den Treiber als WMI-Provider registrieren.... 20
  6.4 WMI-Daten abrufen und ändern.... 21
  6.5 WMI Methoden.... 22
  6.6 WMI Leistungsindikatoren.... 23
  6.7 WMI Events.... 24
  6.8 Tools und Beispiele zur WMI-Nutzung.... 25
7. Der ECHO Treiber.... 28
  7.1 Einführung.... 28
  7.2 Installation des Beispieltreibers.... 29
  7.3 Funktionalität des Treibers.... 29
  7.4 Die Treiber DLL.... 30
  7.5 WMI Funktionalität des Treibers.... 30
  7.6 EchoDevice Testprogramm.... 31
  7.7 EchoWatch.... 32
8. Quellverzeichnis und weiterführende Informationen.... 33
9. Downloadadressen.... 33

Einleitung

Das vorliegende Dokument gibt einen groben Überblick über das Kernel-Mode Driver Framework, welches mit Windows Vista eingeführt wurde. Das Kernel-Mode Driver Framework basiert auf den Windows Driver Modell (WDM) und stellt Schnittstellen und Funktionen bereit, welche die Treiberentwicklung erheblich vereinfachen. Diese Dokument dient als Einführung in die Thematik der Treiberentwicklung unter Windows mit dem neuen Treibermodell, welches neben Windows Vista alle Windows Betriebssysteme ab Windows 2000 unterstützt. Es wird der grundlegende Aufbau eines Treibers nach diesem Modell erläutert und an Beispielen anschaulich erklärt. Weiterhin wird gezeigt wie man diese Treiber für Testzwecke signiert und mithilfe vom Debugger testet. Außerdem wird die Unterstützung von Windows Management Instrumentation und Software Tracing seitens des Treibers beschrieben und auch wie man diese Technologien aus dem User-Mode heraus verwenden kann. Diese Dokument erhebt keinen Anspruch auf Vollständigkeit und soll nur als Einführung in das Thema dienen. Es sei hier auf die Dokumentation des Windows Driver Kit [dok] verwiesen, die in allen Punkten erweiterte und vertiefende Informationen zu den einzelnen Punkten enthält. Weiterhin ist die Internetseite von OSROnline [osr] eine gute Anlaufstelle für alle Treiberentwickler unter dem Betriebssystem Windows.

1. Voraussetzungen

Um einen Treiber mit dem Kernel-Mode Driver Framework zu entwickeln, benötigt man zunächst das aktuelle **Windows Driver Kit [dl-wdk].** Weiterhin ist es sehr empfehlenswert einen zweiten Rechner zum Testen der Treiber zur Verfügung zu haben, da dies die Fehlersuche sehr erleichtert. Zum Debuggen sollten die **Debbuging Tools for Windows [dl-dbg]** installiert sein. Für das Debuggen müssen weiterhin die passenden **Symbole** für das Testsystem vorhanden sein, welche man entweder herunterladen **[dl-sym]** kann oder bei Bedarf über den Symbolserver von Microsoft erhält (siehe hierzu Abschnitt 4). Die Installation des entsprechenden **Windows Software Development Kit [dl-sdk]** bietet sich an, wenn man noch eine zusätzliche Anwendung oder DLL für den Treiber schreiben möchte. Das SDK bietet neben der Dokumentation der Windows API weitere nützliche Tools.

2. Das Kernel-Mode Driver Framework

2.1 Überblick

Das Kernel-Mode Driver Framework stellt eine Bibliothek zur Verfügung, welche zum Erstellen von Treibern verwendet werden kann. Das Framework wurde mit dem Betriebssystem Windows Vista eingeführt und ist Bestandteil des Windows Driver Kit ab der Version 6. Treiber die mit dem Framework programmiert wurden können aber nicht nur Windows Vista, sondern auch in älteren Versionen des Betriebssystems ab Windows 2000 verwendet werden. Bei Neuentwicklungen von Treibern empfiehlt Microsoft deshalb die Programmierung mit dem Kernel-Mode Driver Framework.

Das Kernel-Mode Driver Framework arbeitet objekt- und ereignisorientiert. Ein Objekt des Framework verfügt über Objekteigenschaften und Objektmethoden. Über die Objektmethoden kann ein Treiber auf die bereitgestellten Funktionen eines Objekt zugreifen. Die Objekteigenschaften können über Get- und Set-Methoden verändert werden. Es gibt allerdings auch WDM-Objekte, auf diese wird direkt über die Objektstruktur zugegriffen, beispielsweise die Geräteobjekte. Weiterhin stellt ein Treiber bestimmte Ereignismethoden zur Verfügung, welche dem Framework mitgeteilt werden. Diese werden aufgerufen, wenn ein bestimmtes Ereignis eintritt, beispielsweise das Eintreffen einer E/A-Anfrage (I/O-Request) oder das Wechseln in den Standbymodus.

2.2 Aufbau eines KMDF Treibers

Um einen KMDF Treiber zu schreiben muss man zunächst zwei Headerdateien einbinden.

```
#include <ntddk.h>
#include <wdf.h>
```

Als nächstes muss der Treiber über eine **DriverEntry** Methode verfügen, welche für die Initialisierung des Treibers zwingend notwendig ist und aufgerufen wird, wenn der Treiber geladen wird. In dieser Methode wird **WdfDriverCreate** aufgerufen um ein Treiberobjekt zu erzeugen. Hier wird dem Framework außerdem mitgeteilt, welche Ereignismethode aufgerufen werden soll, wenn das entsprechende Gerät an dem Computer angeschlossen wurde. Die restliche Initialisierungsarbeit übernimmt dann diese Ereignismethode, damit benötigte Systemressourcen erst dann angefordert werden, wenn das Gerät tatsächlich vorhanden ist.

Bei der Initialisierung des eigentlichen Gerätes wird zunächst die Plug'and'Plug- und PowerManagement Funktionalität und zumeist mindestens eine Warteschlange für I/O-Anforderungen eingerichtet. Unter dem Kernel-Mode Driver Framework wurden im Gegensatz zu älteren WDM-Treibern, die PnP- und

Powermanagment-Funktionalität zusammengefasst Die entsprechenden Funktionen werden als PnPPower-Callbacks bezeichnet. (WDF_PNPPOWER_EVENT_CALLBACKS). Für die PnPPower-Funktionalität werden wiederum einige Funktionen angegebenen, welche beim Aktivieren der verschiedenen Betriebsmodi des Gerätes aufgerufen werden, beispielsweise beim Starten oder Unterbrechen des Gerätes. Die PnPPower-Funktionalität sollte nicht vernachlässigt werden, da neuere Betriebssysteme wie Windows XP oder Vista umfangreiche Möglichkeiten zum Energie sparen bereitstellen und der Treiber dies berücksichtigen bzw. unterstützen sollte.

Erhält der I/O-Manager eine Anfrage für das entsprechende Gerät, so übergibt er diese an das Framework, welches wiederum diese Anfrage in die Warteschlange des Gerätes schiebt und anschließend die Ereignismethode des Gerätes zum Behandeln einer I/O-Anfrage aufruft. Um sich einen Überblick über den grundlegenden Aufbau und der Funktionsweise eines KMDF Treibers zu verschaffen, empfehlen sich die Treiberbeispiele aus verschiedenen Gerätegruppen, welche dem Windows Driver Kit beiliegen und der minimale Beispieltreiber ECHO, welcher im Abschnitt 7 dieses Dokumentes vorgestellt wird.

2.3 Die KMDF Objekte

Im Kernel-Mode Framework gibt es eine Vielzahl von Objekten für unterschiedliche Aufgaben. Eine vollständige Übersicht aller Objekte samt Beschreibung aller Methoden und Strukturen findet man in [dok][1]. Die Wichtigsten sind das **Driver - Objekt**, das **Device - Objekt** und das **Queue - Objekt**. Die Objekte werden im Allgemeinen über eine *Create*-Methode mit den gewünschten Einstellungen erzeugt und können über einen eigenen Kontext verfügen. Bei den Einstellungen gibt es die Konfiguration und die Objektattribute, welche jeweils zunächst mittels eines Makros initialisiert und anschließend angepasst werden. Der Kontext ist eine selbst definierte Struktur, in welcher benutzerdefinierte Daten gespeichert werden. Im folgenden ist eine solche Struktur dargestellt.

```
///Der Gerätekontext, welcher zum Abspeichern relevanter Informationen dient
typedef struct _DEVICE_CONTEXT
{
       ULONG                MaxWriteLength;
       EchoWMIClass         WmiData;
       EchoPerfClass        PerfData;
       WDFWMIINSTANCE       EventInstance_BufferSizeError;
       WDFWMIINSTANCE       EventInstance_BufferSizeChanged;
} DEVICE_CONTEXT, *PDEVICE_CONTEXT;

///Definiert eine Methode zum Zugriff auf den Gerätekontext
WDF_DECLARE_CONTEXT_TYPE_WITH_NAME(DEVICE_CONTEXT, GetDeviceData)
```

Gerätekontext - Auszug aus device.h

In dieser Struktur stehen verschiedene Daten, welche beispielsweise die Konfiguration des Gerätes beschreiben und statistische Daten beinhalten könnten. Mit Hilfe des Makros **WDF_DECLARE_CONTEXT_TYPE_WITH_NAME** definert man eine Funktion, mit welcher man über die Angabe des Geräteobjektes auf die besagte Struktur zugreifen kann. In diesem Fall wird die Funktion *GetDeviceData()* für diesen Zweck definiert. Diese Struktur weist man nun dem Objekt beim Erzeugen zu.

```
//Atrribute initialisieren
WDF_OBJECT_ATTRIBUTES_INIT_CONTEXT_TYPE(&deviceAttributes, DEVICE_CONTEXT);
deviceAttributes.SynchronizationScope = WdfSynchronizationScopeDevice;

//Gerät initialisieren
status = WdfDeviceCreate(&DeviceInit, &deviceAttributes, &device);
```

1 Unter Windows Driver Foundation/Kernel-Mode Driver Framework/Reference

Mit dem Makro **WDF_OBJECT_ATTRIBUTES_INIT_CONTEXT_TYPE** initialisiert man die Attribute-Struktur des Gerätes mit dem voher definierten Kontext. Dieser Vorgang, welcher hier am Beispiel der Geräteinitalisierung gezeigt wurde, ist für die meisten Objekte ähnlich.

2.4 Synchronisation

Die Synchronisation gehört zu den größten Schwierigkeiten bei der Treiberprogrammierung. Die Notwendigkeit einer Synchronisation hat im Allgemeinen mehrere Ursachen:

- mehrere Geräte verwenden einen Treiber
- asynchrone Zugriffe aus dem User-Mode auf den Treiber, bspw. durch verschiedene Programme
- asynchrone Treiber-Aufrufe vom Gerät aus
- Threads des Treibers laufen auf verschiedenen Prozessoren

Das Kernel-Mode Framework erleichtert die Synchronisation, indem es in vielen Fällen automatisch synchronisiert. Das Framework achtet bei den *Callback*-Funktionen darauf, dass nur eine Funktion gleichzeitig aufgerufen wird. Dies wiederum gewährleistet auch, dass es beim Zugriff auf den Objektkontext keine Konflikte gibt. Bei den I/O-Requests muss bei der Initialisierung bestimmt werden, wie der Zugriff synchronisiert werden soll. Dazu wird bei der Erstellung des Objektes die entsprechende Synchronisationseinstellung in der Attributstruktur unter **SynchronizationScope** angegeben. Folgende Einstellungen sind dabei möglich:

Einstellung	Beschreibung
`WdfSynchronizationScopeDevice`	Es darf nur ein I/O-Request pro Gerät durchgeführt werden
`WdfSynchronizationScopeQueue`	Es darf ein Request pro Warteschlange (Queue) duchgeführt werden
`WdfSynchronizationScopeNone`	Es erfolgt keine Synchronisation
`WdfSynchronizationScopeInheritFromParent`	Die Synchronisationseinstellung wird vom Eltern-Objekt übernommen

Wird die Option **WdfSynchronizationScopeNone** verwendet, so muss der Treiber die Synchronisation selbst realisieren. Hierzu werden die Methoden **WdfObjectAcquireLock** und **WdfObjectReleaseLock** verwendet, welche beim Betreten eines kritischen Codeabschnittes das entsprechende Objekt für weitere Zugriffe sperren. Als weitere Synchronisationsmittel gibt es noch die Wait Locks und Spin Locks.

Mithilfe der **Wait Locks** kann man einen kritischen Bereich, bspw. der exklusive Zugriff auf einen Datenbereich, sperren. Dieses Synchronisationsmittel kann für Funktionen verwendet werden, die im IRQL=PASSIVE_LEVEL laufen. Der kritische Bereich läuft dabei ebenfalls in diesem Level. Der Wait Lock muss zuvor mit **WdfWaitLockCreate** erstellt werden und kann anschließend mit **WdfWaitLockAcquire** und **WdfWaitLockRelease** verwendet werden.

Die **Spin Locks** dienen dazu in einer die Funktion die im IRQL < DISPATCH_LEVEL läuft, einen kritischen Bereich zu markieren und diesem im IRQL=DISPATCH_LEVEL auszuführen. Dies geschieht mit den Methoden **WdfSpinLockAcquire** und **WdfSpinLockRelease.** Zuvor muss man den SpinLock ebenfalls mit **WdfSpinLockCreate** erstellen.

Verschiedene Funktionen des Framework setzen voraus, dass sie im IRQL=PASSIVE_LEVEL aufgerufen werden. Befindet man sich in einem höheren Level, sollte der Aufruf dieser Funktionen vermieden werden. Im normalen Betrieb merkt man von diesem eigentlichen fehlerhaften Aufruf nichts. Läuft der Treiber jedoch mit der **VerifierOn**-Funktion (siehe Abschnitt 4.4), so wird bei einem solchen Aufruf die Ausführung

des Systems unterbrochen. Man sollte solche Aufrufe prinzipiell vermeiden und beheben, da ein solcher Treiber auch die Tests in den *Windows Hardware Quality Labs* nicht besteht und somit kein Zertifikat erhalten würde. In einem solchen Fall sollte man das **WdfWorkItem-Objekt** verwenden, welchem man bei der Initialisierung eine *Callback*-Funktion mitteilt. In dieser Funktion sollten die Aufrufe stehen die das IRQL=PASSIVE_LEVEL voraussetzen. Anschließend wird das WorkItem mit der Methode **WdfWorkItemEnqueue** der Item-Warteschlange hinzugefügt und alsbald möglich wird die zugeordnete Funktion ausgeführt.

Ferner ist bei Funktionen die auf Interruptdaten zugreifen ebenfalls eine gesonderte spezielle Synchronisation erforderlich, weiteres dazu in [dok][2]

2.5 Kompilieren eines KMDF Treibers

Zum Kompilieren eines KMDF-Treibers sollte die Build-Umgebung des Windows Driver Kit (Version 6 oder höher) verwendet werden. Diese Build-Umgebung setzt gewisse Vorgaben zum Aufbau der Quellen, damit der Build-Vorgang automatisiert geschehen kann. Der Quellcode-Ordner enthält neben den eigentlichen Quellcode, folgende Dateien in denen die Einstellungen zum Kompilieren des Treibers stehen:

Dateiname	Beschreibung
`sources`	Gibt die Quellen des Treiber, den Treibername, die KMDF-Version und weitere Einstellungen an (bspw. Infomationen für Tracing-Präprozessor)
`makefile`	Verweis auf die Makefile.def des WDK, welche verschiedene Einstellungen definiert
`makefile.inc`	Weitere Abläufe die vor dem Kompilieren durchgeführt werden (bspw. INF-Datei anpassen, MOF-Datei umwandeln)

Wichtig sind die Optionen **KMDF_VERSION=1** und **TARGETTYPE=DRIVER** in der *sources*, um einen Kernel_Mode Treiber zu kompilieren, welcher das Framework verwendet. In den meisten Fällen sollte es ausreichen, die Beispiele aus dem WDK anzuschauen und als Vorlage zu verwenden. Alternativ kann man auch in [doc][3] nachschlagen, um die Dateien für seinen Treiber entsprechend anzupassen.

3. Signieren und Ausliefern von KMDF Treibern

3.1 Benötigte Dateien

Um einen Treiber auszuliefern benötigt man neben der eigentlichen Treiberdatei (Endung **sys**) eine **Informationsdatei** (Endung **inf**). Mithilfe dieser Datei ist es Windows möglich den Treiber konkret für ein bestimmtes Gerät zuzuordnen. Weiterhin findet man in dieser Datei Installationsinformationen, sowie den Namen und die Gerätegruppe des Treibers. Diese Informationsdatei legt man im Quellverzeichnis in einer Datei mit der Endung **inx** ab, diese wird beim Kompilieren des Treibers als Vorlage verwendet, ergänzt und liegt dann im Ausgabeverzeichnis als INF-Datei vor. In der *sources*-Datei sollten dazu die Einträge **INF_NAME** und **MISCFILES** stehen.

```
INF_NAME      =     echo
MISCFILES     =     $(OBJ_PATH)\$(O)\$(INF_NAME).inf
```

2 Unter Windows Driver Foundation/Kernel-Mode Driver Framework/Design Guide/Programming techniques for Framework-Based Drivers/Handling Hardware Interrupts/Synchronizing Interrupt Code

3 Unter *Driver Development Tools/Tools for Building Drivers/Build/Using the Build Utility/Build Utility Techniques/*

KMDF-Treiber benötigen zur korrekten Funktion auf dem Zielsystem einen **Windows Driver Foundation - CoInstaller**. In der *INF*-Datei muss deshalb die Installation und die Registrierungseinträge dieses CoInstallers eingetragen werden Den entsprechenden CoInstaller für verschiedene Architekturen des Zielsystems findet man im Unterverzeichnis **/redist/wdf** des Windows Driver Kit. Dieser sollte zusammen mit der inf- und der sys-Datei in ein leeres Verzeichnis kopiert werden, da diese Dateien letztlich für die Auslieferung notwendig sind. Im Folgenden ein Auszug aus der INF-Datei des ECHO Beispieltreibers, in dem die zusätzlichen Einträge für den KMDF-CoInstaller ersichtlich werden.

```
; Co-Installer - Windows Driver Framework

[DestinationDirs]
ECHO_Device_CoInstaller_CopyFiles = 11

[ECHO_Device.NT.CoInstallers]
AddReg=ECHO_Device_CoInstaller_AddReg
CopyFiles=ECHO_Device_CoInstaller_CopyFiles

[ECHO_Device_CoInstaller_AddReg]
HKR,,CoInstallers32,0x00010000, "wdfcoinstaller01005.dll,WdfCoInstaller"

[ECHO_Device_CoInstaller_CopyFiles]
wdfcoinstaller01005.dll

[SourceDisksFiles]
wdfcoinstaller01005.dll=1

[ECHO_Device.NT.Wdf]
KmdfService =  ECHO, ECHO_wdfsect

[ECHO_wdfsect]
KmdfLibraryVersion = 1.5
```

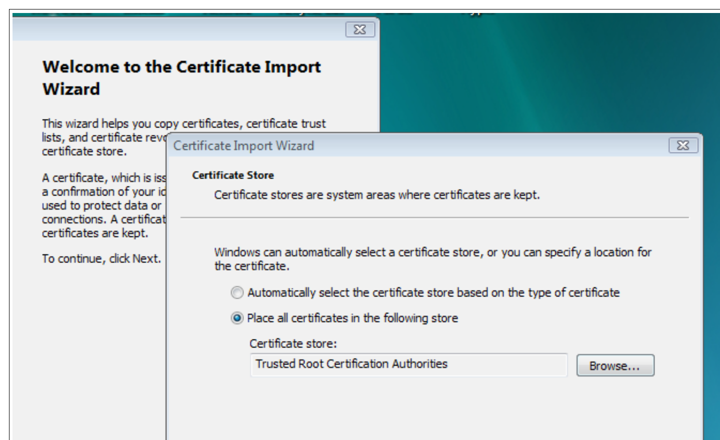
Einträge für den WdfCoInstaller - Auszug aus echo.inx

3.2 Ein Testzertifikat erstellen und installieren

Um den Treiber zu Testzwecken zu installieren, wird von Microsoft empfohlen diesen Treiber mit einem Testzertifikat zu signieren. Um ein solches Testzertifikat zu erstellen benötigt man das Tool **makecert.exe**, welches Bestandteil des Windows Driver Kit ist.

```
Makecert.exe -r -pe -ss PrivateStore -n „CN=CertName“ testcert.cer
```

Hier gibt man als Parameter den Zertifikatsspeicher, den Zertifikatsnamen und den Namen der Ausgabedatei an. Anschließend sollte man eine Datei mit der Endung **cer** vorfinden. Diese Datei ist das erstellte Zertifikat, welches anschließend noch in die Zertifikatsspeicher „**Vertrauenswürdige Stammzertifizierungsstellen**“ und „**Vertraute Herausgeber**“ auf den Testrechner installiert werden muss. Dazu klickt man mit der rechten Maustaste auf das Zertifikat und wählt den Menüpunkt „Zertifikat installieren“. Anschließend wählt man den Zertifikatsspeicher aus in dem das Zertifikat installiert werden soll. Weitere Informationen zum Signieren von Kernel-Mode Treibern findet man in [csw]. Dort wird ebenfalls beschrieben, wie man die Treiber bei der endgültigen Veröffentlichung signiert.



3.3 Den Treiber signieren

Um den Treiber zu signieren benötigt man zunächst eine Katalogdatei, welche mit dem Tool **Inf2Cat** erstellt werden kann. Laut [csw] soll Inf2Cat das Tool **Signability** ablösen, doch bis jetzt ist es allerdings noch nicht einmal im *Windows Driver Kit* enthalten, sondern lediglich Bestandteil der *Winqual Submisson Tools ([dl-wql]).* Mit Inf2Cat wird zunächst eine CAT-Datei erstellt:

```
Inf2Cat.exe /driver:<Treiberpfad> /os:<Betriebssystem>
```

Hier wird der Pfad angegeben, unter welchem die Treiberdateien zu finden sind. Dort sollte das Programm mindestens den eigentlichen Treiber, die INF-Datei und den dazugehörigen CoInstaller vorfinden. Weiterhin wird die Betriebssystemversion angegeben, für Windows Vista (32bit) beispielsweise „Vista_x86“ Weitere Parameter und Einstellungen erfährt man, wann man „INF2CAT /?“ auf der Kommandozeile aufruft.

Signiert wird der Treiber anschließend mit Hilfe des Tools **SignTool**, welches Bestandteil des Windows Driver Kit ist.

```
Signtool.exe sign /v /s <CS> /n <CN> /t <TS> <CAT>
```

Parameter	Bedeutung
<CS>	Zertifikatsspeicher in dem das Zertifikat zu finden ist
<CN>	Name des Zertifikates
<TS>	URL des TimeStamp Servers, üblicherweise wird hier *http://timestamp.verisign.com/scripts/timestamp.dll* verwendet
<CAT>	Der Name der zu signierenden CAB-Datei

Nach diesem Vorgang ist die CAT-Datei digital signiert, wobei man beachten sollte das Windows Vista trotzdem eine Warnmeldung bei der Installation des Treibers anzeigt, da der Herausgeber des Zertifikates nicht ermittelt werden kann. Es ist empfehlenswert die Aufrufe von Inf2Cat und Signtool in einem Skript zu verpacken, um sich die Tipparbeit bei jedem Testlauf des Treibers zu ersparen. Mit Hilfe von Signtool ist es außerdem möglich die Signierung eines Treibers zu überprüfen, dazu verwendet man folgendenden Syntax:

```
SignTool.exe verify /kp <Katalogname.cat>
```

Hier muss man nur den Namen der CAT-Datei angeben, welche überprüft werden soll. Weiterhin ist es empfehlenswert die Treiberdatei selbst auch zu signieren, dies funktioniert ebenfalls über Signtool. Dazu verwendet man den denselben Aufruf, wie er zum Signieren der CAT-Datei verwendet wurde, nur mit dem Unterschied das anstatt der CAT-Datei der Dateiname des Treibers angegeben wird. Hierbei sollte man beachten, dass dieser Vorgang vor dem eigentlichen Signieren durchgeführt werden sollte.

3.4 Besonderheiten unter der 64bit Version von Windows Vista

Die Richtlinien zum Signieren von Treibern haben sich mit dem Erscheinen von Windows Vista drastisch verschärft. So müssen Kernel-Mode-Treiber für die 64bit Version zwangsläufig signiert sein, damit Windows diese überhaupt lädt. Damit man Treiber auf diesen Systemen auch testen kann, ohne diese jedes mal mit einem autorisiertes Zertifikat einer Zertifizierungsstelle zu signieren, gibt es einen so genannten Testmodus des Systems, welcher mit **BCDEdit** aktiviert werden kann. Dazu muss man auf der Kommandozeile den folgenden Befehl eingeben, um den Testmodus für den aktuellen Windows-Booteintrag zu aktivieren.

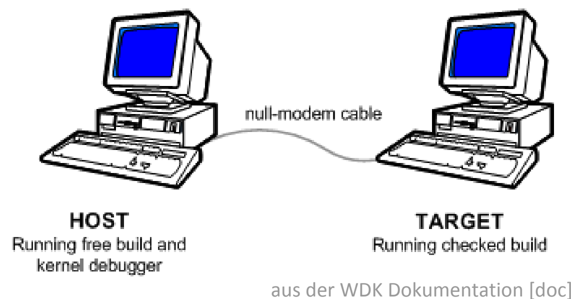
```
Bcdedit.exe -set TESTSIGNING ON
```

Die Treiber müssen in diesem Modus zwar immer noch signiert sein, aber reicht jetzt auch ein selbst erstelltes Testzertifikat aus, um Windows dazu zu bewegen den Treiber zu laden.

4. Debuggen von Kernel-Mode Treibern

4.1 Einrichten einer Debugumgebung

Für das Debuggen eines Kernel-Mode-Treibers werden zwei Computer benötigt, ein Testrechner (Target) und ein Entwicklungsrechner (Host). Es ist zwar ebenfalls möglich nur einen Rechner zu verwenden, indem man den Debugger auf den Testrechner betreibt, von einer solchen Konfiguration kann aber eher abgeraten werden, da sie zu viele Nachteile bringt und nicht produktiv zum Debuggen eingesetzt werden kann. Die folgende Grafik demonstriert den prinzipiellen Aufbau einer Debugumgebung mit zwei Computern:



aus der WDK Dokumentation [doc]

Auf den Testrechner Rechner läuft das Betriebssystem, für welches der Treiber geschrieben wurde, d.h. für das neue Kernel-Mode Driver Framework sollte hier Windows 2000 oder höher installiert sein. Am Besten verwendet man hier die 'checked build' Version von Windows, welches zusätzliche Debugging-Informationen enthält und ohne Optimierung des Codes kompiliert wurde, was ein einfacheres Debuggen des Kernels ermöglicht. Die 'checked build'–Ausgaben des jeweiligen Betriebssystems sind allerdings teilweise nur für MSDN-Abonnenten verfügbar, eine Liste mit Download-Adressen findet man unter [chk]. Auf diesen Rechner muss man das Debuggen des Kernels aktivieren, am Einfachsten kann man dies mit dem Tool **msconfig** bewerkstelligen. In diesem Tool gibt es einen Reiter unter welchem man die Bootoptionen einstellen kann. Dort kann man dann, unter den erweiterten Einstellungen, für den ausgewählten Booteintrag die gewünschten Debugoptionen einstellen. Weitere detaillierte Informationen zum Aktivieren des Debugmodus finden Sie in [doc][4].

Auf den anderen Rechner läuft der Kernel-Debugger. Üblicherweise sind hier ebenfalls die Entwicklungstools installiert. Das Betriebssystem kann ein beliebiges Windows-Betriebssystem ab Windows 2000 sein. Die beiden Rechner müssen über ein Nullmodemkabel, über ein IEEE1394-Anschluss (ab Windows XP) oder über USB 2.0 (ab Windows Vista) miteinander verbunden sein. Als nächstes muss der Debugger auf dem Entwicklungsrechner eingerichtet werden.

4.2 Verwenden von WinDbg zum Debuggen des Kernels

Im Folgenden wird erklärt, wie man den grafischen Debugger **WinDbg** einrichtet, um Treiber im Kernel-Mode debuggen zu können. WinDbg ist Bestandteil der Debugging Tools for Windows, welche man unter [dl-dbg] herunterladen kann.

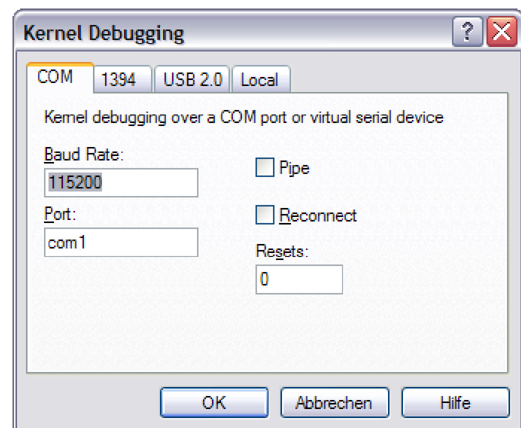
Um mit dem Debuggen zu beginnen, werden zunächst die entsprechenden Symbole für die jeweilige

4 Unter *Driver Development Tools/Boot Options for Driver Testing and Debugging*

Windowsversion benötigt. Unter [dl-sym] kann man die Symbole für verschiedene Windowsversionen herunterladen. Diese sind in einem Ordner zu entpacken und dieser ist dann in WinDbg unter 'File/Symbol File Path...' einzutragen. Hat man allerdings eine schnelle Internetverbindung (ab 1Mbit/s) zu Verfügung, so ist die Verwendung des Symbolservers sehr zu empfehlen. Der Vorteil in der Benutzung des Symbolservers liegt darin, das nur die benötigen Symbole heruntergeladen werden und WinDbg automatisch erkennt für welche Betriebssystemversion die Symbole benötigt werden. Um den Symbolserver zu verwenden legt man zunächst einen leeren Ordner an, welcher die benötigen Symbole lokal zwischenspeichert. Als Symbolpfad gibt man nun unter WinDbg folgendes ein:

```
srv*d:\symbols\websymbols*http://msdl.microsoft.com/download/symbols
```

Wobei der Pfad *d:\symbols\websymbols* auf den angelegten lokalen Ordner verweist. Es können hier auch weitere Symbolpfade durch ein Semikolon getrennt eingetragen werden, beispielsweise um auf die Symbole des Treibers zu verweisen.



Jetzt kann man über 'File/Kernel Debug...' die Verbindungseinstellungen zum Testrechner angeben. Hierbei sollte man darauf achten, exakt die selben Einstellungen zu verwenden, die man in den Bootoptionen des Testrechners eingestellt hat. Wenn alle Einstellungen korrekt sind, sollte eine Verbindung zum Testrechner zustande kommen. Ob eine Verbindung besteht kann man an der Statusleiste des Commandfensters von WinDbg erkennen, bei laufenden Debugger steht dort „Debuggee is running...“ oder ein Eingabefeld „kd>“. Besteht keine Verbindung sollte man zunächst versuchen, über die Tastenkombination 'Strg+Untbr' oder über das Menü 'Debug/Break', einen Breakbefehl an den Testrechner zu senden, um den Debugger somit einen Einsprung in das System zu ermöglichen. Sollte dies funktionieren kann das System mit dem Kommando 'g' oder über die Taste F5 fortsetzt werden. Das Command-Fenster im Debugger liefert bei einer erfolgreichen Verbindung in etwa folgende Ausgabe:

```
Microsoft (R) Windows Debugger  Version 6.6.0007.5
Copyright (c) Microsoft Corporation. All rights reserved.

Opened \\.\com1
Waiting to reconnect...
Connected to Windows Vista 6000 x86 compatible target, ptr64 FALSE
Kernel Debugger connection established.
Symbol search path is:
srv*d:\symbols\websymbols*http://msdl.microsoft.com/download/symbols
Executable search path is:
Windows Vista Kernel Version 6000 MP (1 procs) Free x86 compatible
Product: WinNt, suite: TerminalServer SingleUserTS
Built by: 6000.16386.x86fre.vista_rtm.061101-2205
Kernel base = 0x81c00000 PsLoadedModuleList = 0x81d08ab0
Debug session time: Thu Mar  8 13:45:08.578 2007 (GMT+1)
System Uptime: 0 days 3:55:31.952
```

Damit wäre der Debugger eingerichtet. Es ist empfehlenswert eine Verknüpfung zu WinDbg einzurichten, in welcher die Parameter für die Verbindung mit dem Testrechner direkt übergeben werden. Die folgende Verknüpfung ruft WinDbg beispielsweise mit dem Symbolserver als Symbolsuchpfad und im Kernel-Debug-Modus über die COM1-Schnittstelle auf:

```
windbg.exe -y srv*d:\websymbols*http://msdl.microsoft.com/download/symbols
           -k com:port=1,baud=115200
```

Mehr über die Parameter, mit denen WinDbg aufgerufen werden kann, findet man in der Dokumentation von WinDbg[5].

4.3 Meldungen an den Debugger senden (KdPrintEx)

Um Meldungen an den Kernel-Debugger zu senden, um beispielsweise verschiedene Treiberzustände auszugeben oder um detaillierte Fehlermeldungen anzuzeigen, verwendet man den Befehl *KdPrintEx*. Dieser Befehl löst den bisher genutzten Befehl *KdPrint* ab, welcher aus Kompatibilitätsgründen ebenfalls noch zur Verfügung steht, bei Neuentwicklungen aber nicht mehr genutzt werden sollte. Der *KdPrintEx*–Befehl ist von der Funktionsweise identisch mit dem *DbgPrintEx*-Befehl, welcher ebenfalls verwendet werden kann. Der Unterschied besteht darin, dass der *KdPrintEx*-Befehl ignoriert wird, sobald man den Treiber mit der „Free Build-“Umgebung kompiliert. Der Syntax des Befehl lautet:

```
KdPrintEx((IN ULONG ComponentId, IN ULONG Level, IN PCHAR Format, ... [args]));
```

Die Parameter ComponentID und Level dienen dem Debugger dazu, nur bestimmte Nachrichten anzuzeigen. Dabei gibt ComponentID den Gerätetyp des Treibers an, welcher einer der Folgenden sein muss:

ComponentID	Treibertyp
DPFLTR_IHVVIDEO_ID	Video Treiber
DPFLTR_IHVAUDIO_ID	Audio Treiber
DPFLTR_IHVNETWORK_ID	Netzwerktreiber
DPFLTR_IHVSTREAMING_ID	Kernel Streaming Treiber
DPFLTR_IHVBUS_ID	Bus Treiber
DPFLTR_IHVDRIVER_ID	Sonstiger Treibertyp

Der Level der Nachricht repräsentiert eine bestimmte Nachrichtenklasse bzw. eine bestimmte Wichtigkeit der Nachricht. Die Level 0 bis 3 sind hier bereits vordefiniert, die restlichen Level können frei verwendet werden. Das Level 0 signalisiert beispielsweise eine Fehlermeldung, das Level 3 eine unkritische Informationsnachricht. Die ersten 31 Level repräsentieren aufsteigend jeweils ein Bit in einer 32bit Zahl. Man sollte beachten das Level die größer als 31 sind, anders behandelt werden. Weitere Infomationen findet man in der WDK-Dokumentation[6].

Damit man überhaupt die Debugmeldungen, welche das Programm verschickt im Debugger zu sehen bekommt sind noch ein paar weitere Einstellungennötig. Als erstes muss man die Filtermaske des Debuggers anpassen, damit dieser die gewünschten Meldungen anzeigt. Um die Filtermaske zu editieren muss man zunächst das Windowssystem unterbrechen, in WinDbg kann man dies mittels der Tastenkombination Strg + Unterbr bewerkstelligen.

5 Unter *Debugging Tools for Windows/Debuggers/Debugger Reference/Command-Line Options/WinDbg Command-Line Options*
6 Unter *Driver Development Tools/Tools for Debugging Drivers/Using Debugging Code in a Driver/Debugging Code Overview/Reading and Filtering Debugging Messages*

Nun kann man mittels des Befehls **ed** die Meldungsmaske anpassen.

```
kd> ed Kd_XXXX_Mask 0xYYYY
```

Für **XXXX** muss die jeweilige Treiberklasse angeben, welche den Typen (ComponentID) in der obigen Tabelle entsprechen. Gültige Einträge sind hier: **IHVVIDEO, IHVAUDIO, IHVNETWORK, IHVSTREAMING, IHVBUS, IHVDRIVER**

Für **YYYYY** muss man in eine HEX-Zahl eingeben, welche das gewünschte Bitmuster repräsentiert. Das Bitmuster der Maske wir dem übermittelten Level UND-verknüpft und wenn das Ergebnis ungleich 0 ist, so wird die Nachricht angezeigt. Man sollte noch beachten, dass es eine systemweite Maske 0x1 gibt, welche immer gilt. Möchte man beispielsweise alle Level von 0 bis 3 anzeigen, so muss man die untersten 4 Bit als Maske setzten, also 0xF.

Als Alternative zur Eingabe dieser Meldungsmaske im Debugger, kann man diese auch in der Registry fest definieren. Dazu legt man zunächst den folgenden Schlüssel an:

```
HLM\SYSTEM\CurrentControlSet\Control\Session Manager\Debug Print Filter
```

Unter diesem Schlüssel legt man einen DWORD Wert mit dem Namen der gewünschten Komponente an (siehe oben XXXX), welcher die Maske als HEX-Wert enthält. Diese Maske wird dann vom Debugger als Voreinstellung für die jeweilige Komponente verwendet.

Damit es KMDF-Treiber überhaupt möglich ist Meldungen an den Debugger zu schicken muss in der Windows-Registry zusätzlich noch ein Eintrag angelegt werden, der dies ermöglicht.

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Wdf\Kmdf\Diagnostics
```

Unter dem oben angegebenen Registrypfad (eventuell muss dieser angelegt werden) muss ein Wert mit dem Namen **DbgPrintOn** als DWORD und einem Inhalt **ungleich 0** angelegt werden. Dieser Eintrag teilt dem Framework mit, dass alle KMDF-Treiber Meldungen an den Debugger schicken dürfen.

4.4 Weitere Debugeinstellungen und -möglichkeiten

In der Registry kann man weitere treiberspezifische Debugeinstellungen vornehmen, welche die Debugmöglichkeiten des Framework-basierten Treibers erweitern. Hier eine Übersicht dieser Einstellungen, welche auch in [dok][7] nachgeschlagen werden können. Diese Einstellungen müssen in folgendem Pfad stehen:

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\<Treibername>\Parameters\Wdf
```

RegistryWert	Beschreibung
VerifierOn (REG_DWORD)	Wenn dieser Wert ungleich Null ist, zur überprüft das Framework den Treiber umfangreich als das normalerweise der Fall wäre.
VerifyOn (REG_DWORD)	Ist dieser Wert ungleich Null, so wird das WDFVERIFY Makro aktiviert. Wurde der Wert VerfifierOn gesetzt, so ist diese Option automatisch aktiv.

7 Unter Windows Driver Foundation/Kernel-Mode Driver Framework/Getting Started with Kernel-Mode Driver Framework/Debugging a Framework-based Driver/Registry Values for Debugging Framework-based Drivers

DbgBreakOnError (REG_DWORD)	Wenn dieser Wert ungleich 0 ist, wird der Aufruf WdfVerifierDbgBreakPoint aktiviert. Wurde der Wert VerfifierOn gesetzt, so ist diese Option automatisch aktiv.
VerifierAllocateFailCount (REG_DWORD)	Wurde dieser Wert eine Zahl n, größer Null gesetzt und ist VerifierOn aktiv. So scheitert der Treiber n-mal beim Versuch Speicher für das Treiberobjekt anzufordern. Somit kann man Testen wie der Treiber auf zu wenig Speicher reagiert.
TrackHandles (REG_MULTI_SZ)	Gibt eine Liste mit Framework-Objekten an, welche bei der Ausführung des Treibers vom Debugger nach verfolgt werden sollen. Weiteres hierzu in [dex, S. 20]
VerboseOn (REG_DWORD)	Ist dieser Wert ungleich Null, so zeichnet der Framework Protokolldienst zusätzliche Informationen auf, die beim Debuggen hilfreich sein können. Weiteres hierzu in [log].
LogPages (REG_DWORD)	Gibt die Anzahl der zu verwendeten "Memory pages" für den Protokolldienst an (1-10). Ist dieser Wert nicht definiert wird 1 angenommen. Weiteres hierzu in [log].
ForceLogsInMiniDump (REG_DWORD)	Wenn dieser Wert ungleich Null ist. Zu werden zusätzliche Informationen vom Protokolldienst der Dump-Datei hinzugefügt.
TraceDelayTime (REG_DWORD)	Gibt eine Verzögerungszeit in Millisekunden an, die für das Laden des WPP Software Tracing gilt.

Stehen diese Werte nicht in der Registry, so wird automatisch angenommen, dass diese Einträge den Wert 0 haben, also deaktiviert sind.

Weiterhin ist es möglich **Unterbrechungspunkte** im Programmcode des Treibers zu setzen. Diese Unterbrechungen halten das System an und teilen dies dem verbundenen Debugger mit, welcher das System an dieser Stelle natürlich auch wieder fortsetzen kann. Der dazugehörige Befehl lautet:

```
VOID WdfVerifierDbgBreakPoint (VOID)
```

Hier ist zu beachten das dieser Befehl nur funktioniert wenn der treiberspezifische Registrywert **DbgBreakOnError** oder **VerifierOn** auf einen Wert ungleich 0 gesetzt wurde (siehe obige Tabelle).

5. Software Tracing

5.1 Einführung

Das Software-Tracing erlaubt dem Treiber Nachrichten, ähnlich den Debuggnachrichten, zu verschicken. Die Nachrichten haben einen sehr geringen Overhead und wirken sich kaum auf die Geschwindigkeit des Treibers aus. Dies hat den Vorteil, dass diese Nachrichten auch im Release-Build des Treibers verwendet werden können. Dadurch kann der Treiber mit optimiertem Code kompiliert und anschließend verschiedenen Leistungstest unterzogen werden, ohne auf die Möglichkeit zu verzichten, Informations-, Warn- und Fehlermeldungen seitens des Treibers zu erhalten. Dies funktioniert auch auf einen "FreeBuild"-System, welches ohne Debugger und Debugging-Optionen läuft. Die Nachrichten werden dabei in einem Binärformat versendet und enthalten neben der eigentlichen Nachricht verschiedene hilfreiche Informationen.

Damit diese Nachrichten überhaupt erst geschickt werden und um diese zu empfangen, muss man eine so genannte "Trace-Session" starten. Diese Tracesitzung kann grafisch mit **TraceView** oder kommandozeilenorientiert mit **TraceLog** durchgeführt werden, beide Programme liegen dem Windows Driver Kit bei[8]. In den nachfolgenden Ausführungen wird das Programm TraceView verwendet. Neben der Möglichkeit, die Nachrichten in eine Datei zu schreiben, besteht auch die Option, sich diese in Echtzeit anzeigen zu lassen. Da die Nachrichten im Binärformat vorliegen, benötigt das Traceprogramm noch zusätzliche Informationen zu dem Treiber, welcher die Nachrichten sendet. Hierbei reicht die Angabe der Debug-Informationsdatei (PDB) aus, in dieser stehen alle notwendigen Informationen zum Empfang und zur Auswertung der Nachrichten.

Im Folgenden wird beschrieben, welche Schritte notwendig sind, damit der Treiber das Software-Tracing unterstützt und Nachrichten versendet. Als Beispiel zur Implementierung werden hier die Quellen des ECHO-Treibers verwendet (siehe Abschnitt 7). Weiterhin wird gezeigt wie man die Nachrichten aufzeichnet, filtert und anzeigt.

5.2 Tracing Unterstützung hinzufügen

Zunächst muss der Treiber ein **WPP_CONTROL_GUIDS** Makro definieren, in welchem eine eindeutige GUID und die Tracingflags definiert werden. Die Flags dienen dazu verschiedene Nachrichtengruppen zu trennen, um diese später in einer Tracesitzung filtern zu können. Das Aufrufmuster für dieses Makro ist:

```
#define WPP_CONTROL_GUIDS \
    WPP_DEFINE_CONTROL_GUID(GUIDFriendlyName, (ControlGUID),  \
    WPP_DEFINE_BIT(NameOfTraceFlag1)  \
    WPP_DEFINE_BIT(NameOfTraceFlag2)  \
    ............................     \
    ............................     \
    WPP_DEFINE_BIT(NameOfTraceFlag31) )
```

Es können 31 verschiedene Flags zur Identifizierung der Nachrichten verwendet werden. Die Definition des Makros mit drei Flags könnte beispielsweise folgendermaßen aussehen:

```
//Informationen für das Software-Tracing
//GUID für Tracing: {4870654C-ED7A-46a4-BD9C-09A14AF67881}
#define WPP_CONTROL_GUIDS \
    WPP_DEFINE_CONTROL_GUID(CtlGuid, \
      (4870654C, ED7A, 46a4, BD9C, 09A14AF67881),  \
      WPP_DEFINE_BIT(DRV_INFO) \
      WPP_DEFINE_BIT(IRP_INFO) \
      WPP_DEFINE_BIT(WMI_INFO))
```

Quelltextbeispiel - debug.h

Weiterhin muss in allen Quellcode-Dateien in denen Tracingnachrichten verschickt werden sollen, eine **Tracing-Message-Header-Datei (TMH)** eingebunden werden. In dieser Datei stehen Informationen über die Nachrichten für den Windows Software Trace Preprocessor (WPP), welche später der Debug-Informationsdatei hinzugefügt werden , um die binären Nachrichten zu lesen. Diese Datei hat die Endung .tmh und trägt den selben Namen wie die Quellcodedatei. In der Datei driver.c müsste beispielsweise folgender include-befehl ergänzt werde:

```
#include “driver.tmh“
```

8 Im Ordner /tools/tracing/<arch>

Hierbei ist zu beachten das dieser include-Befehl **nach der Definition** des WPP_CONTROL_GUIDS Makros stehen muss. Damit diese TMH-Dateien beim Kompilieren des Treibers auch vorhanden sind, muss am Ende der **sources** – Datei noch folgender Eintrag hinzugefügt werden:

```
RUN_WPP=  $(SOURCES) -km
```

Dieser Eintrag weist den Präprozessor an, für alle Dateien die als Quellen eingetragen sind, die Tracing Header automatisch zu erstellen. Zu guter Letzt muss der Treiber die Tracing Unterstützung noch initialisieren, dazu muss muss in der **DriverEntry**–Methode folgender Aufruf stehen:

```
WPP_INIT_TRACING(DriverObject, RegistryPath);
```

Dieses Makro darf natürlich erst aufgerufen werden, nachdem das Treiberobjekt mittels **WdfDriverCreate** erfolgreich initialisiert wurde. Anschließend kann man in allen Funktionen des Treibers mit der Methode **DoTraceMessage** Tracenachrichten verschicken. Der Syntax lautet:

```
void DoTraceMessage(IN TraceFlagName, IN FormatString, IN VariableList);
```

Hier wird ein **TraceFlag** und die Nachricht mit einer optionalen Argumentliste übergeben, der Syntax der Nachricht und der Argumente entspricht der von *printf*. Das TraceFlag muss im Makro WPP_CONTROL_GUIDS definiert sein, wie weiter oben beschrieben wurde. Diese Nachricht kann nun in einer Tracesitzung empfangen werden, weiteres dazu steht im Abschnitt 5.4.

Abschließend muss das **WPP_CLEANUP** Makro in der **DriverUnload**-Funktion aufgerufen werden, welches das Tracing wieder deaktiviert, wenn der Treiber entladen wird. Im Folgenden noch ein Ausschnitt aus den Quellcode des ECHO-Treibers, welcher den Aufruf der Initialisierungsroutine des Software Tracing in der *DriverEntry*-Methode, das Schicken einer Tracingnachricht und das Abmelden vom Software Tracing mittels des WPP_CLEANUP Makros demonstriert:

```
NTSTATUS DriverEntry(
      IN PDRIVER_OBJECT DriverObject, IN PUNICODE_STRING RegistryPath)
{
      //... Treiber initialisieren ...
      {...}

      //Tracing initialisieren
      WPP_INIT_TRACING(DriverObject, RegistryPath);

      //Tracingnachricht schicken
      DoTraceMessage(DRV_INFO, "Tracing wurde initalisiert - %s\n", __TIME__);

      //Unload-Funktion definieren
      DriverObject->DriverUnload = EchoDriverUnload;

      //...weitere Initaliasierungsarbeiten ...
      {...}
}

VOID EchoDriverUnload(IN PDRIVER_OBJECT DriverObject)
{
      {...}
      // Tracing deaktivieren
      WPP_CLEANUP(DriverObject);
      {...}
}
```

Quelltextbeispiel - driver.c

5.3 Unterstützung für Tracing-Level hinzufügen

Um die Filterung später zu vereinfachen bietet sich der Einsatz von Tracinglevel an. Dies ermöglicht es, den Nachrichten zusätzlich eine Wichtigkeitsstufe zu geben. Somit kann man mittels der Flags die Nachrichten in Gruppen teilen und mit Hilfe der Levels die Wichtigkeit der jeweiligen Nachricht festlegen. Dadurch ist später während einer Tracesitzung eine detaillierte, angepasste Filterung der Nachrichten möglich (weiteres dazu im Abschnitt 5.4).

Um Tracinglevel verwenden zu können muss der Treiber zunächst zwei weitere Makros aufrufen:

```
#define WPP_LEVEL_EVENT_LOGGER(lvl,event) \
      WPP_LEVEL_LOGGER(event)

#define WPP_LEVEL_EVENT_ENABLED(lvl, event) \
      (WPP_LEVEL_ENABLED(event) && \
      WPP_CONTROL(WPP_BIT_ ##event).Level >= lvl)
```

Da die Funktion DoTraceMessage keine Level unterstützt, muss eine eigene Methode zum Verschicken von Tracingnachrichten definiert werden. Damit der Windows Software Trace Preprocessor Kenntnis von dieser selbst definierten Methode hat, wird diese in der **sources**-Datei definiert:

```
RUN_WPP=      $(SOURCES) -km \
              -func:TracePrint(LEVEL,EVENT,MSG,...)
```

Hier wird die Methode **TracePrin**t definiert, welche neben dem TraceFlag (EVENT) und der Nachricht mit der Parameterliste (MSG, ...), auch ein LEVEL annimmt und weiter gibt. Die Level selbst sind in der Datei *evntrace.h* definiert und lauten wie folgt:

```
#define TRACE_LEVEL_NONE        0 // Tracing is not on
#define TRACE_LEVEL_FATAL       1 // Abnormal exit or termination
#define TRACE_LEVEL_ERROR       2 // Severe errors that need logging
#define TRACE_LEVEL_WARNING     3 // Warnings such as allocation failure
#define TRACE_LEVEL_INFORMATION 4 // Includes non-error cases
#define TRACE_LEVEL_VERBOSE     5 // Detailed from intermediate steps
```

Tracelevel-Definition aus evntrace.h

Diese Schritte reichen aus, damit die selbst definierte Funktion TracePrint zum Versenden von Tracenachrichten verwendet werden kann. Hier noch ein Beispiel für den den Aufruf von TracePrint:

```
TracePrint(TRACE_LEVEL_INFORMATION, DRV_INFO,
      "Software-Tracing wurde initalisiert - %s\n", __TIME__);
```

Traceprint Beispielaufruf

5.4 Nachrichten aufzeichnen, anzeigen und filtern

Um die Nachrichten aufzuzeichnen muss zunächst eine Tracingsitzung gestartet werden. Dazu verwendet man beispielsweise das Programm **TraceView**, welches dem Windows Driver Kit beiliegt.

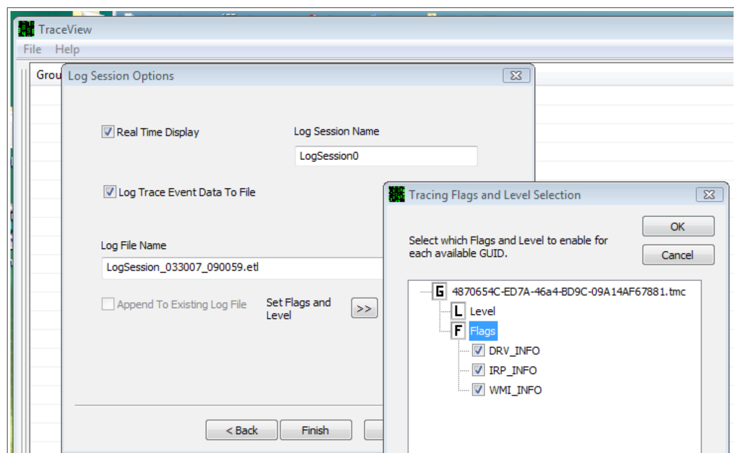
Trace Provider auswählen

Zuerst muss eine neue „**Log Session**“ angelegt werden, dies geschieht über den Menüpunkt „File/Create New Log Session“. Im erscheinenden Einrichtungsassistent muss nun ein „**Trace Provider**“ hinzugefügt

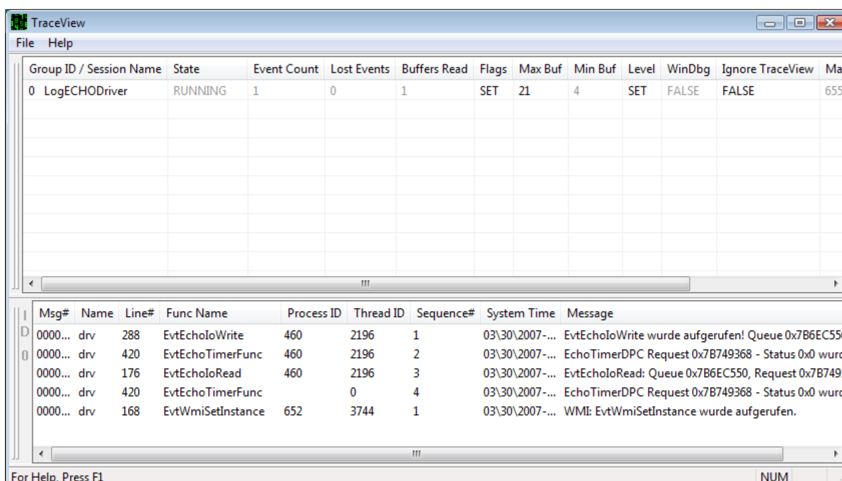
werden, also ein Programm bzw. Treiber, welches Tracenachrichten verschickt. Damit TraceView die binären Tracenachrichten entziffern kann, benötigt es die Debuginformationen des Treibers, in denen die Informationen zu den Tracenachrichten zu finden sind. Dazu wählt man die PDB-Datei des Treibers aus, welche bei der Kompilierung erzeugt wurde. Nun kann man im Assistenten weitere Trace Provider hinzufügen oder einen Schritt weiter gehen, um weitere Einstellungen vorzunehmen.

Art der Aufzeichnung und Filtereinstellungen

Im zweiten Fenster des Einrichtungsassistenten legt man fest, ob die Nachrichten in Echtzeit angezeigt oder ob sie in eine Datei geschrieben werden sollen. Es ist natürlich auch möglich, und in vielen Fällen empfehlenswert, beide Möglichkeiten gleichzeitig zu nutzen. Im Unterpunkt „Set Flags and Level" kann man die Filtereinstellungen zur Aufzeichnung auswählen, hier stehen die benutzerdefinierten Flags und die Tracinglevel (falls die Unterstützung integriert wurde) zur Auswahl. Im Unterpunkt „Advanced Log Session Options" kann man erweiterte Einstellungen zur Tracesitzung vornehmen, hierbei sollte man in der Hilfe von TraceView nachschlagen. Sind alle Einstellungen korrekt, sollte TraceView nun in der Lage sein, die Tracenachrichten des Treibers zu empfangen. Die erstellte Sitzung sollte sich im Zustand „Running" befinden. Die Filtereinstellungen können hier nachträglich durch einen Klick auf die Spalte „Flag" der entsprechenden Sitzung geändert werden.



Mit einem Rechtsklick auf eine Sitzung kann man über den Menüpunkt „Manage Filters...", erweiterte Filtereinstellungen vornehmen. Hier legt man so genannte Regeln fest, welche die Traceausgabe nachträglich filtern können. So kann man beispielsweise alle Meldungen aus einer bestimmten Funktion oder aus einer bestimmten Quelltext-Datei ignorieren. Mit Hilfe dieser zusätzlichen Regeln behält man selbst bei komplexen Treibern, mit vielen Tracenachrichten die Übersicht. Dazu lässt man sich nur die Nachrichten ausgeben, welche man in der aktuellen Situation als relevant betrachtet. Das Ausgabefenster der Nachrichten kann über einen Rechtsklick auf den Spaltenkopf um zusätzliche Spalten ergänzt werden, welche weitere nützliche Informationen zur Nachricht bereithalten. Beispielsweise ist es meist von Nutzen den Funktionsnamen und die zugehörige Zeilennummer anzuzeigen, von welcher aus die Nachricht verschickt wurde.



6. Windows Management Instrumentation

6.1 Einführung

Windows Management Instrumentation ist ein zentraler Dienst von Windows über den das System verwaltet und gewartet werden kann. Er ist Bestandteil des standardisierten Web-Based Enterprise Management, was Funktionen zum Administrieren und Fernwarten von Computersystemen bereitstellt und unabhängig von der verwendeten Hardware und vom verwendeten Betriebssystem arbeitet. Der WMI-Dienst stellt dem Treiber eine Schnittstelle zur Verfügung, über welche Nachrichten, Ereignisse verschickt und Einstellungen gemacht werden können. Damit der Treiber die Schnittstelle verwenden kann und muss sich dieser beim WMI-Dienst als so genannter „WMI Provider" registrieren.

6.2 Vorbereitung zur WMI Unterstützung

Damit der Treiber WMI unterstützt müssen zunächst zwei **Headerdateien** eingebunden werden, welche die WMI-Funktionen und Makros definieren.

```
#include <wmistr.h>
#include <wmilib.h>
```

Ein Treiber bietet dem WMI-Dienst nun ein oder mehrere Klassen an, welche über Eigenschaften und Methoden verfügen. Der Dienst kann nun auf diese Eigenschaften lesend und wenn gewünscht auch schreibend zugreifen. Um diese Klassen zu definieren muss eine Datei im **Managed Object Format (MOF)** erstellt werden, in welcher die bereitgestellten Klassen, sowie deren Methoden und Eigenschaften definiert sind. Jede Klasse muss neben den Werten *InstanceName (string)* und *Active (boolean)* eine eindeutige GUID besitzen. Eine einzelne Klasse repräsentiert einen sogenannten *DataBlock*, welcher *DataItems* und Methoden enthalten kann. Eine detailierte Beschreibung des MOF Syntax findet man unter [doc][9]. Das folgende Beispiel definiert zwei Werte *(DataItems) EnableRead* und *EnableWrite* auf die lesend und schreibend zugeriffen werden kann. Weiterhin wird eine Methode *ChangeMaxLength* deklariert, welche einen Eingabe- und einen Ausgabeparameter besitzt.

```
[WMI, Dynamic, Provider("WMIProv"),
DisplayName("Echo WMI Klasse"),
Guid("9C4EFC6D-3960-409c-A1CF-C9FACB050440")]
class EchoWMIClass
{
       [key, read]    string InstanceName;
       [read]         boolean Active;

       [WmiDataId(1),Read, Write, DisplayName("EnableWrite") : amended,
        Description("Ermöglicht das Schreiben auf das Gerät") : amended]
       boolean EnableWrite;

       [WmiDataId(2),Read, Write, DisplayName("EnableRead") : amended,
        Description("Ermöglicht das Lesen vom Gerät") : amended]
       boolean EnableRead;

       [WmiMethodId(1),Implemented,
        Description("Ändert die Maximallänge") : amended,]
       void ChangeMaxLength([in] uint32 length, [out] boolean success);
};
```

Beispiel einer MOF-Definition

9 Unter Kernel Mode Driver Architecture\Design Guide\Windows Management Instrumentation\MOF Syntax for WMI Data and Event Blocks

Diese MOF-Datei muss anschliessend in eine binäre MOF-Datei (BMF) umgewandelt werden. Aus dieser kann unter Anderem eine Headerdatei und ein Testskript erzeugt werden. Es ist üblich diesen Vorgang automatisch während des Buildvorgang durchzuführen, dazu wird die **makefile.inx** ergänzt. Das Programm mofcomp erzeugt wandelt die MOF-Datei in das binäre Format um. Aus dieser kann das Programm wmimofck anschliessend den Header und Testskript erzeugen. Das folgende Beispiel zeigt einen Eintrag in der **makefile.inx**, der diese Aufgabe erledigt:

```
$(OBJ_PATH)\$O\echowmi.bmf : $(OBJ_PATH)\$O\echowmi.mof
       mofcomp -WMI -B:$(OBJ_PATH)\$O\echowmi.bmf echowmi.mof
       wmimofck -m -h$(OBJ_PATH)\$(O)\echowmi.h -t$(OBJ_PATH)\$(O)\echowmi.vbs
              $(OBJ_PATH)\$O\echowmi.bmf
```

Auszug aus makefile.inx

Der erzeugte Header enthält die in der MOF definierten Klassen als C-Strukturen und die GUID der Klassen, welche zu Registrierung als WMI-Provider benötigt werden. Hier ein Auszug aus der mittels dem obigen Codebeispiel erzeugten Headerdatei.

```
#define EchoWMIClassGuid \
    { 0x9c4efc6d,0x3960,0x409c, { 0xa1,0xcf,0xc9,0xfa,0xcb,0x05,0x04,0x40 } }

typedef struct _EchoWMIClass
{
    // Ermöglicht das Schreiben auf das Gerät
       BOOLEAN EnableWrite;
       #define EchoWMIClass_EnableWrite_SIZE sizeof(BOOLEAN)
       #define EchoWMIClass_EnableWrite_ID 1

    // Ermöglicht das Lesen vom Gerät
       BOOLEAN EnableRead;
       #define EchoWMIClass_EnableRead_SIZE sizeof(BOOLEAN)
       #define EchoWMIClass_EnableRead_ID 2
} EchoWMIClass, *PEchoWMIClass;
```

Auszug aus automatisch erzeugtem Header (mittels wmimofck)

In der Ressourcendatei (RC) des Treibers muss noch eine Zeile eingefügt werden, welche auf den Namen der binären MOF-Datei verweist, damit diese als Ressource des Treibers registriert ist. Auf diesen Verweis kann später bei der Registrierung als WMI-Provider zugeriffen werden, um das Gerät an diese MOF-Ressource zu binden.

```
MofResourceName MOFDATA echowmi.bmf
```

Eintrag in Ressourcendatei des Treibers (*.rc)

Für den schnellen Zugriff auf die Ressource zur Registrierung empfiehlt sich eine Definition in einer Headerdatei.

```
// Name der MOF-Ressource (genauer Dateiname in der .RC)
#define MOFRESOURCENAME L"MofResourceName"
```

Definition der MOF-Ressource - Auszug aus wmihandler.h

6.3 Den Treiber als WMI-Provider registrieren

Die Registrierung sollte während der Initalisierung eines neuen Gerätes durchgeführt werden, es empfielt sich, der Übersichtlichkeit halber, die WMI-Registrierung in einer eigenen Methode unterzubringen. Zuerst

muss die binäre MOF-Datei für das Gerät mittels der Methode **WdfDeviceAssignMofResourceName** registriert werden. Die Registrierung als WMI-Provider besteht nun weiterhin im Wesentlichen darin, dass die Instanzen der definierten Klassen erzeugt werden. Dies wird über die Funktion **WdfWmiInstanceCreate** erreicht. Die Klasse, welche instanziert werden soll, wird über ihre GUID bestimmt. Dazu bindet man den aus der MOF-Datei erzeugten Header ein (siehe Abschnitt 6.2), welcher die GUID der Klasse enthält. Weiterhin werden bei der Instanzierung die Set- und GetFunktionen der Klasse angegeben. Diese werden aufgerufen, wenn ein Programm auf die WMI-Daten zugreifen will bzw. es diese ändern möchte. Ebenfalls wird eine Funktion definiert, welche für die WMI-Methodenaufrufe zuständig ist. Diese Funktionen müssen natürlich nur dann definiert werden, wenn Sie wirklich benötigt werden. Die Registrierung der Beispielsklasse aus Abschnitt 6.2 würde folgendermaßen aussehen:

```
//Lokale Variablen
NTSTATUS status;
WDF_WMI_PROVIDER_CONFIG providerConfig;
WDF_WMI_INSTANCE_CONFIG instanceConfig;
WDF_OBJECT_ATTRIBUTES objectAttributes;
DECLARE_CONST_UNICODE_STRING(mofResourceName, MOFRESOURCENAME);

//MOF registrieren
status = WdfDeviceAssignMofResourceName(Device, &mofResourceName);

//WMI-Instanz der Klasse EchoWMIClass erzeugen
WDF_WMI_PROVIDER_CONFIG_INIT(&providerConfig, &EchoWMIClassGuid);
providerConfig.MinInstanceBufferSize = EchoWMIClass_SIZE;

WDF_WMI_INSTANCE_CONFIG_INIT_PROVIDER_CONFIG(&instanceConfig, &providerConfig);

instanceConfig.Register = TRUE;
instanceConfig.EvtWmiInstanceQueryInstance = EvtWmiGetInstance;
instanceConfig.EvtWmiInstanceSetInstance   = EvtWmiSetInstance;
instanceConfig.EvtWmiInstanceSetItem       = NULL;
instanceConfig.EvtWmiInstanceExecuteMethod = EvtWmiExecuteFunction;

//WMI-Instanz erzeugen
status = WdfWmiInstanceCreate(Device, &instanceConfig,
       WDF_NO_OBJECT_ATTRIBUTES, WDF_NO_HANDLE);
```

Instanzierung der WMI-Klasse

Die Rückgabewerte sollte selbstverständlich noch auf Erfolg der Instanzierung hin gestestet werden, um bei einem Fehler entsprechend reagieren zu können. Weiterführende Infomationen findet man in [doc][10] und in [wmi] (*Achtung!* In [wmi] geht es nicht speziell um KMDF Treiber.)

6.4 WMI-Daten abrufen und ändern

Will ein WMI-Client die Daten einer WMI-Klasse lesen, so wird (falls definiert) die **EvtWmiInstanceQueryInstance** Methode aufgerufen. Dieser Methode wird ein Puffer übergeben, welcher die Größe der jeweiligen Klasse hat. In dieser Methode initialisiert man die zugehörige C-Struktur der entsprechenden Klasse mit den gewünschten Werten und schreibt diese in den übergebenen Puffer.

Ähnlich funktioniert die **EvtWmiInstanceSetInstance**-Methode, hier wird allerdings die entsprechende C-Struktur als Puffer übergeben. Die Werte in der Struktur können nun geprüft und je nach Zweck angewendet werden. Diese Methode behandelt immer die gesamte Instanz der Klasse. Soll allerdings nur ein einzelner Wert der Klasse verändert werden, so kann man bei der Initialisierung dieser Instanz auch

10 Unter Windows Driver Foundation\Kernel-Mode Driver Framework\Design Guide\ Supporting WMI in Framework-Based Drivers\ Initializing WMI Support in Your Driver

eine **EvtWmiInstanceSetItem**-Methode angeben. Die Unterscheidung der einzelnen Werte erfolgt dabei über die eindeutige ID der Werte, wie sie in der MOF-Datei definiert sind. Der Einsatz dieser Methode ist dann beispielsweise sinnvoll, wenn nur wenige Werte der Instanz überhaupt geändert werden können. Da der Puffer immer nur den entsprechenden Wert anstatt die ganze Instanz übergibt, verringert sich der Overhead bei Schreiben der WMI-Werte.

6.5 WMI Methoden

Wie bereits in 6.2 erwähnt wurde ist es auch möglich in der MOF-Datei WMI-Methoden zu definieren. Diese Methoden können beliebig viele Parameter haben, welche allerdings eindeutig als Ein- oder Ausgabeparameter definiert werden müssen. Dies geschieht über ein **[in]** (Wert geht an die Methode) oder **[out]** (Methode gibt diesen Wert zurück) vor dem jeweiligen Parameter bei der Methodendefinition. Der Rückgabewert der Methode muss immer **void** sein.

Bei der Instanzierung der Klasse muss weiterhin eine **EvtWmiInstanceExecuteMethod**-Methode angegeben werden. Diese wird ausgeführt, wenn eine WMI-Methode aus der jeweiligen Instanz ausgerufen wird. Gibt es mehrere Methoden in einer Klasse, so werden diese über ihre MethodenID unterschieden. Diese ID wurde in der MOF-Datei mittels **WmiMethodId()** definiert und ist innerhalb der Klasse eindeutig. Auf die Ein- und Ausgabeparameter wird mittels einer Struktur zugegriffen, welche jeweils als Puffer an die Funktion übergeben werden. Diese Strukturen heißen *Methodenname_IN* bzw. *Methodenname_OUT*. Die genaue Deklaration dieser Struktur findet man in der automatisch erzeugten Headerdatei.

Im folgenden Beispiel wird zunächst bestimmt, welche Methode aufgerufen wurde. Anschließend wird zur Sicherheit noch die Größe des übergeben Puffers getestet, welcher bei Erfolg in die Struktur übertragen wird. Abschließend wird eine Funktion mit der Struktur als Parameter aufgerufen, welche die eigentliche Aufgabe der Methode beinhaltet.

```
NTSTATUS EvtWmiExecuteFunction(IN WDFWMIINSTANCE  WmiInstance,
       IN ULONG  MethodId, IN ULONG  InBufferSize, IN ULONG  OutBufferSize,
       IN OUT PVOID  Buffer, OUT PULONG  BufferUsed)
{
       //Lokale Variablen
       PChangeMaxLength_IN ChangeMaxLengthParams;
       WDFDEVICE device;

       // --> ChangeMaxLenght
       if(MethodId == 1)
       {
              //Puffergrösse überprüfen
              if(InBufferSize != sizeof(ChangeMaxLength_IN))
                     return STATUS_INVALID_PARAMETER;

              //Puffer übertragen
              ChangeMaxLengthParams = Buffer;

              //Weitere Arbeit in einer anderen Funktion
              ChangeMaxWriteLength(ChangeMaxLengthParams->length);
       }
       // --> AndereMethode
       if(MethodId == 2) {...} ...
}
```

EvtWmiInstanceExecuteMethod - Auszug aus wmihandler.c

6.6 WMI Leistungsindikatoren

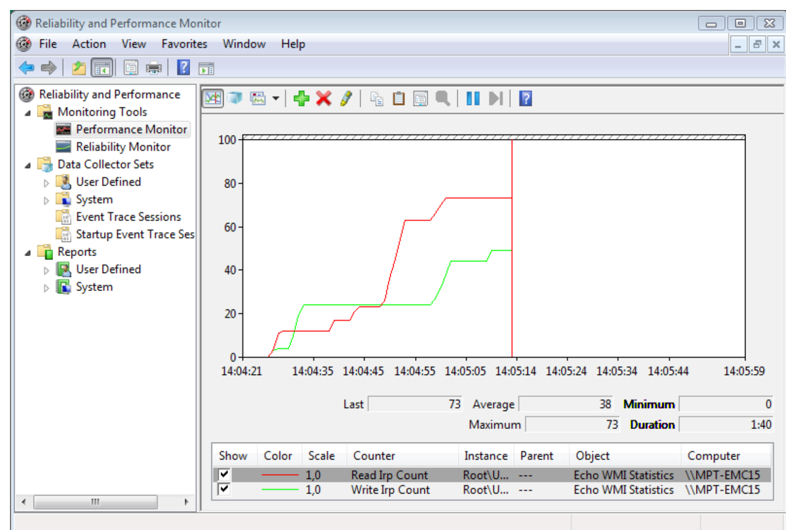
Neben den bereits beschriebenen WMI-Klassen, gibt es noch Klassen, welche so genannte Leistungsindikatoren bereitstellen. Diese Werte beinhalten normalerweise statistische Informationen und Leistungsdaten über das Gerät und können unter Windows angezeigt, aufgezeichnet und überwacht werden. Der Aufbau einer solchen Klasse unterscheidet sich nur geringfügig von den bisher behandelten "normalen" WMI-Klassen. Das folgende Beispiel definiert eine Klasse, welche zwei solcher Werte beinhaltet. Diese Werte repräsentieren die Anzahl der eingegangen Lese- und Schreibanfragen (IRPs).

```
[WMI, Dynamic,
 Provider("WmiProv"),  DisplayName("Echo WMI Statistics") : amended,
 guid("{BC721CDF-ADC1-4c47-A303-A909AA56C75E}"),  PerfDetail(100),  HiPerf]
class EchoPerfClass : Win32_PerfRawData
{
   [key, read]          string InstanceName;
   [read]               boolean Active;

    [WmiDataId(1), DisplayName("Write Irp Count") : amended,
    PerfDefault,  CounterType(0x00000000),    //    PERF_COUNTER_RAWCOUNT
    DefaultScale(0), PerfDetail(100),  read,
    Description("Gesamtanzahl der Schreibanfragen") : amended]
     uint32 WriteCount;

    [WmiDataId(2), DisplayName("Read Irp Count") : amended,
    PerfDefault, CounterType(0x00000000),    //    PERF_COUNTER_RAWCOUNT
    DefaultScale(0), PerfDetail(100), read,
    Description("Gesamtanzahl der Leseanfragen") : amended]
     uint32 ReadCount;
};
```

MOF-Definition einer Leitungsklasse

Die Klasse muss als Superklasse **Win32_PerfRawData** definiert haben, damit der Treiber später als Anbieter von Leistungsdaten korrekt erkannt wird. Die Klasse wird wie andere WMI-Klassen über eine GUID eindeutig bestimmt. Die Instanzierung der Klasse unterscheidet sich nicht von anderen WMI-Klassen (siehe Abschnitt 6.3). Da alle Werte in dieser Klasse Read-Only sind, muss man bei der Initialisierung nur eine **EvtWmiInstanceQueryInstance**-Methode angeben, damit die Werte von außerhalb ausgelesen werden können.

Die Leistungsindikatoren können mithilfe von Bordmitteln unter Windows überwacht werden. Dazu startet man über Start\Ausführen (WIN+R) das Programm **perfmon.msc.** Über das Plus-Zeichen in der Symbolleiste kann man nun neue Leistungsindikatoren zur Überwachung hinzufügen. Man wählt zunächst die entsprechende Klasse aus und anschließend die Indikatoren, welche man überwachen möchte. Nun zeichnet das Programm die Werte der Indikatoren auf und gibt diese grafisch oder in Protokollform zurück (siehe Hilfe zu PerfMon). Die Werte können alternativ auch in einen eigenen Programm überwacht werden, näheres dazu im Abschnitt 6.8.



6.7 WMI Events

Weiterhin gibt es noch die Möglichkeit so genannte WMI-Events auszulösen. Der Treiber kann ein solchen Ereignis an einen Client senden, sofern sich denn ein Client für das entsprechende Ereignis als Empfänger registriert hat. Der Treiber könnte somit beispielsweise ein Programm über einen bestimmten Fehler benachrichtigen. Dieses Programm wiederum ruft eine entsprechende WMI-Methode auf, mit welcher der Fehler behoben werden kann.

Ein Event-Klasse repräsentiert ein einzelnes Event und muss in der MOF-Datei definiert werden. Diese Event-Klasse enthält in der Regel keine weiteren DataItems und muss als Superklasse **WMIEvent** definiert haben. Die eindeutige Zuordnung erfolgt, wie bei allen WMI-Klassen, über eine eindeutige GUID.

```
[WMI, Dynamic, Provider("WMIProv"),
DisplayName("EchoEvent BufferSizeChanged"),
Guid("993AA905-48C0-4168-9B00-59C3B7B26392")]

class EchoEvent_BufferSizeChanged : WMIEvent
{
       [key, read]
       string InstanceName;

       [read]
       boolean Active;
};
```

Definition einer Event-Klasse in der MOF-Datei

Diese Event-Klasse muss zunächst wie alle anderen Klassen instanziert werden. Jedoch muss bei den Eventklassen darauf geachtet werden, dass man später beim Schicken des Ereignisses auf die Instanz der jeweiligen Klasse zugreifen muss. Es dabei empfehlenswert die jeweilige Instanz im Gerätekontext abzuspeichern, da man auf diesem in meisten Funktionen des Treibers zugreifen kann. Weiterhin muss bei der Konfiguration der Instanzierung das Flag **WdfWmiProviderEventOnly** gesetzt werden. Die daraus resultierenden Veränderungen zur bereits angesprochenen Instanzierung in Abschnitt 6.3 wurden im folgenden Beispiel fett hervorgehoben.

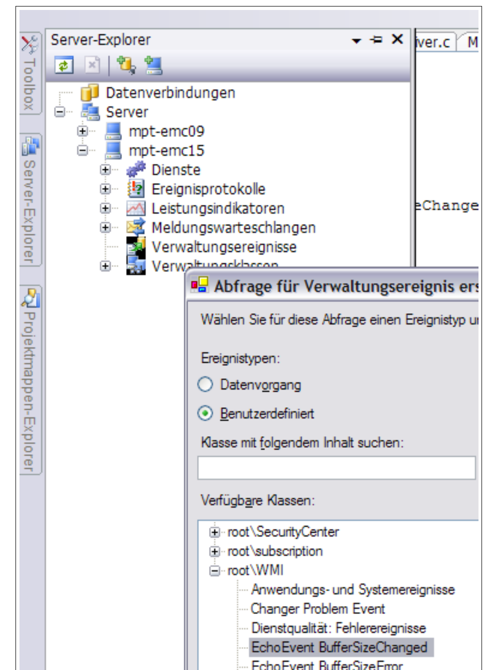
```
//WMI-EventInstanz erzeugen -> BufferSizeChanged
//WMI-Infomationen in Struktur eintragen
WDF_WMI_PROVIDER_CONFIG_INIT(&providerConfig, &eventProviderGuid_BufferChanged);
providerConfig.Flags = WdfWmiProviderEventOnly;
providerConfig.MinInstanceBufferSize = 0;

WDF_WMI_INSTANCE_CONFIG_INIT_PROVIDER_CONFIG(&instanceConfig, &providerConfig);
instanceConfig.Register = TRUE;

//WMI-Instanz erzeugen
deviceData = GetDeviceData(Device); //Gerätekontext abfragen
status = WdfWmiInstanceCreate(Device,
        &instanceConfig,
       WDF_NO_OBJECT_ATTRIBUTES,
       &deviceData->EventInstance_BufferSizeChanged);
```

Instanzierung einer WMI-Ereignisklasse - Auszug aus wmihandler.c

Das Ereignis wird mit der Methode **WdfWmiInstanceFireEvent** geschickt, wobei man zunächst prüfen sollte, ob überhaupt ein Client auf das entsprechende Ereignis wartet. Dies wird mittels der Methode **WdfWmiProviderIsEnabled** realisiert, den Provider der Instanz erhält man über die Methode **WdfWmiInstanceGetProvider.** Das folgende Codebeispiel demonstriert das Verschicken eines Ereignisses und die dazugehörigen Schritte zum Prüfen, ob ein Client auf das Ereignis wartet.

```
//Lokale Variablen
PDEVICE_CONTEXT deviceData;
WDFWMIPROVIDER wmiProvider;

//Gerätekontext abfragen
deviceData = GetDeviceData(Device);

//WMI-Provider der Instanz abfragen
wmiProvider = WdfWmiInstanceGetProvider(deviceData-
>EventInstance_BufferSizeChanged);

//Prüfen ob ein EventListener registiert ist
if(WdfWmiProviderIsEnabled(wmiProvider, WdfWmiEventControl))
{
        //Ereignis abschicken
        WdfWmiInstanceFireEvent(
                deviceData->EventInstance_BufferSizeChanged, 0, NULL);
}
```

Schicken eines Ereignisses - Auszug aus wmihandler.c



Achtung! Man sollte auf alle Fälle das IRQL beachten, in welchem man das Ereignis verschickt. Die Methode **WdfWmiInstanceFireEvent** darf nur im IRQL <= APC_LEVEL aufgerufen werden. Möchte man das Ereignis aber beim Eintreffen eines bestimmten IRP senden, so befindet man sich in der Regel im IRQL=DISPATCH_LEVEL. Sollte dieser Fall auftreten, so muss man ein **WdfWorkItem** verwenden (siehe Abschnitt 2.4).

Zum Testen der Ereignisse eignet sich der **Server Explorer** des Visual Studio, welcher auch in der Lage ist, Ereignisse von entfernten Computern zu empfangen. Man wählt zunächst ein benutzerdefiniertes Ereignis aus, welches abgefragt werden soll. Trifft das Ereignis ein, so wird dies im Ausgabefenster des Visual Studio angezeigt. Das Beispielprogramm **EchoWatch** demonstriert, wie man mittels des .NET Framework in einem eigenen Programm auf ein Ereignis seitens des Treibers wartet und entsprechend darauf reagiert (siehe Abschnitt 7.5).

6.8 Tools und Beispiele zur WMI-Nutzung

VB-Skript zum Testen

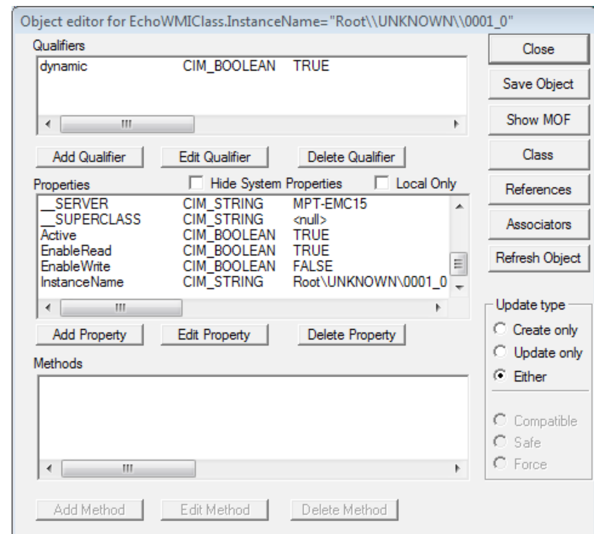
Um auf die WMI Instanz des Treibers zuzugreifen gibt es diverse Möglichkeiten und Tools. Ein grundsätzlicher Funktionstest kann mittels eines **VBSkriptes** durchgeführt werden. Dieses Skript wird beim Kompilieren mit den oben genannten Einstellungen erzeugt, kann aber nachträglich aus der MOF-Datei erzeugt werden.

```
mofcomp -WMI -B:driver.bmf driver.mof          //Komplierung der MOF
wmimofck -m -ttest.vbs driver.bmf              //Erzeugen des Testskripts
```

Dieses Skript wird bei laufenden Treiber ausgeführt und erzeugt eine LOG-Datei, in welcher man die aktuellen Werte der WMI-Klassenattribute wiederfindet. Werden hier keine Werte, sondern nur die Klassennamen angezeigt, sollte man die WMI-Initialisierungsroutine und die GetQuery-Funktionen überprüfen bzw. durch entsprechende Debugmeldungen nachvollziehen, ob Sie korrekt ausgeführt werden.

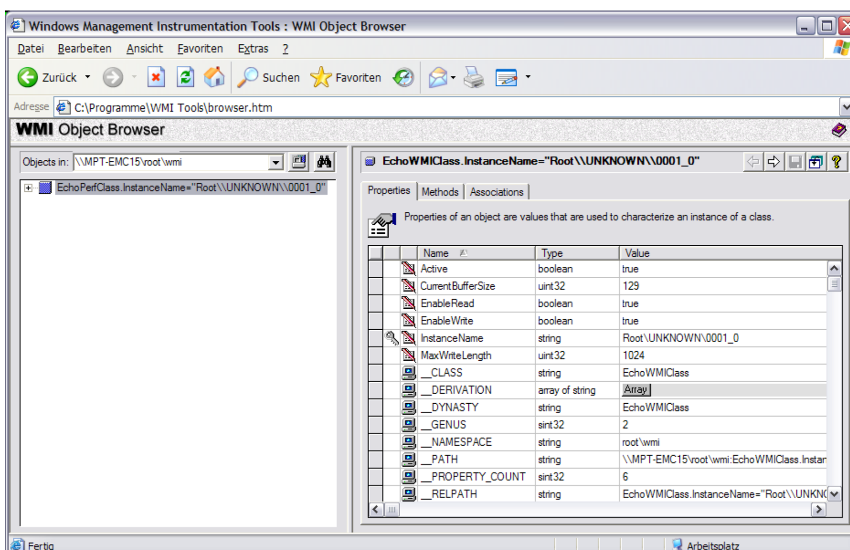
Das Windows Tool WBEMTest

Zu den Bordmitteln von Windows gehört ebenfalls ein Tool zum Testen der WMI-Funktionen, dieses nennt sich **WBEMTest** und befindet sich im Windowsordner unter ".*system32**wbem*\\". Hier muss man sich zunächst mit dem Namespace des WMI-Dienstes verbinden, welcher den Pfad "*root**WMI*" besitzt. Als Nächstes kann man die WMI-Instanzen des Treibers über den Button "Instanzen aufzählen..." abfragen, indem man den gewünschten Klassenname angibt (wie in der MOF-Datei definiert). Das Programm findet normalerweise nur eine Instanz dieser Klasse, welche man dann Auswählen und Anzeigen kann. Nur im Falle, dass es mehrere Geräte gibt, welche den Treiber verwenden, wird man hier mehrere Instanzen dieser Klasse zur Auswahl haben. Im daraufhin erscheinenden Dialog werden die Eigenschaften und Methoden der Klasseninstanz angezeigt. Weiterhin besteht die Möglichkeit die Eigenschaften dieser Klasseninstanz zu ändern, sofern der Treiber die entsprechenden SetInstance- bzw. SetItem- Funktionen bereitstellt.



Der WMI Object Browser

Weiterhin gibt es von Microsoft ein Programmpacket namens WMI Administrative Tools unter [dl-wmi], welches unter anderem das Programm WMI Object Browser enthält. Der WMI Object Browser verwendet ActiveX und kann somit nur mit dem Internet-Explorer aufgerufen werden. Der Browser ermöglicht es auf die Instanzen von WMI-Klassen zuzugreifen, ähnlich wie es das Tool WBEMTest tut. Dazu verbindet man sich zunächst mit dem WMI-Namespace (root\WMI). Mit einem Klick auf den Suchen-Button wird eine Liste mit verfügbaren Klassen angezeigt, aus welcher man ein oder mehrere auswählen kann. Als nächstes werden, genau wie bei dem oben beschriebenen Programm, die Instanzen der Klasse angezeigt. Nach der Auswahl einer dieser Instanzen, wird diese angezeigt. Der WMI Object Browser bietet auch die Möglichkeit an, sich mit einer WMI-Klasse auf einem entfernten Rechner zu verbinden. Dazu klickt man auf den Button links neben der Suchen-Schaltfläche und gibt unter dem Feld "Machine Name" den Namen oder die IP-Adresse des entfernten Rechners an. Die restlichen Schritte unterscheiden sich nicht von der Arbeit auf dem lokalen System.



Nutzung über das .NET Framework

Weiterhin ist es möglich auf die WMI-Daten über eine Programmiersprache zuzugreifen. Neben den Schnittstellen für C++, bietet sich das .NET Framework von Microsoft an. Das Framework stellt im Namespace **System.Management** verschiedene Klassen bereit, welche das Arbeiten mit Windows Management Instrumentation ermöglichen.

Um eine WMI Instanz zu laden fragt man zunächst alle Instanzen einer Klasse in einem bestimmten Namespace ab. Dies geschieht in einem SQL-ähnlichem Dialekt. Anschließend durchläuft man die verschiedenen Instanzen, um auf die Werte und Funktionen dieser Instanz zuzugreifen. Das folgende Codebeispiel demontiert den Ablauf einer Abfrage der WMI-Daten.

```
bool write, read;
uint length;

//WMI Objekt suchen
ManagementObjectSearcher searcher = new ManagementObjectSearcher("root\\WMI",
    "SELECT * FROM EchoWMIClass");

//Durchlaufen der Instanzen
foreach (ManagementObject wmiObj in searcher.Get())
{
    //Auslesen der Daten
    write    = (bool)wmiObj["EnableWrite"];
    read     = (bool)wmiObj["EnableRead"];
    length   = (uint)wmiObj["MaxWriteLength"];

    //Ausgabe der Infomationen
    System.Console.WriteLine("Write = {0}; Read = {1}; Length = {2}",
      write, read, length);
}
```

Codebeispiel: WMI-Abfrage

Kann man auf die entsprechenden Daten auch schreibend zugreifen, so können die entsprechenden Werte über eine simple Wertzuweisung einfach geändert werden. Wurden die entsprechenden Änderungen durchgeführt, muss man abschließend die **Put()**-Methode des WMI-Objektes aufrufen, um die Werte in die entsprechende WMI-Instanz zu schreiben. Ein Änderung des EnableWrite-Attributes aus dem obigen Codebeispiel könnte dementsprechend folgendermaßen aussehen:

```
wmiObj["EnableWrite"] = false;
wmiObj.Put();
```

Codebeispiel: WMI-Daten schreiben

Desweiteren besteht ebenfalls die Möglichkeit auf WMI-Methoden mittels des Framework zuzugreifen. Die Instanz der Klasse wird dabei wie bereits beschrieben abgefragt. Zum Ausführen einer WMI-Methode genügt es die Methode **InvokeMethod()** des WMI-Objektes aufzurufen. Die Ein- und Ausgabeparameter werden dabei in einem *ManagementBaseObject* untergebracht und beim Aufruf mit übergeben. Für die Parameter müssen exakt die gleichen Namen verwendet werden, wie sie in der MOF-Datei der WMI-Klasse deklariert wurden.

```
ManagementBaseObject inParams = wmiObj.GetMethodParameters("ChangeMaxLength");
inParams["length"] = 2048;

ManagementBaseObject outParams = wmiObj.InvokeMethod(
                                   "ChangeMaxLength", inParams, null);
```

Codebeispiel WMI-Methodenaufruf

Um auf WMI-Leistungsindikatoren zuzugreifen bietet das Framework die **PerformanceCounter**-Klasse, welche sich im Namspace **System.Dianostics** befindet. Dieser Klasse teilt man den Klassennamen und den Instanznamen, sowie den Namen des Wertes mit, welcher überwacht werden soll. Jede Klasse kann dabei nur einen Wert überwachen. Wurde die Klasse korrekt initialisiert, kann man über die Methode **NextValue()** den aktuellen Wert des Leistungsindikators abrufen.

```
System.Diagnostics.PerformanceCounter pcIrpCount;

//WMI-Daten eintragen
pcIrpCount.CategoryName = "Echo WMI Statistics";
pcIrpCount.CounterName = "Total Irp Count";
pcIrpCount.InstanceName = "Root\\UNKNOWN\\0001_0";

//Initalisierungen
((System.ComponentModel.ISupportInitialize)(pcIrpCount)).EndInit();

//Daten abfragen
System.Console.Writeln("IRPs gesamt : " + (int)pcIrpCount.NextValue());
```

Codebeispiel: WMI Leistungsindikatoren abfragen

Der Quelltext des ECHO Testprogramms demonstriert alle hier kurz vorgestellten Techniken zum Zugriff auf die WMI Daten (siehe Abschnitt 7.6).

7. Der ECHO Treiber

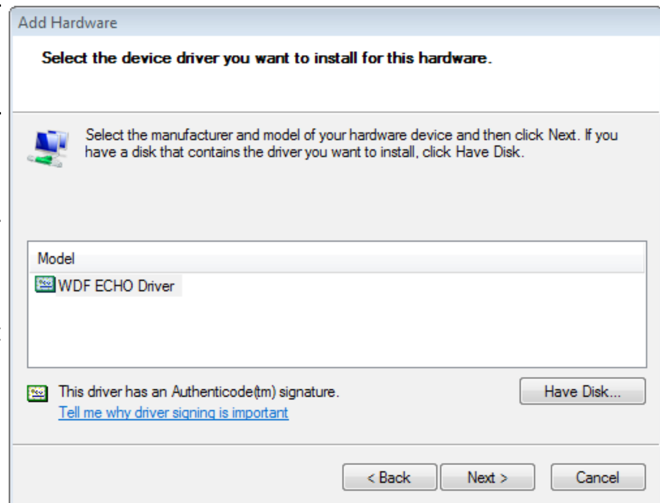
7.1 Einführung

Der ECHO Treiber ist ein minimaler Treiber, welcher auf den Kernel-Mode Driver Framework basiert. Der ECHO-Treiber ist Bestandteil der WDK Beispieltreiber. Er simuliert ein Gerät, welches mit I/O-Anfragen umgehen kann. Es können Bytes an das Gerät gesendet werden und anschließend können diese Bytes wieder aus dem Puffer des Gerätes ausgelesen werden. Den Quellcode dieses Gerätes findet man im Pfad des Windows Driver Kit ([dl-wdk]) unter '\src\kmdf\echo'.

Der Quellcode des Treibers, der diesem Dokument beiliegt basiert auf den Beispieltreiber des WDK, wurde aber an vielen Stellen angepasst und erweitert. So bietet der anpasste ECHO-Treiber Unterstützung für Software Tracing und registriert sich als WMI-Provider. Weiterhin stellt dieser Treiber Leistungsindikatoren bereit. Dieser Treiber beinhaltet alle Technologien, die in diesem Dokument vorgestellt wurden.

Dem Treiber liegen außerdem noch eine Bibliothek und zwei Testprogramme im Quellcode bei bei, um die Ansteuerung des Treibers aus dem User-Mode heraus zu demonstrieren. Die Bibliothek des Treibers ist eine DLL welche in C geschrieben ist und mit welcher der Treiber angesteuert werden kann. Die Testprogramme basieren auf dem .NET Framework.

7.2 Installation des Beispieltreibers



Der Treiber liegt kompiliert im Ordner **ready** für Windows XP und Windows Vista vor (X86, Checked Build). Und kann über den Hardwareassistenten von Windows hinzugefügt werden. Vor der Installation sollte man allerdings noch das Testzertifikat aus dem Ordner **cert** auf den System installieren (siehe Abschnitt 3.2). Bei der Installation unter Windows Vista muss man zusätzlich eine Warnmeldung bestätigen, welche besagt, dass der Herausgeber des Treibers nicht ermittelt werden kann. Nach der erfolgreichen Installation findet man im Gerätemanager eine neue Gerätegruppe namens **DummyTreiber**, welche den **WDF ECHO Driver** enthält. Weiterhin besteht natürlich auch die Möglichkeit den Treiber selbst mit einer **Windows Driver Kit Build Umgebung** zu kompilieren. Dazu kann man das Skript **makeit.cmd** im Hauptordner verwenden, welches die Treiber kompiliert und anschließend auch signiert. Dazu muss man das Skript allerdings zunächst an sein System anpassen, indem man im Abschnitt BUILD-VARIABLEN die Pfade zum WindowsDriver Kit [dl-wdk] und zu dem Tool Inf2Cat [dl-wql] einträgt. Weiterhin muss noch der Name und der Zertifikatsspeicherort des zu verwendeten Zertifikates angegeben werden. Man kann hier auf das Testzertifikat im Ordner **cert** zurückgreifen oder selbst eines erstellen (siehe Abschnitt 3.2). Den kompilierten und signierten Treiber findet man im Ordner **ready**. Ist der Treiber korrekt auf dem System installiert, so kann er mittels des Testprogramms auf seine Funktionalität hin überprüft werden.

7.3 Funktionalität des Treibers

Der Treiber bzw. das virtuelle Gerät dient lediglich zu Demonstrationszwecken und bietet daher eine eher fragwürdige Funktionalität. Der Treiber reagiert auf Read- und WriteRequests (IRP), dabei wird beim Schreiben der Inhalt des übergebenen Puffers abgespeichert und beim nächsten Lesevorgang wieder zurückgegeben. Steht bei einem Schreibvorgang bereits etwas im Puffer, so wird dies überschrieben. Über WMI-Funktionen ist es möglich die Abarbeitung von Lese- oder Schreibanfragen zu blockieren. Weiterhin kann die maximale Puffergröße über eine WMI-Funktion geändert werden. Die Voreinstellung beträgt hier 1024 Byte und das Maximum liegt bei 8096 Byte. Weiterhin übermittelt der Treiber statistische Daten als WMI-Leistungsindikatoren an das Betriebssystem, welche über eine WMI-Funktion zurückgesetzt werden können. Zur Überwachung des Gerätes sendet der Treiber bei bestimmten Ereignissen WMIEvents an einen WMI-Client.

Der Treiber schickt außerdem eine Reihe von Debugmeldungen an den Kernel-Debugger, sofern Windows im Debugmodus läuft und die **DbgPrintOn**-Funktion (siehe Abschnitt 4.5) des Framework aktiviert ist. Desweiteren unterstützt der ECHO-Treiber Software Tracing (siehe Abschnitt 5.4).

7.4 Die Treiber DLL

Die DLL dient zum Ansteuern des Treibers aus dem User-Mode heraus. Dazu stellt die DLL folgende Methoden bereit:

DLL-Funktion	Beschreibung
`BOOLEAN` **`InitEchoDevice`**`(void)`	Initialisiert das Gerät. Diese Methode ermittelt den Gerätepfad und öffnet das Gerät mit wahlfreien Zugriff **Rückgabe** TRUE, wenn die Initialisierung erfolgreich war FALSE, wenn nicht
`ULONG` **`WriteToDevice`**`(` `IN PUCHAR WriteBuffer, IN ULONG Length)`	Schreibt den angegebenen Puffer auf das Gerät **Parameter** WriteBuffer - Pointer auf Puffer Length - Puffergrösse **Rückgabe** Anzahl der geschriebenen Bytes
`ULONG` **`ReadFromDevice`**`(` `OUT PUCHAR ReadBuffer, IN ULONG Length)`	Liest einen Puffer mit der angegebenen Größe vom Gerät **Parameter** ReadBuffer - Pointer auf Ausgabepuffer Length - Puffergrösse **Rückgabe** Anzahl der der gelesenen Bytes
`PUCHAR` **`GetDllVersion`**`(void)`	Gibt die Versionsnummer der DLL als Charpointer zurück

7.5 WMI Funktionalität des Treibers

Der Treiber unterstützt WMI und stellt verschiedene Klassen zur Verfügung.

EchoWMIClass - Echo WMI Klasse

WMIDataItem	Beschreibung
`EnableWrite` `(BOOLEAN, RW)`	Schaltet die Annahme von Schreibanfragen an (TRUE) oder aus (FALSE)
`EnableRead` `(BOOLEAN, RW)`	Schaltet die Annahme von Leseanfragen an (TRUE) oder aus (FALSE)
`MaxWriteLength` `(UINT32, RO)`	Gibt die derzeit maximal mögliche Puffergröße zurück
`CurrentBufferSize` `(UINT32, RO)`	Gibt die aktuelle Puffergröße zurück

WMIMethode	Beschreibung
`ChangeMaxLength``([in] uint32 length)`	Ändert die die maximale Puffergröße auf **length** (maximal 8096)
`ResetStats``(void)`	Setzt die Leistungsindikatoren zurück

EchoPerfClass - Echo WMI Statistics

WMI Leistungsindikator	Beschreibung
`Total Irp Count`	Anzahl aller Anfragen (IRP)
`Success Irp Count`	Anzahl aller erfolgreichen Anfragen
`Write Irp Count`	Anzahl aller Schreibanfragen
`Success Write Irp Count`	Anzahl aller erfolgreichen Schreibanfragen
`Read Irp Count`	Anzahl aller Leseanfragen
`Success Read Irp Count`	Anzahl aller erfolgreichen Leseanfragen

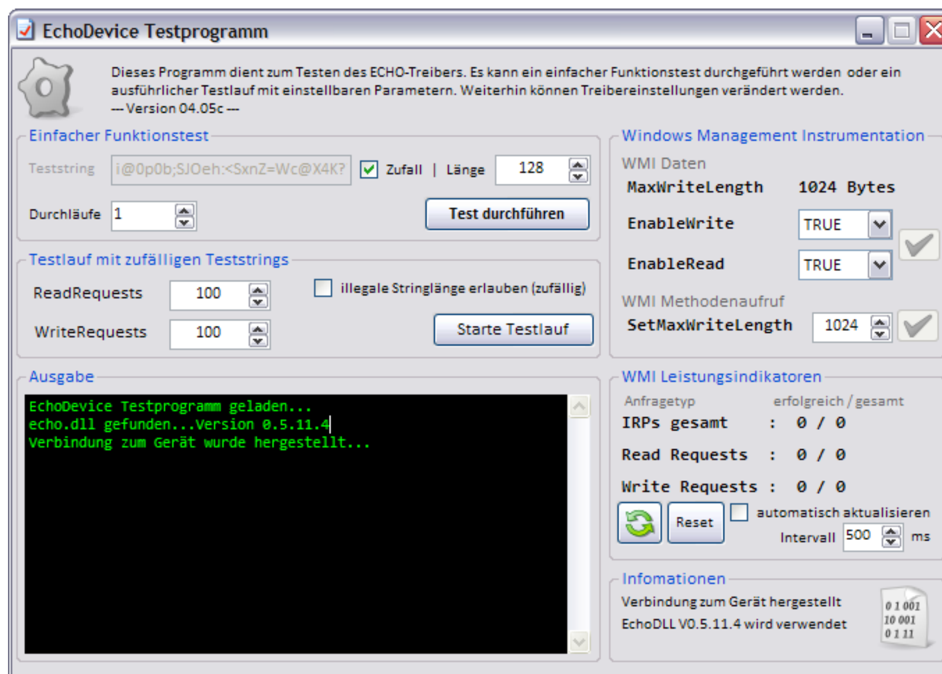
WMIEvents

Ereignis (WMIEvent)	Beschreibung
`EchoEvent_BufferSizeError`	Tritt auf, wenn die übermittelte Puffergröße die eingestellte maximale Puffergröße des Gerätes übersteigt
`EchoEvent_BufferSizeChanged`	Tritt auf, wenn die maximale Puffergröße geändert wurde.

7.6 EchoDevice Testprogramm

Mit dem Testprogramm kann man alle vom Treiber bereitgestellten Funktionen aufrufen und testen. Der **normale Test** erlaubt dabei ein Teststring auf das Gerät zu schreiben und anschließend wieder von diesem zu lesen. Der Eingabe- und Ausgabestring werden anschließend verglichen. Das Ergebnis des Tests kann man im Ausgabefenster des Programms erfahren.

Weiterhin bietet das Programm noch die Möglichkeit einen **Testlauf** zu starten. Hierbei wird vorgegeben, wieviel Lese- und Schreibanfragen an den Treiber geschickt werden sollen. Bei jeder Schreibanfrage wird ein zufälliger Teststring erzeugt, welcher bei Bedarf auch eine ungültige Länge haben kann. Eine Überprüfung findet hierbei nicht statt, da dieser Test in erster Linie dazu dient die Geschwindigkeit und Belastbarkeit des Treibers zu testen. Außerdem kann dieser Testmodus dazu genutzt werden die Funktionsfähigkeit der Leistungsindikatoren zu testen.

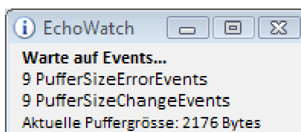


Das Programm kann auch auf auf die bereitgestellten WMI-Funktionen des Treibers zugreifen. Neben der einfachen Abfrage von WMI-Werten (MaxWriteLength), können diese auch geändert werden (EnableRead, EnableWrite). Desweiteren ist das Programm in der Lage die Leistungsindikatoren des Treibers abzufragen und anzuzeigen. Außerdem ist es möglich WMI-Funktionen über die Oberfläche aufzurufen (ChangeMaxLenght, Reset).

7.7 EchoWatch

Das Programm EchoWatch ermöglicht die Überwachung von WMI-Ereignissen, die seitens des Treiber verschickt werden. Das Programm demonstriert wie man automatisch auf solche Ereignisse reagieren kann.

Empfängt das Programm ein **EchoEvent_BufferSizeError**, so erhöht es über die WMI-Methode **ChangeMaxLength** die maximale Puffergröße um 128 Bytes. Wird beispielsweise mit dem Testprogramm ein normaler Test mit einer zunächst ungültigen Länge mehrmals durchlaufen, so sorgt EchoWatch dafür das diese Test ohne weiteres Zutun irgendwann gelingt (sofern die Stringlänge nicht grösser als 8096 Bytes ist, da dies die Obergrenze ist).



8. Quellverzeichnis und weiterführende Informationen

[chk] Interoffice Memorandum - Where's The Checked Build?
http://www.osronline.com/article.cfm?article=259

[csw] Kernel-Mode Code Signing Walkthrough
http://www.microsoft.com/whdc/winlogo/drvsign/kmcs_walkthrough.mspx

[dex] KMDF Debugging Extensions
http://www.microsoft.com/whdc/driver/wdf/KMDF-dbgext.mspx

[dok] Window Driver Kit – Dokumentation
Bestandteil des Windows Driver Kit - http://www.microsoft.com/whdc/devtools/WDK/default.msp

[log] How to Use the KMDF Log
http://www.microsoft.com/whdc/driver/tips/KMDF_IfrLog.mspx

[osr] OSR Online
http://www.osronline.com

[per] WMI Revisited - Instrumentation and Integration with PerfMon
http://www.osronline.com/article.cfm?article=95

[sym] Debugging Tools and Symbols: Getting Started
http://www.microsoft.com/whdc/devtools/debugging/debugstart.mspx

[wmi] WMI - What it is...Why Driver Writers Should Care
http://www.osronline.com/article.cfm?article=67

[wmi2] A Simple Guide to WMI Provider
http://www.codeguru.com/csharp/csharp/cs_network/wmi/article.php/c6035/

9. Downloadadressen

[dl-dbg] Debugging Tools for Windows
http://www.microsoft.com/whdc/devtools/debugging/default.mspx

[dl-sdk] Windows Software Development Kit

> **Für Windows Vista**
> http://www.microsoft.com/downloads/details.aspx?familyid=7614FE22-8A64-4DFB-AA0C-DB53035F40A0&displaylang=en
>
> **Für Windows XP / 2000 / Server 2003**
> http://www.microsoft.com/downloads/details.aspx?FamilyId=484269E2-3B89-47E3-8EB7-1F2BE6D7123A&displaylang=en

[dl-sym] Windows Symbole für verschiedene Betriebssysteme
http://www.microsoft.com/whdc/devtools/debugging/symbolpkg.mspx

[dl-wdk] Windows Driver Kit
http://www.microsoft.com/whdc/devtools/WDK/default.mspx

[dl-wmi] WMI Administrative Tools
http://www.microsoft.com/downloads/details.aspx?FamilyId=6430F853-1120-48DB-8CC5-F2ABDC3ED314&displaylang=en

[dl-wql] WinQual Submission Tools
https://winqual.microsoft.com/member/SubmissionWizard/controls/WinqualSubmissionTool.msi