Das Gerätetreibermodell von Windows Embedded CE 6.0 R2

Treiber sind größter und wichtigster Bestandteil eines Board Support Packages

Das Betriebssystem Windows Embedded CE bringt mit dem OEM Adaption Layer eine wichtige „Isolationsschicht" mit, die hardware-abhängige und hardware-unabhängige Teile des Quellcodes trennt. Genauso ist es mit den Treibern: Auch sie enthalten hardware-abhängige und -unabhängige Teile. Gleichzeitig sind Treiber die wichtigsten Bestandteile eines Board Support Packages, kurz BSP.

Von Ralf Ebert und Rudi Swiontek

Gerätetreiber sind Software-Komponenten, über die das Betriebssystem und die Anwendungen mit den Peripheriegeräten eines Zielgeräts kommunizieren. Das kann der PCI-Bus sein, es können aber auch Tastatur, Maus, serielle Ports, Bildschirm, Netzwerkadapter und Speichergeräte sein. Anstatt direkt auf die Hardware zuzugreifen, lädt das Betriebssystem die entsprechenden Gerätetreiber und spricht die Geräte mittels Funktionen und E/A-Diensten (APIs, Application Programming Interfaces) an. Auf diese Art bleibt die Windows-Embedded-CE-6.0-Architektur flexibel, erweiterbar und unabhängig von der Hardware.

Die Gerätetreiber enthalten den hardware-spezifischen Code. Zusätzlich zu den Standardtreibern in Windows Embedded CE können benutzerdefinierte Treiber implementiert werden, um weitere Peripheriegeräte zu unterstützen. Die Gerätetreiber bilden den größten Teil eines Board Support Packages (BSP). Das BSP ist ein hardware-spezifisches Bindeglied zwischen dem Betriebssystem und der Hardware. Derzeit unterstützt Windows Embedded CE vier verschiedene Prozessorfamilien: ARM, MIPS, SH4 und x86. Ebenfalls zum BSP gehört der Bootloader, der den Startup-Code im OAL (OEM Adaptation Layer) enthält, der die komplette Software lädt und startet.

User-Mode- und Kernel-Mode-Treiber

Ein Gerätetreiber in Windows Embedded CE ist eine DLL (Dynamic-Link Library). Die Treiber umfassen Funktionen und Programmlogik zum Initialisieren und Kommunizieren mit der Hardware. Software-Entwickler verwenden wiederum die Funktionen der Treiber in ihren Anwendungen, um auf die Hardware zuzugreifen (**Bild 1**). Es können User-Mode-Treiber entwickelt werden, die zwar sehr robust sind und in einen *Udevice.exe*-Prozess eingebettet sind, die aber keine privilegierten Befehle (Interrupts) ausführen können. Durch einen fehlerhaften User-Mode-Treiber wird nur der User-Prozess beendet.

Die Kernel-Mode-Treiber haben vollen Zugriff auf alle Systemressourcen und laufen sehr schnell ab. Ein

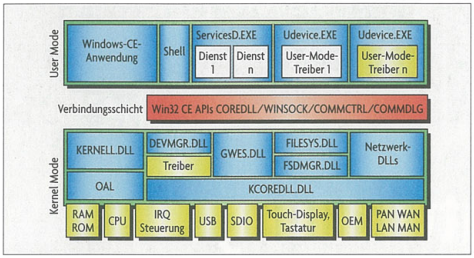

Bild 1. User-Mode-Treiber können nicht viel Schaden im System anrichten, haben aber auch nur beschränkten Zugriff auf die Hardware. Für Kernel-Mode-Treiber gibt es keine Schranken, dafür können sie das gesamte System zum Absturz bringen.

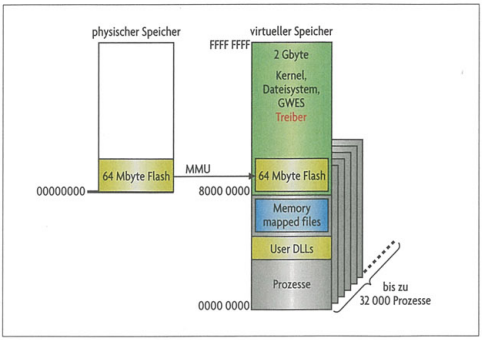

Bild 2. Aufteilung des physikalischen Speichers in Windows Embedded CE. Der virtuelle Adressraum von 4 Gbyte setzt sich zusammen aus 2 Gbyte für den Kernel, in den auch die Treiber geladen werden, sowie je 512 Mbyte für gemeinsam genutzten MMF- und DLL-Speicher. 1 Gbyte bleibt für Programmcode.

Programmfehler im Treiber führt zum Anhalten des Systems. Alle Kernel-Mode-Treiber werden zum Kernel hinzugebunden und sind mit ihm in der Datei *NK.exe* zusammengefasst.

Wenn ein Gerätetreiber ein bekanntes API verwendet, beispielsweise das Device Driver Interface (DDI), kann der Treiber als Bestandteil des Betriebssystems geladen werden, was z.B. bei einem Bildschirmtreiber oder dem Treiber für ein Speichergerät der Fall ist. Anwendungen können dann die Windows-API-Standardfunktionen aufrufen, beispielsweise *ReadFile* oder *WriteFile*, um auf das Peripheriegerät zuzugreifen, unabhängig von den Informationen über die physische Hardware. Dank der Peripherietreiber lassen sich Peripheriegeräte gut ansteuern, ohne dass die Anwendungen neu programmiert werden müssen. Der Peripherietreiber wird einfach dem Betriebssystem hinzugefügt.

Bild 2 zeigt, wie der physikalische Speicher durch die MMU (Memory Management Unit) in den 2 Gbyte großen, virtuellen Adressbereich des Kernels eingeblendet wird. Der virtuelle Speicher eines Prozesses beträgt ebenfalls 2 Gbyte. Darin enthalten sind der 512 Mbyte große MMF-Shared-Speicher für Interprozess-Kommunikation und der 512 Mbyte große Speicherbereich für die Anwender-DLLs, über die man auch über Prozessgrenzen hinweg kommunizieren kann. Denn der Programmcode einer DLL kann von mehreren Prozessen genutzt werden, der verwendete Speicher kann entweder prozessglobal oder „shared" sein. Der verbleibende 1 Gbyte große Speicher ist für den Programmcode pro Prozess.

Was sind systemeigene Treiber und Streamtreiber?

Ein Windows-Embedded-CE-Gerätetreiber ist eine DLL, die die *DllMain*-Standardfunktion als Einsprungspunkt verwendet, damit ein übergeordneter Prozess den Treiber durch Aufruf der Funktion *LoadLibrary* oder *LoadDriver* laden kann. Mit der Funktion LoadLibrary geladene Treiber können ausgelagert werden (Paging). Das Betriebssystem lagert mittels LoadDriver geladene Treiber jedoch nicht aus. Ob-

wohl alle Treiber den DllMain-Einsprungspunkt verwenden, unterstützt Windows Embedded CE zwei Treibertypen: systemeigene Treiber (native driver) und Streamtreiber (stream driver). Zu den systemeigenen CE-Treibern zählen Eingabe- und Ausgabetreiber, Tastaturtreiber und Touchscreen-Treiber. Die Spezifikation der Treiber wird von Microsoft zur Verfügung gestellt. Das GWES (Graphics, Windowing and Events Subsystem) lädt und verwaltet diese Treiber direkt.

Streamtreiber verwenden hingegen standardisierte Funktionen. Diese Treiber werden über den Gerätemanager geladen und verwaltet. Damit der Gerätemanager mit einem Streamtreiber interagieren kann, muss der Treiber die Funktionen *xxx_Init, xxx_Deinit, xxx_Open, xxx_Close, xxx_Read, xxx_Write, xxx_Seek* und *xxx_IOControl* implementieren. In vielen Streamtreibern sorgen die Funktionen xxx_Read, xxx_Write und xxx_Seek für den Zugriff auf den Streaminhalt, obwohl nicht alle Peripheriegeräte Streamgeräte sind. Wenn die Anforderungen des Geräts über xxx_Read, xxx_Write und xxx_Seek hinausgehen, kann die Funktion xxx_IOControl verwendet werden, um die erforderlichen Funktionen zu implementieren. Die universelle Funktion xxx_IOControl kann die besonderen Anforderungen eines Streamtreibers erfüllen. Beispielsweise lassen sich die Treiberfunktionen erweitern, indem man den benutzerdefinierten IOCTL-Befehlscode sowie Eingabe- und Ausgabepuffer übergibt.

Monolithische und mehrschichtige Treiber

Systemeigene Treiber und Streamtreiber unterscheiden sich ausschließlich in den bereitgestellten APIs. Beide Treibertypen können ein monolithisches oder mehrschichtiges Design verwenden. Ein monolithischer Treiber verwendet eine DLL, um die Schnittstelle zum Betriebsystem und zu den Anwendungen sowie die Logik für die Hardware zu implementieren. Die Entwicklungskosten für monolithische Treiber sind im Allgemeinen höher als für mehrschichtige Treiber. Monolithische Treiber bieten jedoch einige Vorteile. Der wichtigste Vorteil ist die Leistungssteigerung, die durch das Verhindern zusätzlicher Funktionsaufrufe zwischen separaten Schichten der Treiberarchitektur erzielt wird. Die Speicheranforderungen sind im Vergleich zu mehrschichtigen Treibern ebenfalls etwas niedriger. Ein monolithischer Treiber ist möglicherweise auch für unübliche Hardware-Komponenten besser geeignet. Wenn für ein Treiberprojekt kein mehrschichtiger Treibercode vorhanden ist, kann ein Treiber in einer monolithischen Architektur implementiert werden. Dies trifft insbesondere dann zu, wenn monolithischer Quellcode verfügbar ist.

Um die Wiederverwendung von Code zu vereinfachen und die Entwicklungskosten sowie den Mehraufwand zu reduzieren, unterstützt Windows Embedded CE eine mehrschichtige Treiberarchitektur, basierend auf MDD (Model Device Driver) und PDD (Platform Device Driver). MDD und PDD stellen einen zusätzlichen Abstraktionslayer für Treiberupdates und die Beschleunigung der Treiberentwicklung für neue Hardware dar. Der MDD-Layer umfasst die Schnittstelle zum Betriebssystem und zu den Anwendungen. Auf der Hardwareseite stützt sich MDD hingegen auf den PDD-Layer. Der PDD-Layer implementiert die Funktionen für die Kommunikation mit der Hardware.

Beim Portieren eines mehrschichtigen Treibers auf neue Hardware muss der Code im MDD-Layer normalerweise nicht geändert werden. Außerdem kann ein mehrschichtiger Treiber problemlos dupliziert werden; Funktionen können hinzugefügt oder entfernt werden, anstatt einen neuen Treiber zu erstellen. Viele der in Windows Embedded CE integrierten Treiber nutzen die Vorteile der mehrschichtigen Treiberarchitektur.

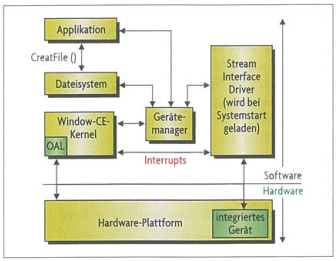

Bild 3. Streamtreiber werden in Windows Embedded CE vom Gerätemanager verwaltet.

Implementieren eines Streamtreibers

In Windows Embedded CE ist ein Streamtreiber ein Gerätetreiber, der das Streamschnittstellen-API implementiert. Unabhängig von den Hardware-Anforderungen stellen alle CE-Streamtreiber die Streamschnittstellenfunktionen für das Betriebssystem bereit, damit der Gerätemanager in Windows Embedded CE diese Treiber laden und verwalten kann. Streamtreiber sind für E/A-Geräte geeignet, die als Quellen für Datenstreams agieren, beispielsweise als integrierte Hardware-Komponenten und Peripheriegeräte. Ein Streamtreiber kann jedoch auch auf andere Treiber zugreifen, um Anwendungen den Zugriff auf die Hardware zu gewähren. Ein Entwickler muss mit den Streamschnittstellenfunktionen und deren Implementierung vertraut sein, um funktionelle und zuverlässige Streamtreiber zu entwickeln.

Der Gerätemanager (**Bild 3**) von Windows Embedded CE ist die Betriebssystemkomponente, die die Streamgerätetreiber auf dem System verwaltet. Der OAL (Oal.exe) lädt den Kernel (Kernel.dll) und der Kernel lädt den Gerätemanager während des

Funktion	Beschreibung
XXX_Init	Der Gerätemanager ruft diese Funktion auf, um während des Starts einen Treiber zu laden oder einen *ActivateDeviceEx*-Aufruf zu beantworten, um die Hardware und Speicherstrukturen für das Gerät zu initialisieren.
XXX_PreDeinit	Der Gerätemanager ruft diese Funktion vor XXX_Deinit auf, damit der Treiber deaktivierte Threads reaktivieren und offene Handles außer Kraft setzen kann, um die Neuinitialisierung zu beschleunigen. Anwendungen rufen diese Funktion nicht auf.
XXX_Deinit	Der Gerätemanager ruft diese Funktion auf, um die Speicherstrukturen und anderen Ressourcen neu zu initialisieren oder zuzuordnen, wenn *DeActivateDevice* aufgerufen wird, nachdem ein Treiber deaktiviert oder entladen wurde.
XXX_Open	Der Gerätemanager ruft diese Funktion auf, wenn eine Anwendung *CreateFile* aufruft, um den Zugriff auf das Gerät für Lese- oder Schreibvorgänge anzufordern.
XXX_PreClose	Der Gerätemanager ruft diese Funktion auf, damit der Treiber deaktivierte Threads reaktivieren und Handles außer Kraft setzen kann, um das Entladen zu beschleunigen. Anwendungen rufen diese Funktion nicht auf.
XXX_Close	Der Gerätemanager ruft diese Funktion auf, wenn eine Anwendung eine offene Instanz des Treibers schließt, beispielsweise durch den Aufruf von *CloseHandle*. Der Streamtreiber muss den Speicher und die Ressourcen neu zuordnen, die während des vorherigen XXX_Open-Aufrufs zugeordnet wurden.
XXX_Read	Der Gerätemanager ruft diese Funktion nach einem *ReadFile*-Aufruf auf, um die Daten vom Gerät zu lesen und zu übergeben. Auch wenn das Gerät keine Daten bereitstellt, muss der Streamtreiber die Funktion zwecks Kompatibilität mit dem Gerätemanager implementieren.
XXX_Write	Der Gerätemanager ruft diese Funktion nach einem *WriteFile*-Aufruf auf, um die Daten an das Gerät zu übergeben. Wie XXX_Read ist XXX_Write erforderlich, kann aber einen leeren Wert haben, wenn das Gerät, beispielsweise ein Kommunikationseingang, Schreibvorgänge nicht unterstützt.
XXX_Seek	Der Gerätemanager ruft diese Funktion nach einem *SetFilePointer*-Aufruf auf, um den Datenzeiger für Lese- und Schreibvorgänge auf einen bestimmten Punkt im Datenstream zu verweisen. Wie XXX_Read und XXX_Write kann die Funktion einen leeren Wert haben, muss aber für die Kompatibilität mit dem Gerätemanager exportiert werden.
XXX_IOControl	Der Gerätemanager ruft diese Funktion nach einem *DeviceIOControl*-Aufruf auf, um gerätespezifische Steuerungsaufgaben auszuführen. Anwendungen können die Funktion DeviceIOControl verwenden, um Gerätetreiberdaten zu lesen und zu schreiben, die XXX_Write und XXX_Read nicht verwenden. Diese Methode ist für viele Streamtreiber üblich.
XXX_PowerUp	Der Gerätemanager ruft diese Funktion auf, wenn das Betriebssystem aus einem niedrigeren Energiemodus reaktiviert wird. Anwendungen rufen diese Funktion nicht auf. Die Funktion wird im Kernelmodus ausgeführt, kann externe APIs nicht aufrufen und kann nicht ausgelagert werden, da das Betriebssystem im nicht ausgelagerten Single-Thread-Modus ausgeführt wird. Microsoft empfiehlt, dass Treiber die Energieverwaltung basierend auf Power Manager und Energieverwaltungs-IOCTLs implementieren, um die Funktionalität eines Treibers auszusetzen oder fortzusetzen.
XXX_PowerDown	Der Gerätemanager ruft diese Funktion auf, wenn das Betriebssystem in den Standby-Modus übergeht. Wie XXX_PowerUp wird diese Funktion im Kernelmodus ausgeführt, kann keine externen APIs aufrufen und kann nicht ausgelagert werden. Anwendungen rufen diese Funktion nicht auf.

Über die Streamschnittstelle tauschen Gerätemanager und Treiber Daten und Steuerbefehle aus. Die Schnittstelle besteht aus 12 Funktionen.

Startprozesses. Der Kernel lädt die Gerätemanager-Shell (Device.dll), die wiederum den Gerätemanager-Basiscode (Devmgr.dll) lädt, der die Streamtreiber lädt und entlädt.

Streamtreiber können als Teil des Betriebssystems beim Start oder bei Bedarf geladen werden, wenn die entsprechende Hardware Plug & Play unterstützt. Die Benutzeranwendungen greifen über die Dateisystem-APIs oder *DeviceIOControl*-Aufrufe auf die Streamtreiber zu, beispielsweise ReadFile und WriteFile. Streamtreiber, die der Gerätemanager über das Dateisystem bereitstellt, werden von den Anwendungen als reguläre Dateiressourcen mit speziellen Dateinamen behandelt.

Die Funktion DeviceIOControl erlaubt Anwendungen das Ausführen direkter E/A-Vorgänge. Die Anwendungen interagieren jedoch in beiden Fällen über den Gerätemanager mit den Streamtreibern. Der Gerätemanager entkoppelt alle API-Aufrufe des Anwenders und ruft dann die zugehörige Treiber-Funktion auf. Er weiß auch, in welchem Zustand sich der Treiber befindet. Rückgabe-Werte der Treiber werden z.B. im xxx_Init() in ein Handle umgewandelt.

Namenskonvention

Damit eine Anwendung einen Streamtreiber über das Dateisystem verwenden kann, muss der Streamtreiber als Datei-Ressource dargestellt werden. Die Anwendung kann die Gerätedatei in einem *CreateFile*-Aufruf angeben, um ein Gerätehandle abzurufen. Nachdem das Handle abgerufen wurde, führt die Anwendung über *ReadFile* oder *WriteFile* E/A-Vorgänge aus, die der Gerätemanager in die entsprechenden Aufrufe umwandelt, um die erforderlichen Lese- und Schreibvorgänge auszuführen. Damit Windows Embedded CE 6.0 die Streamgeräte-Ressourcen erkennt und E/A-Dateivorgänge an das entsprechende Streamlaufwerk umleitet, müssen die Streamtreiber einer bestimmten Namenskonvention entsprechen, um die Ressourcen von anderen Dateien zu unterscheiden.

Streamschnittstellen-API

Damit der Gerätemanager einen Streamtreiber laden und verwalten kann, muss der Treiber eine Schnittstelle exportieren, die als Streamschnittstelle bezeichnet wird. Die Streamschnittstelle besteht aus zwölf Funktionen (**Tabelle**), die das Gerät initialisieren und öffnen, Daten lesen und schreiben, das Gerät ein- oder ausschalten oder das Gerät deaktivieren.

Um einen Gerätetreiber zu erstellen, kann man ein Teilprojekt für eine Windows-Embedded-CE-DLL zum OS-Design hinzufügen. Normalerweise werden hierzu die Quelldateien des Gerätetreibers in den Ordner *Drivers* des Board Support Packages (BSP) kopiert. Ein guter Ausgangspunkt für einen Gerätetreiber ist die Vorlage A „Simple Windows Embedded CE DLL Subproject", die sich auf der Seite „Auto-Generated Subproject Files" im Windows-Embedded-CE-Subproject-Wizard auswählen lässt. Die Quellcodedatei mit einer Definition des DllMain-Einsprungspunkts für die DLL, einige Parameterdateien, beispielsweise .def- und .reg-Dateien, werden automatisch erstellt. Außerdem wird die Sources-Datei konfiguriert, um die Ziel-DLL erstellen zu können. Alle Anwendungen werden durch eine *.bib-Dateidefinition zum Image dazugelegt und in einem bestimmten Adressbereich abgelegt. Eine *.dat-Datei bestimmt die Dateistruktur, in der der Treiber abgelegt werden soll.

```
BOOL APIENTRY DllMain(HANDLE hModule, DWORD ul_reason_for_call, LPVOID lpReserved)
{
      return TRUE;
}

DWORD LED_Init(LPCTSTR pContext, LPCVOID lpvBusContext)
{
    return 0x1;
}
```

Listing 1. Definition von Streamschnittstellenfunktionen in einem DLL-Teilprojekt.

QUADMO747
7x7 cm
Nicht heiss
aber scharf
Q S E V E N
Standard
Intel® ATOM™
1.1 GHz und 1.6 GHz
Bis zu 8 GB SSD Flash
und
1 GB DDR2 on board
SECO BIOS
µSD-Card SLOT
SECOCQ7-mITX
Mini-ITX form factor
Development Carrier
SECOCQ7-pITX
Pico-ITX form factor
Compact Ready Solution
Open Frame Chassis opt.
SECO
Frischer Wind aus Italien
www.seco.it
Ihr direkter Draht zu SECO in D A CH
Harald Ernst Baumann
seco@toscco.de 0049-991-2705706

```
LIBRARY LEDDriver

EXPORTS
    LED_Init
    LED_Open
    LED_Close
    LED_Deinit
    LED_Read
    LED_Write
    LED_Seek
    LED_PowerDown
    LED_PowerUp
    LED_IOControl
```

Listing 2. Mittels einer DEF-Datei bestimmt der Linker, welche Funktionen exportiert werden. Auf diese Funktionen kann der Gerätemanager dann zugreifen.

Nachdem das DLL-Teilprojekt erstellt wurde, kann die Quellcodedatei in Visual Studio geöffnet und die erforderlichen Funktionen können hinzugefügt werden, um die Streamschnittstelle und die Treiberfunktionen zu implementieren. Das Codesegment in **Listing 1** veranschaulicht die Definition der Streamschnittstellenfunktionen.

Exportieren von Streamfunktionen

Um die Streamfunktionen in der Treiber-DLL für externe Anwendungen bereitzustellen, muss der Linker die Funktionen während des Buildprozesses exportieren. C++ enthält hierzu mehrere Optionen. Für kompatible Treiber-DLLs mit dem Gerätemanager müssen die Funktionen exportiert werden, indem diese in der .def-Datei des DLL-Teilprojekts definiert werden. Der Linker bestimmt mittels der .def- Datei, welche Funktionen exportiert werden. Für einen Standardstreamtreiber müssen die Streamschnittstellenfunktionen mit dem Präfix exportiert werden, das in der Quelldatei des Treibers und in den Registrierungseinstellungen angegeben ist. (**Listing 2**). Für den Zugriff auf einen Streamtreiber kann eine Anwendung die Funktion *CreateFile* verwenden und den gewünschten Gerätenamen angeben.

Dynamisches Laden eines Treibers

Wie bereits erwähnt, kann eine Anwendung mit einem Streamgerätetreiber kommunizieren, nachdem die Funktion *ActivateDevice* oder *ActivateDeviceEx* aufgerufen wurde. ActivateDeviceEx ist flexibler als ActivateDevice, aber beide Funktionen stellen sicher, dass der Gerätemanager den Streamtreiber lädt und die XXX_Init-Funktion des Treibers aufruft. ActivateDevice ruft ActivateDeviceEx auf. ActivateDeviceEx allein erlaubt keinen Zugriff auf einen bereits geladenen Treiber. Die Funktion ActivateDeviceEx wird in erster Linie verwendet, um den im Funktionsaufruf angegebenen treiberspezifischen Registrierungsschlüssel (**Listing 3**) zu lesen, um den DLL-Namen, das Gerätepräfix und andere Werte zu bestimmen, die relevanten Werte zur aktiven Geräteliste hinzuzufügen und den Gerätetreiber in den Prozessbereich des Gerätemanagers zu laden. Der Funktionsaufruf gibt ein Handle zurück, das die Anwendung später zum Entladen des Treibers mit dem Aufruf der Funktion *DeactivateDevice* verwenden kann. ActivateDeviceEx ersetzt die Funktion RegisterDevice als Methode zum Laden eines erforderlichen Treibers und wird gemäß **Listing 4** verwendet.

Das Laden eines Treibers durch den Aufruf der Funktion ActivateDeviceEx führt zum gleichen Ergebnis wie das automatische Laden des Treibers während des Startprozesses über die im Schlüssel HKEY_LOCAL_MACHINE\Drivers\BuiltIn definierten Parameter. *jk*

```
[HKEY_LOCAL_MACHINE\Drivers\LEDDriver]
"Dll"="LEDDriver.dll"
"Prefix"="LED"
"Order"="dword:30"
"Index"="dword:1"
"FriendlyName"="LEDDriver"
```

Listing 3. Der treiberspezifische Geräteschlüssel wird von der Funktion ActivateDeviceEx aus der Registrierung gelesen. Anhand dieser Angaben wird der Treiber in den Prozessbereich des Gerätemanagers geladen.

```
hActiveDriver = ActivateDeviceEx(L"\\Drivers\\LED_Treiber",
 NULL, 0, NULL);

if (hActiveDriver == INVALID_HANDLE_VALUE)
{
      ERRORMSG(1, (L"Treiber konnte nicht geladen werden"));
      return -1;
}
```

Listing 4. Die Funktion ActiveDeviceEx lädt einen Treiber und ersetzt die Funktion RegisterDevice.

Ralf Ebert
ist Produktspezialist beim Microsoft-Embedded-Distributor Avnet Silica und dort unter anderem für Design-In-Unterstützung und Beratung zuständig. Zuvor arbeitete er fünf Jahre direkt für Microsoft als Embedded System Engineer. Avnet Silica vertreibt die genannten Windows-CE-Lizenzen und kann sowohl kostenlose Evalversion als auch den kompletten Windows CE Platformbuilder CE 6.0 R3 zur Verfügung stellen.
ralf.ebert@silica.com

Dipl.-Inf. Rudi Swiontek
ist im Bereich Schulung und Entwicklung bei der HILFIGmbH tätig. Als erfahrener Produktspezialist ist er u.a. für die Windows-Embedded-Betriebssysteme zuständig.
Rudi.Swiontek@hilf.de