MCTS

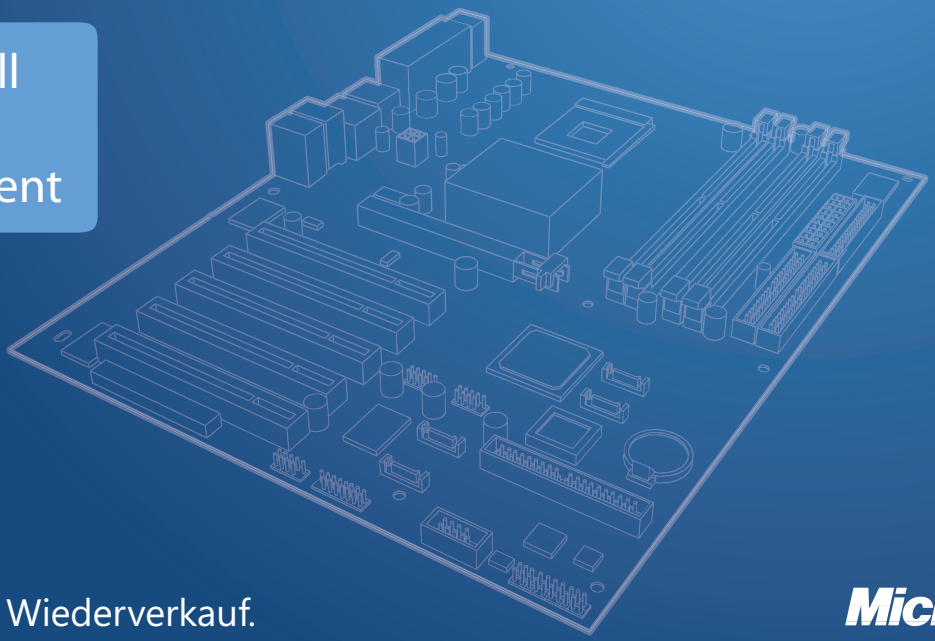
Prüfung 70-571

Windows® Embedded CE 6.0

Preparation Kit

Vorbereitung auf die Zertifizierungsprüfung

Aktuell
mit
R2 Content



Microsoft®

Herausgegeben von

Microsoft Corporation

Konrad-Zuse-Straße 1

85716 Unterschleißheim

Redakteur:	Sondra Webber, Microsoft Corporation
Autoren:	Nicolas Besson, Adeneo Corporation Ray Marcilla, Adeneo Corporation Rajesh Kakde, Adeneo Corporation
Autorenteamleitung:	Warren Lubow, Adeneo Corporation
Technischer Redakteur:	Brigette Huang, Microsoft Corporation
Redaktionelle Überarbeitung:	Biblioso Corporation

Body Part No. 098-110196

Übersicht

Inhaltsverzeichnis

Vorwort

Es scheint erst gestern gewesen zu sein, als wir Windows CE 1.0 auf den Markt brachten, obwohl mittlerweise 12 erfolgreiche Jahre vergangen sind und zahlreiche Änderungen vorgenommen wurden. Neue Technologien wurden entwickelt, während andere Technologien verschwunden sind. Wir werden weiterhin zusammen mit unseren Partnern alle Vorteile neuer Hardware- und Softwareinnovationen nutzen. Windows Embedded CE wird ständig weiterentwickelt, bleibt aber ein eingebettetes Small Footprint- und Echtzeit-Betriebssystem, das auf zahlreichen Multiprozessor-Architekturen und Geräten ausgeführt werden kann, einschließlich Robotern, tragbaren Ultraschallsystemen, Kassenterminals, Mediastreamern, Spielkonsolen, Thin Clients und anderen Geräten, die niemand mit einem Microsoft-Betriebssystem in Zusammenhang bringen würde. Möglicherweise wird Windows Embedded CE eines Tages auf Geräten auf dem Mond verwendet. Dies wäre keine Überraschung. Windows Embedded CE kann überall eingesetzt werden, wo Geräte das Leben vereinfachen und mehr Spaß bringen.

Wir haben uns von Anfang an auf die Anforderungen von professionellen Entwicklern konzentriert und eine umfassende Suite an Entwicklungstools erstellt sowie Windows-Programmierschnittstellen und Frameworks unterstützt. Außerdem haben wir die Windows Embedded CE-Entwicklungstools mit Visual Studio 2005 integriert, damit Entwickler das Betriebssystem einfacher anpassen und Anwendungen erstellen können. Windows Embedded CE 6.0 unterstützt x86-, ARM-, MIPS- und SH4-Prozessoren und umfasst ca. 700 auswählbare Betriebssystemkomponenten. CE umfasst sowohl die Tools, die zum Konfigurieren, Erstellen, Herunterladen, Debuggen und Testen von Betriebssystem-Images und Anwendungen benötigt werden, als auch den Quellcode für den Kernel, die Gerätetreiber und andere Features. Anwendungsentwickler können Win32-, MFC- und ATL-Anwendungen in nativem Code oder .NET-Anwendungen, die auf .NET Compact Framework basieren, erstellen. Im Zuge der Microsoft Shared Source Initiative stellen wir mehr als 2,5 Millionen Zeilen mit CE-Quellcode bereit, um Entwicklern zu ermöglichen, den geänderten Quellcode anzuzeigen, zu bearbeiten, neu zu erstellen und freizugeben. Das vor kurzem ins Leben gerufene Programm "Spark your Imagination" bietet kostengünstige Hardware und CE-Entwicklungstools für Hobbyentwickler an.

Weitere Informationen über das CE-Betriebssystem, die Entwicklungstools und Konzepte finden Sie im Preparation Kit für Microsoft Certified Technology Specialist (TS),

Prüfung 70-571 "Windows Embedded CE 6.0-Entwicklung", das im Mai 2008 erschienen ist. Die Prüfung 70-571 stellt einen wichtigen Meilenstein in der Windows Embedded CE-Erfolgsgeschichte dar. Die Entwickler können nun zum ersten Mal ihre Kenntnisse bezüglich der Entwicklung eingebetteter Lösungen auf Basis von Windows Embedded-Technologien prüfen lassen und sicherstellen, dass ihr Wissen anerkannt wird. Jeder Benutzer, der sich mit CE 6.0 befasst, sollte diese Prüfung ablegen. Wir hoffen, dass dieses Buch die Vorbereitung auf die Prüfung beschleunigt, so wie Windows Embedded CE 6.0 den Entwicklungsprozess beschleunigt. Die besten Wünsche dazu vom Microsoft-Entwicklungsteam!

Mike Hall

Windows Embedded Architect
Microsoft Corporation

Einleitung

Willkommen beim Microsoft Windows Embedded CE 6.0 R2 Exam Preparation Kit. Dieses Preparation Kit soll Windows Embedded CE-Entwicklern bei der Vorbereitung auf die Zertifizierungsprüfung Microsoft Certified Technology Specialist (MCTS) Windows Embedded CE 6.0-Anwendungsentwicklung helfen.

Mit diesem Kit können Sie Ihre Kenntnisse in folgenden Prüfungsbereichen vertiefen:

- Anpassen des OS Designs.
- Klonen von Windows Embedded CE-Komponenten und Verwalten der Katalogelemente.
- Generieren eines Software Development Kits (SDK).
- Erstellen eines Run-Time Images und Analysieren der Buildergebnisse.
- Bereitstellen, Überwachen und Optimieren eines Run-Time Images.
- Entwickeln von Multithread-Systemanwendungen.
- Implementieren der Ausnahmeverarbeitung.
- Unterstützen der Energieverwaltung in Anwendungen, Gerätetreibern und im OAL (OEM Adaptation Layer).
- Konfigurieren eines Board Support Packages (BSP), einschließlich Boot Loader und Speicherzuordnungen.
- Entwickeln von umfassenden Streamschnittstellentreibern.
- Implementieren von Interrupt Service Routines (ISRs) und Interrupt Service Threads (ISTs) sowie Marshalling von Daten zwischen Kernelmodus- und Benutzermoduskomponenten.
- Debuggen der Kernelmodus- und Benutzermoduskomponenten, um Softwarefehler auszuschließen.
- Verwenden des Windows Embedded CE Test Kit (CETK), um auf einem Entwicklungscomputer und einem Zielgerät Standardtests und benutzerdefinierte Tests auszuführen.
- Entwickeln von Tux-Erweiterungskomponenten, um benutzerdefinierte Gerätetreiber in CETK-Tests einzubeziehen.

Zielgruppe

Dieses Preparation Kit ist für Systementwickler mit Grundkenntnissen im OS Design, Programmieren von Systemkomponenten und Debuggen auf der Windows Embedded CE-Plattform bestimmt.

Das Kit wurde insbesondere für Entwickler mit folgenden Kenntnissen geschrieben:

- Grundkenntnisse in der Windows- und Windows Embedded CE-Entwicklung.
- Mindestens zwei Jahre Erfahrung in der C/C++-Programmierung und der Win32 Application Programming Interface (API).
- Vertrautheit mit Microsoft Visual Studio 2005 und Platform Builder für Windows Embedded CE 6.0.
- Grundkenntnisse im Debuggen mit den Windows-Standarddebugtools.

WEITERE INFORMATIONEN Zielgruppenprofil für die Prüfung 70-571

Weitere Informationen zu den Prüfungsanforderungen finden Sie im Abschnitt „Zielgruppenprofil" im Vorbereitungshandbuch für Exam 70-571 unter *http://www.microsoft.com/learning/exams/70-571.mspx*.

Gliederung des Buchs

Alle Kapiteln enthalten die jeweiligen Prüfungsziele und den Abschnitt „Bevor Sie beginnen", der Sie auf das Kapitel vorbereitet. Die Kapitel sind in Lektionen aufgeteilt. Jede Lektion beginnt mit einer Liste der Prüfungsziele und der für die Lektion erforderlichen Zeitdauer. Eine Lektion ist entsprechend der Themen und Ziele gegliedert.

Jedes Kapitel endet mit praktischen Übungen und einer kurzen Zusammenfassung der Lektionen. Anschließend folgen eine kurze Überprüfung der Schlüsselbegriffe und vorgeschlagene Übungen, um Ihre Kenntnisse der Themen im Kapitel zu prüfen und Ihnen zu helfen, die Prüfungsziele zu erreichen.

Anhand der praktischen Übungen können Sie Ihre Kenntnisse bezüglich eines Konzepts testen. Alle praktischen Übungen umfassen schrittweise Verfahren, die mit Aufzählungszeichen aufgegliedert sind. Die Arbeitsblätter mit schrittweisen Anweisungen für die Labs, die sich ebenfalls auf der Begleit-CD befinden, helfen Ihnen diese Verfahren erfolgreich auszuführen.

Für die praktischen Übungen benötigen Sie einen Entwicklungscomputer, auf dem Microsoft Windows XP oder Microsoft Windows Vista, Visual Studio 2005 Service Pack 1 und Platform Builder für Windows Embedded CE 6.0 installiert ist.

Hardwareanforderungen

Der Entwicklungscomputer muss folgende Mindestanforderungen erfüllen und die Hardwarekomponenten müssen in der Windows XP- oder Windows Vista-Hardwarekompatibilitätsliste aufgeführt sein.

- 1 GHz 32 Bit (x86) oder 64 Bit (x64) Prozessor
- 1 GB RAM
- 40 GB Festplatte mit 20 GB freiem Speicherplatz für Visual Studio 2005 und Platform Builder
- DVD-Laufwerk
- Microsoft-Maus oder ein kompatibles Zeigegerät
- Eine Auslagerungsdatei mit mindestens doppelter RAM-Größe
- VGA-kompatibler Bildschirm

Softwareanforderungen

Für die Übungen ist folgende Software erforderlich:

- Microsoft Windows XP SP2 oder Windows Vista
- Microsoft Visual Studio 2005 Professional Edition
- Microsoft Windows Embedded CE 6.0
- Microsoft Visual Studio 2005 Professional Edition SP1
- Microsoft Windows Embedded CE 6.0 SP1
- Microsoft Windows Embedded CE 6.0 R2

HINWEIS Evaluierungsversionen von Visual Studio 2005 und Windows Embedded CE 6.0

Installationsrichtlinien und die Evaluierungsversionen von Visual Studio 2005 und Windows Embedded CE 6.0 sind auf der Microsoft-Website unter *http://www.microsoft.com/windows/embedded/products/windowsce/getting-started.mspx* verfügbar.

Schreibweise

- Zeichen oder Befehle, die Sie eintippen, sind in Kleinbuchstaben geschrieben und **fett** formatiert.
- Spitze Klammern < > in Syntaxangaben enthalten Platzhalter für Variableninformationen.
- Buchtitel und Webadressen sind kursiv formatiert.
- Die Namen von Dateien und Ordnern werden in Großbuchstaben angezeigt, außer wenn Sie diese direkt eingeben. Sofern nicht anders angegeben, können Sie Dateiennamen in einem Dialogfeld oder in der Befehlszeile klein schreiben.
- Dateierweiterungen sind klein geschrieben.
- Akronyme sind groß geschrieben.
- `Monospace`-Zeichen geben Codebeispiele, Bildschirmtext oder Einträge an, die Sie in der Befehlszeile oder in Initialisierungsdateien eingeben.
- Eckige Klammern { } schließen optionale Elemente in Syntaxangaben ein. Beispielsweise bedeutet [*Dateiname*] in einer Befehlssyntax, dass Sie im Befehl einen Dateinamen angeben können. Geben Sie nur die Informationen in den Klammern ein, nicht die Klammern.
- Geschweifte Klammern { } schließen in Syntaxangaben die erforderlichen Elemente ein. Geben Sie nur die Informationen in den Klammern ein, nicht die Klammern.

Tastaturkonventionen

- Ein Pluszeichen (+) zwischen zwei Tastenbezeichnungen bedeutet, dass Sie diese Tasten gleichzeitig drücken müssen. Beispielsweise bedeutet "Drücken Sie ALT+TAB", dass Sie die Alt-Taste gedrückt halten, während Sie die TAB-Taste drücken.
- Ein Komma (,) zwischen zwei oder mehreren Tastenbezeichnungen bedeutet, dass Sie die Tasten nacheinander, nicht gleichzeitig, drücken müssen. Beispielsweise bedeutet "Drücken Sie ALT, F, X", dass Sie die Tasten nacheinander in der angegebenen Reihenfolge drücken müssen. "Drücken Sie ALT+W, L" zeigt an, dass Sie zuerst die ALT-Taste und den Buchstaben W zusammen und anschließend den Buchstaben L drücken müssen.
- Sie können Menübefehle mit der Tastatus auswählen. Drücken Sie die ALT-Taste, um die Menüleiste zu aktivieren, und anschließend den Buchstaben, der

im Menünamen oder im Menübefehl hervorgehoben bzw. unterstrichen ist. Für einige Befehle können Sie auch die im Menü angegebene Tastenkombination drücken.

- Sie können Kontrollkästchen und Optionsfelder über die Tastatur aktivieren oder deaktivieren. Halten Sie die ALT-Taste gedrückt und drücken Sie den Buchstaben, der im Optionsnamen unterstrichen ist. Sie können die Option auch mit der TAB-Taste auswählen und die Leertaste drücken, um das Kontrollkästchen oder die Option zu aktivieren bzw. zu deaktivieren.
- Sie können ein Dialogfeld schließen, indem Sie die ESC-Taste drücken.

Hinweise

Die Lektionen enthalten mehrere Hinweistypen.

- **Tipp** enthält Erklärungen der möglichen Ergebnisse oder alternative Methoden.
- **Wichtig** enthält Informationen, die für eine Aufgabe wesentlich sind.
- **Hinweis** enthält zusätzliche Informationen.
- **Achtung** enthält Warnungen bezüglich dem möglichen Datenverlust.
- **Prüfungstipp** enthält hilfreiche Tipps zu den Prüfungsthemen und Zielen.

Die Begleit-CD

Die Begleit-CD enthält Informationen zu allen Lektionen im Buch. Auf der Begleit-CD sind außerdem die Arbeitsblätter mit schrittweisen Anweisungen und der Quellcode, die in den praktischen Übungen verwendet werden, sowie technische Informationen und Artikel von Microsoft-Entwicklern gespeichert.

Eine elektronische Version (eBook) des Buchs, für die mehrere Ansichtsoptionen verfügbar sind, ist ebenfalls beigelegt. Auf der Begleit-CD befindet sich ein vollständiger Satz Dateien für das Selbststudium, die nach der Veröffentlichung erstellt wurden, um ein Buch zu drucken. Hierbei handelt es sich um PDF-Dateien (Portable Document Format) mit den für die professionelle Buchbindung erforderlichen Schnittzeichen.

Microsoft Certified Professional-Programm

Mit dem MCP-Programm (Microsoft Certified Professional) können Sie Ihre Fachkenntnisse in aktuellen Microsoft-Produkten und Technologien nachweisen. Die Prüfungen und entsprechenden Zertifizierungen wurden entwickelt, um Ihre Kompetenz im Entwickeln, Implementieren und Warten von Lösungen zu bestätigen, die auf Microsoft-Produkten und Technologien beruhen. In der Computerbranche besteht eine hohe Nachfrage für Experten, die ein Microsoft-Zertifikat besitzen. Die Zertifizierung bringt somit zahlreiche Vorteile für Bewerber, Arbeitgeber und Unternehmen mit sich.

WEITERE INFORMATIONEN Alle Microsoft-Zertifizierungen

Eine vollständige Liste der Microsoft-Zertifizierungen finden Sie unter *http://www.microsoft.com/learning/mcp/default.asp*.

Technischer Support

Microsoft Press bemüht sich stets um die Richtigkeit der in diesem Buch sowie der auf der Begleit-CD-ROM enthaltenen Informationen. Falls Sie Feedback, Fragen oder Ideen bezüglich der Windows Embedded CE-Entwicklung haben, wenden Sie sich über Microsoft Product Support Services (PSS), Microsoft Developer Network (MSDN) oder folgende Blogsites an einen Windows Embedded CE-Experten:

- **Nicolas BESSONs Weblog** Falls Sie Feedback und Vorschläge für neue Artikel bezüglich dieser Themen haben, wenden Sie sich unter *http://nicolasbesson.blogspot.com* an den Hauptautor des Windows Embedded CE 6.0 Exam Preparation Kits.
- **Windows Embedded Blog** Tricks, Tipps und Hinweise zu Windows Embedded von Mike Hall finden Sie unter *http://blogs.msdn.com/mikehall/default.aspx*.
- **Windows CE Base Team Blog** Unter *http://blogs.msdn.com/ce_base/default.aspx* erhalten Sie Hintergrundinformationen zum Windows Embedded CE-Kernel und zu Speichertechnologien direkt von Microsoft-Entwicklern.

WEITERE INFORMATIONEN Windows Embedded CE-Produktsupport

Weitere Informationen zu den verfügbaren Windows Embedded CE-Supportoptionen finden Sie unter *http://www.microsoft.com/windows/embedded/support/products/default.mspx*.

Kapitel 1

Anpassen des OS Designs

Um Windows® Embedded CE 6.0 R2 auf einem Zielgerät bereitzustellen, müssen Sie ein Run-Time Image verwenden, das die erforderlichen Betriebssystemkomponenten, Features, Treiber und Konfigurationseinstellungen umfasst. Das Run-Time Image ist die binäre Darstellung des OS Designs. Mit Microsoft® Platform Builder für Windows Embedded CE 6.0 können Sie ein OS Design erstellen oder anpassen und das entsprechende Run-Time Image generieren. Erstellen Sie für jedes OS Design ein neues Entwicklungsprojekt in Microsoft® Visual Studio® 2005, das ausschließlich die für das Zielgerät und Anwendungen erforderlichen Komponenten umfasst. Dies reduziert den Speicherbedarf des Betriebssystems sowie die Hardwareanforderungen. Um jedoch kompakte und funktionelle Run-Time Images zu generieren, müssen Sie mit dem Platform Builder vertraut sein, einschließlich der Benutzeroberfläche, der Katalogkomponenten und dem Buildverfahren. In diesem Kapitel wird das Erstellen eines OS Designs und das Generieren eines neuen Run-Time Images für Windows Embedded CE erklärt.

Prüfungsziele in diesem Kapitel

- Erstellen und Anpassen des OS Designs
- Konfigurieren von Windows Embedded CE-Teilprojekten
- Klonen von Komponenten
- Verwalten der Katalogelemente
- Generieren eines Software Development Kits (SDK)

Bevor Sie beginnen

Damit Sie die Lektionen in diesem Kapitel durcharbeiten können, benötigen Sie:

- Mindestens Grundkenntnisse in der Windows Embedded CE-Softwareentwicklung.
- Grundkenntnisse der Verzeichnisstruktur und des Buildprozesses von Platform Builder für Windows Embedded CE 6.0 R2.
- Kenntnisse im Erstellen von binären Windows Embedded CE-Run-Time Images und im Herunterladen von Run-Time Images auf Zielgeräte.
- Erfahrung in der Verwendung eines SDKs, zum Entwickeln von Anwendungen für Windows Embedded CE.
- Einen Entwicklungscomputer, auf dem Microsoft Visual Studio 2005 Service Pack 1 und Platform Builder für Windows Embedded CE 6.0 installiert ist.

Lektion 1: Erstellen und Anpassen des OS Designs

Mit Platform Builder in Visual Studio 2005 können Sie ein OS Design mit den in Windows Embedded CE 6.0 R2 verfügbaren Features erstellen, die für Ihr Projekt erforderlich sind. Beispielsweise können Sie ein OS Design für ein bestimmtes Zielgerät (z.B. ein tragbares Multimediagerät) oder einen fernprogrammierbaren digitalen Thermostaten entwerfen. Diese beiden Zielgeräte verwenden möglicherweise die gleiche Hardware, aber der jeweilige Verwendungszweck hat unterschiedliche Anforderungen an das OS Design.

Nach Abschluss dieser Lektion können Sie:

- Die Funktion und Besonderheiten eines OS Designs verstehen.
- OS Designs erstellen, anpassen und verwenden.

Veranschlagte Zeit für die Lektion: 30 Minuten.

OS Design – Übersicht

Das OS Design definiert die Komponenten und Features im Run-Time Image. Dies entspricht im Wesentlichen einem Visual Studio mit Platform Builder für Windows Embedded CE 6.0 R2-Projekt. Das OS Design kann folgende Elemente umfassen:

- Katalogelemente, einschließlich Softwarekomponenten und Treiber
- Zusätzliche Softwarekomponenten in Form von Teilprojekten
- Benutzerdefinierte Registrierungseinstellungen
- Buildoptionen, beispielsweise für die Lokalisierung oder das Debugging basierend auf KITL (Kernel Independent Transport Layer)

Jedes OS Design umfasst außerdem eine Referenz auf mindestens ein BSP (Board Support Package) mit Gerätetreibern, hardwarespezifischen Dienstprogrammen und einem OAL (OEM Adaptation Layer).

Erstellen eines OS Designs

Windows Embedded CE umfasst den OS Design Wizard, der eine praktische Methode zum Erstellen von OS Designs bietet. Um auf den Design Wizard zuzugreifen und das Dialogfeld **New Project** zu öffnen, starten Sie Visual Studio 2005 mit Platform Builder für Windows Embedded CE 6.0 R2, zeigen Sie im Menü **File** auf **New** und klicken Sie auf **Project**. Wählen Sie in diesem Dialogfeld unter **Project**

Types den **Platform Builder für CE 6.0** und unter **Visual Studio Installed Templates** die Option **OS Design** aus. Geben Sie einen Namen für das Design im Feld **Name** ein und klicken Sie auf **OK**, um den **Windows Embedded CE 6.0 OS Design Wizard** zu starten.

Der OS Design Wizard ermöglicht die Auswahl eines BSP und einer Designvorlage mit häufig verwendeten Optionen und Katalogkomponenten. Da Sie alle im OS Design Wizard festgelegten Einstellungen zu einem späteren Zeitpunkt ändern können, halten Sie sich nicht zu lange mit den einzelnen Einstellungen auf. Abhängig von der auf der Seite **Design Templates** ausgewählten Vorlage, zeigt der OS Design Wizard möglicherweise eine weitere Seite mit Vorlagenvarianten an, die bestimmte Optionen für diese Vorlage umfasst. Da beispielsweise **Windows Thin Client**, **Enterprise Terminal** und **Windows Network Projector** Geräte sind, die RDP (Remote Desktop Protocol) verwenden, sind diese jeweils Varianten der gleichen **Thin Client**-Designvorlage. Abhängig von der ausgewählten Vorlage und Variante zeigt der OS Design Wizard weitere Seiten an, um bestimmte Komponenten in das OS Design einzubeziehen, beispielsweise ActiveSync®, WMV/MPEG-4 Videocodec oder IPv6.

Die OS Design-Vorlage

Eine CE 6.0 OS-Designvorlage entspricht der Teilmenge von Katalogkomponenten, die erforderlich sind, um Windows Embedded CE für einen bestimmten Zweck zu verwenden. Obwohl es nicht erforderlich ist, die Erstellung eines neuen OS Designs mit einer Vorlage zu beginnen, kann dies jedoch den Zeitaufwand erheblich reduzieren. Die Katalogkomponenten können zu einem späteren Zeitpunkt einfach geändert werden, indem Sie diese im Fenster **Catalog Items View** (Katalogelementansicht) auswählen.

Die Auswahl einer geeigneten Vorlage kann den Entwicklungsaufwand verringern. Wenn Sie beispielsweise die Features eines neuen Entwicklungsboards auf einer Messe demonstrieren möchten, können Sie mit einer Designvorlage für ein PDA-Gerät oder ein Mediengerät beginnen und die erforderlichen Komponenten sowie Windows-Anwendungen im OS Design Wizard hinzufügen, z.B. .NET Compact Framework 2.0, Internet Explorer® und WordPad. Wenn Sie einen Treiber für einen CAN-Controller (Controller Area Network) entwickeln, sollten Sie mit der Designvorlage **Small Footprint Device** beginnen und nur die unbedingt erforderlichen Komponenten hinzufügen, um die Größe des Run-Time Images und der Startzeiten zu verringern.

Der OS Design Wizard ist flexibel und unterstützt benutzerdefinierte Designvorlagen. Vorlagendateien sind XML-Dokumente (Extensible Markup Language), die im Ordner *%_WINCEROOT%\Public\CEBase\Catalog* gespeichert sind. Sie können mit einer Kopie einer vorhandenen Platform Builder Catalog XML-Datei (PBCXML) beginnen und die PBCXML-Strukturen an Ihre Anforderungen anpassen. Platform Builder listet automatisch alle .pbcxml-Dateien im Ordner *Katalog* auf, wenn Sie Visual Studio starten oder das Fenster **Catalog Items View** in Visual Studio aktualisieren.

Anpassen des OS Designs mit Katalogkomponenten

Nachdem der OS Design Wizard abgeschlossen ist, können Sie das OS Design anpassen. Der Katalog umfasst alle Komponenten, die zu einem OS Design hinzugefügt werden können. Sie können direkt in der integrierten Entwicklungsumgebung (Integrated Development Environment, IDE) auf den Katalog zugreifen. Klicken Sie im Solution Explorer auf **Catalog Items View**. Fast alle CE-Features sind in separate auswählbare Katalogkomponenten aufgeteilt (von ActiveSync bis TCP/IP). Sie können diese Komponenten direkt in der Benutzeroberfläche auswählen. Jedes Katalogelement entspricht dabei einem Verweis auf alle Komponenten, die zum Erstellen und Integrieren des Features im Run-Time Image erforderlich sind.

Wenn Sie ein Katalogelement hinzufügen, das von anderen Katalogelementen abhängig ist, werden diese Elemente ebenfalls als Abhängigkeiten zum OS Design hinzugefügt. Im Fenster **Catalog Items View** sind diese Elemente durch ein grünes Quadrat im Kontrollkästchen markiert, um die Abhängigkeit anzuzeigen. Die manuell ausgewählten Elemente und die über eine Designvorlage hinzugefügten Elemente sind durch ein grünes Häkchen gekennzeichnet.

In der **Catalog Items View** können Sie alle Katalogelemente auflisten oder einen Filter aktivieren, der ausschließlich die ausgewählten Katalogelemente anzeigt. Klicken Sie im Solution Explorer in der linken oberen Ecke der **Catalog Items View** auf den Pfeil neben **Filter**, um den Filter anzuwenden, oder wählen Sie die Option **All Catalog Items** (Alle Katalogelemente) aus, um eine vollständige Liste der Katalogelemente anzuzeigen.

Wenn Ihnen der Name des Katalogelements oder die SYSGEN-Variable einer Komponentengruppe bekannt ist, können Sie schnell nach dem Katalogelement suchen, das Sie hinzufügen oder entfernen möchten. Um nach dem Namen oder der SYSGEN-Variablen zu suchen, geben Sie den Suchbegriff in das Textfeld ein und klicken Sie auf den grünen Pfeil.

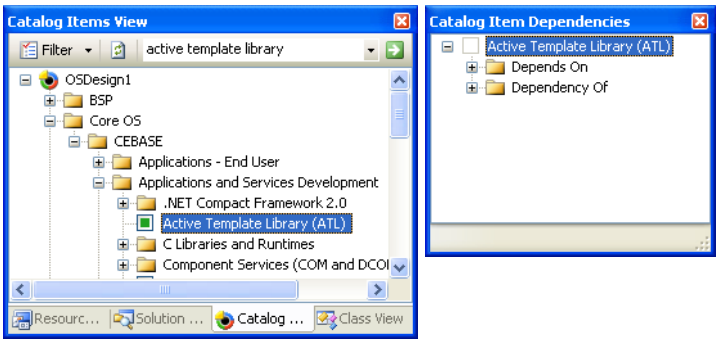


Abbildung 1.1 Catalog Items View mit dem Suchfeld und dem Fenster Catalog Items Dependencies

Um die Abhängigkeiten eines Katalogelements zu analysieren, klicken Sie mit der rechten Maustaste auf das Element und wählen Sie **Show Dependencies** aus, um das Fenster **Catalog Item Dependencies** zu öffnen (siehe Abbildung 1.1). Sie können dieses Feature unter anderem verwenden, um die Ursache für die Einbeziehung eines Katalogelements als Abhängigkeit zu überprüfen. Platform Builder in CE 6.0 R2 durchsucht den Katalog dynamisch, um alle Komponenten aufzulisten, die sowohl vom ausgewählten Element als auch von den Komponenten abhängig sind, die wiederum von diesem Element abhängen.

Verwalten der Buildkonfiguration

Windows Embedded CE unterstützt mehrere Buildkonfigurationen, die separat geändert werden können. Die beiden Standardkonfigurationen sind **Release** und **Debug**. Diese Buildkonfigurationen sind beim Erstellen eines OS Design automatisch verfügbar. In der Debug-Buildkonfiguration generiert der Compiler die Debuginformationen, verwaltet Verknüpfungen zum Quellcode in den Programmdatenbankdateien (.pdb) und unterstützt das Debuggen. Die schrittweise Codeausführung optimiert den Code nicht. Die in der Debug-Buildkonfiguration kompilierten Windows Embedded CE Run-Time Images sind normalerweise 50 bis 100 Prozent größer als die in der Release-Konfiguration kompilierten Images. Um eine Buildkonfiguration auszuwählen, öffnen Sie das Menü **Build** in Visual Studio und klicken Sie auf **Configuration Manager**. Wählen Sie im Dialogfeld **Configuration Manager** unter **Configuration** die gewünschte Buildkonfiguration aus. Sie können die Buildkonfiguration auch im Pulldown-Menü in der Standardsymbolleiste auswählen.

OS Design Property Pages

Für jede Buildkonfiguration können mehrere Projekteigenschaften konfiguriert werden, beispielsweise das Gebietsschema, KITL, benutzerdefinierte Buildaktionen, Teilprojekte im Binärimage und benutzerdefinierte SYSGEN-Variablen. Um auf diese Optionen zuzugreifen, öffnen Sie das Dialogfeld **Property Pages**, indem Sie mit der rechten Maustaste im Solution Explorer auf den Knoten **OS Design** klicken und **Properties** auswählen. Der Knoten **OS Design** ist das erste untergeordnete Objekt unter dem Knoten **Solution**. Der Dialogfeldtitel entspricht dem Projektnamen, beispielsweise ***OSDesign1***. Wenn der Solution Explorer nicht angezeigt wird, klicken Sie im Menü **View** auf **Solution Explorer**. Wenn im Solution Explorer die **Catalog Items View** oder die **Class View** geöffnet ist, klicken Sie auf die Registerkarte **Solution Explorer**, um die **Solution**-Struktur anzuzeigen.

TIPP Festlegen von Eigenschaften für mehrere Konfigurationen

In der oberen linken Ecke des Dialogfelds Property Pages befindet sich ein Listenfeld zum Auswählen der Buildkonfiguration. Beispielsweise können Sie die Option All Configurations oder Multiple Configurations auswählen. Diese Optionen sind hilfreich, wenn Sie die Eigenschaften für mehrere Buildkonfigurationen gleichzeitig festlegen möchten.

Gebietsschemaoptionen

Im Dialogfeld **Property Pages** unter **Configuration Properties** befindet sich der Knoten **Locale**, der das Konfigurieren der Spracheinstellungen für das Windows Embedded CE-Image ermöglicht (siehe Abbildung 1.2). Die Optionen unter **Locale** erfüllen die Anforderungen für die meisten Sprachen, um das OS Design zu lokalisieren. Einige Sprachen, insbesondere die ostasiatischen Sprachen, beispielsweise Japanisch, erfordern jedoch zusätzliche Katalogelemente. Beachten Sie, dass einige Katalogkomponenten für die Internationalisierung das Run-Time Image erheblich vergrößern.

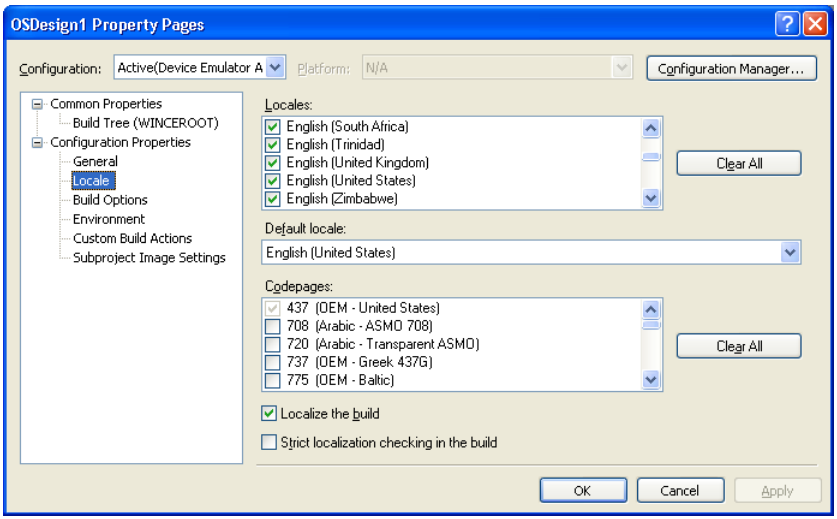


Abbildung 1.2 Eigenschaftenseite Locale

Die Eigenschaftenseite **Locale** ermöglicht das Konfigurieren folgender Optionen für das Run-Time Image:

- **Locales** Wählt die Sprachen aus, die zum Lokalisieren des Run-Time Images verfügbar sind. Wenn eine ausgewählte Sprache standardmäßig eine ANSI- und OEM-Codeseite umfasst, wird die Codeseite automatisch zum OS Design hinzugefügt (der entsprechende Codeseiteneintrag in der Liste **Codepages** ist markiert).
- **Default Locale** Definiert das Standardgebietsschema für das OS Design. Die Standardsprache ist **English (United States)**, die die Standardcodeseite **437 (OEM–United States)** verwendet.
- **Code Pages** Gibt die ANSI- und OEM-Codeseiten an, die im OS Design verfügbar sind.
- **Localize The Build** Weist den Buildprozess an, lokalisierte String- und Imageressourcen zu verwenden. Platform Builder führt die Lokalisierung des OS Designs während der Erstellung des Images im OS Design-Buildprozess aus. In den Binärdateien für die allgemeinen Komponenten werden die lokalisierten Ressourcen über *res2exe* integriert.
- **Strict Localization Checking In The Build** Bewirkt, dass der Buildprozess fehlschlägt, wenn die Lokalisierungsressourcen nicht vorhanden sind (anstatt die Ressourcen basierend auf dem Standardgebietsschema zu verwenden).

Buildoptionen

Direkt unter dem Knoten **Locale** im Dialogfeld **Property Pages** befindet sich der Knoten **Build Options**, der Optionen zum Nachverfolgen der Steuerereignisse, zum Debuggen und weitere Buildoptionen für das aktive OS Design umfasst (siehe Abbildung 1.3).

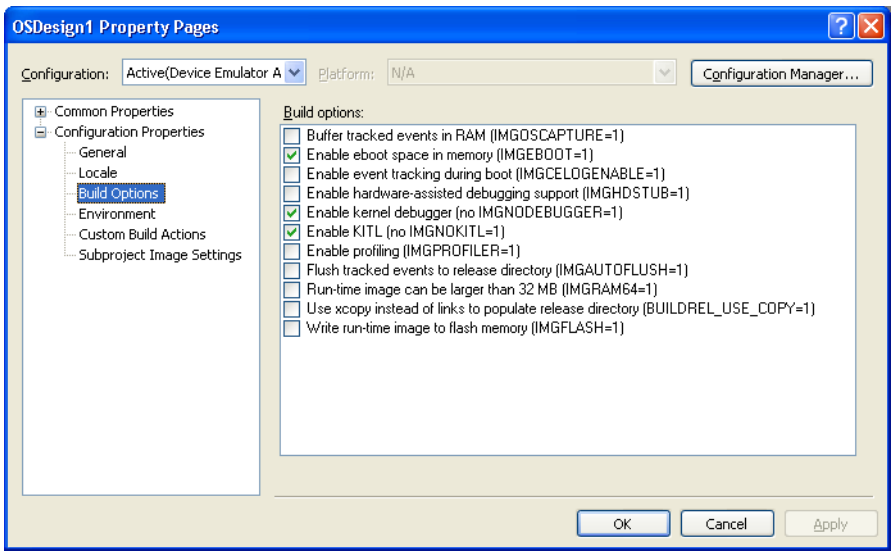


Abbildung 1.3 Eigenschaftenseite Build Options

Die Eigenschaftenseite **Build Options** ermöglicht das Konfigurieren folgender Optionen für das Run-Time Image:

- **Buffer Tracked Events In RAM** Verursacht, dass Platform Builder die Datei *OSCapture.exe* in das CE-Image einbezieht. Diese Option ermöglicht außerdem das Protokollieren von Betriebssystemereignissen, die von *OSCapture.exe* im RAM erfasst und zur späteren Überprüfung in eine Datei kopiert werden können.
- **Enable Eboot Space In Memory** Ermöglicht dem Ethernet Boot Loader (EBOOT) das Übergeben von Daten an das Windows Embedded CE-Betriebssystem zur Startzeit.
- **Enable Event Tracking During Boot** Ermöglicht das Zusammenstellen der CE-Ereignisprotokolldaten während des Startprozesses. Wenn Sie diese Option aktivieren, beginnt die Ereignisnachverfolgung bevor die Initialisierung des Kernels und des Dateisystems abgeschlossen ist.

- **Enable Hardware-Assisted Debugging Support** Diese Option ist für einige Hardware-Debugtools von Drittanbietern erforderlich (JTAG überprüft die Kompatibilität mit *exdi2*).
- **Enable Kernel Debugger** Aktiviert den Windows Embedded CE Debugger, damit Sie den Code im Run-Time Image durchlaufen können. Das Kerneldebuggen erfordert KITL, um zur Laufzeit mit Platform Builder zu kommunizieren.
- **Enable KITL** Fügt den KITL zum Run-Time Image hinzu. KITL ist ein nützliches Debugfeature, das dem Entwickler die Verwendung des Kerneldebuggers, die Interaktion mit dem Dateisystem des Remotegeräts, der Registrierung und anderen Komponenten sowie das Ausführen von Code ermöglicht. Beziehen Sie den KITL nicht in den endgültigen Build des Betriebssystems ein, da dieser Overhead verursacht und den Startprozess beim Herstellen der Verbindung mit einem Hostcomputer verlängert.
- **Enable Profiling** Aktiviert den Kernel Profiler im Run-Time Image, mit dem Sie Zeit- und Leistungsdaten zusammenstellen und überprüfen können. Der Kernel Profiler ist ein nützliches Tool zum Optimieren der Leistung von Windows Embedded CE auf dem Zielgerät.
- **Flush Tracked Events To Release Directory** Fügt die Datei *CeLogFlush.exe* zum Run-Time Image hinzu, die die Protokolldateien automatisch entfernt, die von *OSCapture.exe* in der Datei *Celog.clg* im Verzeichnis *Release* auf dem Entwicklungscomputer erfasst werden.
- **Run-Time Image Can Be Larger Than 32 MB** Ermöglicht das Erstellen eines Images, das größer als 32 MB ist. Aktivieren Sie diese Option nicht, wenn ein Image größer als 64 MB sein soll. In diesem Fall müssen Sie eine Umgebungsvariable auf die entsprechende Größe festlegen (beispielsweise IMGRAM128).
- **Use Xcopy Instead Of Links To Populate Release Directory** Erstellt Kopien der Dateien mit *xcopy* anstatt mit *copylink*. *Copylink* kann nur Verknüpfungen zu den Dateien erstellen, anstatt diese zu kopieren, und erfordert das NTFS-Dateisystem auf dem Entwicklungscomputer.
- **Write Run-Time Image To Flash Memory** Weist EBOOT an, das Run-Time Image in den Flashspeicher auf dem Zielgerät zu schreiben.

Umgebungsoptionen

Das Dialogfeld **Property Pages** enthält die Option **Environment**, mit der die Umgebungsvariablen für den Buildprozess konfiguriert werden. Sie können die meisten Features in Windows Embedded CE 6.0 R2 über die Katalogkomponenten aktivieren. Für einige Optionen müssen Sie jedoch eine SYSGEN-Variable festlegen, damit Platform Builder den erforderlichen Code kompiliert und in das Run-Time Image einbezieht. Das Festlegen von Umgebungsvariablen, die den Buildprozess beeinflussen, ist hilfreich beim Entwickeln eines BSP. Sie können während des Windows Embedded CE-Buildprozesses über die Befehlszeile auf die Umgebungsvariablen zugreifen. Außerdem können Sie mit Umgebungsvariablen flexible Informationen in den Quelldateien, .bib- (Buildprozess Image Builder) und .reg-Dateien (Registrierung) angeben.

TIPP Wenn es in Debug, aber nicht in Release funktioniert

Wenn Sie ein Run-Time Image in der Debug-Konfiguration erstellen können, aber nicht in der Release-Konfiguration, öffnen Sie das Dialogfeld Property Pages, wählen Sie im Listenfeld Configuration die Option All Configurations aus und aktivieren Sie die Option Environment, um die Umgebungsvariablen für Debug und Release auf den gleichen Wert festzulegen.

Erweiterte OS Design-Konfigurationen

Dieser Abschnitt befasst sich mit erweiterten OS Designs. Sie erfahren, wie mehrere Plattformen mit dem gleichen OS Design unterstützt werden. Außerdem werden die Dateiverzeichnisse und Dateitypen erklärt, die ein OS Design normalerweise umfasst.

Zuordnen eines OS Designs zu mehreren Plattformen

Wenn Sie mit dem OS Design Wizard ein OS Design-Projekt erstellen, können Sie auf der Assistentenseite **Board Support Packages** ein oder mehrere BSPs auswählen. Wenn Sie ein OS Design mehreren BSPs zuordnen, können Sie separate Run-Time Images mit identischem Inhalt für mehrere Plattformen generieren. Dies ist insbesondere für Projekte nützlich, an denen mehrere Entwicklungsteams arbeiten und die Zielhardware derzeit nicht verfügbar ist. Beispielsweise können Sie ein Run-Time Image für eine Emulator-basierte Plattform generieren, damit das Entwicklungsteam mit der Arbeit beginnen kann, bevor die Hardware verfügbar ist. Das Entwicklungsteam kann die APIs (Application Programming Interfaces) verwenden, bevor die endgültige Zielplattform zur Verfügung steht. Die APIs werden

in das Ziel einbezogen, da die beiden Run-Time Images die gleichen Komponenten und Konfigurationseinstellungen verwenden.

Nach der ursprünglichen Erstellung können Sie die Unterstützung für mehrere Plattformen zu einem OS Design hinzufügen. Aktivieren Sie hierzu die entsprechenden Kontrollkästchen unter BSP in der **Catalog Items View** des Solution Explorers. Wenn Sie ein BSP auswählen, wird die zusätzliche Plattform automatisch zur Release- und Debug-Konfiguration hinzugefügt. Anschließend können Sie zwischen den Plattformen und Buildkonfigurationen wechseln, indem Sie den **Configuration Manager** verwenden, der im Menü **Build** in Visual Studio verfügbar ist. Sie müssen jedoch für jede Plattform den gesamten Buildprozess ausführen, einschließlich der zeitaufwendigen Sysgen-Phase.

Pfade und Dateien im OS Design

Um Ihre OS Designs zu verwenden und neu zu verteilen, müssen Sie wissen, welche Dateien das OS Design umfasst und wo die Dateien auf dem Entwicklungscomputer gespeichert sind. Standardmäßig sind die OS Designs im Verzeichnis *%_WINCEROOT%\OSDesigns* gespeichert. Jedes Projekt entspricht einem separaten untergeordneten Verzeichnis. OS Designs haben normalerweise folgende Datei- und Verzeichnisstruktur:

- ***<Solution Name>*** Das übergeordnete Verzeichnis, das Visual Studio für das Projekt erstellt.
  - ***<Solution Name>*.sln** Die Visual Studio sln-Datei (Solution) zum Speichern der Einstellungen für das OS Design-Projekt. Der Dateiname ist normalerweise mit dem Namen des OS Designs identisch.
  - ***<Solution Name>*.suo** Die Visual Studio suo-Datei (Solution User Options), die die Benutzerinformationen enthält, beispielsweise den Status der Solution Explorer-Ansichten. Der Dateiname ist normalerweise mit dem Namen des OS Designs identisch.
  - ***<OS Design Name>*** Das übergeordnete Verzeichnis für die restlichen Dateien des OS Design-Projekts.
    - ***<OS Design Name>*.pbxml** Die Katalogdatei des OS Designs. Diese Datei enthält Referenzen auf ausgewählte Katalogkomponenten und alle Einstellungen für das OS Design.
    - **Subprojects** Dieses Verzeichnis umfasst einen separaten untergeordneten Order für jedes Teilprojekt, das für das OS Design erstellt wird.

- **SDKs** Dieses Verzeichnis enthält die Software Development Kits (SDKs), die für das OS Design erstellt werden.
- **Reldir** Das Releaseverzeichnis. Platform Builder kopiert die Dateien beim Erstellen des Run-Time Images in dieses Verzeichnis. Die Dateien können anschließend auf das Zielgerät heruntergeladen werden.
- **WinCE600** Die Dateien, einschließlich die Ressourcen- und Konfigurationsdateien für das aktuelle OS Design, werden nach Abschluss der SYSGEN-Phase in dieses Verzeichnis kopiert.

Software für die Quellcodeverwaltung

Ein OS Design besteht im wesentlichen aus Konfigurationsdateien für Platform Builder, mit denen das Windows Embedded CE Run-Time Image generiert wird. Wenn Sie die Arbeit Ihres Entwicklungsteams mit Software für die Quellcodeverwaltung koordinieren, müssen Sie die Konfigurationsdateien nur im Quellcodeverwaltungsspeicher speichern. Sie müssen keine anderen Dateien aus den Verzeichnisse *CESysgen* oder *Reldir* einbeziehen, die während des Buildprozesses des Run-Time Images verwendet werden, da diese mit Platform Builder und dem BSP auf jeder Arbeitsstation wiederhergestellt werden können. Beziehen Sie außerdem weder die user- oder suo-Dateien noch die ncb-Dateien ein, da diese benutzerspezifische Einstellungen für die IDE bzw. nur IntelliSense®-Daten enthalten.

Zusammenfassung

Platform Builder für Windows Embedded CE 6.0 R2 umfasst einen OS Design Wizard, mit dem Sie die grundlegenden Schritte zum Erstellen eines OS Designs ausführen können. Sie können ein oder mehrere BSPs, um alle hardwarespezifischen Gerätetreiber und Dienstprogramme für die Zielplattform einzubeziehen, sowie eine Designvorlage mit verschiedenen Vorlagenvarianten auswählen, um weitere Katalogelemente hinzuzufügen. Nachdem der OS Design Wizard abgeschlossen ist, können Sie das OS Design anpassen. Sie können nicht erforderliche Katalogelemente ausschließen, weitere Komponenten hinzufügen und die Projekteigenschaften, beispielsweise Debug- und Release-Buildoptionen, konfigurieren. In der Debug-Buildkonfiguration bezieht Platform Builder die Debuginformationen in das Run-Time Image ein, das 50 bis 100 Prozent größer als für Release-Builds ist. Ein Debug-Build unterstützt jedoch das Debuggen und die schrittweise Codeausführung während des Entwicklungsprozesses. Da Sie die Debug- und Release-Buildoptionen separat konfigurieren können, wird das OS Design möglicherweise in der Debug-

Konfiguration, aber nicht in der Release-Konfiguration, kompiliert. In diesem Fall müssen Sie die Umgebungsvariablen für Debug und Release auf die gleichen Werte festlegen. Um die OS Designs weiterzugeben, benötigen Sie die Quelldateien, die standardmäßig im Verzeichnis *%_WINCEROOT%\OSDesigns* gespeichert sind. Mit Software für die Quellcodeverwaltung können Sie die Arbeit eines Entwicklungsteams koordinieren.

Lektion 2: Konfigurieren von Windows Embedded CE-Teilprojekten

Ein Teilprojekt ist ein Visual Studio-Projekt, das in ein übergeordnetes Projekt eingefügt wird, um relativ unabhängige Komponenten in eine Gesamtlösung aufzunehmen. Normalerweise entspricht das übergeordnete Projekt einem OS Design. Teilprojekte können folgende Formate haben:

- Eine Anwendung (.NET oder nativ).
- Eine DLL (Dynamic Link Library)
- Eine statische Bibliothek.
- Ein leeres Projekt, das ausschließlich Konfigurationseinstellungen umfasst.

Teilprojekte eignen sich zum Einbeziehen einer Anwendung, eines Gerätetreibers oder eines anderen Codemoduls in ein OS Design und zum Verwalten des Codes und des OS Designs als eine Lösung.

Nach Abschluss dieser Lektion können Sie:

- Teilprojekte anlegen und konfigurieren.
- Teilprojekte erstellen und verwenden.

Veranschlagte Zeit für die Lektion: 20 Minuten.

Übersicht der Windows Embedded-Teilprojekte

Platform Builder für Windows Embedded CE ermöglicht das Erstellen von Teilprojekten als Bestandteil des OS Designs. Da Teilprojekte modular und verteilbar sind, stellen sie eine praktische Methode zum Hinzufügen von Anwendungen, Treibern oder anderen Dateien zum OS Design dar, ohne diese manuell in die Buildstruktur als Teil des BSP einfügen zu müssen. Sie können Teilprojekte auch für interne Testanwendungen und Entwicklungstools verwenden, die schnell und einfach erstellt und auf einem Testgerät ausgeführt werden können.

Teilprojekttypen

Windows Embedded CE unterstützt folgende Teilprojekttypen:

- **Anwendungen** Win32®-Anwendungen mit einer Benutzeroberfläche, die in C oder C++ programmiert sind.

- **Konsolenanwendungen** Win32-Anwendungen ohne Benutzeroberfläche, die in C oder C++ programmiert sind.
- **Dynamic-Link Library (DLL)** Treiber oder Codebibliotheken, die zur Laufzeit geladen werden.
- **Statische Bibliothek** Codemodule in Form von Bibliothekdateien (.lib), die Sie verknüpfen, um andere Teilprojekte zu erstellen, oder als Teil des SDK des OS Designs exportieren können.
- **TUX Dynamic-Link Library** Benutzerdefinierte Windows Embedded CE-Testkomponenten für den Microsoft Windows CE Test Kit (CETK). Diese Komponenten sind in Kapitel 4 beschrieben.

Erstellen und Hinzufügen von Teilprojekten zu einem OS Design

Das Erstellen eines neuen Teilprojekts oder das Hinzufügen eines vorhandenen Projekts als Teilprojekt zu einem OS Design ist unkompliziert. Sie können den **Windows Embedded CE Subproject Wizard** größtenteils zum Ausführen dieser Aufgabe verwenden. Klicken Sie hierzu mit der rechten Maustaste im Solution Explorer auf den Ordner **Subprojects** und wählen Sie **Add New Subproject** oder **Add Existing Subproject** aus. Sie sollten jedoch mit den Details vertraut sein, einschließlich den Teilprojekttypen, den Dateien und Einstellungen, die der CE Subproject Wizard erstellt, sowie den Anpassungsoptionen für Teilprojekte.

Der CE Subproject Wizard erstellt einen untergeordneten Ordner im OS Design-Ordner, der alle erforderlichen Konfigurationsdateien enthält, einschließlich:

- **<*Name*>.pbpxml** Eine XML-Datei, die die Metadateninformationen über das Teilprojekt enthält. Die Datei verweist auf die bib-, reg-, SOURCES- und DIRS-Dateien zum Erstellen des Teilprojekts.
- **<*Name*>.bib** Eine bib-Datei (Binary Image Builder), die während der makeimg-Phase des Buildprozesses verwendet wird, um Dateien zum Binärimage hinzuzufügen.
- **<*Name*>.reg** Eine Registrierungsdatei mit den Einstellungen für das endgültige Run-Time Image.
- **Sources** Eine Windows Embedded CE-Quelldatei. Diese Datei enthält die Buildoptionen zum Steuern des Windows Embedded CE-Buildprozesses.
- **Makefile** Eine Datei, die in Verbindung mit der SOURCES-Datei im Windows Embedded CE-Buildprozess verwendet wird.

Um eine Kopie eines Teilprojekts zu erstellen, öffnen Sie den OSDesigns-Order (*%_WINCEROOT%\OSDesigns*) und anschließend den Ordner für Ihr OS Design. Dieser Ordner enthält normalerweise die *<OS Design Name>.sln*-Datei und einen Ordner mit dem Namen des OS Designs. In diesem Ordner finden Sie die Definitionsdatei des OS Designs *<OS Design Name>.pbxml* und mehrere Unterverzeichnisse. Eines dieser Unterverzeichnisse ist der Teilprojektordner (siehe Abbildung 1.4). Sie sollten diesen Ordner sichern. Sie können das Teilprojekt später zu einem OS Design hinzufügen, indem Sie mit der rechten Maustaste im Solution Explorer auf den Container **Subprojects** klicken und **Add Existing Subproject** auswählen.

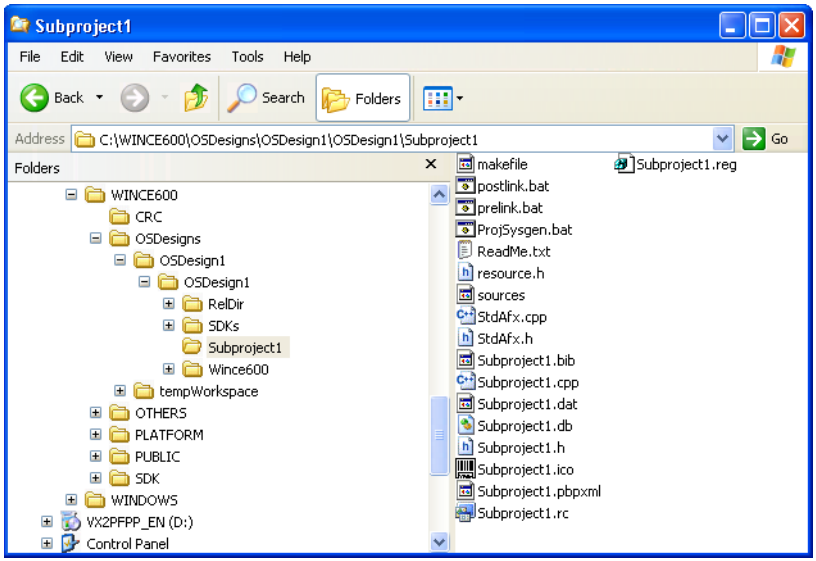


Abbildung 1.4 Ein Teilprojektordner in einem OS Design-Projekt

Erstellen von Windows Embedded CE-Anwendungen und Dynamic-Link Libraries

Um eine Windows Embedded CE-Anwendung oder DLL zu einem OS Design hinzuzufügen, erstellen Sie mit dem CE Subproject Wizard das entsprechende Teilprojekt. Sie können zwar mit einem leeren Teilprojekt beginnen, aber es ist praktischer, eine einfache Konsolen- oder GUI-Anwendungsvorlage auszuwählen und den erforderlichen Code hinzuzufügen.

Erstellen von statischen Bibliotheken

Der CE Subproject Wizard umfasst auch eine Option zum Erstellen einer statischen Bibliothek, die Sie mit einem anderen Teilprojekt verknüpfen oder als Teil eines SDK exportieren können. Dies ist hilfreich zum Aufteilen anspruchsvollerer Teilprojekte oder zum Bereitstellen weiterer Optionen für Anwendungsentwickler, die Hardware- oder Firmwarelösungen entwickeln. Wenn andere Teilprojekte im OS Design von einer statischen Bibliothek abhängen, müssen Sie die Buildreihenfolge der Teilprojekte möglicherweise ändern, um die Bibliothek effizient einzusetzen. Wenn die statische Bibliothek beispielsweise von einer Windows Embedded CE-Anwendung verwendet wird, erstellen Sie zuerst die Bibliothek, damit die aktualisierte Bibliothek für den Anwendungsbuildprozess verfügbar ist.

Erstellen eines Teilprojekts zum Hinzufügen von Dateien oder Umgebungsvariablen zu einem Run-Time Image

Teilprojekte müssen nicht unbedingt Quellcode umfassen. Beispielsweise können Sie mit dem CE Subproject Wizard ein leeres Teilprojekt erstellen, die SOURCES-Datei ändern und **TARGETTYPE=NOTARGET** festlegen, wenn Sie keine Binärdateien generieren möchten. Anschließend können Sie Dateien zum Run-Time Image hinzufügen, indem Sie die entsprechenden Referenzen in die bib-Datei des Teilprojekts einfügen. Sie können Registrierungseinstellungen zur reg-Datei des Teilprojekts und SYSGEN-Variablen hinzufügen, indem Sie die Datei *Projsysgen.bat* bearbeiten. Obwohl es schneller und praktischer ist, die reg- und bib-Dateien sowie die Projekteigenschaften des OS Designs direkt zu ändern, ist das Erstellen eines Teilprojekts vorteilhaft, wenn Sie die Änderungen in mehreren OS Designs übernehmen möchten.

Konfigurieren eines Teilprojekts

Visual Studio umfasst zahlreiche Optionen in den Projekteigenschaften, die Sie konfigurieren können, um den Buildprozess für Teilprojekte anzupassen. Um diese Einstellungen zu konfigurieren, öffnen Sie das Dialogfeld **Property Pages**. Die Eigenschaften für das Teilprojekt werden unter **Subproject Image Settings** angezeigt. Für ein Teilprojekt, das im aktuellen OS Design erstellt oder hinzugefügt wird, können Sie folgende Parameter konfigurieren:

- **Exclude From Build** Diese Option schließt das Teilprojekt aus dem Buildprozess des OS Designs aus. Das heißt, das Buildmodul verarbeitet die Quelldateien nicht, die zum ausgewählten Teilprojekt gehören.

- **Exclude From Image** Wenn ein Teilprojekt geändert wird, kann das Bereitstellen eines Run-Time Images zeitaufwendig sein. Sie müssen die Verbindung mit der Zielplattform trennen, das Projekt neu erstellen, ein neues Image erstellen, die Verbindung mit der Zielplattform wiederherstellen und das aktualisierte Image herunterladen. Um den Aufwand zu reduzieren, sollten Sie das Teilprojekt aus dem Run-Time Image ausschließen, indem Sie die Option **Exclude From Image** aktivieren. Aktualisieren Sie die Datei über KITL, ActiveSync oder mit einer anderen Methode zur Laufzeit auf dem Gerät.
- **Always Build And Link As Debug** Diese Option erstellt das Teilprojekt in der Debug-Buildkonfiguration. Der OS Design-Buildprozess verwendet die Release-Konfiguration. Auf diese Art können Sie das Teilprojekt mit dem Kernel Debugger debuggen, während das Betriebssystem in der Release-Version ausgeführt wird (die Option aktiviert den Kernel Debugger nicht automatisch).

HINWEIS Ausschließen aus dem Run-Time Image

Wenn Sie ein Teilprojekt aus dem Run-Time Image ausschließen, werden die Teilprojektdateien nicht in die *Nk.bin*-Datei einbezogen, die auf das Zielgerät heruntergeladen wird. Windows Embedded CE greift bei Bedarf direkt über KITL im Releaseverzeichnis auf die Teilprojektdateien zu. Das bedeutet, dass Sie den Code in einem Treiber oder Anwendungsteilprojekt ändern können, ohne das Run-Time Image erneut bereitstellen zu müssen. Stellen Sie sicher, dass der Code derzeit nicht auf dem Remotegerät ausgeführt wird. Anschließend können Sie den Code erneut erstellen und ausführen.

Zusammenfassung

Sie können Windows Embedded CE-Teilprojekte verwenden, um Anwendungen, Treiber, DLLs und statische Bibliotheken zu einem OS Design hinzuzufügen. Dies ist nützlich, wenn Sie ein komplexes Windows Embedded CE-Entwicklungsprojekt verwalten, das zahlreiche Anwendungen und Komponenten umfasst. Beispielsweise können Sie eine benutzerdefinierte Shellanwendung oder einen Gerätetreiber für ein USB-Peripheriegerät in Form eines Teilprojekts in ein OS Design einbeziehen. Diese Komponenten können anschließend von verschiedenen Entwicklungsteams implementiert werden. Mit Windows Embedded CE-Teilprojekten können Sie außerdem Registrierungseinstellungen, Umgebungsvariablen oder andere Dateien zum OS Design hinzufügen, beispielsweise Run-Time Images für die Core Connectivity-Schnittstellen (CoreCon) oder eine Testanwendung. Teilprojekte können separat gesichert oder zu vorhandenen Teilprojekten hinzugefügt werden.

Lektion 3: Klonen von Komponenten

Platform Builder für Windows Embedded CE 6.0 R2 umfasst öffentlichen Quellcode, den Sie wiederverwenden und an Ihre Anforderungen anpassen können. Sie können den Quellcode der meisten Komponenten in Windows Embedded CE analysieren und ändern, von der Shell bis zur MDD (Model Device Driver)-Schicht des seriellen Treibers. Sie können den Quellcode jedoch nicht direkt ändern. Stattdessen müssen Sie eine funktionelle Kopie des öffentlichen Codes erstellen, um die gewünschten Komponenten anzupassen, ohne die ursprüngliche Windows Embedded CE 6.0 R2-Codebasis zu beeinträchtigen.

Nach Abschluss dieser Lektion können Sie:

- Eine zu klonende Komponente identifizieren.
- Eine vorhandene Komponente klonen.

Veranschlagte Zeit für die Lektion: 15 Minuten.

Ändern der Struktur Public und Klonen von Komponenten

Wenn sich der gewünschte Code im Ordner *%_WINCEROOT%\Public* befindet, möchten Sie den Code möglicherweise in diesem Verzeichnis ändern und erstellen, ohne ihn zuerst in einen anderen Ordner zu verschieben. Es gibt jedoch mehrere Gründe, die Struktur *Public* nicht zu ändern:

- Sie müssen das Verzeichnis *Public* sichern und für jedes OS Design-Projekt separate Verzeichnisversionen verwalten, beispielsweise *WINCE600\PUBLIC_Company1*, *WINCE600\PUBLIC_Company2* und *WINCE600\PUBLIC_Backup*.
- Windows Embedded CE-Updates, QFEs (Quick Fix Engineering) und Service Packs können Ihre Änderungen überschreiben oder sind nicht mit diesen kompatibel.
- Die Neuverteilung des Codes ist schwierig und fehleranfällig.
- Wenn Sie den Code in der Verzeichnisstruktur *Public* ändern, dauert das Erstellen des Betriebssystems bis zu drei Stunden. Wenn Sie bereits so umfassend mit dem CE-Buildprozess vertraut sind, dass Sie bestimmten Code erneut erstellen können, ohne den gesamten Ordner *Public* neu zu erstellen, können Sie auch Komponenten klonen.

ACHTUNG Ändern des öffentlichen Codes

Ändern Sie den Inhalt der Ordnerstruktur *Public* nicht. Auch wenn das Klonen von Komponenten auf den ersten Blick schwierig erscheint, kann es den Entwicklungsaufwand langfristig reduzieren.

Klonen von öffentlichem Code

Platform Builder unterstützt das sofortige Klonen einiger Windows Embedded CE-Komponenten. Um diese Komponenten zu klonen, klicken Sie mit der rechten Maustaste in der **Catalog Items View** des Solution Explorers auf das Katalogelement und wählen Sie **Clone Catalog Item** aus. Platform Builder erstellt automatisch ein Teilprojekt für die ausgewählte Komponente im OS Design mit einer Kopie des Codes. Bevor Sie eine andere Methode verwenden, beispielsweise das Sysgen Capture-Tool, sollten Sie überprüfen, ob die gewünschte Katalogkomponente die Option **Clone Catalog Item** unterstützt. Wenn die Option von der Katalogkomponente unterstützt wird, sind nur zwei weitere Mausklicks erforderlich (siehe Abbildung 1.5).

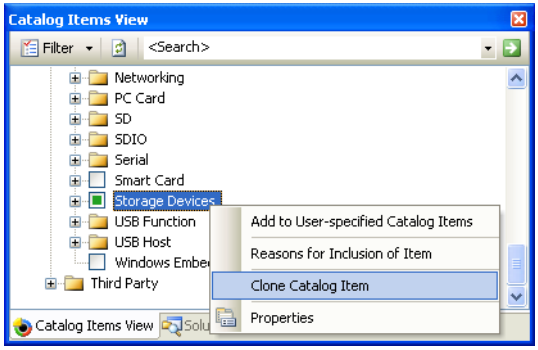


Abbildung 1.5 Klonen eines Katalogelements

Wenn Sie eine Komponente nicht automatisch mit IDE klonen können, müssen Sie dies manuell ausführen. Beim Überprüfen der Sources-Datei für eine dll- oder exe-Datei in der Verzeichnisstruktur *Public* werden Sie jedoch feststellen, dass diese Datei nicht mit der Sources-Datei im Plattformverzeichnis oder in einem Teilprojekt identisch ist. Dies ist darauf zurückzuführen, dass sich der Buildprozess für die Verzeichnisstruktur *Public* vom BSP-Buildprozess unterscheidet. Die Buildanweisungen sind in der Makefile-Datei definiert, die immer im gleichen Verzeichnis wie die zugehörige Sources-Datei gespeichert ist. Die Verzeichnisstruktur

Public muss die SYSGEN-Phase unterstützen, in der die erforderlichen Komponenten relativ miteinander verknüpft werden.

Das Konvertieren einer Komponente in der Verzeichnisstruktur Public in eine BSP-Komponente oder ein Teilprojekt umfasst mehrere Schritte, die in der Produktdokumentation für Platform Builder für Microsoft Windows Embedded CE unter „Using the Sysgen Capture Tool" beschrieben sind (*http://msdn2.microsoft.com/en-us/library/aa924385.aspx*).

Sie müssen folgende Schritte ausführen:

1. Kopieren Sie den Code der gewünschten öffentlichen Komponente in ein neues Verzeichnis.
2. Bearbeiten Sie die Sources-Datei im neuen Verzeichnis und fügen Sie die Zeile **RELEASETYPE=PLATFORM** hinzu oder ändern Sie den Wert in **PLATFORM**, wenn die Zeile bereits vorhanden ist, damit das Buildmodul die Buildausgabe im Ordner *%_TARGETPLATROOT%* speichert.
3. Fügen Sie **WINCEOEM=1** zur Sources-Datei hinzu und erstellen Sie die Komponente im neuen Verzeichnis. Möglicherweise müssen Sie weitere Änderungen vornehmen, um Buildfehler zu beheben.
4. Verwenden Sie das Sysgen Capture-Tool, um modulare Sources- und Dirs-Dateien zu erstellen.
5. Benennen Sie die mit dem Sysgen Capture-Tool erstellten Dateien um und verwenden Sie die Dateien zusammen mit einer Makefile-Datei, um das geklonte Modul neu zu erstellen.

Nachdem Sie alle erforderlichen Änderungen an der geklonten Komponente vorgenommen haben, können Sie diese wie jeden anderen Code anpassen und neu verteilen.

Zusammenfassung

Windows Embedded CE umfasst die Verzeichnisstruktur *Public* mit Quellcode für die meisten CE-Komponenten, den Sie jedoch nicht direkt in der Verzeichnisstruktur ändern sollten. Klonen Sie die Elemente stattdessen automatisch oder manuell. Das Ändern des Quellcodes in der Verzeichnisstruktur *Public* verursacht Probleme, außer Sie sind umfassend mit dem Buildsystem vertraut. In diesem Fall sind Ihnen jedoch alle Gründe bekannt, warum Sie die Klonmethode verwenden sollten.

Lektion 4: Verwalten der Katalogelemente

Eines der nützlichsten Features von Windows Embedded CE ist das Katalogsystem. Unter Verwendung des Katalogs kann der Entwickler die Windows Embedded CE-Firmware schnell und zweckdienlich an seine Anforderungen anpassen. Wenn Sie für jede Ihrer Komponenten ein benutzerdefiniertes Katalogelement erstellen, wird die Installation und Konfiguration der Komponenten vereinfacht. Dies ist der Unterschied zwischen Ad-Hoc-Lösungen und professionellen Windows Embedded CE-Lösungen. Für Ad-Hoc-Lösungen sind Basisinstallationshinweise und eine Liste der erforderlichen SYSGEN-Variablen möglicherweise ausreichend. Professionelle Software sollte jedoch Katalogelemente mit den entsprechenden Werten für die SYSGEN-Variablen und Konfigurationseinstellungen umfassen.

Nach Abschluss dieser Lektion können Sie:

- Den Inhalt des Katalogs anpassen.
- Einen neuen Komponenteneintrag zum BSP-Katalog hinzufügen.

Veranschlagte Zeit für die Lektion: 20 Minuten.

Katalogdateien - Übersicht

Der Windows Embedded CE-Katalog verwendet Dateien im XML-Format (Extensible Markup Language) mit der Dateierweiterung pbcxml. Der Katalog umfasst zahlreiche pbcxml-Dateien im *WINCEROOT*-Verzeichnis. Platform Builder listet diese Dateien auf, um die **Catalog Items View** im Solution Explorer zu generieren.

Platform Builder durchsucht folgende Verzeichnisse, um die Katalogelemente aufzulisten:

- **Öffentliche Katalogdateien** *%_WINCEROOT%\Public\<Unterverzeichnis>\Catalog*
- **BSP-Katalogdateien** *%_WINCEROOT%\Platform\<Unterverzeichnis>\Catalog*
- **Katalogdateien von Drittanbietern** *%_WINCEROOT%\3rdParty\<Unterverzeichnis>\Catalog*
- **Allgemeine SOC-Dateien (System-On-Chip) files** *%_WINCEROOT%\Platform\Common\Src\soc\<Unterverzeichnis>\Catalog*

HINWEIS 3rdParty-Ordner

Der Ordner *3rdParty* enthält normalerweise eigenständige Anwendungen oder Quellanwendungen, die einbezogen und als Teil eines OS Designs verteilt werden können. Durch Auflisten der pbcxml-Dateien ermöglicht Platform Builder das Hinzufügen von Einträgen für diese Komponenten zur Catalog Items View.

Erstellen und Ändern von Katalogeinträgen

Um ein neues Katalogelement zum Windows Embedded CE-Katalog hinzuzufügen, können Sie eine Kopie einer vorhandenen Katalogdatei (pbcxml-Datei) erstellen und den Dateiinhalt anschließend mit dem Catalog Editor im Platform Builder bearbeiten. Sie können eine neue Katalogdatei im Platform Builder erstellen, indem Sie im Menü **File** in Visual Studio auf **New** zeigen und **File** auswählen. Wählen Sie im Dialogfeld **New File** unter **Platform Builder for CE 6.0 R2** die Option **Platform Builder Catalog File** aus und klicken Sie auf **Open**.

HINWEIS Bearbeiten von Katalogdateien

Bearbeiten Sie Katalogdateien immer mit dem Catalog Editor im Platform Builder. Es sind keine Einstellungen vorhanden, die die Verwendung eines Texteditors, beispielsweise Notepad, erfordern. Das manuelle Öffnen und Bearbeiten der Katalogdateien außerhalb des Platform Builders ist unnötig und zeitaufwendig.

Katalogeintragseigenschaften

Für jeden Katalogeintrag sind mehrere Eigenschaften vorhanden, die Sie im Platform Builder ändern können (siehe Abbildung 1.6). Die wichtigsten Eigenschaften sind:

- **Unique ID** Eine eindeutige ID-Zeichenfolge.
- **Name** Der Name der Katalogkomponente in der **Catalog Items View**.
- **Description** Eine Beschreibung der Komponente, die angezeigt wird, wenn der Benutzer mit der Maus auf ein Katalogelement zeigt.
- **Modules** Eine Liste der Dateien, die zur Katalogkomponente gehören.
- **SYSGEN Variable** Eine Umgebungsvariable für das Katalogelement. Wenn die Katalogkomponente eine SYSGEN-Variable festlegt, wird diese hier angezeigt.
- **Additional Variables** Eine Gruppe weiterer Umgebungsvariablen für das Katalogelement. Dies ist der wichtigste Teil der Katalogkomponente in einem BSP, da dieses Feld das Festlegen von Umgebungsvariablen in Sources-, bib- und

reg-Dateien zum Steuern des Buildprozesses ermöglicht. Sie können dieses Feld auch verwenden, um Abhängigkeiten von anderen Komponenten zu generieren.

- **Platform Directory** Das Verzeichnis der Katalogdateien. Legen Sie diese Eigenschaft für ein neues BSP auf den Namen des BSP-Verzeichnisses fest.

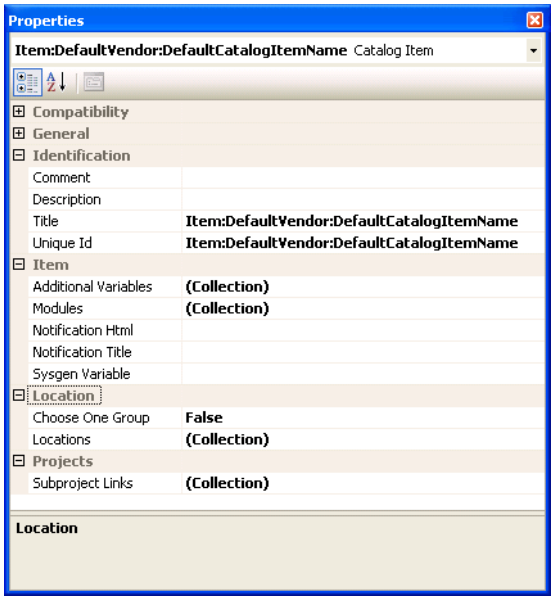


Abbildung 1.6 Katalogelementeigenschaften

HINWEIS Eindeutige Namen

Jede Katalogkomponente muss eine eindeutige ID haben, die sich normalerweise aus dem Anbieter- und Komponentennamen zusammensetzt. Wenn Sie ein BSP unter Verwendung der Option Clone Catalog Item klonen, erstellt der Platform Builder für die geklonte Komponente einen eindeutigen Namen. Stellen Sie beim manuellen Bearbeiten von Katalogdateien sicher, dass Sie eindeutige IDs verwenden.

Hinzufügen eines neuen Katalogelements zu einem OS Design

Um eine neue Katalogdatei oder ein neues Katalogelement zu verwenden, stellen Sie sicher, dass die entsprechende pbcxml-Datei im Unterverzeichnis *Catalog* im Verzeichnis *3rdParty* oder *Platform* vorhanden ist, und klicken Sie in der **Catalog Items View** in Visual Studio auf **Refresh Catalog Tree**. Platform Builder generiert den Katalog dynamisch erneut, indem er die Verzeichnisse *3rdParty* und *Platform* durchsucht und alle vorhandenen Katalogdateien verarbeitet. Um die neue in der

Catalog Items View angezeigte Komponente in das OS Design einzubeziehen, aktivieren Sie das entsprechende Kontrollkästchen.

Verwenden eines Katalogelements für die BSP-Entwicklung

Nachdem Sie die neue Katalogkomponente hinzugefügt und die elementspezifischen Umgebungsvariablen festgelegt haben, können Sie mit dieser Methode die Komponente in ein BSP einbeziehen, die C/C++-Builddirektiven festlegen und die Systemregistrierungseinstellungen im Run-Time Image ändern. Wenn andere Entwickler mit diesem BSP Ihr Katalogelement in einem OS Design-Projekt auswählen, werden die von Ihnen festgelegten Einstellungen verwendet. Um ein Katalogelement in ein BSP einzubeziehen, müssen Sie die Datei *Platform.bib* des BSPs bearbeiten und eine auf Ihren Einstellungen basierende Bedingungsanweisung hinzufügen. Sie können eine Komponente einbeziehen, wenn eine Variable unter Verwendung von *if-else*-Anweisungen definiert oder nicht definiert wird. Beachten Sie, dass Sie möglicherweise den Befehl **Rebuild Current BSP And Subprojects** in Visual Studio im Menü **Build** unter **Advanced Build Commands** ausführen müssen, um die an den bib- und reg-Dateien vorgenommenen Änderungen zu übernehmen. In Kapitel 2 ist der Befehl **Rebuild Current BSP And Subprojects** detailliert beschrieben.

Um eine C/C++-Direktive basierend auf einer Umgebungsvariablen festzulegen, die Sie in den Eigenschaften des Katalogelements angegeben haben, verwenden Sie eine Bedingungsanweisung in der Sources-Datei, die auf der Variablen basiert, und fügen Sie einen **CDEFINES**-Eintrag hinzu. Vermeiden Sie das Festlegen von C/C++-Builddirektiven, die auf Katalogelementeigenschaften basieren, da diese das Verteilen einer Binärversion des BSP erschweren.

Außerdem können Sie mit Bedingungsanweisungen die Einträge in der Systemregistrierung ändern. Sie müssen die reg-Dateien nur bearbeiten, um bestimmte Registrierungsdateien für die neue Komponente einzubeziehen oder auszuschließen.

Exportieren eines Katalogelements aus dem Katalog

Das direkte Klonen wird von einigen Katalogelementen nicht unterstützt. Um diese Komponenten zu klonen, müssen Sie eine neue Katalogdatei (wenn Sie unter dem *3rdParty*-Ordner einen Eintrag erstellen) oder einen Eintrag in der Katalogdatei eines BSP erstellen. Überprüfen Sie in jedem Fall, ob die ursprünglichen Werte für alle SYSGEN-Variablen und zusätzlichen Variablen beibehalten werden. Vergessen Sie nicht, die ID zu ändern, da jedes Element im Katalog eine eindeutige ID erfordert.

Abhängigkeiten von Katalogkomponenten

Der Katalog in Platform Builder für Windows Embedded CE 6.0 R2 unterstützt Komponentenabhängigkeiten. Um anzugeben, dass eine Komponente von einer anderen Komponente abhängig ist, müssen Sie das Feld **SYSGEN** oder **Additional Variables** für die Komponente des Katalogelements festlegen und diesen Wert als zusätzliche Umgebungsvariable in die abhängige Komponente einbeziehen. Wenn Ihr BSP beispielsweise Katalogkomponenten für einen Bildschirmtreiber und einen Hintergrundbeleuchtungs-Treiber umfasst, können Sie das Feld **Additional Variables** für den Bildschirmtreiber auf **BSP_DISPLAY** und für den Hintergrundbeleuchtungs-Treiber auf **BSP_BACKLIGHT** festlegen. Wenn Sie möchten, dass der Bildschirmtreiber vom Hintergrundbeleuchtungs-Treiber abhängig ist, bearbeiten Sie den Katalogeintrag für **BSP_DISPLAY** im Catalog Editor und fügen Sie **BSP_BACKLIGHT** zu den zusätzlichen Umgebungsvariablen hinzu. Wenn Sie anschließend den Bildschirmtreiber in ein OS Design einbeziehen, fügt der Platform Builder den Hintergrundbeleuchtungs-Treiber automatisch hinzu. Das Kontrollkästchen für den Hintergrundbeleuchtungs-Treiber wird in der **Catalog Items View** mit einem grünen Quadrat angezeigt, um die Abhängigkeit vom Bildschirmtreiber anzugeben.

Zusammenfassung

Platform Builder für Windows Embedded CE 6.0 R2 umfasst ein dateibasiertes Katalogsystem, das Sie für Ihre Katalogelemente verwenden können, indem Sie diese in separaten Katalogdateien im Verzeichnis *Platform* oder *3rdParty* in der Verzeichnisstruktur *%_WINCEROOT%* speichern. Die Katalogdateien haben das Dateiformat XML und die Dateierweiterung pbcxml. Platform Builder listet die pbcxml-Dateien automatisch auf, wenn Sie Visual Studio starten oder die **Catalog Items View** im Solution Explorer aktualisieren. Um ein neues Katalogelement zum Windows Embedded CE-Katalog hinzuzufügen, können Sie eine neue Katalogdatei verwenden oder eine Kopie eines vorhandenen Katalogelements erstellen und den Dateiinhalt anschließend mit dem Catalog Editor bearbeiten. Sie müssen die pbcxml-Dateien nicht mit einem Texteditor, beispielsweise Notepad, bearbeiten, da alle Einstellung direkt im Platform Builder verfügbar sind. Unter anderem können Sie SYSGEN und zusätzliche Variablen für bedingte C/C++-Builddirektiven, Registrierungsänderungen und Abhängigkeitsdefinitionen angeben.

Lektion 5: Generieren eines Software Development Kits

Entwickler, die Anwendungen für ein Zielgerät erstellen, benötigen ein Software Development Kit (SDK). Ein SDK entspricht Ihrem OS Design, damit der Entwickler nur die Features verwendet, die tatsächlich verfügbar sind. Das SDK umfasst im OS Design vorhandene Features, um zu verhindern, dass der Anwendungsentwickler versehentlich Code erstellt, der aufgrund eines nicht unterstützten API nicht ausgeführt werden kann.

Nach Abschluss dieser Lektion können Sie:

- Den Zweck eines SDK identifizieren.
- Ein SDK generieren.
- Die SDK-Dateien auf der Festplatte auffinden.
- Ein SDK verwenden.

Veranschlagte Zeit für die Lektion: 20 Minuten.

Software Development Kit - Übersicht

Um gültige Anwendungen für das OS Design zu erstellen und zu kompilieren, muss der Entwickler die erforderlichen Headerdateien einbeziehen und mit den entsprechenden Bibliotheken im Entwicklungsprojekt verknüpfen. Sie müssen sicherstellen, dass das SDK für Ihr OS Design alle erforderlichen Headerdateien und Bibliotheken umfasst, einschließlich der Header und Bibliotheken für die benutzerdefinierten Komponenten für Anwendungsentwickler. Platform Builder für Windows Embedded CE 6.0 R2 ermöglicht das Erstellen von SDKs für OS Designs, indem alle erforderlichen Headerdateien und Bibliotheken exportiert werden.

Generieren eines SDK

Benutzerdefinierte SDKs werden normalweise vom Ersteller des OS Designs generiert und bereitgestellt. Platform Builder umfasst zu diesem Zweck ein SDK-Exportfeature. Das SDK-Exportfeature erstellt das angepasste SDK für das OS Design zusammen mit einer msi-Datei für den SDK Setup Wizard.

Konfigurieren und Generieren eines SDK

Um ein SDK mit dem SDK-Exportfeature im Platform Builder zu erstellen und zu konfigurieren, führen Sie folgende Schritte aus:

1. Konfigurieren Sie das OS Design und erstellen Sie dieses mindestens einmal in der Release-Konfiguration.
2. Öffnen Sie den Solution Explorer, klicken Sie mit der rechten Maustaste auf **SDKs** und wählen Sie **Add New** auf, um das Dialogfeld **SDK Property Pages** anzuzeigen.
3. Konfigurieren Sie im Dialogfeld **SDK Property Pages** die allgemeinen Eigenschaften des SDK und definieren Sie den **MSI Folder Path**, **MSI File Name** und **Locale** (siehe Abbildung 1.7). Sie können auch einige benutzerdefinierte Einstellungen festlegen.
4. Um zusätzliche Dateien einzubeziehen, wählen Sie den Knoten **Additional Folders** im Dialogfeld **SDK Property Pages** aus.
5. Klicken Sie auf **OK**.

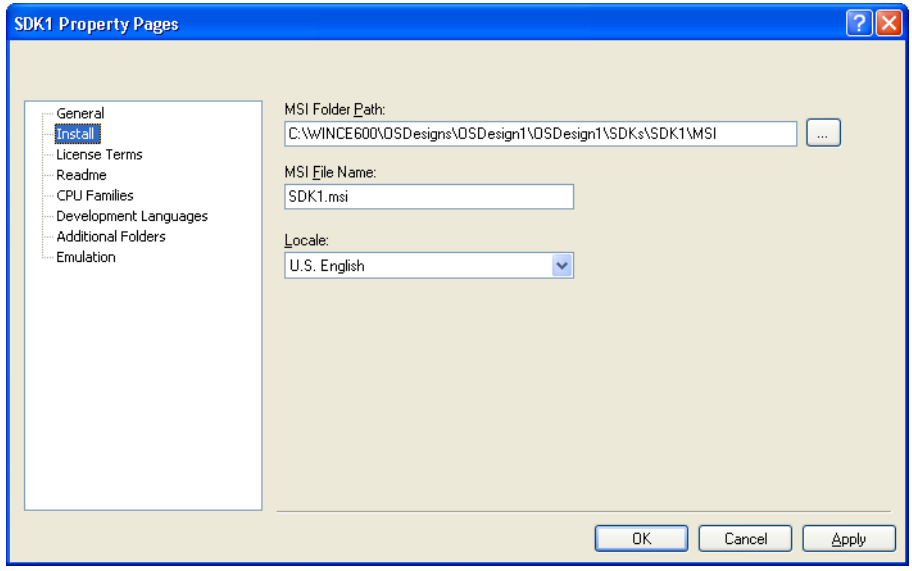


Abbildung 1.7 Dialogfeld **SDK Property Pages**

Hinzufügen neuer Dateien zu einem SDK

Sie können Dateien manuell zu einem SDK hinzufügen, indem Sie die Option **Additional Folders** im Dialogfeld **SDK Property Pages** aktivieren oder die Dateien in das SDK-Verzeichnis für das OS Design kopieren (normalerweise in *\OSDesigns\<Solution Name>\<OS Design Name>\WinCE600\<Platform Name>\SDK*). Sie können diesen Prozess auch unter Verwendung von bat- und

Sources-Dateien automatisieren, damit das Buildmodul die neueste Version der Dateien in das SDK kopiert, wenn Sie einen Build ausführen.

Kopieren Sie die Dateien in folgende SDK-Unterverzeichnisse:

- **Inc** Enthält die Headerdateien für das SDK.
- **Lib**\<*Processor Type*>\<*Build Type*> Enthält die Bibliotheken für das SDK.

Installieren eines SDK

Nachdem der SDK-Buildprozess abgeschlossen ist, befindet sich die msi-Datei im *SDK*-Unterverzeichnis im OS Design-Ordner. Der Pfad ist normalerweise *%_WINCEROOT%\OSDesigns\<Solution Name>\<OS Design Name>\SDKs\SDK1\MSI\<SDK Name>.msi*. Sie können die msi-Datei entsprechend Ihren Lizenzvereinbarungen für Platform Builder und denen für die Komponenten von Drittanbietern weitergeben.

Sie können das MSI-Paket nun auf einem Visual Studio 2005-Computer installieren und zum Entwickeln von Windows Embedded CE-Anwendungen für das Zielgerät verwenden. Auf einem Computer, auf dem das SDK installiert ist, werden die Dateien unter *%PROGRAMFILES%\Windows Embedded CE Tools\WCE600* gespeichert.

Zusammenfassung

Windows Embedded CE 6.0 R2 ist ein Betriebssystem mit Komponenten, was bedeutet, dass die Anwendungsentwickler ein angepasstes SDK benötigen, das dem OS Design entspricht, um Anwendungen zu entwickeln, die auf dem Zielgerät ausgeführt werden können. Das angepasste SDK sollte nicht nur die erforderlichen Windows Embedded CE-Komponenten umfassen, sondern auch die Header und Bibliotheken für die benutzerdefinierten Komponenten, die Sie in das OS Design einbezogen haben, um Probleme aufgrund fehlender Dateien oder Bibliotheken zu vermeiden. Platform Builder umfasst ein SDK-Exportfeature zum Generieren von SDKs und zum Erstellen eines MSI-Pakets für die SDK-Bereitstellung auf Entwicklungscomputern mit Hilfe eines SDK Setup Wizards.

Lab 1: Erstellen und Konfigurieren eines OS Designs

In diesen Lab erstellen Sie ein OS Design und passen dieses an, indem Sie weitere Komponenten aus dem Katalog hinzufügen. Es ist wichtig, dass Sie alle Schritte in diesem Lab ausführen, da dieses Lab die Grundlage für die übrigen Kapitel in diesem Microsoft Windows Embedded CE 6.0 R2 Exam Preparation Kit bildet.

HINWEIS Schrittweise Anleitungen

Um die Schritte in diesem Lab erfolgreich auszuführen, lesen Sie das Dokument *Detailed Step-by-Step Instructions for Lab 1"* im Begleitmaterial.

▶ Erstellen eines OS Designs

1. Wählen Sie in Visual Studio 2005 mit Platform Builder für Windows Embedded CE 6.0 R2 das Menü **File**, das Untermenü **New** und die Option **Project** aus, um ein neues OS Design-Projekt zu erstellen.
2. Verwenden Sie den Standardnamen für das OS Design (***OSDesign1***).
3. Visual Studio startet den **Windows Embedded CE 6.0 OS Design Wizard**.
4. Aktivieren Sie das Kontrollkästchen für den **Device Emulator: ARMV4I** in der BSP-Liste, und klicken Sie auf **Next**.
5. Wählen Sie in der Liste **Available Design Templates** die Option **PDA Device** aus. Wählen Sie in der Liste der verfügbaren Designvarianten die Option **Mobile Handheld** aus.
6. Deaktivieren Sie auf der nächsten Assistentenseite die Optionen **.NET Compact Framework 2.0** und **ActiveSync** (siehe Abbildung 1.8). Die Kontrollkästchen **Internet Browser** und **Quarter VGA Resources - Portrait Mode** müssen aktiviert sein.
7. Deaktivieren Sie auf der Seite **Networking Communications** die Optionen **TCP/IPv6 Support** und **Personal Area Network (PAN)**, um die Unterstützung für Bluetooth und Infrared Data Association (IrDA) zu deaktivieren. Die Option **Local Area Network (LAN)** muss aktiviert sein.
8. Klicken Sie auf **Finish**, um den **Windows Embedded CE 6.0 OS Design Wizard** zu beenden. Anschließend öffnet Visual Studio das OS Design-Projekt. Im Solution Explorer wird das neue OS Design-Projekt unter dem Container **Solution** angezeigt.

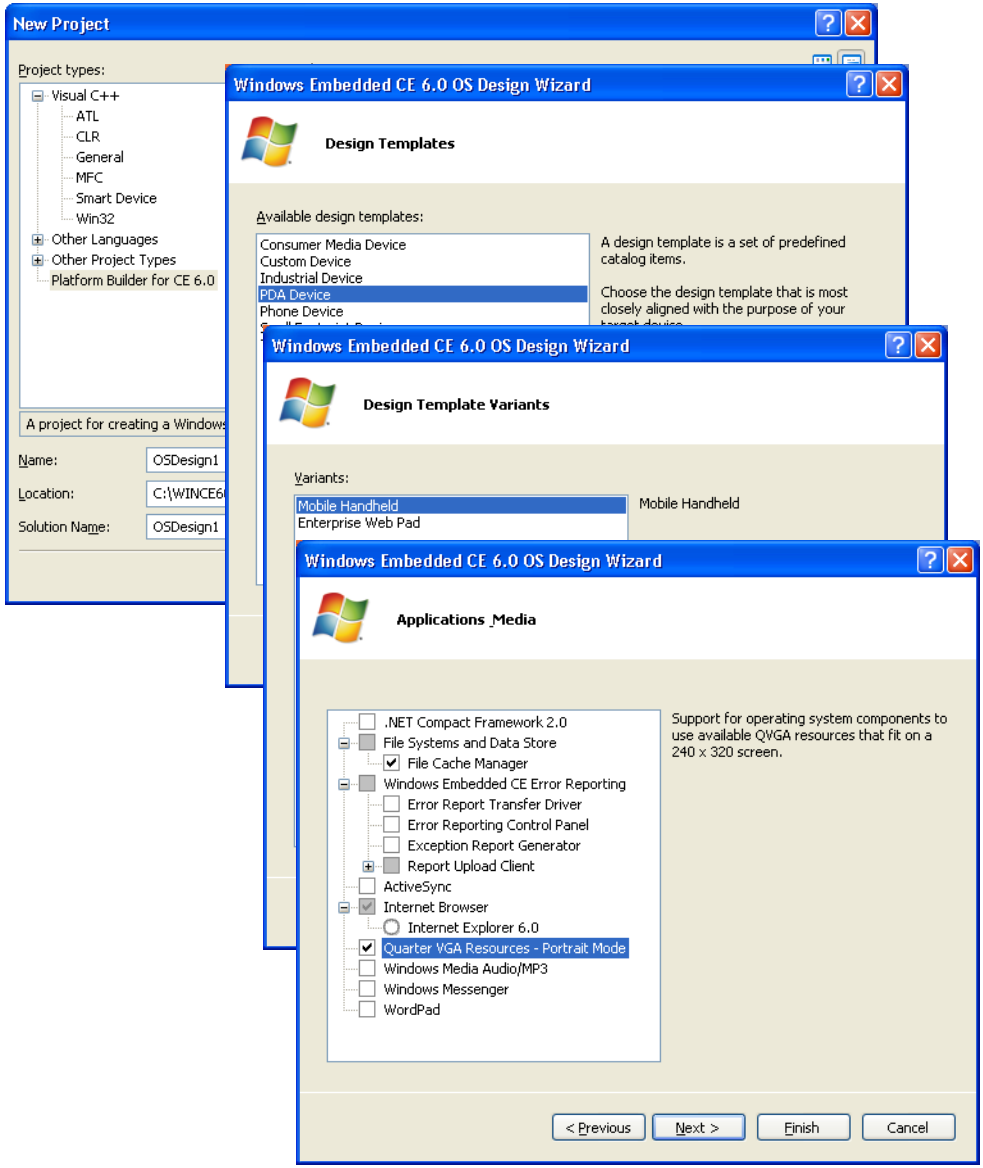


Abbildung 1.8 Erstellen eines OS Designs für ein PDA-Gerät

HINWEIS Nachträgliche Änderungen des OS Designs

Der OS Design Wizard erstellt die ursprüngliche Konfiguration für das OS Design. Nachdem der OS Design Wizard abgeschlossen ist, können Sie Änderungen am OS Design vornehmen.

▶ Überprüfen des OS-Katalogs

1. Klicken Sie in Visual Studio im Solution Explorer auf die Registerkarte **Catalog Items View**.
2. Erweitern Sie die Containerknoten, um die aktivierten Kontrollkästchen und Symbole im Katalog zu analysieren. Die Kontrollkästchen mit einem grünen Häkchen zeigen die Elemente an, die als Bestandteil des OS Designs hinzugefügt wurden. Die Kontrollkästchen mit einem grünen Quadrat zeigen die Elemente an, die aufgrund von Abhängigkeiten zum OS Design hinzugefügt wurden. Nicht aktivierte Kontrollkästchen zeigen die Elemente an, die nicht in das OS Design einbezogen wurden, aber verfügbar sind.
3. Suchen Sie ein Katalogelement mit einem grünen Quadrat im Kontrollkästchen.
4. Klicken Sie mit der rechten Maustaste auf das Katalogelement und wählen Sie **Reasons For Inclusion Of Item** aus. Im Dialogfeld **Remove Dependent Catalog Item** werden die Katalogelemente angezeigt, die das Einbeziehen des ausgewählten Elements in das OS Design verursacht haben (siehe Abbildung 1.9).
5. Erweitern Sie den Knoten **Core OS | CEBASE | Applications – End User | Active Sync** im Katalog.

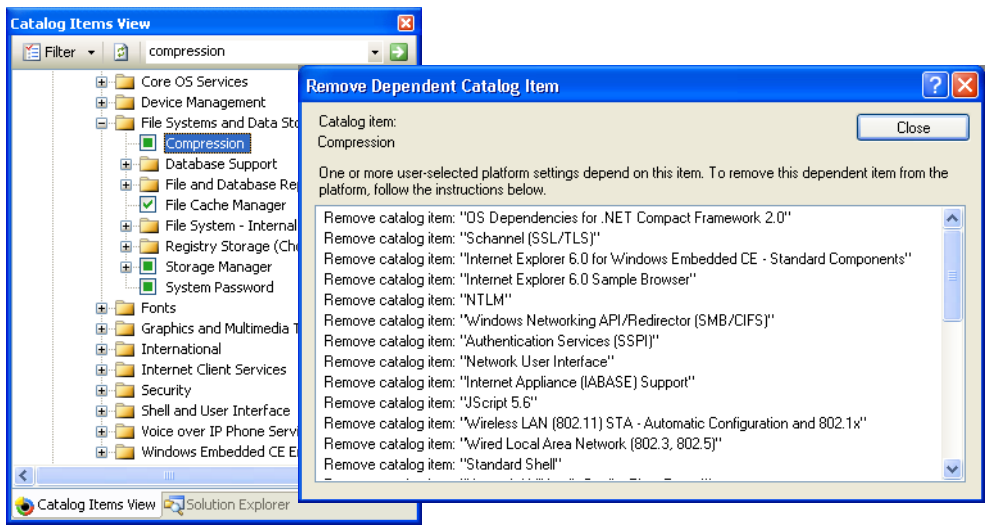


Abbildung 1.9 Gründe zum Einbeziehen eines Katalogelements als Abhängigkeit

6. Klicken Sie mit der rechten Maustaste auf **ActiveSync System Cpl Items** und wählen Sie **Display In Solution View** aus. Im Solution Explorer wird das Teilprojekt angezeigt, das die ActiveSync-Komponente umfasst. In Windows Embedded CE 6.0 können Sie mit dieser Methode durch den Quellcode navigieren.

▶ Hinzufügen der Unterstützung für das Katalogelement Internet Explorer 6.0 Sample Browser

1. Wählen Sie die Registerkarte **Catalog Items View** aus, um den OS Design-Katalog anzuzeigen. Stellen Sie sicher, dass die Filteroption auf **All Catalog Items In Catalog** festgelegt ist.
2. Geben Sie **Internet Explorer 6.0 Sample** im Suchfeld rechts neben **Catalog Item View Filter** ein und drücken Sie die Eingabetaste oder klicken Sie auf den grünen Pfeil.
3. Überprüfen Sie, ob das Katalogelement **Internet Explorer 6.0 Sample Browser** gefunden wird. Aktivieren Sie das entsprechende Kontrollkästchen, um das Katalogelement in das OS Design einzubeziehen (siehe Abbildung 1.10).

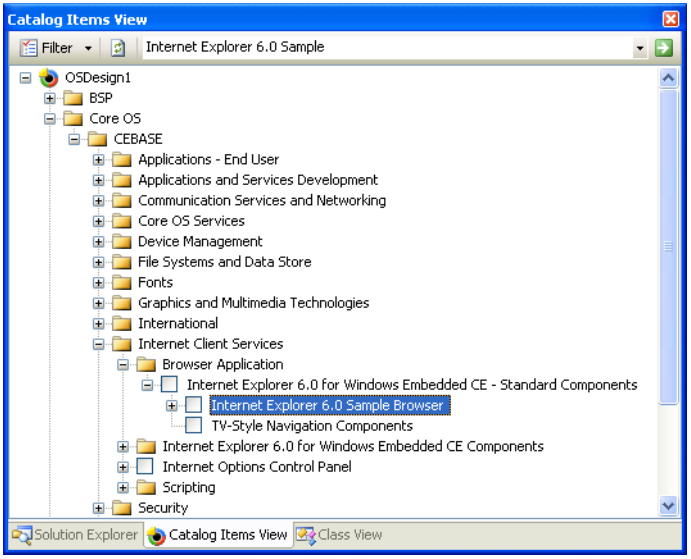


Abbildung 1.10 Einbeziehen des Katalogelements **Internet Explorer 6.0 Sample Browser** in das OS Design

▶ Hinzufügen der Unterstützung für die Entwicklung von verwaltetem Code zum OS Design

1. Geben Sie ***ipconfig*** im Suchfeld ein und drücken Sie die Eingabetaste.
2. Überprüfen Sie, ob **Network Utilities (IpConfig, Ping, Route)** gefunden wird.
3. Fügen Sie das Katalogelement **Network Utilities (IpConfig, Ping, Route)** zum OS Design hinzu, indem Sie das entsprechende Kontrollkästchen aktivieren.
4. Geben Sie ***wceload*** im Suchfeld ein und drücken Sie die Eingabetaste.
5. Überprüfen Sie, ob das Katalogelement **CAB File Installer/Uninstaller** gefunden wird. Dieses Katalogelement wird gefunden, da die SYSGEN-Variable den Wert **wceload** hat.
6. Fügen Sie das Katalogelement **Cab File Installer/Uninstaller** zum OS Design hinzu.
7. Verwenden Sie das Suchfeature, um den Container **OS Dependencies for .NET Compact Framework 2.0** anzuzeigen. Überprüfen Sie, ob das Katalogelement **OS Dependencies for .NET Compact Framework 2.0** im OS Design einbezogen ist (siehe Abbildung 1.11).

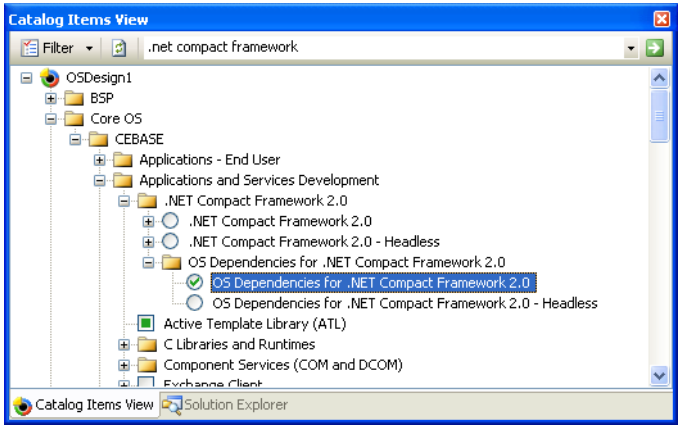


Abbildung 1.11 Hinzufügen des Katalogelements **OS Dependencies for .NET Compact Framework Catalog Item** zum OS Design

HINWEIS Headless .NET Compact Framework

Diese Kategorie umfasst zwei separate Komponenten. Wählen Sie die Komponente ohne die Angabe Headless Modifier aus, da diese Version für Geräte ohne Bildschirm bestimmt ist.

Lernzielkontrolle

Um Microsoft Windows Embedded CE 6.0 R2 auf einem Zielgerät bereitzustellen, müssen Sie ein OS Design erstellen, das die erforderlichen Betriebssystemkomponenten, Features, Treiber und Konfigurationseinstellungen umfasst. Anschließend können Sie mit dem Builder das entsprechende Run-Time Image für die Bereitstellung erstellen. Sie müssen folgende Aufgaben ausführen, um ein angepasstes OS Design zu erstellen, das Ihre Anforderungen erfüllt:

- Ein OS Design-Projekt in Visual Studio mit dem OS Design Wizard erstellen.
- Komponenten manuell oder über Abhängigkeiten zum OS hinzufügen bzw. aus dem OS entfernen.
- Umgebungsvariablen oder SYSGEN-Variablen über den Catalog Editor festlegen.
- Die regionalen Einstellungen für die Lokalisierung des OS Designs konfigurieren.
- Katalogelemente automatisch durch Klicken auf **Clone Catalog Item** oder manuell mit dem Tool Sysgen Capture klonen.
- Ein angepasstes SDK für das OS Design exportieren, um die Anwendungsentwicklung für das Zielgerät zu unterstützen.

Schlüsselbegriffe

Kennen Sie die Bedeutung der folgenden wichtigen Begriffe? Sie können Ihre Antworten überprüfen, wenn Sie die Begriffe im Glossar am Ende dieses Buches nachschlagen.

- OS Design
- Komponente
- SYSGEN-Variable
- Umgebungsvariable
- Software Development Kit

Empfohlene Vorgehensweise

Um die in diesem Kapitel behandelten Prüfungsschwerpunkte erfolgreich zu beherrschen, sollten Sie die folgenden Aufgaben durcharbeiten.

Erstellen eines benutzerdefinierten OS Designs

Erstellen Sie mit dem OS Design Wizard ein OS Design basierend auf dem Geräteemulator: ARMV4I BSP und der Designvorlage **Custom Device**. Führen Sie nach der Erstellung des OS Designs folgende Aufgaben aus:

- **Fügen Sie .Net Compact Framework 2.0 hinzu** Fügen Sie dieses Katalogelement unter Verwendung des Suchfeatures in der **Catalog Items View** hinzu.
- **Lokalisieren Sie Ihr Run-Time Image** Zeigen Sie die OS Design **Property Pages** an und lokalisieren Sie das OS Design in Deutsch.

Generieren und Testen eines SDK

Führen Sie basierend auf dem in Lab 1 generierten OS Design folgende Aufgaben aus:

- **Erstellen und generieren Sie das binäre Image** Erstellen und generieren Sie das Binärimage für das OS Design in der Release-Buildkonfiguration.
- **Erstellen und installieren Sie das SDK** Überprüfen Sie, ob der Buildprozess erfolgreich abgeschlossen wurde. Erstellen Sie anschließend ein neues SDK und installieren Sie dieses auf einem Anwendungsentwicklungscomputer.
- **Verwenden Sie das SDK** Verwenden Sie eine andere Instanz von Visual Studio und erstellen Sie eine **Win32 Smart Device**-Anwendung. Verwenden Sie Ihr angepasstes SDK als Referenz für das Projekt und erstellen Sie die Anwendung.

Kapitel 2

Erstellen und Bereitstellen eines Run-Time Images

Der Microsoft® Windows® Embedded CE 6.0 R2-Buildprozess ist äußerst komplex. Dieser Prozess umfasst mehrere Phasen und hängt von verschiedenen Tools ab, um die Windows Embedded CE-Buildumgebung zu initialisieren, den Quellcode zu kompilieren, Module und Dateien in ein Versionsverzeichnis zu kopieren sowie das Run-Time Image zu erstellen. Diese Prozesse werden mit Batchdateien und Buildtools, beispielsweise Sysgen (*Sysgen.bat*) und Make Binary Image (*Makeimg.exe*), automatisiert. Sie können diese Tools direkt über die Befehlszeile ausführen oder den Buildprozess in Microsoft Platform Builder für Windows Embedded CE 6.0 R2 starten. Die Platform Builder IDE (Integrated Development Environment) stützt sich auf die gleichen Prozesse und Tools. Sie sollten unbedingt mit dem Buildprozess und dem Verfahren zum Bereitstellen von Run-Time Images vertraut sein, wenn Sie Run-Time Images effizient erstellen, Buildprobleme beheben oder BSPs (Board Support Packages) und Teilprojekte auf einem Zielgerät bereitstellen möchten.

Prüfungsziele in diesem Kapitel

- Erstellen eines Run-Time Images
- Analysieren der Buildergebnisse und Builddateien
- Bereitstellen eines Run-Time-Images auf einem Zielgerät

Bevor Sie beginnen

- Damit Sie die Lektionen in diesem Kapitel durcharbeiten können, benötigen Sie:
- Kenntnisse in allen Aspekten des OS Designs, einschließlich der Katalogelemente und dem Konfigurieren von Umgebungsvariablen und SYSGEN-Variablen, die in Kapitel 1 erklärt wurden.
- Mindestens Grundkenntnisse in der Windows Embedded CE-Softwareentwicklung, einschließlich dem Kompilieren und Linken von Quellcode.
- Einen Entwicklungscomputer, auf dem Microsoft Visual Studio® 2005 Service Pack 1 und Platform Builder für Windows Embedded CE 6.0 installiert ist.

Lektion 1: Erstellen eines Run-Time Images

Der Windows Embedded CE-Buildprozess ist der letzte Schritt im Entwicklungszyklus des Run-Time Images. Platform Builder kompiliert alle Komponenten, einschließlich der Teilprojekte und des BSPs, basierend auf den im OS Design definierten Einstellungen, und erstellt anschließend das Run-Time Image, das Sie auf das Zielgerät herunterladen können. Der Buildprozess umfasst mehrere Phasen, die Sie mit Batchdateien automatisieren können. Sie müssen mit den Buildphasen und den Buildtools vertraut sein, um die Buildoptionen korrekt zu konfigurieren, das Run-Time Image effizient zu erstellen und Buildprobleme zu beheben.

Nach Abschluss dieser Lektion können Sie:

- Den Buildprozess verstehen.
- Buildprobleme analysieren und beheben.
- Ein Run-Time-Image auf einem Zielgerät bereitstellen.

Veranschlagte Zeit für die Lektion: 40 Minuten.

Übersicht des Buildprozesses

Der Windows Embedded CE-Buildprozess umfasst vier Hauptphasen, die der Reihe nach ausgeführt werden (siehe Abbildung 2.1). Sie können die einzelnen Buildphasen jedoch unabhängig voneinander ausführen, wenn Ihnen die für die jeweilige Phase benötigten Tools bekannt sind. Indem Sie die Buildtools selektiv verwenden, können Sie die erforderlichen Schritte gezielt ausführen, um Zeit zu sparen und produktiver zu arbeiten.

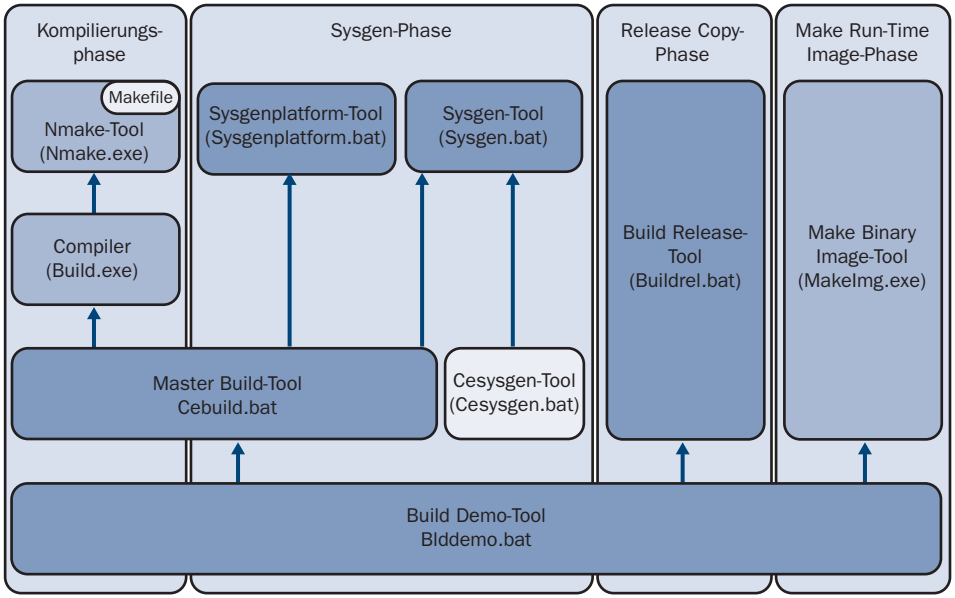


Abbildung 2.1 Buildphasen und Buildtools

Der Buildprozess besteht aus den folgenden Hauptphasen:

- **Kompilierungsphase** Der Compiler und Linker verwenden den Quellcode und die Ressourcendateien, um ausführbare Dateien (.exe), statische Bibliotheken (.lib), DLL-Dateien und binäre Ressourcendateien (.res) basierend auf den ausgewählten Gebietsschemas zu generieren. Beispielsweise kompiliert das Buildsystem während dieser Phase den Quellcode in lib-Dateien in den Ordnern Private und Public. Dieser Prozess kann mehrere Stunden dauern. Glücklicherweise ist es jedoch selten erforderlich, dass diese Komponenten erneut erstellt werden müssen, da die Binärdateien von Microsoft bereitgestellt werden. Sie sollten den Quellcode in den Ordnern *Private* und *Public* jedoch nicht ändern.
- **Sysgen-Phase** Das Buildsystem aktiviert oder deaktiviert die SYSGEN-Variablen basierend auf den Katalogelementen und Abhängigkeitsstrukturen im OS Design, filtert die Headerdateien, erstellt Importbibliotheken für die Software Development Kits (SDKs), erstellt Konfigurationsdateien für das Run-Time Image und erstellt das BSP entsprechend den Quelldateien im *Platform*-Verzeichnis.

- **Buildphase** Das Buildsystem verarbeitet die Quelldateien des BSPs und der Anwendungen unter Verwendung der während der Sysgen-Phase generierten Dateien. Zu diesem Zeitpunkt werden die Hardwaretreiber und der OAL (OEM Adaptation Layer) erstellt. Obwohl die Buildprozesse automatisch während der Sysgen-Phase ausgeführt werden, sollten Sie beachten, dass Sie das BSP und die Teilprojekte ohne das Sysgen-Tool erneut erstellen können, wenn Sie nur das BSP und die Teilprojekte ändern.
- **Release Copy-Phase** Das Buildsystem kopiert alle Dateien, die zum Erstellen des Run-Time Images erforderlich sind, in das Releaseverzeichnis des OS Designs. Diese Dateien umfassen die lib-, dll- und exe-Dateien, die während der Kompilierungs- und Sysgen-Phase erstellt werden sowie Binary Image Builder- (.bib) und Registrierungsdateien (.reg). Das Buildsystem überspringt diese Phase, wenn die Header und Bibliotheken aktuell sind.
- **Make Run-Time Image-Phase** Das Buildsystem kopiert die projektspezifischen Dateien (*Project.bib*, *Project.dat*, *Project.db* und *Project.reg*) in das Releaseverzeichnis und erstellt mittels dieser Dateien ein Run-Time Image. Die auf Umgebungsvariablen basierenden Direktiven in den reg- und bib-Dateien legen fest, welche Katalogelemente in das Run-Time Image einbezogen werden. Sie können das Run-Time Image, das normalerweise den Namen *Nk.bin* hat, herunterladen und auf dem Zielgerät ausführen.

Erstellen von Run-Time Images in Visual Studio

Während der Installation von Windows Embedded CE 6.0 R2 auf dem Entwicklungscomputer wird Platform Builder mit Visual Studio 2005 integriert und das Menü **Build** erweitert, damit Sie den Buildprozess direkt in der Visual Studio IDE steuern können. In Abbildung 2.2 sind die Platform Builder-Befehle dargestellt, die im Menü **Build** verfügbar sind, wenn Sie den OS **OSDesign**-Knoten im Solution Explorer auswählen.

Sie können die Platform Builder-Befehle im Menü **Build** verwenden, um ausgewählte oder mehrere Buildschritte auszuführen, die sich über mehrere Buildphasen erstrecken. Beispielsweise können Sie den Befehl **Copy Files To Release Directory** ausführen, um sicherzustellen, dass das Buildsystem die aktualisierten bib- und reg-Dateien auch dann in das Releaseverzeichnis kopiert, wenn die Headerdateien und Bibliotheken nicht geändert wurden. Ansonsten überspringt das Buildsystem die Release Copy-Phase und die an der bib- oder reg-Datei vorgenommenen Änderungen werden nicht in das Run-Time Image übernommen.

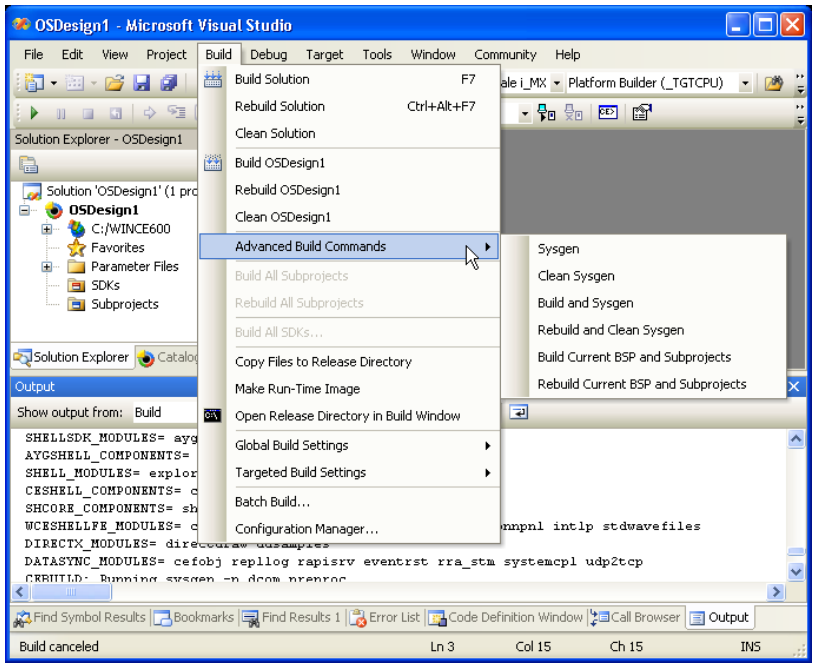


Abbildung 2.2 Windows Embedded CE-Buildbefehle in Visual Studio 2005

In Tabelle 2.1 sind die Windows Embedded CE-Buildbefehle zusammen mit einer Beschreibung aufgeführt.

Tabelle 2.1 Windows Embedded CE-Buildbefehle

Menüoption	Beschreibung
Build Solution	Entspricht dem Befehl Sysgen im Untermenü Advanced Build Commands.
Rebuild Solution	Entspricht dem Befehl Clean Sysgen im Untermenü Advanced Build Commands.
Clean Solution	Bereinigt das Releaseverzeichnis, indem alle temporären Dateien gelöscht werden.
Build <*OS Design Name*>	Hilfreich für Lösungen, die mehrere OS Designs umfassen. In Lösungen mit einem einzigen OS Design entsprechen diese Optionen den Befehlen Build Solution, Rebuild Solution und Clean Solution.
Rebuild <*OS Design Name*>	
Clean <*OS Design Name*>	

Tabelle 2.1 Windows Embedded CE-Buildbefehle (Fortsetzung)

Menüoption		Beschreibung
Advanced Build Commands	Sysgen	Führt das Sysgen-Tool aus und linkt die lib-Dateien in den Ordnern *Public* und *Private*, um die Dateien für das Run-Time Image zu erstellen. Die Dateien verbleiben im *WinCE*-Ordner des OS Designs. Abhängig von den globalen Buildeinstellungen, kann der Buildprozess automatisch mit der Release Copy- und Make Run-Time Image-Phase fortfahren.
	Clean Sysgen	Bereinigt die während den vorherigen Builds erstellten temporären Dateien, bevor das Tool Sysgen ausgeführt wird. Verwenden Sie diese Option, wenn Sie Dateien oder Katalogelemente nach einer Sysgen-Sitzung hinzugefügt bzw. entfernt haben, um das Risiko von Buildfehlern zu verringern.
	Build And Sysgen	Kompiliert den gesamten Inhalt der Ordner *Public* und *Private* und linkt die Dateien anschließend basierend auf den Einstellungen im OS Design. Dieser Prozess dauert mehrere Stunden und ist nur erforderlich, wenn der Inhalt des Ordners *Public* geändert wurde. Sie sollten diese Option nur verwenden, wenn Sie die Windows Embedded CE-Codebasis geändert haben.
	Rebuild And Sysgen	Bereinigt die während vorherigen Builds erstellten temporären Dateien in den Ordnern *Public* und *Private* und führt anschließend die Build- und Sysgen-Vorgänge aus. Sie sollten diese Option nicht verwenden.

Tabelle 2.1 Windows Embedded CE-Buildbefehle (Fortsetzung)

Menüoption		Beschreibung
Advanced Build Commands (continued)	Build Current BSP And Subprojects	Erstellt die Dateien im Verzeichnis für das aktuelle BSP und die Teilprojekte im OS Design und führt anschließend das Tool Sysgen aus. Beachten Sie, dass diese Option zusätzliche BSPs außerhalb des aktuellen OS Designs erstellt. Stellen Sie deshalb sicher, dass Ihre BSPs miteinander kompatibel sind oder entfernen Sie nicht verwendete BSPs.
	Rebuild Current BSP And Subprojects	Bereinigt die während vorherigen Builds erstellten temporären Dateien und führt anschließend die Build Current BSP And Subprojects-Vorgänge aus.
Build All Subprojects		Kompiliert und linkt alle Teilprojekte und überspringt aktuelle Dateien.
Rebuild All Subprojects		Bereinigt, kompiliert und linkt die Teilprojekte.
Build All SDKs		Erstellt alle SDKs im Projekt und die entsprechenden MSI-Pakete (Microsoft Installer). Da es normalweise nicht erforderlich ist, Debugversionen von MSI-Paketen zu erstellen, verwenden Sie diese Option ausschließlich in Releasebuildkonfigurationen.
Copy Files To Release Directory		Kopiert die während der Kompilierungs- und Sysgen-Phase für das BSP generierten Dateien und andere Komponenten in das Releaseverzeichnis, um diese Dateien in das Run-Time Image einzubeziehen.
Make Run-Time Image		Erstellt das Run-Time Image mit allen Dateien im Releaseverzeichnis. Anschließend können Sie das Run-Time Image auf das Zielgerät herunterladen.

Tabelle 2.1 Windows Embedded CE-Buildbefehle (Fortsetzung)

Menüoption		Beschreibung
Open Release Directory In Build Window		Öffnet eine Eingabeaufforderung, wechselt zum Releaseverzeichnis und legt alle erforderlichen Umgebungsvariablen fest, um die Batchdateien und Buildtools manuell auszuführen. Die Standardbefehlseingabe initialisiert die Entwicklungsumgebung nicht, um die Buildtools auszuführen.
Global Build Settings	Copy Files To Release Directory After Build	Aktiviert oder deaktiviert das automatische Fortsetzen mit der Release Copy-Phase für alle Befehle.
	Make Run-Time Image After Build	Aktiviert oder deaktiviert das automatische Fortsetzen mit der Make Run-time Image-Phase nach einem Buildvorgang.
Targeted Build Settings	Make Run-Time Image After Build	Aktiviert oder deaktiviert die Make Run-time Image-Phase.
Batch Build		Ermöglicht Ihnen, mehrere Builds nacheinander auszuführen.
Configuration Manager		Ermöglicht das Hinzufügen oder Entfernen von Buildkonfigurationen.

Das Untermenü **Advanced Build Commands** bietet Zugriff auf mehrere Platform Builder-spezifische Buildbefehle, die Sie häufig verwenden werden. Beispielsweise müssen Sie den **Sysgen**- oder **Clean Sysgen**-Befehl ausführen, wenn Sie Katalogkomponenten zum OS Design hinzufügen oder aus dem Design entfernen möchten, um binäre Versionen für das Run-Time Image zu erstellen. Ausnahmen sind Komponenten, die die SYSGEN-Variablen nicht ändern, beispielsweise die Komponenten im Ordner *ThirdParty*. Sie müssen **Sysgen** oder **Clean Sysgen** nicht ausführen, wenn Sie diese Elemente aktiviert oder deaktiviert haben. Im Anschluss an die Sysgen-Phase setzt der Platform Builder den Buildprozess ähnlich wie mit dem Befehl **Build Current BSP And Subprojects** fort.

Sie können den Befehl **Build Current BSP And Subprojects** oder **Rebuild Current BSP And Subprojects** in Visual Studio auswählen, wenn Sie den Quellcode im *Platform*-Verzeichnis und die Teilprojekte des OS Designs kompilieren und linken möchten, und den Code in das Zielverzeichnis *Platform*\<*BSP Name*>*Target* and *Platform*\<*BSP Name*>*Lib* kopieren. Dies ist beispielsweise erforderlich, um den Quellcode im *Platform*-Verzeichnis zu ändern. Abhängig von den Optionen **Copy Files To Release Directory After Build** und **Make Run-Time Image After Building** kopiert der Platform Builder die Dateien in das Releaseverzeichnis und erstellt das Run-Time Image. Sie können diese Schritte auch einzeln über das Menü oder mit den Tools *Buildrel.exe* und *Makeimg.exe* über die Befehlszeile ausführen.

ACHTUNG Clean Sysgen beeinflusst mehrere Buildkonfigurationen

Wenn Sie Clean Sysgen für eine Buildkonfiguration ausführen, müssen Sie diesen Befehl später auch für die anderen Buildkonfigurationen ausführen. Beachten Sie, dass der Befehl Clean Sysgen alle von anderen Buildkonfigurationen und von der aktuellen Konfiguration generierten Dateien löscht.

Erstellen von Run-Time Images über die Befehlszeile

Das Platform Builder für CE6 R2-PlugIn für Visual Studio 2005 ermöglicht den Zugriff auf Batchdateien und Buildtools. Sie können die Batchdateien und Buildtools jedoch auch direkt über die Befehlszeile ausführen. Jeder Befehl in Visual Studio mit Platform Builder entspricht einem bestimmten Buildbefehl (siehe Tabelle 2.2). Verwenden Sie den Befehl **Open Build Window** in Visual Studio, um die Eingabeaufforderung zu öffnen. Die Umgebungsvariablen für die Buildtools werden von der Standardeingabeaufforderung nicht initialisiert. Der Buildprozess schlägt fehl, wenn die erforderlichen Umgebungsvariablen nicht verfügbar sind.

Tabelle 2.2 Buildbefehle und Befehlszeile

Buildbefehl	Befehlszeile
Build	blddemo -q
Rebuild	blddemo clean -q
Sysgen	blddemo -q
Clean Sysgen	blddemo clean -q
Build And Sysgen*	Blddemo

Tabelle 2.2 Buildbefehle und Befehlszeile (Fortsetzung)

Buildbefehl	Befehlszeile
Rebuild And Clean Sysgen*	blddemo clean cleanplat -c
Build Current BSP And Subprojects	blddemo -qbsp
Rebuild Current BSP And Subprojects	blddemo -c -qbsp

* Nicht empfohlen

Inhalt von Windows Embedded CE Run-Time Images

Ein Run-Time Image umfasst alle Elemente und Komponenten, die als Bestandteil des OS Designs auf einem Zielgerät bereitgestellt und ausgeführt werden, beispielsweise Kernelkomponenten, Anwendungen und Konfigurationsdateien (siehe Abbildung 2.3). Die wichtigsten Konfigurationsdateien für Entwickler sind bib-Dateien (Binary Image Builder), Registrierungsdateien (.reg), Datenbankdateien (.db) und Dateisystemdateien (.dat). Diese Dateien legen das Speicherlayout sowie die Initialisierung des Dateisystems und der Systemregistrierung fest. Sie müssen mit diesen Dateitypen vertraut sein. Sie können die reg- und bib-Dateien für ein BSP direkt im OS Design ändern oder ein Teilprojekt erstellen, um benutzerdefinierte Einstellungen zum Run-Time Image hinzuzufügen. Wie in Kapitel 1 bereits erwähnt, ist es normalerweise zwar schneller und praktischer, die reg- und bib-Dateien eines OS Designs direkt zu ändern, aber Teilprojekte unterstützen die Wiederverwendung der Anpassungen in anderen OS Designs.

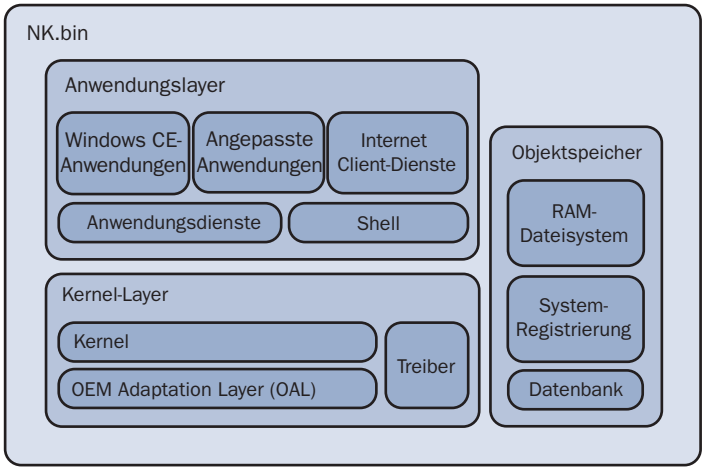


Abbildung 2.3 Inhalt eines Run-Time Images

Binary Image Builder-Dateien

Der Windows Embedded CE-Buildprozess verwendet bib-Dateien, um den Inhalt des Run-Time Images zu generieren und das Speicherlayout des Geräts zu definieren. Während der Make Run-Time Image-Phase ruft das Tool Make Binary Image (*Makeimg.exe*) das Tool File Merge (*Fmerge.exe*) auf, um alle erforderlichen bib-Dateien, beispielsweise *Config.bib* und *Platform.bib* im Ordner *Platform\<BSP Name>\Files*, *Project.bib*, *Common.bib*, und alle bib-Dateien des Teilprojekts in der Datei *Ce.bib* im Releaseverzeichnis zusammenzufassen. Das Tool Make Binary Image ruft anschließend das Tool ROM Image Builder (*Romimage.exe*) auf, um diese Datei zu verarbeiten und festzulegen, welche Binärdateien und Dateien in das Run-Time Image einbezogen werden.

Eine bib-Datei kann folgende Abschnitte enthalten:

- **MEMORY** Definiert die Parameter für das Speicherlayout. Dieser Abschnitt befindet sich normalerweise in der Datei *Config.bib* für das BSP, beispielsweise *C:\Wince600\Platform\DeviceEmulator\Files\Config.bib*.
- **CONFIG** Definiert die Konfigurationsoptionen für *Romimage.exe* zum Anpassen des binären Run-Time Images. Dieser optionale Abschnitt befindet sich normalerweise in der Datei *Config.bib* für das BSP, beispielsweise *C:\Wince600\Platform\DeviceEmulator\Files\Config.bib*.
- **MODULES** Gibt eine Liste der von *Romimage.exe* markierten Dateien aus, die in den RAM geladen werden müssen oder welche direkt aus dem ROM ausgeführt werden (Execute In Place, XIP). Ausschließlich nicht komprimierte Objektmodule können direkt aus dem ROM ausgeführt werden. Sie können

systemeigene ausführbare Dateien in diesem Abschnitt auflisten, aber keine verwalteten (.NET-) Binärdateien, da die CLR (Common Language Runtime) den MISL-Inhalt (Microsoft Intermediate Language) zur Laufzeit in systemeigenen Computercode konvertieren muss.

- **FILES** Verweist auf ausführbare und andere Dateien, die das Betriebssystem zum Ausführen in den RAM laden muss. Geben Sie in diesem Abschnitt die verwalteten (.NET-) Codemodule an.

MEMORY-Abschnitt der BIB-Datei Der MEMORY-Abschnitt in der Datei *Config.bib* definiert die reservierten Speicherbereiche und weist jedem Bereich einen Namen, eine Adresse, eine Größe und einen Typ zu. Ein gutes Beispiel ist der MEMORY-Abschnitt in der Datei *Config.bib* im Device Emulator BSP. Das Device Emulator BSP ist im Platform Builder für CE 6.0 R2 verfügbar. Sie finden die Datei *Config.bib* im Verzeichnis *Platform\<BSP Name>\Files*. In Abbildung 2.4 ist der MEMORY-Abschnitt in Visual Studio 2005 dargestellt.

config.bib

```
MEMORY

;
; NK and RAM region definitions.
;
IF IMGFLASH !
#define    NKNAME    NK
#define    NKSTART   80070000
#define    NKLEN     02000000

#define    RAMNAME   RAM
#define    RAMSTART  82070000
#define    RAMLEN    01E7F000

ELSE

#define    NKNAME    NK
#define    NKSTART   88001000
#define    NKLEN     05fff000 // 96mb less 4k

#define    RAMNAME   RAM
#define    RAMSTART  80070000
#define    RAMLEN    03E7F000

ENDIF ; IMGFLASH

    PTS             80000000       00020000     RESERVED
    ARGS            80020000       00000800     RESERVED
    SLEEPSTATE      80020800       00000800     RESERVED
    EBOOT           80021000       00040000     RESERVED
    EBOOT_STACK     80061000       00004000     RESERVED
    EBOOT_RAM       80065000       00006000     RESERVED

    $(NKNAME)       $(NKSTART)     $(NKLEN)     RAMIMAGE
    $(RAMNAME)      $(RAMSTART)    $(RAMLEN)    RAM
```

Abbildung 2.4 MEMORY-Abschnitt in einer bib-Datei

Die Felder im MEMORY-Abschnitt definieren folgende Parameter:

- **Name** Der Name des MEMORY-Abschnitts. Dieser Name muss eindeutig sein.
- **Address** Diese Hexadezimalzahl repräsentiert die Startadresse des MEMORY-Abschnitts.
- **Size** Diese Hexadezimalzahl definiert die Gesamtlänge des MEMORY-Abschnitts in Bytes.
- **Type** Dieses Feld kann einen der folgenden Wert haben:
  - ***RESERVED*** Gibt an, dass der Bereich reserviert ist. *Romimage.exe* überspringt diese Abschnitte während der Erstellung des Images. Beispielsweise umfasst die Datei *Ce.bib* mehrere RESERVED-Abschnitte, beispielsweise den ARGS-Abschnitt, um einen freigegebenen Speicherbereicht für den Boot Loader (EBOOT) bereitzustellen und Daten nach dem Start an das System zu übergeben (ARGS), sowie den DISPLAY-Abschnitt für einen Anzeigepuffer (siehe Abbildung 2.4). Die Datei *Ce.bib* eines anderen OS Designs kann weitere RESERVED-Abschnitte für Speicherbereiche enthalten, die der Kernel nicht als Systemspeicher verwendet.
  - ***RAMIMAGE*** Definiert den Speicherbereich, den das System zum Laden des Kernelimages und der Module verwendet, die Sie in den Abschnitten MODULES und FILES der bib-Dateien angeben. Ein Run-Time Image kann nur einen RAMIMAGE-Abschnitt enthalten und der Adressbereich muss zusammenhängend sein.
  - ***RAM*** Definiert einen Speicherbereich für das RAM-Dateisystem und zum Ausführen von Anwendungen. Dieser MEMORY-Abschnitt muss zusammenhängend sein. Wenn Sie einen nicht zusammenhängenden MEMORY-Abschnitt benötigen, beispielsweise für erweiterten DRAM (Dynamic RAM) auf dem Gerät, können Sie diesen reservieren, indem Sie die Funktion *OEMGetExtensionDRAM* im OAL des BSP implementieren. Windows Embedded CE unterstützt bis zu zwei Abschnitte für nicht zusammenhängenden physischen Speicher.

CONFIG-Abschnitt der BIB-Datei Der CONFIG-Abschnitt definiert zusätzliche Parameter für das Run-Time Image, einschließlich der folgenden Optionen:

- **AUTOSIZE** Fasst die Abschnitte RAMIMAGE und RAM automatisch zusammen und weist dem RAM den nicht belegten Speicher im RAMIMAGE-

Abschnitt oder gegebenenfalls Speicher aus dem RAM-Abschnitt dem RAMIMAGE zu.

- **BOOTJUMP** Dieser Parameter verschiebt die Boot Jump-Seite in einen bestimmten Bereich im RAMIMAGE-Abschnitt, anstatt den Standardbereich zu verwenden.
- **COMPRESSION** Komprimiert schreibbare Speicherbereiche im Image automatisch. Der Standardwert für diese Option ist **ON**.
- **FIXUPVAR** Initialisiert während der Make Binary Image-Phase eine globale Kernelvariable.
- **FSRAMPERCENT** Legt den RAM für das RAM-Dateisystem in Prozent fest.
- **KERNELFIXUPS** Weist *Romimage.exe* an, den Speicher zu verschieben, in den der Kernel schreiben kann. Normalerweise ist diese Option aktiviert (**ON**).
- **OUTPUT** Wechselt das Verzeichnis, das *Romimage.exe* als Ausgabeverzeichnis für die Datei *Nk.bin* verwendet.
- **PROFILE** Gibt an, ob das Image den Profiler enthält.
- **RAM_AUTOSIZE** Erweitert den RAM bis zum Ende des letzten XIP-Abschnitts.
- **RESETVECTOR** Verschiebt die Jump-Seite an eine festgelegte Stelle. Dies ist erforderlich, damit MIPS-Prozessoren von 9FC00000 starten.
- **ROM_AUTOSIZE** Ändert die Größe der XIP-Bereiche unter Berücksichtigung der ROMSIZE_AUTOGAP-Einstellung.
- **ROMFLAGS** Konfiguriert folgende Optionen für den Kernel:
  - ***Demand Paging*** Kopiert eine Datei vor der Ausführung oder Seitenzuordnung in den RAM.
  - ***Full Kernel Mode*** Da alle OS-Threads im Kernelmodus ausgeführt werden, ist das System nicht vor Angriffen geschützt, aber die Leistung wird verbessert.
  - ***Trust only ROM modules*** Kennzeichnet ausschließlich die Dateien im ROM als vertrauenswürdig.
  - ***Flush the X86 TLB on X86 systems*** Verbessert die Leistung, aber stellt ein Sicherheitsrisiko dar.
  - ***Honor the /base linker setting*** Legt fest, ob die Linkereinstellung **/base linker** in DLLs verwendet wird.

- **ROMOFFSET** Ermöglicht das Ausführen des Run-Time Images in einem anderen Speicherbereich. Beispielsweise können Sie das Run-Time Image im FLASH-Speicher speichern und anschließend kopieren und aus dem RAM ausführen.
- **ROMSIZE** Gibt die Größe des ROM in Bytes an.
- **ROMSTART** Gibt die Startadresse des ROM an.
- **ROMWIDTH** Legt fest, wie die Datenbits und Romimage.exe das Run-Time Image aufteilen. Romimage.exe kann das gesamte Run-Time Image in eine Datei packen, in zwei Dateien mit geraden und ungeraden 16-Bit-Wörtern oder in vier Dateien mit geraden und ungeraden 8-Bit-Bytes aufteilen.
- **SRE** Legt fest, ob Romimage.exe eine sre-Datei generiert. Das Dateiformat SRE (Motorola S-record) wird von den meisten ROM-Brennern erkannt.
- **X86BOOT** Gibt an, ob bei der x86-Reset-Vektoradresse eine JUMP-Anweisung hinzugefügt wird.
- **XIPSCHAIN** Ermöglicht das Erstellen der Dateien *Chain.bin* und *Chain.lst*, um eine XIP-Kette zum Aufteilen eines Images in mehrere Dateien zu konfigurieren.

Die Abschnitte MODULES und FILES der BIB-Datei BSP- und OS Design-Entwickler müssen die Abschnitte MODULES und FILES einer bib-Datei häufig bearbeiten, um neue Komponenten zu einem Run-Time Image hinzuzufügen. Das Format der Abschnitte MODULES und FILES ist beinahe identisch, aber der MODULES-Abschnitt unterstützt mehr Konfigurationsoptionen. Der wichtigste Unterschied ist, dass der MODULES-Abschnitt Dateien auflistet, die nicht im Speicher komprimiert sind, um XIP zu unterstützen. Der FILES-Abschnitt listet komprimierte Dateien auf. Das Betriebssystem muss die Daten dekomprimieren, um auf die Dateien zuzugreifen.

Im Folgenden sind zwei kleine MODULES- und FILES-Abschnitte in der Datei *Platform.bib* aufgeführt. Ein vollständiges Beispiel finden Sie in der Datei *Platform.bib* des Device Emulator BSPs.

```
MODULES
;  Name                      Path                                 Memory Type
;  --------------            -----------------------------        -----------
; @CESYSGEN IF CE_MODULES_DISPLAY
IF BSP_NODISPLAY !
  DeviceEmulator_lcd.dll   $(_FLATRELEASEDIR)\DeviceEmulator_lcd.dll   NK SHK
IF BSP_NOBACKLIGHT !
  backlight.dll            $(_FLATRELEASEDIR)\backlight.dll            NK SHK
ENDIF BSP_NOBACKLIGHT !
ENDIF BSP_NODISPLAY !
; @CESYSGEN ENDIF CE_MODULES_DISPLAY

FILES

;  Name                 Path                                 Memory Type
;  --------------       -------------------------------      -----------
; @CESYSGEN IF CE_MODULES_PPP
dmacnect.lnk            $(_FLATRELEASEDIR)\dmacnect.lnk            NK  SH
; @CESYSGEN ENDIF CE_MODULES_PPP
```

Sie können für die Dateiverweise in den Abschnitten MODULES und FILES folgende Optionen definieren:

- **Name** Der Name des Moduls oder der Datei in der Speichertabelle. Dieser Name ist normalerweise mit dem Dateinamen im Run-Time Image identisch.
- **Path** Der vollständige Pfad zur Datei, die Romimage.exe in das Run-Time Image einbezieht.
- **Memory** Referenziert den Namen eines Speicherbereichs im MEMORY-Abschnitt der Datei Config.bib, in dem *Romimage.exe* das Modul oder die Datei lädt. Diese Option ist normalerweise auf NK festgelegt, um die Datei im NK-Bereich zu integrieren, der im MEMORY-Abschnitt definiert ist.

Tabelle 2.3 Dateitypdefinitionen für die Abschnitte MODULES und FILES

MODULES- und FILES-Abschnitt	Nur MODULES-Abschnitt
■ **S** Die Datei ist eine Systemdatei. ■ **H** Die Datei ist ausgeblendet. ■ **U** Die Datei ist nicht komprimiert. (Die Standardeinstellung für Dateien ist Komprimiert.) ■ **N** Das Modul ist nicht vertrauenswürdig. ■ **D** Das Modul ist nicht zum Debuggen geeignet.	■ **K** Weist Romimage.exe an, eine feste virtuelle Adresse zu den öffentlichen Exporten der DLL zuzuordnen und führt das Modul im Kernelmodus anstatt im Benutzermodus aus. Treiber müssen für den direkten Zugriff auf die Hardware im Kernelmodus ausgeführt werden. ■ **R** Komprimiert Ressourcendateien. ■ **C** Komprimiert alle Daten in der Datei. Wenn die Datei bereits im RAM gespeichert ist, wird sie erneut in einen neuen RAM-Abschnitt dekomprimiert. Dies führt zu einer höheren RAM-Belegung. ■ **P** Überprüft den CPU-Typ nicht pro Modul. ■ **X** Signiert das Modul und speichert die Signatur im ROM. ■ **M** Signalisiert, dass der Kernel das Modul nicht auf Anforderung aufrufen muss. (In Kapitel 3 finden Sie weitere Informationen zum Demand Paging.) ■ **L** Weist Romimage.exe an, die ROM DLL nicht aufzuteilen.

- **Section Override** Ermöglicht die Angabe von Modulen im FILES-Abschnitt und von Dateien im MODULES-Abschnitt. Romimage.exe ignoriert den Abschnitt, in dem sich der Eintrag befindet, und behandelt den Eintrag als Member des angegebenen Abschnitts. Dieser Parameter ist optional.

- **Type** Gibt den Dateityp an und kann aus einer Kombination aus Flags bestehen (siehe Tabelle 2.3).

Bedingte Verarbeitung der bib-Datei Beachten Sie, dass bib-Dateien Bedingungsanweisungen basierend auf Umgebungsvariablen und SYSGEN-Variablen unterstützen. Sie können Umgebungsvariablen über Katalogelemente festlegen und diese Variablen in IF-Anweisungen in einer bib-Datei überprüfen, um bestimmte Module oder Dateien einzubeziehen bzw. auszuschließen. Verwenden Sie @CESYSGEN IF-Anweisungen für SYSGEN-Variablen.

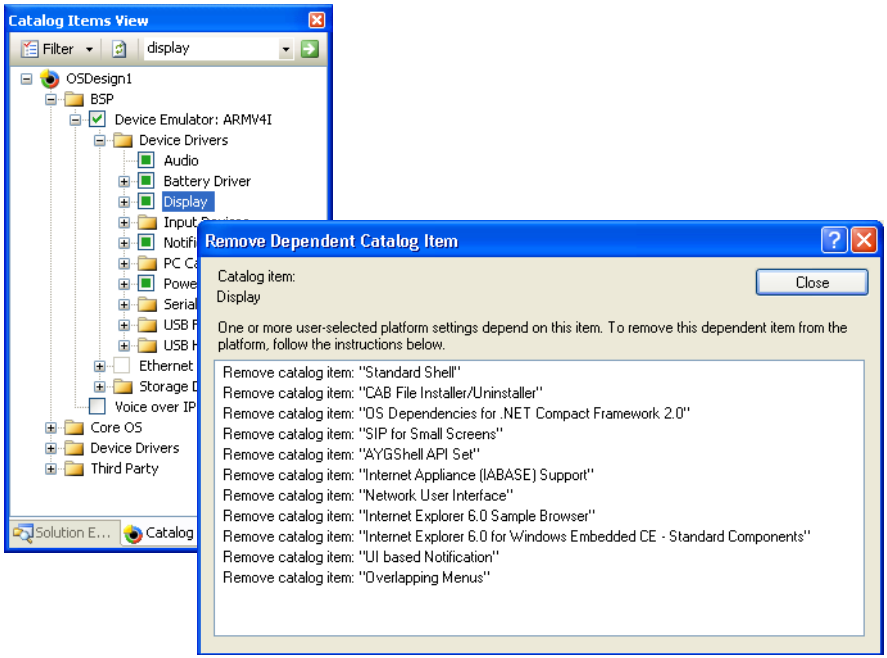


Abbildung 2.5 Die Kernkomponenten eines Betriebssystems, die vom Bildschirmelement abhängig sind

Die MODULES- und FILES-Auflistung im vorherigen Abschnitt veranschaulicht die Verwendung von @CESYSGEN IF- und IF-Anweisungen für die Verarbeitung von Bedingungen basierend auf SYSGEN-Variablen und Umgebungsvariablen. Die @CESYSGEN IF CE_MODULES_DISPLAY-Anweisung in den MODULES-Abschnitten gibt beispielsweise an, dass das BSP den Bildschirmtreiber automatisch einbezieht, wenn das OS Design eine Bildschirmkomponente umfasst. Sie können in der **Catalog Items View** für ein OS Design in Visual Studio überprüfen, ob der Platform Builder die Bildschirmkomponente automatisch zum BSP hinzufügt (siehe Abbildung 2.5).

Registrierungsdateien

Registrierungsdateien (.reg) initialisieren die Systemregistrierung auf dem Remotegerät. Diese Dateien sind beinahe mit den Registrierungsdateien der Windows-Desktopbetriebssysteme identisch, mit dem Unterschied, dass die reg-Dateien in CE nicht mit einem Header und Versionsinformationen beginnen. Wenn Sie auf einem Entwicklungscomputer auf eine reg-Datei von CE doppelklicken und bestätigen, dass Sie die Einstellungen zur Desktopregistrierung hinzufügen möchten, wird eine Meldung angezeigt, die besagt, dass die reg-Datei kein gültiges Registrierungsskript ist. Ein weiterer Unterschied ist, dass die reg-Dateien in CE, ähnlich wie bib-Dateien, Bedingungsanweisungen enthalten können, um Registrierungseinstellungen entsprechend den ausgewählten Katalogelementen zu importieren. Das folgende Codesegment in der Datei Platform.reg des Device Emulator BSP veranschaulicht die Verwendung von Vorverarbeitungsbedingungen.

```
; Our variables
#define BUILTIN_ROOT HKEY_LOCAL_MACHINE\Drivers\BuiltIn
;#define PCI_BUS_ROOT $(BUILTIN_ROOT)\PCI
#define DRIVERS_DIR $(_PUBLICROOT)\common\oak\drivers

; @CESYSGEN IF CE_MODULES_RAMFMD
; @CESYSGEN IF FILESYS_FSREGHIVE
; HIVE BOOT SECTION
[HKEY_LOCAL_MACHINE\init\BootVars]
   "Flags"=dword:1          ; see comment in common.reg
; END HIVE BOOT SECTION
; @CESYSGEN ENDIF FILESYS_FSREGHIVE
; @CESYSGEN IF CE_MODULES_PCCARD
; @XIPREGION IF DEFAULT_DEVICEEMULATOR_REG

IF BSP_NOPCCARD !
#include "$(_TARGETPLATROOT)\src\drivers\pccard\pcc_smdk2410.reg"
#include "$(DRIVERS_DIR)\pccard\mdd\pcc_serv.reg"
[HKEY_LOCAL_MACHINE\Drivers\PCCARD\PCMCIA\TEMPLATE\PCMCIA]
   "Dll"="pcmcia.dll"
   "NoConfig"=dword:1
   "NoISR"=dword:1 ; Do not load any ISR.
   "IClass"=multi_sz:"{6BEAB08A-8914-42fd-B33F-61968B9AAB32}=
                                                PCMCIA Card Services"
ENDIF ; BSP_NOPCCARD !
; @XIPREGION ENDIF DEFAULT_DEVICEEMULATOR_REG
; @CESYSGEN ENDIF CE_MODULES_PCCARD
```

Datenbankdateien

Windows Embedded CE verwendet Datenbankdateien (.db), um den Standardobjektspeicher zu konfigurieren. Der Objektspeicher ist eine auf Transaktionen

basierende Speichermethode. Das heißt, der Objektspeicher ist ein Speicher für Datenbanken im RAM, den das Betriebssystem und die Anwendungen als permanenten Datenspeicher verwenden können. Beispielsweise verwendet das Betriebssystem den Objektspeicher, um Stacks und Speicherheaps zu verwalten, Dateien zu komprimieren bzw. zu dekomprimieren und ROM-basierte Anwendungen und RAM-basierte Daten zu integrieren. Die transaktionsorientierte Speichermethode stellt die Datenintegrität auch bei einem Stromausfall sicher, während die Daten in den Objektspeicher geschrieben werden. Beim Neustart des Systems beendet Windows Embedded CE die anstehende Transaktion oder stellt die letzte als funktionierend bekannte Konfiguration wieder her. Für Systemdateien muss Windows Embedded CE zum Wiederherstellen der letzten als funktionierend bekannten Konfiguration möglicherweise die ursprünglichen Einstellungen erneut aus dem ROM laden.

Dateisystemdateien

Dateisystemdateien (.dat), insbesondere *Platform.dat* und *Project.dat*, enthalten Einstellungen zum Initialisieren des RAM-Dateisystems. Wenn Sie das Run-Time Image auf dem Zielgerät kalt starten, verarbeitet *Filesys.exe* die dat-Dateien, um die RAM-Dateisystemverzeichnisse, Dateien und Verknüpfungen auf dem Zielgerät zu erstellen. Die Datei *Platform.dat* wird normalerweise für Hardwareeinträge und die Datei *Project.dat* für das OS Design verwendet. Sie können die Dateisystemeinstellungen jedoch in einer beliebigen dat-Datei festlegen, da das Buildsystem alle dat-Dateien in der Datei *Initobj.dat* zusammenfasst.

Beispielsweise können Sie für ein Run-Time Image zusätzlich zum Windows-Verzeichnis die Stammverzeichnisse definieren, indem Sie die Datei *Project.dat* entsprechend anpassen. Die Elemente im ROM-Image werden standardmäßig im Windows-Verzeichnis angezeigt. Unter Verwendung einer dat-Datei können Sie die Dateien auch außerhalb des Windows-Verzeichnisses anzeigen. Außerdem können Sie die Dateien im ROM Windows-Verzeichnis auch kopieren oder verknüpfen. Dies ist insbesondere nützlich, wenn Sie Verknüpfungen auf dem Desktop oder zu Anwendungen im Startmenü hinzufügen möchten. Ähnlich wie für reg- und bib-Dateien können Sie die Bedingungen IF und IF ! in dat-Dateien verwenden.

Im Folgenden ist die Verwendung der Datei *Project.dat* zum Erstellen der beiden Stammverzeichnisse *Program Files* und *My Documents* und des Unterverzeichnisses *My Projects* in *Program Files* sowie zum Zuordnen der Datei *Myfile.doc* im *Windows*-Verzeichnis zum Verzeichnis *My Documents* veranschaulicht.

```
Root:-Directory("Program Files")
Root:-Directory("My Documents")
Directory("\Program Files"):-Directory("My Projects")
Directory("\My Documents"):-File("MyFile.doc", "\Windows\Myfile.doc")
```

Zusammenfassung

Umfassende Kenntnisse des Buildsystems helfen Ihnen die Entwicklungszeit (und somit die Projektkosten) zu reduzieren. Sie müssen mit den während jeder Phase des Buildprozesses ausgeführten Schritten vertraut sein, um am Quellcode vorgenommene Änderungen schnell und ohne unnötige Kompilierungszyklen zu testen. Außerdem müssen Sie über den Verwendungszweck und den Speicherort der Konfigurationsdateien für das Run-Time Image Bescheid wissen, beispielsweise der reg-, bib-, db- und dat-Dateien, um OS Designs effizient erstellen und verwalten zu können.

Das Windows Embedded CE-Buildsystem fasst die reg-, bib-, db- und dat-Dateien während der Make Run-Time Image-Phase in konsolidierten Dateien zusammen, mit denen das Buildsystem das fertige Run-Time Image konfiguriert. Sie sollten diese Dateien überprüfen, um sicherzustellen, dass eine bestimmte Einstellung oder Datei im Image enthalten ist, bevor Sie das Run-Time Image auf dem Zielgerät laden. Die Konfigurationsdateien für das Run-Time Image sind im Releaseverzeichnis des OS Designs gespeichert. Wenn Einträge fehlen, überprüfen Sie sowohl die Bedingungsanweisungen als auch die Umgebungsvariablen und SYSGEN-Variablen, die in den Katalogelementen definiert sind.

Das Buildsystem erstellt während der Make Run-Time Image-Phase die folgenden Konfigurationsdateien für das Run-Time Image:

- ***Reginit.ini*** Fasst die Dateien *Platform.reg*, *Project.reg*, *Common.reg*, *IE.reg*, *Wceapps.reg* und *Wceshell.reg* zusammen.
- ***Ce.bib*** Fasst die Dateien *Config.bib*, *Platform.bib*, *Project.bib* und die bib-Dateien des Teilprojekts zusammen.
- ***Initdb.ini*** Fasst die Dateien *Common.db*, *Platform.db* und *Project.db* zusammen.
- ***Initobj.dat*** Fasst die Dateien *Common.dat*, *Platform.dat* und *Project.dat* zusammen.

Lektion 2: Bearbeiten der Buildkonfigurationsdateien

Zusätzlich zu den Konfigurationsdateien für das Run-Time Image verwendet Windows Embedded CE Buildkonfigurationsdateien, um den Quellcode in funktionelle binäre Komponenten zu kompilieren. Das Buildsystem hängt für die Konfiguration des Quellcodes insbesondere von drei Dateitypen ab: *Dirs*, *Sources* und *Makefile*. Diese Dateien enthalten Informationen über die zu durchsuchenden Quellcodeverzeichnisse, die zu kompilierenden Quellcodedateien und den Typ der zu erstellenden binären Komponenten für das Buildtool (*Build.exe*), den Compiler und den Linker (*Nmake.exe*). Ein CE-Entwickler muss diese Dateien, beispielsweise beim Klonen der öffentlichen Katalogelemente, mit den in Kapitel 1 beschriebenen Verfahren bearbeiten.

Nach Abschluss dieser Lektion können Sie:

- Die Quellcode-Konfigurationsdateien, die während des Buildprozesses verwendet werden, identifizieren.
- Die Buildkonfigurationsdateien bearbeiten, um Anwendungen, DLLs und statische Bibliotheken zu generieren.

Veranschlagte Zeit für die Lektion: 25 Minuten.

Dirs-Dateien

Dirs-Dateien identifizieren die Verzeichnisse, die die Quellcodedateien enthalten, die in den Buildprozess einbezogen werden. Wenn *Build.exe* eine Dirs-Datei im Ordner findet, in dem das Tool ausgeführt wird, werden die in der Dirs-Datei referenzierten Unterverzeichnisse durchlaufen, um den Quellcode in diesen Unterverzeichnissen zu erstellen. Diese Methode ermöglicht Ihnen unter anderem Teile des Run-Time Images selektiv zu aktualisieren. Wenn Sie Änderungen am Quellcode in *Subproject1* vornehmen, können Sie dieses Teilprojekt erneut erstellen, indem Sie *Build.exe* im Verzeichnis *Subproject1* ausführen. Sie können Verzeichnisse in der Quellcodestruktur auch aus dem Buildprozess ausschließen, indem Sie die entsprechenden Verzeichnisverweise aus der Dirs-Datei löschen oder Bedingungsanweisungen verwenden.

Dirs-Dateien sind Textdateien mit einer unkomplizierten Inhaltsstruktur. Sie können mit DIRS, DIRS_CE oder OPTIONAL_DIRS die Unterverzeichnisse in einer oder in mehreren Zeilen angeben (beenden Sie jede Zeile mit einem Backslash). Verzeichnisse, die mit dem DIRS-Schlüsselwort referenziert sind, werden immer in

den Buildprozess einbezogen. Wenn Sie das Schlüsselwort DIRS_CE angeben, erstellt *Build.exe* den Quellcode nur, wenn dieser ausdrücklich für ein Windows Embedded CE Run-Time Image geschrieben wurde. Das Schlüsselwort OPTIONAL_DIRS definiert optionale Verzeichnisse. Beachten Sie jedoch, dass Dirs-Dateien nur eine DIRS-Direktive enthalten können. Da *Build.exe* die Verzeichnisse in der angegebenen Reihenfolge verarbeitet, listen Sie die erforderlichen Verzeichnisse zuerst auf. Mit dem Platzhalter "*" können Sie alle Verzeichnisse einbeziehen.

Das folgende Codesegment aus Windows Embedded CE-Standardkomponenten veranschaulicht das Einbeziehen der Quellcodeverzeichnisse in den Buildprozess mit Dirs-Dateien.

```
# C:\WINCE600\PLATFORM\DEVICEEMULATOR\SRC\Dirs
#-----------------------------------------------------
DIRS=common     \
     drivers    \
     apps       \
     kitl       \
     oal        \
     bootloader

# C:\WINCE600\PLATFORM\H4SAMPLE\SRC\DRIVERS\Dirs
#-----------------------------------------------------
DIRS=        \
# @CESYSGEN IF CE_MODULES_DEVICE
    buses \
    dma \
    triton \
# @CESYSGEN IF CE_MODULES_KEYBD
    keypad \
# @CESYSGEN ENDIF CE_MODULES_KEYBD
# @CESYSGEN IF CE_MODULES_WAVEAPI
    wavedev \
# @CESYSGEN ENDIF CE_MODULES_WAVEAPI
# @CESYSGEN IF CE_MODULES_POINTER
    touch \
    tpagent \
# @CESYSGEN ENDIF CE_MODULES_POINTER
# @CESYSGEN IF CE_MODULES_FSDMGR
    nandflsh \
# @CESYSGEN ENDIF CE_MODULES_FSDMGR
# @CESYSGEN IF CE_MODULES_SDBUS
    sdhc \
# @CESYSGEN ENDIF CE_MODULES_SDBUS
# @CESYSGEN IF CE_MODULES_DISPLAY
    backlight \
# @CESYSGEN ENDIF CE_MODULES_DISPLAY
# @CESYSGEN IF CE_MODULES_USBFN
    usbd \
```

```
# @CESYSGEN ENDIF CE_MODULES_USBFN
# @CESYSGEN ENDIF CE_MODULES_DEVICE
# @CESYSGEN IF CE_MODULES_DISPLAY
    display \
# @CESYSGEN ENDIF CE_MODULES_DISPLAY
```

HINWEIS Bearbeiten von Dirs-Dateien in Solution Explorer

Der Solution Explorer in Visual Studio mit Platform Builder für Windows Embedded CE 6.0 R2 verwendet Dirs-Dateien, um eine dynamische Ansicht der Windows Embedded CE-Verzeichnisstruktur in einem OS Designprojekt zu generieren. Sie sollten im Solution Explorer jedoch keine Verzeichnisse hinzufügen oder entfernen, da die Bearbeitung von Dirs-Dateien im Solution Explorer die Buildreihenfolge ändern kann. Dies verursacht möglicherweise Buildfehler, für deren Behebung ein zweiter Build erforderlich ist.

Sources-Dateien

Wenn Sie die Ordner und Dateien eines OS Designs überprüfen, beispielsweise *C:\Wince600\OSDesigns\OSDesign1*, werden Sie feststellen, dass das Projekt standardmäßig keine Dirs-Dateien umfasst. Wenn Sie Teilprojekte für benutzerdefinierte Komponenten und Anwendungen einbeziehen, enthalten die Stammordner der Teilprojekte jeweils eine Sources-Datei. Die Sources-Datei enthält detaillierte Informationen zu den Quellcodedateien, einschließlich Builddirektiven, die eine Dirs-Datei nicht bereitstellen kann. Ein Quellcodeverzeichnis kann jedoch nur eine Dirs-Datei oder eine Sources-Datei enthalten, nicht beide. Das heißt, dass ein Verzeichnis mit einer Sources-Datei keine Unterverzeichnisse mit weiterem Code enthalten kann. *Nmake.exe* verwendet die Sources-Dateien während des Buildprozesses, um zu bestimmen, welcher Dateityp (.lib, .dll oder .exe) erstellt wird. Ähnlich wie Dirs-Dateien erwarten Sources-Dateien die Deklarationen in einer Zeile. Sie können die Zeile jedoch mit einem Backslash beenden, um die Deklaration in der nächsten Zeile fortzusetzen.

Das folgende Codesegment zeigt den Inhalt einer Sources-Datei im Device Emulator BSP an. Standardmäßig ist diese Datei im Ordner *C:\Wince600\Platform\DeviceEmulator\Src\Drivers\Pccard* gespeichert.

```
WINCEOEM=1

TARGETNAME=pcc_smdk2410
TARGETTYPE=DYNLINK
RELEASETYPE=PLATFORM
TARGETLIBS=$(_COMMONSDKROOT)\lib\$(_CPUINDPATH)\coredll.lib \
           $(_SYSGENOAKROOT)\lib\$(_CPUINDPATH)\ceddk.lib
```

```
SOURCELIBS=$(_SYSGENOAKROOT)\lib\$(_CPUINDPATH)\pcc_com.lib

DEFFILE=pcc_smdk2410.def
DLLENTRY=_DllEntryCRTStartup

INCLUDES=$(_PUBLICROOT)\common\oak\drivers\pccard\common;$(INCLUDES)

SOURCES= \
    Init.cpp \
    PDSocket.cpp \
    PcmSock.cpp  \
    PcmWin.cpp

#xref VIGUID {549CAC8D_8AF0_4789_9ACF_2BB92599470D}
#xref VSGUID {0601CE65_BF4D_453A_966B_E20250AD2E8E}
```

Sie können die folgenden Direktiven in einer Sources-Datei definieren:

- **TARGETNAME** Der Name der Zieldatei ohne Dateierweiterung.
- **TARGETTYPE** Definiert den Typ der Datei, die erstellt wird, wie folgt:
- **DYNLINK** Eine DLL (Dynamic-Link Library).
- **LIBRARY** Eine LIB-Datei (Static-Link Library).
- **PROGRAM** Eine ausführbare Datei (EXE).
- **NOTARGET** Es wird keine Datei erstellt.
- **RELEASETYPE** Gibt das Verzeichnis an, in dem Nmake.exe die Zieldatei speichert:
  - ***PLATFORM*** *PLATFORM\(BSP Name)\<Target>.*
  - ***OAK, SDK, DDK*** *%_PROJECTROOT%\Oak\<Target>.*
  - ***LOCAL*** Das aktuelle Verzeichnis.
  - ***CUSTOM*** Das in TARGETPATH angegebene Verzeichnis.
  - ***MANAGED*** *%_PROJECTROOT%\Oak\<Target>\Managed.*
- **TARGETPATH** Definiert den Pfad für **RELEASETYPE=CUSTOM**.
- **SOURCELIBS** Gibt die Bibliotheken an, die mit der in TARGETNAME festgelegten Zieldatei verknüpft werden, um die binäre Ausgabe zu erstellen. Diese Option wird normalerweise zum Erstellen einer lib-Datei verwendet, aber nicht von dll- oder exe-Dateien.
- **TARGETLIBS** Gibt zusätzliche Bibliotheken und Objektdateien für die binäre Ausgabe an (normalerweise zum Erstellen von dll- oder exe-Dateien, aber nicht von lib-Dateien.

- **INCLUDES** Listet zusätzliche Verzeichnisse auf, die nach Include-Dateien durchsucht werden.
- **SOURCES** Definiert die Quelldateien für eine bestimmte Komponente.
- **ADEFINES** Gibt die Parameter für den Assembler an.
- **CDEFINES** Gibt die Parameter für den Compiler an, die als zusätzliche DEFINE-Anweisungen in IFDEF-Anweisungen verwendet werden.
- **LDEFINES** Legt die Linkerdefinitionen fest.
- **RDEFINES** Gibt DEFINE-Anweisungen für den Ressourcencompiler an.
- **DLLENTRY** Definiert den Einsprungspunkt für eine DLL.
- **DEFFILE** Definiert die def-Datei, die die exportierten Symbole einer DLL enthält.
- **EXEENTRY** Legt den Einsprungspunkt für eine ausführbare Datei fest.
- **SKIPBUILD** Markiert den Build als erfolgreich, ohne das Ziel tatsächlich zu erstellen.
- **WINCETARGETFILE0** Gibt nicht dem Standard entsprechende Dateien an, die vor dem aktuellen Verzeichnis erstellt werden müssen.
- **WINCETARGETFILES** Diese Makrodefinition gibt die nicht dem Standard entsprechenden Zieldateien an, die Build.exe erstellen muss, nachdem alle anderen Ziele im aktuellen Verzeichnis gelinkt wurden.
- **WINCE_OVERRIDE_CFLAGS** Definiert Compilerflags, um die Standardeinstellungen zu überschreiben.
- **WINCECPU** Gibt an, dass der Code einen bestimmten CPU-Typ erfordert und nur für diesen CPU-Typ erstellt wird.

HINWEIS Ausführen von Aktionen vor und nach dem Build

Zusätzlich zu den Standarddirektiven unterstützen die Sources-Dateien in Windows Embedded CE die Direktiven PRELINK_PASS_CMD und POSTLINK_PASS_CMD. Mit diesen Direktiven können Sie vor und nach dem Buildprozess benutzerdefinierte Aktionen basierend auf Befehlszeilentools oder Batchdateien ausführen, beispielsweise PRELINK_PASS_CMD=pre_action.bat und POSTLINK_PASS_CMD=post_action.bat. Dies ist unter anderem hilfreich, wenn Sie beim Entwickeln einer benutzerdefinierten Anwendung zusätzliche Dateien in das Releaseverzeichnis kopieren möchten.

Makefile-Dateien

In einem Teilprojektordner ist die Datei Makefile gespeichert, die Standarddirektiven für die Vorverarbeitung, Befehle, Makros und andere Ausdrücke für *Nmake.exe* enthält. Die Datei *Makefile* in Windows Embedded CE enthält jedoch nur eine einzige Zeile, die auf *%_MAKEENVROOT%\Makefile.def* verweist. Die Umgebungsvariable %_MAKEENVROOT% verweist standardmäßig auf den Ordner *C:\Wince600\Public\Common\Oak\Misc* und die Datei *Makefile.def* in diesem Ordner ist die Standarddatei für alle CE-Komponenten. Sie sollten diese Datei nicht ändern. Die Datei *Makefile.def* enthält unter anderem Include-Anweisungen, beispielsweise !INCLUDE $(MAKEDIR)\sources, um die Sources-Datei im Teilprojektordner anzugeben. Bearbeiten Sie die Sources-Datei im Teilprojektordner, um die Erstellung der Zieldatei mit *Nmake.exe* anzupassen.

Zusammenfassung

Die Windows Embedded CE 6.0 R2-Entwicklungsumgebung steuert mit Makefile-, Sources- und Dirs-Dateien, wie *Build.exe* und *Nmake.exe* den Quellcode in funktionelle binäre Komponenten für das Run-Time Image kompilieren. Sie können die Dirs-Dateien verwenden, um die Quellcodeverzeichnisse für den Buildprozess festzulegen. Mit Sources-Dateien können Sie die Kompilierungs- und Builddirektiven detailliert angeben. Die Makefile-Datei erfordert keine Anpassung, sondern verweist auf die Standarddatei *Makefile.def*, die allgemeine Vorverarbeitungsdirektiven, Befehle, Markos und andere Verarbeitungsanweisungen für das Buildsystem enthält. Sie müssen mit dem Verwendungszweck der Dateien und dem Steuern des Buildprozesses vertraut sein, um öffentliche Katalogelemente zu klonen oder neue Komponenten zu erstellen.

Lektion 3: Analysieren der Buildergebnisse

Während der Softwareentwicklung treten mit größter Wahrscheinlichkeit Buildfehler auf. Es ist auch nicht ungewöhnlich, die Syntax des Quellcodes mittels Kompilierungsfehlern zu überprüfen, obwohl IntelliSense® und andere Codierhilfen in Visual Studio 2005 die Tippfehler und anderen Syntaxfehler erheblich reduzieren können. Syntaxfehler sind relativ einfach zu beheben, da Sie auf die entsprechende Fehlermeldung im Ausgabefenster doppelklicken und direkt zur fehlerhaften Zeile in der Quellcodedatei springen können. Kompilierungsfehler sind jedoch nicht die einzigen Fehler, die während des Buildprozesses auftreten können. Andere häufig auftretende Buildfehler sind mathematische Fehler, Ausdrucksfehler, Linkerfehler und Fehler in den Konfigurationsdateien für das Run-Time Image. Außer Fehlermeldungen generiert das Buildsystem auch Statusmeldungen und Warnungen, die Ihnen helfen, Buildprobleme zu analysieren. Die während des Buildprozesses generierten Informationen sind möglicherweise überwältigend. Sie müssen mit den verschiedenen Typen und dem Format von Buildmeldungen vertraut sein, um Buildfehler erkennen und beheben zu können.

Nach Abschluss dieser Lektion können Sie:

- Buildberichte suchen und analysieren.
- Buildprobleme analysieren und beheben.

Veranschlagte Zeit für die Lektion: 15 Minuten.

Buildberichte

Wenn Sie einen Build in der Visual Studio IDE oder über die Befehlszeile erstellen, werden umfangreiche Buildinformationen ausgegeben. Das Buildsystem protokolliert diese Informationen in der Datei *Build.log*. Die Dateien *Build.wrn* und *Build.err* enthalten ebenfalls Informationen über Kompilierung- bzw. Linker-Warnungen und Fehler. Wenn Sie für ein OS Design einen vollständigen Build oder einen Sysgen-Vorgang mit den entsprechenden Befehlen im Menü **Build** in Visual Studio starten, schreibt das Buildsystem diese Dateien in den Ordner *%_WINCEROOT%* (standardmäßig *C:\Wince600*). Wenn Sie einen Build jedoch nur für eine bestimmte Komponente ausführen (indem Sie beispielsweise mit der rechten Maustaste im Solution Explorer auf einen Teilprojektordner klicken und im Kontextmenü den Befehl **Build** auswählen), erstellt das Buildsystem diese Dateien im angegebenen Verzeichnis. Die Dateien *Build.wrn* und *Build.err* werden nur erstellt, wenn während

des Buildprozesses Warnungen und Fehler auftreten. Sie müssen die Dateien nicht in Notepad oder einem anderen Texteditor öffnen und durchsuchen. Visual Studio 2005 mit Platform Builder für CE 6.0 R2 zeigt die Informationen während des Buildprozesses im Ausgabefenster an. Sie können Statusmeldungen, Warnungen und Fehler auch im Fenster **Error List** anzeigen, indem Sie im Menü **View** unter **Other Windows** auf **Error List** klicken.

In Abbildung 2.6 sind die Fenster **Output** und **Error List** dargestellt. Im Fenster **Output** wird der Inhalt der Datei *Build.log* angezeigt, wenn Sie im Listenfeld **Show Output From** die Option **Build** auswählen. Das Fenster **Error List** zeigt den Inhalt der Dateien *Build.wrn* und *Build.err* an.

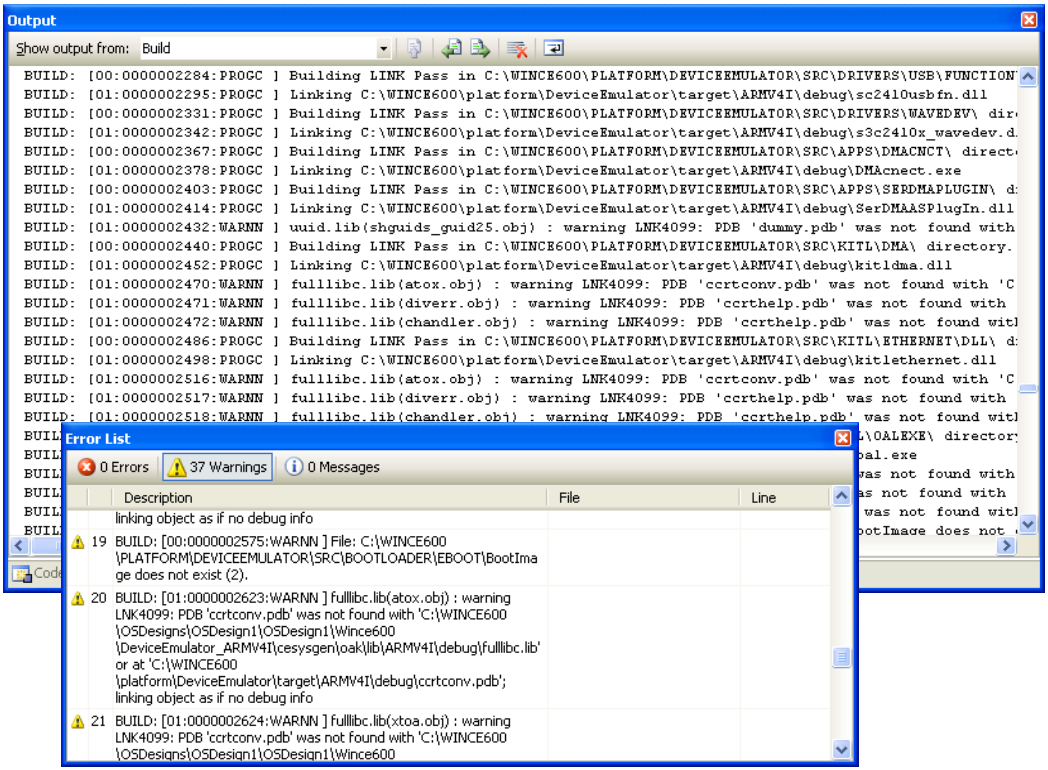


Abbildung 2.6 Die Fenster Output und Error List mit Buildinformationen in Visual Studio

Die Buildprotokolldateien enthalten folgende Informationen:

- ***Build.log*** Enthält Informationen zu den Befehlen, die während der Buildphasen ausgeführt werden. Diese Informationen sind nützlich, um den Buildprozess und Buildfehler zu analysieren.
- ***Build.wrn*** Enthält Informationen zu den Warnungen, die während des Buildprozesses generiert werden. Versuchen Sie, die Ursachen für die Warnungen zu beheben oder mindestens zu identifizieren. Die Informationen in *Build.wrn* sind ebenfalls in *Build.log* enthalten.
- ***Build.err*** Enthält Informationen zu den Fehlern, die während des Buildprozesses aufgetreten sind. Diese Informationen sind ebenfalls in der Datei *Build.log* verfügbar. Diese Datei wird nur erstellt, wenn ein Fehler auftritt.

HINWEIS Identifizieren der Buildschritte

Das Buildsystem protokolliert übersprungene und ausgeführte Buildphasen in der Datei *Build.log*. Beispielsweise zeigt der Eintrag *CEBUILD: Skipping directly to SYSGEN phase* an, dass das Buildsystem die Kompilierungsphase für eine Komponente übersprungen hat. Sie können überprüfen, an welcher Stelle die Sysgen-Phase beginnt und wie der Buildprozess von SYSGEN zu BUILD bzw. von BUILD zu MAKEIMG übergeht.

Beheben von Buildproblemen

Die Buildprotokolldateien, die Ihnen Einblick in den allgemeinen Buildprozess gewähren, sind ausgesprochen hilfreich beim Beheben von Buildfehlern. Wenn sich eine Fehlermeldung auf eine Quellcodedatei bezieht, können Sie zur relevanten Codezeile springen, indem Sie im Fenster **Error List** auf den Meldungseintrag doppelklicken. Nicht alle Buildfehler beziehen sich jedoch auf den Quellcode. Linkerfehler aufgrund nicht vorhandener Bibliothekreferenzen, Sysgen-Fehler aufgrund fehlender Komponentendateien, Kopierfehler aufgrund nicht ausreichenden Speicherplatzes und Make Run-Time Image-Fehler aufgrund ungültiger Einstellungen in den Konfigurationsdateien können ebenfalls verursachen, dass der Build fehlschlägt.

Fehler während der Sysgen-Phase

Sysgen-Fehler sind normalerweise das Ergebnis fehlender Dateien. Die Datei *Build.log* enthält detaillierte Informationen zur Fehlerursache. Komponenten, die kürzlich zum OS Design hinzugefügt oder aus diesem entfernt wurden, können diesen Fehlertyp verursachen, wenn die erforderlichen Abhängigkeiten nicht verfügbar sind. Um einen Sysgen-Fehler zu analysieren, sollten Sie alle an den Katalogelementen und

deren Abhängigkeiten vorgenommenen Änderungen überprüfen. Beachten Sie, dass einige Komponenten einen neuen Sysgen-Build, anstatt einen normalen Sysgen-Zyklus, erfordern. Führen Sie den Befehl **Clean Sysgen** möglichst nicht aus, da dieser in einer Release- oder Debug-Buildkonfiguration erfordert, dass Sie einen normalen Sysgen-Zyklus in den anderen Buildkonfigurationen ausführen müssen. Sollten nach dem Hinzufügen oder Entfernen von Katalogelementen Sysgen-Buildfehler auftreten, müssen Sie während der nächsten normalen Sysgen-Phase möglicherweise den **Clean Sysgen**-Befehl ausführen, um das Problem zu beheben.

Fehler während der Buildphase

Buildfehler werden normalerweise vom Compiler oder Linker verursacht. Kompilierungsfehler werden von Syntaxfehlern, fehlenden oder unzulässigen Parametern in Funktionsaufrufen, Divisionen durch Null und ähnlichen Problemen verursacht, die verhindern, dass der Compiler gültigen Binärcode generiert. Um zur relevanten Codezeile zu springen, doppelklicken Sie auf den Kompilierungsfehler. Beachten Sie jedoch, dass Kompilierungsfehler von anderen Kompilierungsfehlern verursacht werden können. Beispielsweise kann eine ungültige Variablendeklaration unterschiedliche Kompilierungsfehler verursachen, wenn die Variable in mehreren Funktionen verwendet wird. Normalerweise ist es sinnvoll, den ersten Fehler in der Liste zu beheben und den Code neu zu kompilieren. Auch geringfügige Codeänderungen können oft zahlreiche Fehler beseitigen.

Linker-Fehler sind schwieriger zu beheben als Kompilierungsfehler, da diese normalerweise von fehlenden oder inkompatiblen Bibliotheken verursacht werden. Falsch implementierte APIs können ebenfalls zu Linker-Fehlern führen, wenn der Linker die externen Verweise auf exportierte DLL-Funktionen nicht auflösen kann. Eine weitere häufige Ursache sind falsch initialisierte Umgebungsvariablen. Builddateien, insbesondere Sources-Dateien, verwenden Umgebungsvariablen anstatt hart codierte Verzeichnisnamen, um auf referenzierte Bibliotheken zu verweisen. Wenn diese Umgebungsvariablen nicht festgelegt sind, kann der Linker die Bibliotheken nicht finden. Beispielsweise muss %_WINCEROOT% auf *C:\Wince600* verweisen, wenn Sie Windows Embedded CE in der Standardkonfiguration installiert haben, und %_FLATRELEASEDIR% muss auf das aktuelle Releaseverzeichnis verweisen. Um die Werte der Umgebungsvariablen zu überprüfen, wählen Sie in Visual Studio im Menü **Build** die Option **Open Release Directory in Build Window** aus. Führen Sie anschließend in der Befehlszeile den Befehl **set** mit oder ohne Umgebungsvariable aus, beispielsweise **set _winceroot**. Der Befehl **set** (ohne Parameter) zeigt alle Umgebungsvariablen an, aber diese Liste ist lang.

Fehler während der Release Copy-Phase

Während der Release Copy-Phase auftretende Buildrel-Fehler weisen normalerweise auf nicht genügend Speicherplatz hin. Während der Release Copy-Phase kopiert das Buildsystem Dateien in das Releaseverzeichnis. Möglicherweise müssen Sie Speicherplatz freigeben oder den OS Design-Ordner auf ein anderes Laufwerk verschieben. Stellen Sie sicher, dass der neue Pfad zum OS Design-Ordner keine Leerzeichen enthält, da Leerzeichen im Pfad oder im Namen des OS Designs Fehler im Buildprozess verursachen.

Fehler während der Make Run-Time Image-Phase

Fehler, die während dieser letzten Phase des Buildprozesses auftreten, werden normalerweise von fehlenden Dateien verursacht. Dieses Problem kann auftreten, wenn eine Komponente in einem vorherigen Schritt nicht erstellt werden konnte, aber die Make Run-Time Image-Phase trotzdem fortgesetzt wurde. Syntaxfehler in den reg- oder bib-Dateien können diesen Fehler verursachen, wenn das Buildsystem die Datei *Reginit.ini* oder *Ce.bib* nicht erstellen kann. *Makeimg.exe* ruft während des Buildprozesses das Tool FMerge (*FMerge.exe*) auf, um diese Dateien zu erstellen. Wenn die Dateien nicht erstellt werden können, beispielsweise aufgrund ungültiger Bedingungsanweisungen, tritt ein Make-Image-Fehler auf. Ein weiterer möglicher Fehler ist Error: Image Exceeds (*X*). Diese Meldung gibt an, dass das Image die in der Datei *Config.bib* festgelegte maximale Größe überschreitet.

Zusammenfassung

Platform Builder für Windows Embedded CE 6.0 R2 ist mit dem Buildprotokollierungssystem von Visual Studio 2005 integriert, um den Zugriff auf Statusinformationen, Warnungen und Fehlermeldungen zu ermöglichen, die während des Buildprozesses generiert und in den Dateien *Build.log*, *Build.wrn* und *Build.err* protokolliert werden. Abhängig von der Initialisierung des Buildprozesses in Visual Studio werden diese Dateien im Ordner *%_WINCEROOT%* oder in einem Teilprojektverzeichnis gespeichert. Der Speicherort dieser Dateien ist jedoch nicht wichtig, da Sie die Inhalte der Dateien direkt in den Fenstern **Output** und **Error List** in Visual Studio analysieren können. Sie müssen die Dateien nicht in Notepad oder einem anderen Texteditor öffnen.

Durch die Analyse der Buildprotokolldateien erhalten Sie Einblick in den Buildprozess und Buildprobleme. Die Buildprobleme umfassen Kompilierungsfehler, Linker-Fehler, Sysgen-Fehler, Buildfehler und andere Fehler, die während der Release

Copy- und Make Run-Time Image-Phase auftreten. Wenn ein Buildfehler direkt mit einer Zeile in der Quellcodedatei in Beziehung steht, können Sie im Fenster **Error List** auf den Meldungseintrag doppelklicken, um die entsprechende Datei zu öffnen und zur relevanten Codezeile zu springen. Andere Probleme, beispielsweise Buildrel-Fehler, die von unzureichendem Speicherplatz verursacht werden, müssen Sie außerhalb der Visual Studio IDE beheben.

Lektion 4: Bereitstellen eines Run-Time Images auf der Zielplattform

Nachdem Sie alle Buildprobleme behoben und ein Run-Time Image generiert haben, können Sie Windows Embedded CE auf dem Zielgerät bereitstellen. Für diese Aufgabe stehen Ihnen mehrere Methoden zur Verfügung. Die ausgewählte Methode hängt vom Startprozess ab, mit dem Windows Embedded CE auf dem Zielgerät geladen wird. Sie können ein Windows Embedded CE 6.0 Run-Time Image auf mehrere Arten starten. Sie können ein Image direkt aus dem ROM starten. In diesem Fall müssen Sie das Run-Time Image auf dem Zielgerät mit einem ROM-Tool bereitstellen. Sie können auch einen Boot Loader verwenden und das Run-Time Image entweder bei jedem Gerätestart herunterladen oder das Image in einem permanenten Speicher speichern. Windows Embedded CE 6.0 R2 umfasst allgemeinen Boot Loader-Code, den Sie an Ihre Anforderungen anpassen können. Das Implementieren des Boot Loaders eines Drittanbieters ist ebenfalls ohne weiteres möglich. Windows Embedded CE kann die Anforderungen beinahe aller Startumgebungen erfüllen. Neue Run-Time Images können während der Entwicklungsphase und für die Freigabe für den Endbenutzer schnell und einfach heruntergeladen werden.

Nach Abschluss dieser Lektion können Sie:

- Entscheiden, wie ein Run-Time-Image auf einem Zielgerät bereitgestellt wird.
- Platform Builder konfigurieren, um die korrekte Bereitstellungsweise auszuwählen.

Veranschlagte Zeit für die Lektion: 15 Minuten.

Auswählen einer Bereitstellungsmethode

Um ein Run-Time-Image bereitzustellen, müssen Sie eine Verbindung mit dem Zielgerät herstellen. Sie müssen hierzu mehrere Parameter für die Kommunikation zwischen Platform Builder und dem Gerät konfigurieren.

Die Core Connectivity-Infrastruktur von Windows Embedded CE unterstützt verschiedene Download- und Transportmethoden, um die Anforderungen von Hardwareplattformen mit unterschiedlichen Kommunikationsfähigkeiten zu erfüllen. Um die Kommunikationsparameter für das Zielgerät festzulegen, wählen Sie in Visual Studio im Menü **Target** die Option **Connectivity Options** aus, um das Dialogfeld **Target Device Connectivity Options** zu öffnen. Platform Builder zeigt im Listenfeld **Target Device** standardmäßig das Gerät **CE Device** an (siehe Abbildung 2.7). Sie können jedoch Geräte mit eindeutigen Namen erstellen, indem Sie auf **Add Device** klicken.

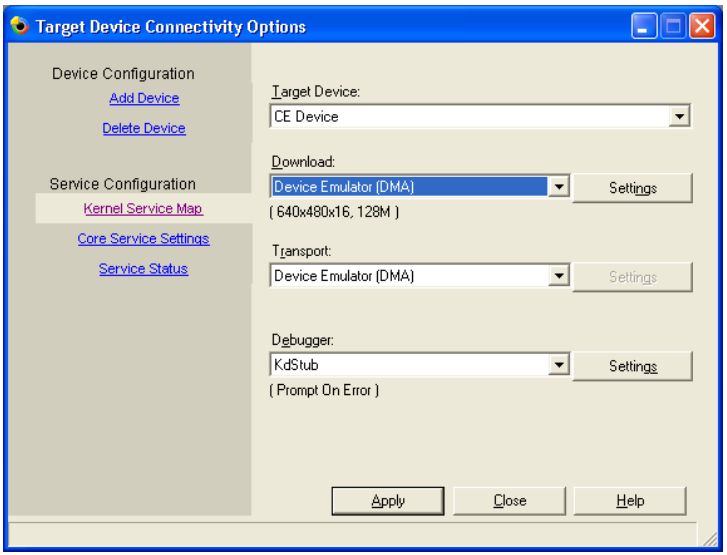


Abbildung 2.7 Das Dialogfeld **Target Device Connectivity Options**

Downloadschicht

Klicken Sie auf **Settings** neben dem Listenfeld **Download**, um den Downloaddienst zu konfigurieren, der das Run-Time Image auf dem Zielgerät lädt. Die Core Connectivity-Infrastruktur unterstützt für die Bereitstellung eines Run-Time Images folgende Downloadoptionen:

- **Ethernet** Lädt das Run-Time Image über eine Ethernet-Verbindung. Klicken Sie auf **Settings**, um den Ethernet-Downloaddienst zu konfigurieren. Der Entwicklungscomputer und das Zielgerät müssen sich im gleichen Subnetz befinden, damit Sie die Verbindung mit dem Zielgerät herstellen können.
- **Serial** Lädt das Run-Time Image über eine RS232-Verbindung. Klicken Sie auf **Settings**, um den Port, die Baudrate und andere serielle Kommunikationsparameter zu konfigurieren.
- **Device Emulator (DMA)** Lädt das Run-Time Image über DMA (Direct Memory Access) auf einen Geräteemulator. Klicken Sie auf **Settings**, um den Geräteemulator zu konfigurieren.
- **USB** Lädt das Run-Time Image über eine USB-Verbindung (Universal Serial Bus). Sie müssen keine Einstellungen konfigurieren.
- **Image Update** Aktualisiert das Image im Flash-Speicher des Geräts. Sie müssen keine Einstellungen konfigurieren.

- **None** Wählen Sie diese Option aus, wenn Sie das Run-Time Image weder laden noch aktualisieren möchten.

Transportschicht

Nachdem Sie das Run-Time Image auf das Remotegerät übertragen haben, können Sie das Gerät zuordnen, wenn KITL (Kernel Independent Transport Layer) im OS Design aktiviert ist. Der ausgewählte Kerneltransportdienst sollte mit dem im Listenfeld **Download** angegebenen Downloaddienst übereinstimmen. Die Core Connectivity-Infrastruktur unterstützt folgende Transportschichtoptionen:

- **Ethernet** Kommuniziert über eine Ethernet-Verbindung mit dem Zielgerät. Die Verbindung verwendet die gleichen Einstellungen wie der Downloaddienst.
- **Serial** Kommuniziert über eine RS232-Verbindung mit dem Zielgerät. Die Verbindung verwendet die gleichen Einstellungen wie der Downloaddienst.
- **Device Emulator (DMA)** Kommuniziert über DMA (Direct Memory Access) mit dem Geräteemulator.
- **USB** Kommuniziert über eine USB-Verbindung mit dem Zielgerät.
- **None** Deaktiviert die Kommunikation mit dem Zielgerät.

Debugger-Optionen

Wenn die Unterstützung für einen oder mehrere Debugger im OS Design aktiviert ist, werden die Namen der Debugger als Optionen im Listenfeld **Debugger** aufgeführt. Standardmäßig sind folgende Debugger-Optionen verfügbar:

- **Sample Device Emulator eXDI2 Driver** Ein Extensible Resource Identifier (XRI) Data Interchange (XDI) Treiber in Windows Embedded CE 6.0 R2. XDI ist eine Standard-Hardwaredebugschnittstelle.
- **KdStub** Der Kerneldebugger. KdStub steht für Kernel Debugger Stub, der den Platform Builder und Visual Studio anweist, den Softwaredebugger zu verwenden.
- **CE Dump File Reader** Wenn Sie das Katalogelement Error Report Generator zum OS Design hinzugefügt haben, können Sie diese Option für das Postmortem-Debuggen verwenden.
- **None** Wählen Sie diese Option aus, wenn Sie keine Debugger verwenden möchten.

Zuordnen eines Geräts

Nachdem Sie die Geräteverbindung konfiguriert haben, können Sie das Run-Time Image unter Verwendung der Core Connectivity-Infrastruktur auf das Zielgerät oder den Geräteemulator übertragen. Führen Sie hierzu in Visual Studio 2005 den Befehl **Attach Device** im Menü **Target** aus. Sie müssen das Gerät zuordnen, damit der Platform Builder das Run-Time Image laden kann, auch wenn Sie KITL oder die Core Connectivity-Infrastruktur nicht zum Debuggen verwenden.

Im Anschluss an den Image-Download wird der Startprozess ausgeführt, KITL wird aktiviert und Sie können die Betriebssystemkomponenten und Anwendungsprozesse mit dem Kerneldebugger debuggen. Mit KITL können Sie außerdem die Remotetools in Visual Studio mit Platform Builder im Menü **Target** verwenden. Diese Tools umfassen den File Viewer zum Interagieren mit dem Dateisystem des Geräts, den Registrierungs-Editor für den Zugriff auf die Registrierungseinstellungen des Geräts, den Systemmonitor zum Analysieren der Ressourcenauslastung und Antwortzeiten und den Kernel Tracker sowie andere Tools zum Anzeigen detaillierter Systeminformationen. Weitere Informationen zum Systemdebuggen finden Sie in Kapitel 4.

Zusammenfassung

Windows Embedded CE unterstützt das Bereitstellen von Run-Time Images über zahlreiche Geräteverbindungen, um die Anforderungen von Hardwareplattformen mit unterschiedlichen Fähigkeiten zu erfüllen, einschließlich Ethernet-Verbindungen, serielle Verbindungen, DMA und USB-Verbindungen. Beispielsweise ist DMA die beste Wahl, um CE 6.0 R2 auf einem Geräteemulator bereitzustellen. Nachdem Sie die Kommunikationsparameter konfiguriert haben, können Sie Windows Embedded CE bereitstellen, indem Sie in Visual Studio 2005 mit Platform Builder im Menü **Target** auf **Attach Device** klicken.

PRÜFUNGSTIPP

Um die Zertifizierungsprüfung zu bestehen, müssen Sie mit den verschiedenen Methoden zum Bereitstellen eines Windows Embedded CE Run-Time Images vertraut sein. Stellen Sie sicher, dass Sie ein Run-Time Image für einen Geräteemulator bereitstellen können.

Lab 2: Erstellen und Bereitstellen eines Run-Time Images

In diesem Lab erstellen Sie ein OS Design basierend auf dem Device Emulator BSP, stellen das OS Design bereit, analysieren die Buildinformationen im Visual Studio-Fenster **Output**, um den Start der verschiedenen Buildphasen zu identifizieren, und konfigurieren eine Verbindung mit einem Zielgerät, um das Run-Time Image herunterzuladen. Sie ändern die Geräteemulatorkonfiguration, um eine höhere Bildschirmauflösung zu unterstützen und die Netzwerkkommunikation zu aktivieren. Anschließend laden Sie das Run-Time Image herunter und ordnen dieses mit dem Kerneldebugger dem Zielgerät zu, um den Windows Embedded CE-Startprozess zu überprüfen. Um das ursprüngliche OS Design in Visual Studio zu erstellen, verwenden Sie die in Kapitel 1 beschriebenen Verfahren.

HINWEIS Detailierte schrittweise Anleitungen

Um die Verfahren in diesem Lab erfolgreich auszuführen, lesen Sie das Dokument Detailed Step-by-Step Instructions for Lab 2 im Begleitmaterial.

Erstellen eines Run-Time Images für ein OS Design

1. Nachdem Sie Lab 1 abgeschlossen haben, wählen Sie die Befehle **Sysgen** unter **Advanced Build** im Menü **Build** in Visual Studio aus (siehe Abbildung 2.8). Sie können auch die Option **Build Solution** im Menü **Build** auswählen, um den Build mit Sysgen zu starten.

TIPP Sysgen-Vorgänge

Sysgen-Vorgänge können bis zu 30 Minuten dauern. Um Zeit zu sparen, führen Sie Sysgen nicht bei jeder Änderung des OS Designs aus. Führen Sie Sysgen aus, nachdem Sie alle gewünschten Komponenten hinzugefügt und entfernt haben.

2. Verfolgen Sie den Buildprozess im Fenster **Output**. Überprüfen Sie die Buildinformationen, um die SYSGEN-, BUILD-, BUILDREL- und MAKEIMG-Schritte zu identifizieren. Drücken Sie Strg+F, um das Dialogfeld **Find And Replace** zu öffnen und suchen Sie nach folgendem Text, der den Start dieser Phasen anzeigt:
   a. **Starting Sysgen Phase For Project** Der SYSGEN-Schritt wird gestartet.
   b. **Build Started With Parameters** Der BUILD-Schritt wird gestartet.
   c. ***C:\WINCE600\Build.log*** Der BUILDREL-Schritt wird gestartet.

d. BLDDEMO: Calling Makeimg—Please Wait Der MAKEIMG-Schritt wird gestartet.

3. Öffnen Sie den Ordner *C:\Wince600* im Windows Explorer. Stellen Sie sicher, dass die Dateien *Build.** vorhanden sind.
4. Öffnen Sie die *Build.**-Dateien in einem Texteditor, beispielsweise Notepad, und überprüfen Sie den Inhalt der Dateien.

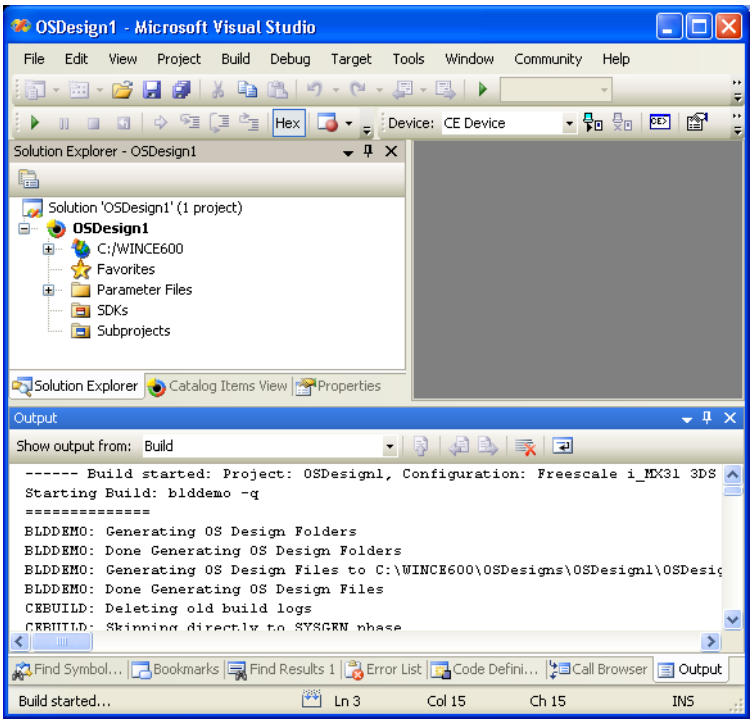


Abbildung 2.8 Erstellen eines OS Designs

Konfigurieren der Verbindungssoptionen

1. Wählen Sie in Visual Studio im Menü **Target** die Option **Connectivity Options** aus, um das Dialogfeld **Target Device Connectivity Options** zu öffnen.
2. Stellen Sie sicher, dass im Listenfeld **Target Device** die Option **CE Device** ausgewählt ist.
3. Wählen Sie **Device Emulator (DMA)** im Listenfeld **Download** aus.
4. Wählen Sie **Device Emulator (DMA)** im Listenfeld **Transport** aus.

5. Wählen Sie **KdStub** im Listenfeld **Debugger** aus (siehe Abbildung 2.9).

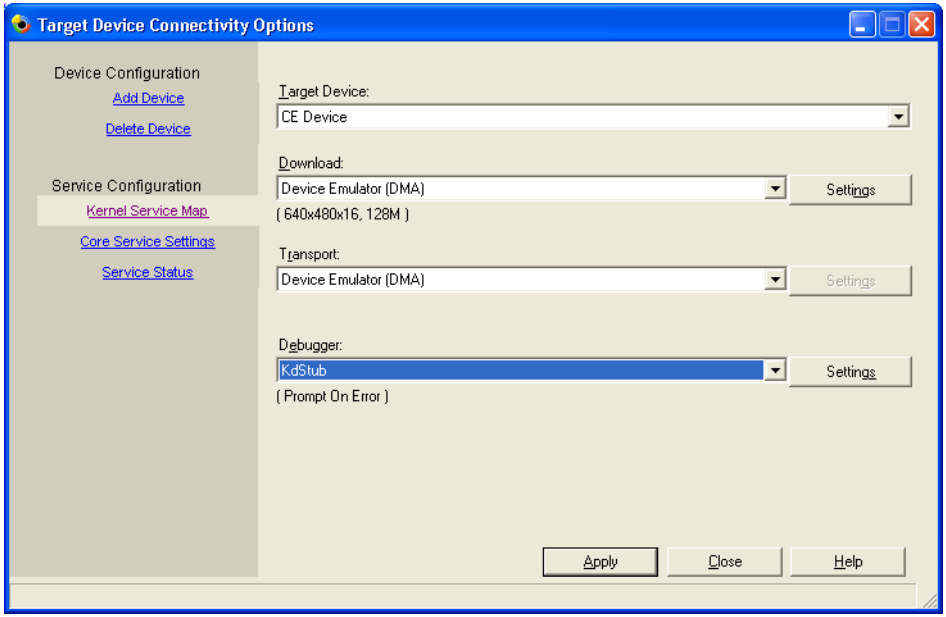


Abbildung 2.9 Festlegen der Optionen im Dialogfeld Target Device Connectivity Options

Ändern der Emulatorkonfiguration

1. Klicken Sie neben dem Listenfeld Download auf Settings.
2. Wählen Sie im Dialogfeld **Emulator Properties** die Registerkarte **Display** aus.
3. Ändern Sie die Bildschirmbreite in ***640*** Pixel und die Bildschirmhöhe in ***480*** Pixel.
4. Wechseln Sie zur Registerkarte **Network**.
5. Aktivieren Sie das Kontrollkästchen **Enable NE2000 PCMCIA Network Adapter And Bind To**, wählen Sie die Option **Connected Network Card** im Listenfeld aus und klicken Sie auf **OK** (siehe Abbildung 2.10).
6. Klicken Sie auf **Apply**, um die neue Gerätekonfiguration zu speichern.
7. Klicken Sie auf **Close**, um das Dialogfeld **Target Device Connectivity Options** zu schließen.

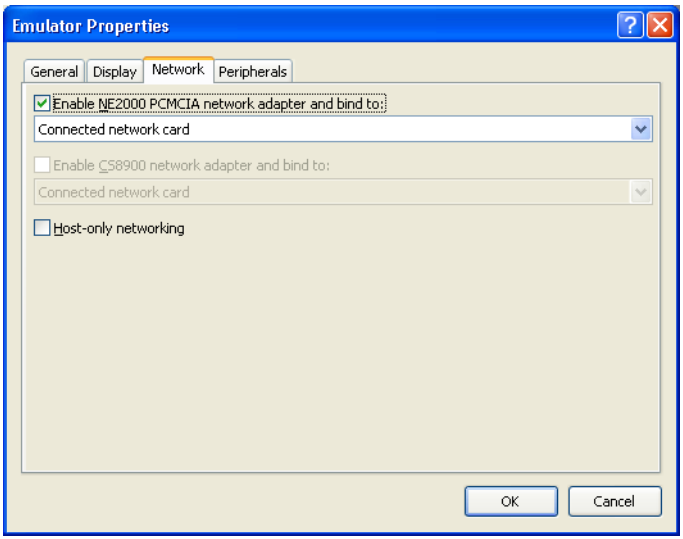


Abbildung 2.10 Geräteemulator-Netzwerkoptionen

Testen eines Run-Time-Images auf dem Geräteemulator

1. Klicken Sie in Visual Studio im Menü **Target** auf **Attach Device**.
2. Überprüfen Sie, ob das Run-Time Image auf das Zielgerät heruntergeladen wird. Der Download kann mehrere Minuten dauern.
3. Verfolgen Sie die Debugmeldungen während des Startprozesses im Visual Studio-Fenster **Output**.
4. Warten Sie bis der Startprozess abgeschlossen ist, um mit dem Geräteemulator zu arbeiten und die Features des OS Designs zu testen (siehe Abbildung 2.11).

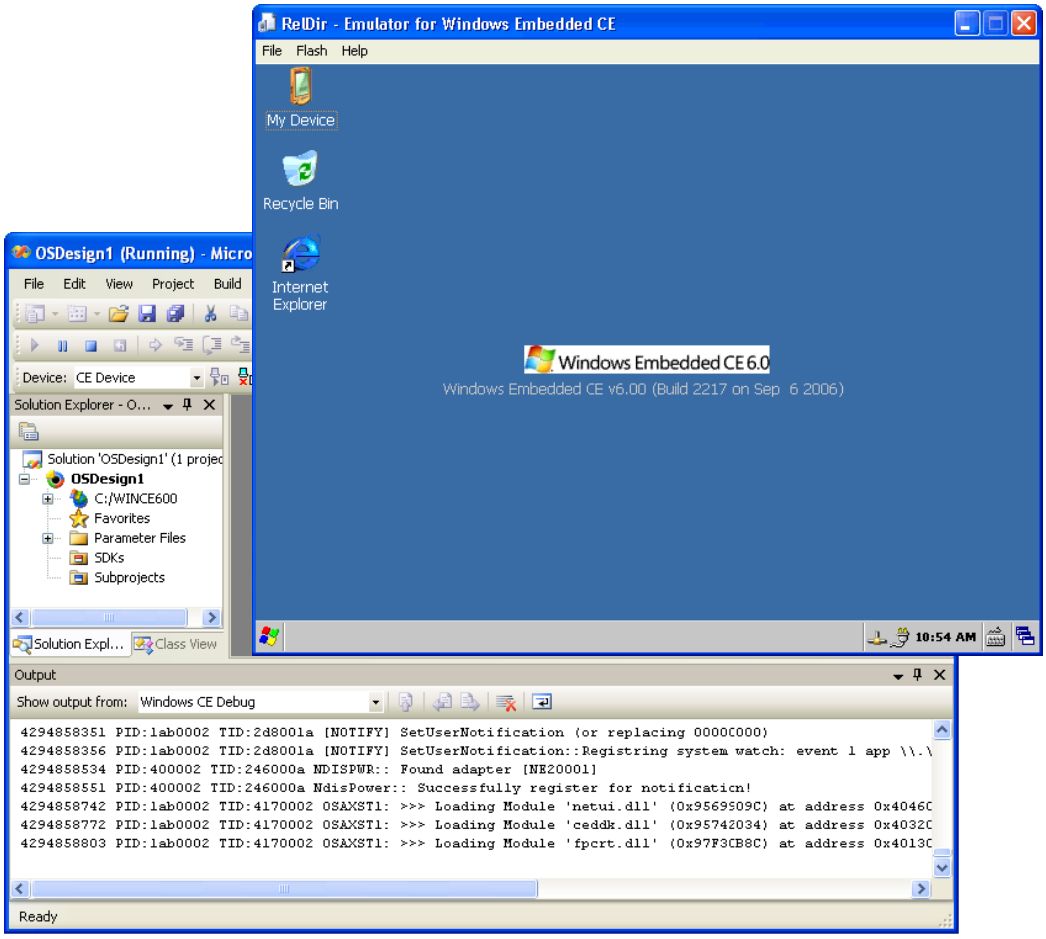


Abbildung 2.11 Windows Embedded CE-Geräteemulator

Lernzielkontrolle

Der Windows Embedded CE-Buildprozess umfasst mehrere Phasen und verwendet zahlreiche Build- und Run-Time Image-Konfigurationsdateien, um den Quellcode zu kompilieren und das Run-Time Image zu erstellen. In der Kompilierungsphase werden die .exe-Dateien, statischen Bibliotheken, DLLs und binären Ressourcendateien (.res) für das BSP und die Teilprojekte generiert. In der Sysgen-Phase wird der Quellcode basierend auf den SYSGEN-Variablen im Ordner Public für die im OS Design ausgewählten Katalogelemente gefiltert und kopiert. Außerdem werden in dieser Phase die Konfigurationsdateien für das Run-Time Image erstellt. In der Release Copy-Phase werden die Dateien, die zum Erstellen des Run-Time Images erforderlich sind, aus dem BSP und den Teilprojekten in das Releaseverzeichnis kopiert. In der Make Run-Time Image-Phase wird das Run-Time Image aus dem Inhalt des Releaseverzeichnisses entsprechend den Einstellungen in den .bib-, .reg-, .db- und .dat-Dateien erstellt.

Sie können den Buildprozess überprüfen, indem Sie die Informationen analysieren, die der Platform Builder generiert und in den Dateien *Build.log*, *Build.wrn* und *Build.err* protokolliert. Die Datei *Build.log* enthält detaillierte Informationen zu den Buildbefehlen, die während der Buildphasen ausgeführt werden. Die Dateien *Build.wrn* und *Build.err* enthalten die gleichen Informationen, aber gefiltert nach Warnungen und Fehler, die während des Buildprozesses auftreten. Sie müssen diese Dateien nicht in Notepad öffnen, da die Statusinformationen und Fehlermeldungen in Visual Studio angezeigt werden. Die Fenster **Output** und **Error List** ermöglichen den Zugriff auf diese Informationen.

Buildfehler können verschiedene Ursachen haben. Die häufigsten Ursachen sind Kompilierungs- und Linker-Fehler. Das Ausführen eines Buildbefehls in einer falsch initialisierten Buildumgebung verursacht beispielsweise einen Linker-Fehler, wenn die Umgebungsvariablen, die die Bibliothekenverzeichnisse in der Sources-Datei angeben, auf ein ungültiges Verzeichnis verweisen. Andere wichtige Buildkonfigurationsdateien, beispielsweise Dirs-Dateien und *Makefile.def*, hängen ebenfalls von SYSGEN-Variablen und Umgebungsvariablen in Bedingungsanweisungen und Verzeichnispfaden ab.

Nachdem Sie das Run-Time-Image generiert haben, können Sie Windows Embedded CE auf einem Zielgerät bereitstellen. Sie müssen hierzu eine Geräteverbindung konfigurieren, die auf der Core Connectivity-Infrastruktur basiert. Der letzte Bereitstellungsschritt besteht im Klicken auf den Befehl **Attach Device** im Menü **Target** in Visual Studio mit Platform Builder für Windows Embedded CE 6.0 R2.

Sie müssen mit folgenden Konfigurationsdateien vertraut sein, da diese den Windows Embedded CE-Buildprozess steuern:

- **Binary Image Builder-Datei (.bib)** Konfiguriert das Speicherlayout und legt die Dateien fest, die in das Run-Time Image einbezogen werden.
- **Registrierungsdatei (.reg)** Initialisiert die Systemregistrierung auf dem Zielgerät.
- **Datenbankdatei (.db)** Konfiguriert den Standardobjektspeicher.
- **Dateisystemdatei (.dat)** Initialisiert das RAM-Dateisystemlayout zur Startzeit.
- **Dirs-Datei** Legt fest, welche Verzeichnisse in den Buildprozess einbezogen werden.
- **Sources-Datei** Definiert die Vorverarbeitungsdirektiven, Befehle, Makros und andere Anweisungen für den Compiler und Linker. Ersetzt die Makefile-Dateien in Visual Studio durch Platform Builder IDE.
- **Makefile-Datei** Verweist auf die Standarddatei *Makefile.def* und sollte nicht bearbeitet werden.

Schlüsselbegriffe

Kennen Sie die Bedeutung der folgenden wichtigen Begriffe? Sie können Ihre Antworten überprüfen, wenn Sie die Begriffe im Glossar am Ende dieses Buches nachschlagen.

- Sysgen
- Build Rel
- Flaches Releaseverzeichnis
- Verbindungsoptionen
- KITL

Empfohlene Vorgehensweise

Um die in diesem Kapitel behandelten Prüfungsschwerpunkte erfolgreich zu beherrschen, sollten Sie die folgenden Aufgaben durcharbeiten.

Starten Sie den Buildprozess über die Befehlszeile

Um Ihre Kenntnisse des Windows Embedded CE-Buildprozesses zu vertiefen, führen Sie folgende Schritte aus:

1. **Sysgen-Prozess** Führen Sie **sysgen –q** in der Befehlszeile aus, nachdem Sie in einem Katalogelement eine Umgebungsvariable festgelegt haben.
2. **Buildprozess** Wechseln Sie über die Befehlszeile zum aktuellen BSP-Ordner (%_TARGETPLATROOT%) und führen Sie den Befehl build-c mit und ohne die WINCEREL-Umgebungsvariable mit dem Wert 1 aus (set WINCEREL=1). Überprüfen Sie den Inhalt des Ordners *%_FLATRELEASEDIR%* vor und nach dem Build.

Stellen Sie Run-Time Images bereit

Stellen Sie ein Windows Embedded CE Run-Time Image auf einem Geräteemulator bereit, indem Sie unterschiedliche Download-, Transport- und Debugger-Einstellungen für das Zielgerät im Platform Builder festlegen.

Klonen Sie eine öffentliche Katalogkomponente manuell

Klonen Sie eine Komponente im Ordner *%_WINCEROOT%\Public*, indem Sie die Quelldateien in ein BSP kopieren (siehe Kapitel 1). Führen Sie den Befehl **sysgen_capture** aus, um eine Sources-Datei zu erstellen, die die Komponentenabhängigkeiten definiert. Ändern Sie die neue Sources-Datei, um die Komponente als Teil des BSP zu erstellen. Detaillierte schrittweise Anweisungen zum Ausführen dieser fortgeschrittenen Entwicklungsaufgaben finden Sie im Abschnitt „Using the Sysgen Capture Tool“ in der Platform Builder für Microsoft Windows Embedded CE-Produktdokumentation auf der Microsoft MSDN®-Website unter *http://msdn2.microsoft.com/en-us/library/aa924385.aspx*.

Kapitel 3

Systemprogrammierung

Die Systemleistung ist ausschlaggebend für die Produktivität des Benutzers. Sie hat außerdem direkten Einfluss auf den Eindruck, den der Benutzer von einem Gerät erhält. Der Benutzer beurteilt die Nützlichkeit eines Geräts häufig basierend auf der Systemleistung und der Benutzerfreundlichkeit der Oberfläche. Eine zu komplexe Benutzeroberfläche kann für den Benutzer verwirrend sein und das Gerät potenziellen Sicherheitsrisiken oder unerwarteten Manipulationen aussetzen. Die Verwendung von inkorrekten APIs oder Anwendungsarchitekturen in einer Multithread-Umgebung kann die Leistung wesentlich beeinträchtigen. Die Leistungsoptimierung und Systemanpassung stellen eine große Herausforderung für die Anbieter von Firmware dar. In diesem Kapitel werden die Tools und bewährte Verfahren beschrieben, mit denen Sie optimale Systemantwortzeiten auf Zielgeräten erreichen können.

Prüfungsziele in diesem Kapitel

- Überwachen und Optimieren der Systemleistung
- Implementieren von Systemanwendungen
- Programmieren mit Threads und Threadsynchronisierungsobjekten
- Implementieren der Ausnahmebehandlung für Treiber und Anwendungen
- Unterstützen der Energieverwaltung auf Systemebene

Bevor Sie beginnen

Damit Sie die Lektionen in diesem Kapitel durcharbeiten können, benötigen Sie:

- Umfassende Kenntnisse der Konzepte für das Echtzeit-Systemdesign, beispielsweise der Scheduler-Funktionen in einem Betriebssystem, der Interrupts und Timer.
- Grundkenntnisse in der Multithread-Programmierung, einschließlich der Synchronisierungsobjekte.
- Einen Entwicklungscomputer, auf dem Microsoft® Visual Studio® 2005 Service Pack 1 und Platform Builder für Microsoft Windows® Embedded CE 6.0 installiert ist.

Lektion 1: Überwachen und Optimieren der Systemleistung

Das Überwachen und Optimieren der Leistung sind wichtige Aufgaben bei der Entwicklung von Small-Footprint-Geräten. Das Optimieren der Systemleistung ist wichtig, da sowohl die Anzahl komplexer Anwendungen als auch die Anforderung für intuitive, und somit ressourcenintensive Benutzeroberflächen, zunimmt. Die Leistungsoptimierung erfordert, dass Firmwarearchitekten und Softwareentwickler die Ressourcenbelegung in ihren Systemkomponenten und Anwendungen einschränken, damit die anderen Komponenten und Anwendungen die verfügbaren Ressourcen verwenden können. Unabhängig davon, ob Gerätetreiber oder Benutzeranwendungen entwickelt werden, können optimierte Verarbeitungsalgorithmen die Prozessorauslastung reduzieren, und effiziente Datenstrukturen können die Arbeitsspeicherbelegung verringern. Auf allen Systemebenen sind Tools zum Identifizieren von Leistungsproblemen in Treibern, Anwendungen und anderen Komponenten verfügbar.

Nach Abschluss dieser Lektion können Sie:

- Die Latenz einer ISR (Interrupt Service Routine) identifizieren.
- Die Leistung eines Windows Embedded CE-Systems verbessern.
- Informationen zur Systemleistung protokollieren und analysieren.

Veranschlagte Zeit für die Lektion: 20 Minuten

Echtzeit-Leistung

Treiber-, Anwendungs- und OAL-Code (OEM Adaptation Layer) beeinflusst die Systemleistung und die Echtzeit-Leistung. Obwohl Windows Embedded CE in Konfigurationen mit und ohne Echtzeitverarbeitungsunterstützung eingesetzt werden kann, ist es wichtig, zu berücksichtigen, dass die Verwendung von Komponenten und Anwendungen ohne Echtzeitverarbeitungsunterstützung die Systemleistung eines Echtzeitbetriebssystems beeinträchtigen können. Beachten Sie beispielsweise, dass das Demand Paging, Geräte-E/A und die Energieverwaltung nicht für Echtzeitgeräte ausgelegt sind. Setzen Sie diese Features vorsichtig ein.

Demand Paging

Demand Paging unterstützt die Speicherfreigabe zwischen mehreren Prozessen auf Geräten mit begrenzter RAM-Kapazität. Wenn Demand Paging aktiviert ist, verwirft

und entfernt Windows Embedded CE die Speicherseiten aktiver Prozesse, wenn nicht genügend Arbeitsspeicher verfügbar ist. Um den Code aller aktiven Prozesse im Speicher beizubehalten, deaktivieren Sie Demand Paging für das gesamte Betriebssystem oder ein bestimmtes Modul, beispielsweise eine DLL (Dynamic-Link Library) oder einen Gerätetreiber.

Sie können Demand Paging unter Verwendung einer der folgenden Methoden deaktivieren:

- **Betriebssystem** Bearbeiten Sie die Datei *Config.bib* und aktivieren Sie die Option ROMFLAGS im Abschnitt CONFIG.
- **DLLs** Laden Sie die DLL mit der Funktion LoadDriver, anstatt der Funktion *LoadLibrary*, in den Speicher.
- **Gerätetreiber** Fügen Sie das Flag DEVFLAGS_LOADLIBRARY zum Registrierungseintrag Flags für den Treiber hinzu. Dieses Flag stellt sicher, dass der Geräte-Manager die Funktion *LoadLibrary* anstatt die Funktion *LoadDriver* zum Laden des Treibers verwendet.

Windows Embedded CE ordnet den Speicher wie gewöhnlich zu, aber leert diesen nicht automatisch, wenn Demand Paging deaktiviert ist.

Systemzeitgeber

Der Systemzeitgeber ist ein Hardwarezeitgeber, der pro Millisekunde einen Systemtick generiert. Der Systemscheduler bestimmt mittels des Zeitgebers, welche Threads zu welchem Zeitpunkt ausgeführt werden. Ein Thread ist die kleinste ausführbare Einheit in einem Prozess, dem zum Ausführen von Anweisungen im Betriebssystem eine bestimmte Prozessorzeit zugeordnet wird. Mit der *Sleep*-Funktion können Sie einen Thread für eine angegebene Zeitdauer anhalten. Der Mindestwert für die *Sleep*-Funktion ist 1 (**Sleep(1)**), der den Thread für ca. 1 Millisekunde anhält. Eine *Sleep*-Einheit beträgt jedoch nicht genau 1 Millisekunde, da sie den aktuellen Systemtick plus den Rest der vorherigen Zeiteinheit umfasst. Die *Sleep*-Zeit ist außerdem mit der Priorität des Threads verknüpft. Die Threadpriorität bestimmt die Reihenfolge, in der das Betriebssystem die Threads auf dem Prozessor ausführt. Aus diesem Grund sollten Sie die *Sleep*-Funktion nicht verwenden, wenn Sie genaue Zeitgeber für Echtzeit-Anwendungen benötigen. Verwenden Sie stattdessen dedizierte Zeitgeber mit Interrupts oder Multimedia-Zeitgeber.

Energieverwaltung

Die Energieverwaltung kann sich auf die Systemleistung auswirken. Wenn der Prozessor in den Leerlauf übergeht, verursachen alle von einem Peripheriegerät oder dem Systemscheduler generierten Interrupts, dass der Prozessorstatus gewechselt, der vorherige Kontext wiederhergestellt und der Scheduler aufgerufen wird. Der Energiekontextwechsel ist ein zeitaufwendiger Prozess. Weitere Informationen zu den Energieverwaltungsfeatures in Windows Embedded CE finden Sie im Abschnitt "Power Management" in der Windows Embedded CE 6.0-Dokumentation auf der Microsoft MSDN®-Website unter *http://msdn2.microsoft.com/en-us/library/aa923906.aspx*.

Systemspeicher

Der Kernel reserviert und verwaltet den Systemspeicher für Heaps, Prozesse, Critical Sections, Mutexe, Events und Semaphores. Der Kernel gibt den Systemspeicher jedoch nicht vollständig frei, wenn diese Kernelobjekte abgerufen werden. Stattdessen verwendet der Kernel den Systemspeicher erneut für die nächste Zuordnung. Da die Wiederverwendung des zugeordneten Speichers schneller ist, initialisiert der Kernel während des Startprozesses den Systemspeicherpool und ordnet weiteren Speicher nur dann zu, wenn im Pool kein Speicher mehr verfügbar ist. Die Systemleistung kann abnehmen, abhängig davon, wie die Prozesse den virtuellen Speicher, die Heapobjekte und den Stack verwenden.

APIs ohne Echtzeitunterstützung

Beachten Sie beim Aufrufen von System-APIs oder GWES-APIs (Graphical Windows Event System), dass einige APIs von den Features ohne Echtzeitunterstützung abhängen, bespielweise für die Fensteranzeige. Das Aufrufen von APIs ohne Echtzeitunterstützung kann die Systemleistung drastisch reduzieren. Sie sollten deshalb sicherstellen, dass Ihre APIs in Echtzeit-Anwendungen Echtzeit-kompatibel sind. Andere APIs, beispielsweise für den Zugriff auf ein Dateisystem oder Hardware, können die Leistung ebenfalls beeinträchtigen, da sie möglicherweise Methoden blockieren, beispielsweise Mutexe oder Critical Sections, um Ressoucen zu schützen.

HINWEIS APIs ohne Echtzeitunterstützung

APIs ohne Echtzeitunterstützung können die Echtzeitleistung beeinträchtigen. Die Win32® API-Dokumentation enthält bedauerlicherweise nur wenige Informationen zu Echtzeitproblemen. Praktische Erfahrung und das Testen der Leistung helfen Ihnen bei der Auswahl der richtigen Funktionen.

Tools für die Beurteilung der Echtzeitleistung

Windows Embedded CE umfasst zahlreiche Tools für die Leistungsüberwachung und Problembehandlung, mit denen Sie die Auswirkungen der Win32 APIs auf die Systemleistung auswerten können. Mit diesen Tools können Sie eine ineffiziente Speicherbelegung identifizieren, beispielsweise wenn eine Anwendung den zugeordneten Systemspeicher nicht freigibt.

Die folgenden Windows Embedded CE-Tools sind insbesondere zum Auswerten der Echtzeitleistung der Systemkomponenten und Anwendungen nützlich:

- **ILTiming** Wertet ISR-Latenzen (Interrupt Service Routine) und IST-Latenzen (Interrupt Service Thread) aus.
- **OSBench** Wertet die Systemleistung aus, indem die Zeitdauer, die der Kernel zum Verwalten der Kernelobjekte benötigt, überwacht wird.
- **Remote Performance Monitor** Wertet die Systemleistung aus, einschließlich die Speicherbelegung, den Netzwerkdurchsatz und andere Aspekte.

ILTiming (Interrupt Latency Timing)

Das Tool ILTiming ist insbesondere für OEMs (Original Equipment Manufacturer) nützlich, die die ISR- und IST-Latenzen auswerten möchten. ILTiming ermöglicht das Auswerten der zum Aufrufen eines ISR erforderlichen Zeitdauer nach einem Interrupt (ISR-Latenz) und der Zeitdauer zwischen dem ISR und dem Start eines IST (IST-Latenz). Dieses Tool verwendet einen hardwarebasierten Systemzeitgeber (Tick Timer). Es ist jedoch auch möglich, alternative Zeitgeber (Leistungsindikatoren) zu verwenden.

HINWEIS Hardwarezeitgeber-Einschränkungen
Nicht alle Hardwareplattformen unterstützen das Tool ILTiming.

ILTiming hängt von der Funktion *OALTimerIntrHandler* in OAL ab, um den ISR zum Verwalten der Systemtick-Interrupts zu implementieren. Der Zeitgeber-Interrupthandler speichert die aktuelle Zeit und gibt ein *SYSINTR_TIMING*-Interrupterreignis zurück, das von einem ILTiming-Anwendungsthread erwartet wird. Dieser Thread ist der IST. Die zwischen dem Empfang des Interrupts im ISR und dem Empfang des *SYSINTR_TIMING*-Events im IST verstrichene Zeitdauer stellt die vom Tool ILTiming ausgewertete IST-Latenz dar.

Der Quellcode des Tools ILTiming ist im Ordner *%_WINCEROOT%\Public\Common\Oak\Utils* auf Ihrem Entwicklungscomputer gespeichert, wenn Microsoft Platform Builder für Windows Embedded CE 6.0 R2 installiert ist. Das Tool ILTiming unterstützt mehrere Befehlszeilenparameter, über die Sie die IST-Priorität und den Typ unter Verwendung folgender Syntax festlegen können:

iltiming [-i0] [-i1] [-i2] [-i3] [-i4] [-p priority] [-ni] [-t interval] [-n interrupt] [-all] [-o file_name] [-h]

In Tabelle 3.1 sind die ILTiming-Befehlszeilenparameter aufgeführt.

Tabelle 3.1 ILTiming-Parameter

Befehlszeilen-parameter	Beschreibung
-i0	Kein Leerlaufthread. Dieser Parameter entspricht dem Parameter **-ni**.
-i1	Thread Spinning ohne tatsächliche Verarbeitung.
-i2	Thread Spinning, Aufruf von *SetThreadPriority (THREAD_PRIORITY_IDLE)*.
-i3	Zwei Threads; abwechselnd SetEvent und *WaitForSingleObject* mit einem Timeout von 10 Sekunden.
-i4	Zwei Threads; abwechselnd *SetEvent* und *WaitForSingleObject* mit einem unbegrenzten Timeout.
-i5	Thread Spinning, Aufruf von *VirtualAlloc* (64 KB) oder *VirtualFree* oder beiden. Leert den Cache und den TLB (Translation Look-aside Buffer).
-p *priority*	Gibt die IST-Priorität an (0 bis 255). Die Standardeinstellung ist 0 für die höchste Priorität.
-ni	Gibt keinen Leerlauf-Prioritätsthread an. Die Standard-einstellung entspricht der Anzahl der Leerlauf-Prioritäts-threadspins. Dieser Parameter entspricht dem Parameter -i0.
-t *interval*	Gibt das *SYSINTR_TIMING*-Zeitgeberintervall mit Ticks in Millisekunden an. Die Standardeinstellung ist 5.

Tabelle 3.1 ILTiming-Parameter (Fortsetzung)

Befehlszeilen-parameter	Beschreibung
-n *interrupt*	Gibt die Anzahl der Interrupts an. Mit diesem Parameter können Sie die Zeitdauer für den Test festlegen. Die Standardeinstellung ist 10.
-all	Gibt an, dass alle Daten ausgegeben werden. Standardmäßig wird nur die Zusammenfassung angezeigt.
-o *file_name*	Gibt an, dass die Daten in einer Datei ausgegeben werden. Standardmäßig wird die Ausgabe im Debuggerfenster angezeigt.

HINWEIS **Leerlauf-Threads**

ILTiming kann Leerlauf-Threads (Befehlszeilenparameter- i1, -i2, -i3 und -i4) erstellen, um Aktivitäten im System zu generieren. Dies ermöglicht den nicht unterbrochenen Kernelaufruf, der vor der Verarbeitung des IST beendet werden muss. Diese Funktion ist nützlich, um Leerlauf-Threads in Hintergrundaufgaben zu aktivieren.

OSBench (Operating System Benchmark)

Das Tool OSBench hilft Ihnen beim Auswerten der Systemleistung, indem die Zeitdauer ermittelt wird, die der Kernel zum Verwalten der Kernelobjekte benötigt. OSBench stellt basierend auf dem Scheduler die Zeitgebungsauswertungen mittels Leistungszeitmessungstests zusammen. Ein Leistungszeitmessungstest überprüft die für Kernelbasisvorgänge erforderliche Zeitdauer, beispielsweise für die Threadsynchronisierung.

OSBench ermöglicht das Überwachen der Zeitinformationen für folgende Kernelvorgänge:

- Abrufen oder Freigeben einer Critical Section.
- Warten oder Signalisieren eines Events.
- Erstellen eines Semaphores oder Mutex.
- Zurückgeben eines Threads.
- Aufrufen von System-APIs.

HINWEIS OSBench-Test

Um Leistungsprobleme in unterschiedlichen Systemkonfigurationen zu identifizieren, verwenden Sie OSBench zusammen mit einer Stresstestsuite, beispielsweise den Microsoft Windows CE Test Kit (CETK).

Das Tool OSBench unterstützt mehrere Befehlszeilenparameter, mit denen Sie entsprechend folgender Syntax die Zeitsamples für Kernelvorgänge zusammenstellen können:

osbench [-all] [-t test_case] [-list] [-v] [-n number] [-m address] [-o file_name] [-h]

In Tabelle 3.2 sind die OSBench-Befehlszeilenparameter aufgeführt.

Tabelle 3.2 OSBench-Parameter

Befehlszeilen-parameter	Beschreibung
-all	Führt alle Test aus (standardmäßig werden ausschließlich die mit dem Parameter **-t** angegebenen Tests ausgeführt). Test-ID 0: Critical Sections Test-ID 1: Eventaktivierung Test-ID 2: Semaphore-Freigabe/Aufruf Test-ID 3: Mutex Test-ID 4: Absichtlicher Vorrang Test-ID 5: PSL API-Aufrufoverhead Test-ID 6: Verknüpfte APIs (decrement, increment, testexchange, exchange)
-t *test_case*	ID des auszuführenden Tests (für jeden Test ist ein separates **-t** erforderlich)
-list	Listet die Test-IDs mit einer Beschreibung auf
-v	Ausführlich: Zeigt zusätzliche Auswertungsinformationen an
-n *number*	Anzahl der Samples pro Test (Standard = 100)
-m *address*	Virtuelle Adresse zum Schreiben der Markerwerte (Standard = <*none*>)

Tabelle 3.2 OSBench-Parameter (Fortsetzung)

Befehlszeilen-parameter	Beschreibung
-o *file_name*	Ausgabe mit durch Kommas getrennten Werten (CSV-Datei) (Standard: Ausgabe nur in Debug)

Überprüfen Sie den OSBench-Quellcode, um den Testinhalt zu identifizieren. Der Quellcode ist in folgenden Verzeichnissen gespeichert:

- *%_WINCEROOT%\Public\Common\Oak\Utils\Osbench*
- *%_WINCEROOT%\Public\Common\Oak\Utils\Ob_load*

Sie können die Testergebnisse, die standardmäßig an die Debugausgabe übergeben werden, in eine CSV-Datei umleiten.

HINWEIS OSBench-Anforderungen

Das Tool OSBench verwendet Systemzeitgeber. Der OAL muss deshalb die Funktionen *QueryPerformanceCounter* und *QueryPerformanceFrequency* unterstützen, die in der *OEMInit*-Funktion initialisiert werden.

Remote Performance Monitor

Der Remote Performance Monitor kann sowohl die Echtzeit-Leistung des Betriebssystems als auch die Speicherauslastung, Netzwerklatenzen und andere Elemente überwachen. Jedem Systemelement sind Indikatoren zugeordnet, die Informationen über die Auslastung, Warteschlangenlänge und Verzögerungen bereitstellen. Der Remote Performance Monitor ist ein Remotetool und kann die auf einem Zielgerät generierten Protokolldateien analysieren. Die Anwendung überwacht Geräte, die sowohl noch entwickelt als auch bereits eingesetzt werden, wenn Sie auf ein Gerät zugreifen und die Anwendung bereitstellen können.

Der Remote Performance Monitor überwacht folgende Objekte:

- RAS (Remote Access Server)
- ICMP (Internet Control Message Protocol)
- TCP (Transmission Control Protocol)
- IP (Internet Protocol)

- UDP (User Datagram Protocol)
- Arbeitsspeicher
- Batterie
- System
- Prozess
- Thread

Sie können die Liste mit einer Remote Performance Monitor-Erweiterungs-DLL erweitern. Beispielcode finden Sie im Ordner *%COMMONPROGRAMFILES%\Microsoft Shared\Windows CE Tools\Platman\Sdk\WCE600\Samples\CEPerf*.

Ähnlich wie das Tool Systemmonitor auf einer Windows-Arbeitsstation kann der Remote Performance Monitor Leistungsdiagramme erstellen, Warnungen konfigurieren, die bei festgelegten Schwellwerten ausgelöst werden, nicht formatierte Protokolldateien schreiben und Leistungsberichte kompilieren, die auf den Leistungsobjekten auf dem Zielgerät basieren. In Abbildung 3.1 ist ein Leistungsdiagramm dargestellt.

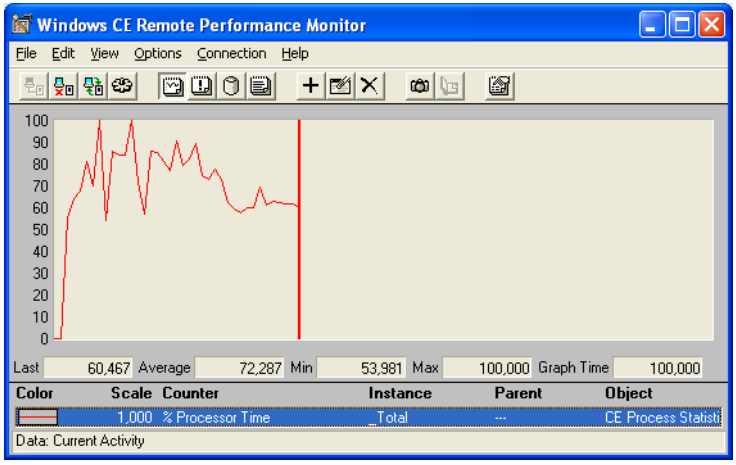


Abbildung 3.1 Ein Leistungsdiagramm im Remote Performance Monitor

Hardwareüberprüfung

ILTiming, OSBench und Remote Performance Monitor erfüllen die meisten Anforderungen an die Leistungsüberwachung. In einigen Fällen sind jedoch möglicherweise andere Methoden zum Zusammenstellen der Systemleistungsinformationen erforderlich. Wenn Sie beispielsweise exakte Interrupt-Latenzzeit-

messungen benötigen oder die Hardwareplattform die für das Tool ILTiming erforderlichen Zeitgeber nicht unterstützt, müssen Sie Messmethoden für die Hardwareleistung verwenden, die auf der GPIO-Schnittstelle (General Purpose Input/Output) des Prozessors und einem Waveform-Generator basieren.

Unter Verwendung eines Waveform-Generators für einen GPIO können Interrupts generiert werden, die über ISRs und ISTs verarbeitet werden. Diese ISRs und ISTs verwenden einen anderen GPIO, um eine Waveform als Antwort auf den empfangenen Interrupt zu generieren. Die zwischen den beiden Waveforms (die Eingangswaveform vom Generator und die Ausgangswaveform vom ISR oder IST) verstrichene Zeitdauer ist die Latenzzeit des Interrupts.

Zusammenfassung

Windows Embedded CE umfasst zahlreiche Tools, die in einer Entwicklungsumgebung verwendet werden können, um die Systemleistung auszuwerten und die Echtzeit-Geräteleistung zu überprüfen. Mit dem Tool ILTiming können Sie die Interruptlatenzen auswerten. Das Tool OSBench ermöglicht Ihnen zu analysieren, wie der Kernel die Systemobjekte verwaltet. Der Remote Performance Monitor erstellt sowohl Diagramme und Protokolle mit Leistungs- und Statistikdaten als auch Berichte über Geräte, die noch entwickelt oder bereits eingesetzt werden. Der Remote Performance Monitor kann Warnungen basierend auf konfigurierbaren Leistungsschwellwerten generieren. Außer diesen Tools können Sie auch die Hardwareüberwachung für die Latenz- und Leistungsbeurteilung verwenden.

Lektion 2: Implementieren von Systemanwendungen

Wie in Kapitel 1 erklärt, ist Windows Embedded CE ein auf Komponenten basierendes Betriebssystem und eine Entwicklungsplattform für zahlreiche Small-Footprint-Geräte. Hierbei kann es sich um Geräte mit beschränktem Zugriff für bestimmte Aufgaben handeln, beispielsweise hochzuverlässige Industriesteuergeräte, aber auch offene Plattformen, die den Zugriff auf das Betriebssystem ermöglichen, einschließlich alle Einstellungen und Anwendungen, wie beispielsweise PDAs (Personal Digital Assistant). Praktisch alle Windows Embedded CE-Geräte benötigen jedoch Systemanwendungen, um eine Benutzerschnittstelle bereitzustellen.

Nach Abschluss dieser Lektion können Sie:

- Eine Anwendung beim Systemstart starten.
- Die Standardshell ersetzen.
- Die Shell anpassen.

Veranschlagte Zeit für die Lektion: 25 Minuten

Übersicht der Systemanwendungen

Entwickler unterscheiden zwischen Systemanwendungen und Benutzeranwendungen, da diese Anwendungen für unterschiedliche Verwendungszwecke ausgelegt sind. In Zusammenhang mit Windows Embedded CE-Geräten bezieht sich der Begriff Systemanwendung auf eine Anwendung, die eine Schnittstelle zwischen dem Benutzer und dem System bereitstellt. Im Gegensatz dazu ist eine Benutzeranwendung ein Programm, das eine Schnittstelle zwischen dem Benutzer und den anwendungsspezifischen Daten bereitstellt. Wie Benutzeranwendungen können Systemanwendungen eine Benutzeroberfläche oder Eingabeaufforderung umfassen. Systemanwendungen werden jedoch normalerweise automatisch mit dem Betriebssystem gestartet.

Starten einer Anwendung beim Systemstart

Sie können Anwendungen so konfigurieren, dass diese automatisch während des Windows Embedded CE-Initialisierungsprozesses gestartet werden. Dieses Feature kann so angepasst werden, dass die Anwendungen entweder vor oder nach dem Laden der Shellbenutzeroberfläche starten. Eine Methode zum Festlegen des Features ist das Ändern mehrerer Registrierungseinstellungen, die das Startverhalten

der Anwendung steuern. Eine andere Methode ist das Erstellen einer Verknüpfung zur Anwendung im Ordner Start, damit die Standardshell die Anwendung startet.

Registrierungsschlüssel HKEY_LOCAL_MACHINE\INIT

Die Windows Embedded CE-Registrierung enthält mehrere Einträge zum Starten der Betriebssystemkomponenten und Anwendungen, beispielsweise des Geräte-Managers und von GWES (Graphical Windows Event System). Diese Registrierungseinträge befinden sich unter dem Registrierungsschlüssel *HKEY_LOCAL_MACHINE\INIT* (siehe Abbildung 3.2). Sie können zusätzliche Einträge erstellen, um die im Run-Time Image enthaltenen Anwendungen auszuführen, damit Sie diese nicht manuell auf dem Zielgerät laden müssen. Unter anderem unterstützt der automatische Start einer Anwendung das Debuggen während der Softwareentwicklung.

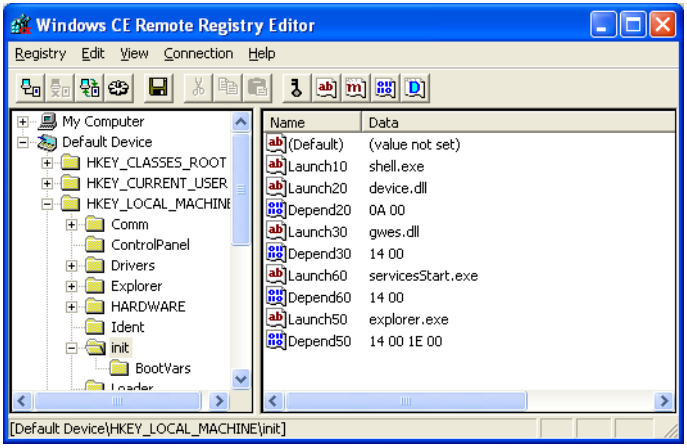


Abbildung 3.2 Der Registrierungsschlüssel *HKEY_LOCAL_MACHINE\INIT*

In Tabelle 3.3 sind drei Registrierungseinträge aufgeführt, die Windows Embedded CE-Komponenten beim Laden des Run-Time Images starten.

Tabelle 3.3 Registrierungsparameter für den Systemstart

Pfad	**HKEY_LOCAL_MACHINE\INIT**		
Komponente	Geräte-Manager	GWES	Explorer
Binär	Launch20= "*Device.dll*"	Launch30= "*Gwes.dll*"	Launch50= "*Explorer.exe*"

Tabelle 3.3 Registrierungsparameter für den Systemstart (Fortsetzung)

Pfad	HKEY_LOCAL_MACHINE\INIT		
Abhängigkeiten	Depend20= hex:0a,00	Depend30= hex:14,00	Depend50= hex:14,00, 1e,00
Beschreibung	Der Registrierungseintrag *LaunchXX* gibt die Binärdatei der Anwendung an und der Registrierungseintrag *DependXX* definiert die Abhängigkeiten zwischen Anwendungen.		

Der Registrierungseintrag *Launch50* in Tabelle 3.3 gibt an, dass die Windows Embedded CE-Standardshell (*Explorer.exe*) erst nach dem Start der Prozesse 0x14 (20) und 0x1E (30) ausgeführt wird (dem Geräte-Manager und GWES). Die Hexadezimalwerte im Eintrag *DependXX* beziehen sich auf die dezimale Startnummer *XX*, die im Namen der *LaunchXX*-Einträge angegeben ist.

Das *SignalStarted* API unterstützt die Verwaltung der Prozessabhängigkeiten zwischen den im Registrierungsschlüssel *HKEY_LOCAL_MACHINE\INIT* registrierten Anwendungen. Die Funktion *SignalStarted* teilt dem Kernel mit, dass die Anwendung gestartet und initialisiert wurde. Dieser Vorgang ist im folgenden Codeausschnitt dargestellt.

```
int WINAPI WinMain(HINSTANCE hInstance,
    HINSTANCE hPrevInstance,
    LPTSTR lpCmdLine,
    int nCmdShow)
    {
        // Perform initialization here...

        // Initialization complete,
        // call SignalStarted...
        SignalStarted(_wtol(lpCmdLine));

        // Perform application work and eventually exit.
        return 0;
    }
```

Die Verarbeitung von Abhängigkeiten ist unkompliziert. Der Kernel ermittelt die Startnummer im *Launch*-Registrierungseintrag, die er als Sequenz-ID verwendet, und übergibt diese als Startparameter in *lpCmdLine* an den *WinMain*-Einsprungspunkt. Die Anwendung führt die erforderliche Initialisierung aus und informiert den Kernel über den Abschluss der Initialisierung, indem sie die Funktion *SignalStarted* aufruft. Der Aufruf der Funktion *_wtol* in der *SignalStarted*-Codezeile bewirkt die

Konvertierung der Startnummer aus einer Zeichenfolge in eine ganze Zahl, da die Funktion *SignalStarted* einen DWORD-Parameter erwartet. Beispielsweise muss der Geräte-Manager den *SignalStarted*-Wert 20 und GWES muss den Wert 30 an den Kernel zurückgeben, um *Explorer.exe* zu starten.

Der Startordner

Wenn Sie auf dem Zielgerät die Standardshell verwenden, können Sie die Anwendung oder eine Verknüpfung zur Anwendung im Ordner *Windows\Start* speichern. *Explorer.exe* startet alle Anwendungen in diesem Ordner.

HINWEIS StartupProcessFolder-Funktion

Verwenden Sie den Ordner *Windows\Start* nur, wenn auf dem Zielgerät die Windows Embedded CE-Standardshell ausgeführt wird. Wenn Sie die Standardshell nicht verwenden, erstellen Sie eine benutzerdefinierte Startanwendung und initialisieren Sie diese beim Systemstart basierend auf den Einträgen im Registrierungsschlüssel *HKEY_LOCAL_MACHINE\INIT*. Beispiele des Codes zum Überprüfen des Startordners und zum Starten der Anwendungen finden Sie in der Datei *Explorer.cpp* im Ordner *%_WINCEROOT%\Public\Shell\OAK\HPC\Explorer\Main*. Sie können die Funktion *StartupProcessFolder* als Ausgangspunkt für Ihre Implementierungen verwenden.

Die Windows Embedded CE-Standardshell kann ausführbare Dateien und Verknüpfungsdateien verarbeiten. Die Windows Embedded CE-Verknüpfungsdateien unterscheiden sich von den Verknüpfungsdateien in Windows XP, aber unterstützen ähnliche Funktionen. CE-Verknüpfungsdateien sind Textdateien mit der Dateierweiterung lnk. Diese Dateien enthalten die Befehlszeilenparameter für die Verknüpfung entsprechend der folgenden Syntax:

nn# command [optional parameters]

Der Platzhalter *nn* steht für die Anzahl der Zeichen gefolgt von einem Pfund-Zeichen (#) und dem eigentlichen Befehl, beispielsweise **27#\Windows\iexplore.exe -home** zum Starten von Internet Explorer® und zum Öffnen der Homepage. Nachdem Sie die .lnk-Datei für das Run-Time Image erstellt und hinzugefügt haben, bearbeiten Sie die Datei *Platform.dat* oder *Project.dat*, um die .lnk-Datei zuzuordnen, wie in folgendem .dat-Dateieintrag:

```
Directory("\Windows\Startup"):-File("Home Page.lnk", "\Windows\homepage.lnk")
```

In Kapitel 2 sind diese Konfigurationsaufgaben ausführlich erklärt.

HINWEIS Startordnereinschränkungen

Der wichtigste Vorteil des Startordners ist, dass die Anwendungen in diesem Ordner das Signal-Started API nicht implementieren müssen, um den Kernel über den erfolgreichen Abschluss des Initialisierungs- und Startprozesses zu benachrichtigen. Dies bedeutet jedoch auch, dass das Betriebssystem die Abhängigkeiten zwischen den Anwendungen nicht verwalten und keine bestimmte Startsequenz erzwingen kann. Das Betriebssystem startet alle Anwendungen im Startordner gleichzeitig.

Verzögerter Start

Eine weitere Option für den automatischen Start von Anwendungen ist die Verwendung von *Services.exe* (Services Host Process). Windows Embedded CE enthält zwar keinen umfassenden Service Control Manager (SCM), aber integrierte Dienste und den Beispieldienst Svcstart, der Anwendungen startet.

Svcstart ist insbesondere nützlich für Anwendungen, die von Systemkomponenten und Diensten abhängig sind, die nicht unmittelbar nach dem Startprozess zur Verfügung stehen. Beispielsweise kann das Abrufen einer IP-Adresse für eine Netzwerkkarte (Network Interface Card, NIC) von einem DHCP-Server (Dynamic Host Configuration Protocol) oder das Initialisieren eines Dateisystems mehrere Sekunden dauern. In diesem Fall unterstützt der Svcstart-Dienst einen Delay-Parameter, der die Zeitdauer angibt, bevor eine Anwendung gestartet wird. Sie finden den Svcstart-Beispielcode im Ordner *%_WINCEROOT%\Public\Servers\SDK\Samples\Services\Svcstart*. Kompilieren Sie den Beispielcode in *Svcstart.dll*, fügen Sie die DLL zum Run-Time Image hinzu und führen Sie den Befehl **sysgen -p servers svcstart** aus, um den Svcstart-Dienst mit dem Betriebssystem zu registrieren. Laden Sie den Dienst mit *Services.exe*.

In Tabelle 3.4 sind die vom Svcstart-Dienst zum Starten von Anwendungen unterstützten Registrierungseinstellungen aufgeführt.

Tabelle 3.4 Svcstart-Registrierungsparameter

Pfad	HKEY_LOCAL_MACHINE\Software\Microsoft\Svcstart\1
Anwendungspfad	@="*iexplore.exe*"
Befehlszeilenparameter	Args="-home"

Tabelle 3.4 Svcstart-Registrierungsparameter (Fortsetzung)

Pfad	HKEY_LOCAL_MACHINE\Software\Microsoft\Svcstart\1
Verzögerungszeit	Delay=dword:4000
Beschreibung	Startet die Anwendung mit den angegebenen Befehlszeilenparametern nach einer in Millisekunden festgelegten Verzögerungszeit. Weitere Informationen finden Sie in der Datei *Svcstart.cpp*.

Windows Embedded CE-Shell

Platform Builder stellt standardmäßig drei Shells bereit, um die Schnittstelle zwischen dem Zielgerät und dem Benutzer zu implementieren: die Befehlsprozessorshell, die Standardshell und eine Thin Client-Shell. Alle Shells unterstützen unterschiedliche Features, um mit dem Zielgerät zu kommunizieren.

Befehlsprozessorshell

Die Befehlsprozessorshell ermöglicht die Konsoleneingabe und Ausgabe mit eingeschränkten Befehlen. Die Shell ist für Geräte sowohl mit Bildschirm als auch ohne Bildschirm und Tastatur verfügbar. Beziehen Sie für Geräte mit Bildschirm die Console Window-Komponente (*Cmd.exe*) ein, damit die Befehlsprozessorshell die Eingabe und die Ausgabe über ein Befehlsfenster verarbeiten kann. Geräte ohne Bildschirm verwenden normalerweise einen seriellen Port für die Eingabe und Ausgabe.

In Tabelle 3.5 sind die Registrierungseinstellungen aufgeführt, die Sie auf dem Zielgerät konfigurieren müssen, um einen seriellen Port zusammen mit der Befehlsprozessorshell zu verwenden.

Tabelle 3.5 Konsolenregistrierungsparameter

Pfad	HKEY_LOCAL_MACHINE\Drivers\Console	
Registrierungseintrag	OutputTo	COMSpeed
Typ	REG_DWORD	REG_DWORD
Standardwert	Keiner	19600

Tabelle 3.5 Konsolenregistrierungsparameter (Fortsetzung)

Pfad	HKEY_LOCAL_MACHINE\Drivers\Console	
Beschreibung	Definiert den seriellen Port, den die Befehlsprozessorshell für die Eingabe und Ausgabe verwendet. ■ Der Wert 1 leitet die Eingabe und Ausgabe an einen Debugport um. ■ Legen Sie den Wert auf 0 fest, um die Umleitung zu deaktivieren. ■ Ein Wert zwischen 0 und 10 leitet die Eingabe und Ausgabe an einen seriellen Port um.	Gibt die Datenübertragungsrate des seriellen Ports in Bits pro Sekunde (Bit/s) an.

Windows Embedded CE-Standardshell

Die Standardshell hat eine Benutzeroberfläche, die dem Windows XP-Desktop ähnlich ist. Der Hauptverwendungszweck der Standardshell ist das Starten und Ausführen von Benutzeranwendungen auf dem Zielgerät. Die Shell umfasst einen Desktop mit einem Startmenü und einer Symbolleiste, die dem Benutzer das Wechseln zwischen Anwendungen ermöglicht. Die Standardshell umfasst außerdem einen Systembenachrichtigungsbereich, um weitere Informationen anzuzeigen, beispielsweise den Status der Netzwerkschnittstellen und die aktuelle Systemzeit.

Die Windows Embedded CE-Standardshell ist ein erforderliches Katalogelement, wenn Sie mit dem OS Design Wizard die Enterprise Terminal-Designvorlage zum Erstellen eines OS-Designprojekts in Visual Studio auswählen. Wenn Sie die Shell klonen und anpassen möchten, verwenden Sie den Quellcode im Ordner *%_WINCEROOT\Public\Shell\OAK\HPC*. In Kapitel 1 ist das Klonen von Katalogelementen sowie das Hinzufügen der Elemente zu einem OS Design erklärt.

Thin Client-Shell

Die Thin Client-Shell, die auch als WBT-Shell (Windows-based Terminal) bezeichnet wird, umfasst eine Benutzeroberfläche für Thin Client-Geräte, die Benutzeranwendungen nicht lokal ausführen. Sie können Internet Explorer zu einem Thin-Client OS Design hinzufügen. Alle anderen Benutzeranwendungen müssen jedoch auf einem Terminalserver im Netzwerk ausgeführt werden. Die Thin Client-Shell verwendet RDP (Remote Desktop Protocol), um die Verbindung mit einem Server herzustellen und den Windows-Remotedesktop anzuzeigen. Die Thin Client-Shell zeigt den Remotedesktop standardmäßig im Vollbildmodus an.

Task-Manager

Sie können Ihre eigene Shell implementieren, indem Sie die Windows-Shellanwendung Task-Manager (TaskMan) klonen und anpassen. Der Quellcode im Ordner *%_WINCEROOT%\Public\Wceshellfe\Oak\Taskman* ist ein guter Ausgangspunkt für diese Aufgabe.

Windows Embedded CE-Systemsteuerung

Die Systemsteuerung ermöglicht den Zugriff auf System- und Anwendungskonfigurationstools. In der Produktdokumentation werden diese Konfigurationstools als Applets bezeichnet, da diese in der Systemsteuerung eingebettet sind. Jedes Applet dient einem anderen Verwendungszweck und ist unabhängig von den anderen Applets. Sie können den Inhalt der Systemsteuerung anpassen, indem Sie Ihre eigenen Applets hinzufügen oder vorhandene Applets entfernen.

Systemsteuerungskomponenten

Die Systemsteuerung ist ein Konfigurationssystem, das auf den folgenden drei Hauptkomponenten basiert:

- **Front-End (*Control.exe*)** Diese Anwendung zeigt die Benutzeroberfläche an und unterstützt das Starten der Applets in der Systemsteuerung.
- **Host Application (*Ctlpnl.exe*)** Diese Anwendung lädt und führt die Applets in der Systemsteuerung aus.
- **Applets** Dies sind die einzelnen Konfigurationstools, die als cpl-Dateien mit einem Symbol und Namen in der Benutzeroberfläche der Systemsteuerung implementiert sind.

Weitere Informationen zum Implementieren der Windows Embedded CE-Systemsteuerung finden Sie im Quellcode im Ordner *%_WINCEROOT%\Public\Wceshellfe\Oak\Ctlpnl*. Sie können den Systemsteuerungscode auch klonen und anpassen, um Ihre Version der Systemsteuerung zu implementieren.

Implementieren von Applets in der Systemsteuerung

Wie bereits erwähnt, ist ein Applet in der Systemsteuerung ein Konfigurationstool für eine Systemkomponente oder eine Benutzeranwendung. Ein Applet wird als cpl-Datei implementiert und im Windows-Ordner auf dem Zielgerät gespeichert. Eine cpl-Datei ist im Wesentlichen eine DLL, die das *CPlApplet* API implementiert. Eine cpl-Datei kann mehrere Systemsteuerungsanwendungen enthalten, aber ein Applet kann nicht mehrere cpl-Dateien umfassen. Da alle cpl-Dateien das *CPlApplet* API implementieren, kann *Control.exe* beim Start detaillierte Informationen zu den implementierten Applets abrufen, um die *verfügbaren* Applets auf der Benutzeroberfläche anzuzeigen. *Control.exe* muss alle cpl-Dateien im Windows-Ordner auflisten und in allen Dateien die Funktion *CPlApplet* aufrufen.

Entsprechend dem DLL-Verhalten und den *CPlApplet* API-Anforderungen müssen die cpl-Dateien die folgenden beiden öffentlichen Einsprungspunkte implementieren:

- **BOOL APIENTRY DllMain(HANDLE hModule, DWORD ul_reason_for_call, LPVOID lpReserved)** Initialisiert die DLL. Das System ruft *DllMain* auf, um die DLL zu laden. Die DLL gibt *true* zurück, wenn die Initialisierung erfolgreich war, und *false*, wenn die Initialisierung fehlgeschlagen ist.
- **LONG CALLBACK CPlApplet(HWND hwndCPL, UINT message, LPARAM lParam1, LPARAM lParam2)** Eine Callback-Funktion, die als Einsprungspunkt für die Systemsteuerung dient, um Aktionen für das Applet auszuführen.

HINWEIS DLL-Einsprungspunkte

Sie müssen die *DllMain*- und *CPlApplet*-Einsprungspunkte exportieren, damit die Systemsteuerungsanwendung auf diese Funktionen zugreifen kann. Nicht exportierte Funktionen der DLL werden nicht erkannt. Stellen Sie sicher, dass die Funktionsdefinitionen in **export "C" { }**-Blöcken gespeichert sind, um die C-Schnittstelle zu exportieren.

Die Systemsteuerung ruft die Funktion *CPlApplet* auf, um das Applet zu initialisieren, Informationen abzurufen, Informationen zu Benutzeraktionen bereitzustellen und das Applet zu entladen. Das Applet muss die in Tabelle 3.6 aufgeführten Meldungen der Systemsteuerung unterstützen, um eine voll funktionsfähige *CPlApplet*-Schnittstelle zu implementieren:

Tabelle 3.6 Meldungen der Systemsteuerung

Meldung der Systemsteuerung	Beschreibung
CPL_INIT	Die Systemsteuerung sendet diese Meldung, um das Applet global zu initialisieren. Die Speicherinitialisierung ist beispielsweise eine typische Aufgabe.
CPL_GETCOUNT	Die Systemsteuerung sendet diese Meldung, um die Anzahl der Systemsteuerungsanwendungen zu bestimmen, die in der cpl-Datei implementiert sind.
CPL_NEWINQUIRE	Die Systemsteuerung sendet diese Meldung für alle von *CPL_GETCOUNT* angegebenen Systemsteuerungsanwendungen. Jede Systemsteuerungsanwendung muss eine *NEWCPLINFO*-Struktur zurückgeben, um das Symbol und den Namen anzugeben, die in der Benutzeroberfläche angezeigt werden.
CPL_DBLCLK	Die Systemsteuerung sendet diese Meldung, wenn der Benutzer auf ein Appletsymbol in der Systemsteuerung doppelklickt.
CPL_STOP	Die Systemsteuerung sendet diese Meldung einmal für jede von *CPL_GETCOUNT* angegebene Instanz.
CPL_EXIT	Die Systemsteuerung sendet diese Meldung für das Applet, bevor das System die DLL freigibt.

HINWEIS **NEWCPLINFO-Informationen**

Speichern Sie die *NEWCPLINFO*-Informationen für alle Systemsteuerungsanwendungen, die Sie in einem Applet in einer Ressource implementieren, die in der cpl-Datei eingebettet ist. Dies unterstützt die Lokalisierung von Symbolen, Namen und Appletbeschreibungen, die als Antwort auf *CPL_NEWINQUIRE*-Meldungen zurückgegeben werden.

Erstellen von Applets in der Systemsteuerung

Um ein Applet zu erstellen und die entsprechende cpl-Datei zu generieren, ermitteln Sie den Quellcodeordner des Applet-Teilprojekts und fügen Sie folgende CPL-Builddirektive am Ende der Sources-Datei in einer neuen Zeile hinzu:

CPL=1

Außerdem müssen Sie in Visual Studio im Applet-Teilprojekt auf der Registerkarte **C/C++** den Pfad zur Systemsteuerungs-Headerdatei zum **Include Directories**-Eintrag hinzufügen (siehe Abbildung 3.3).

$(_PROJECTROOT)\CESysgen\Oak\Inc

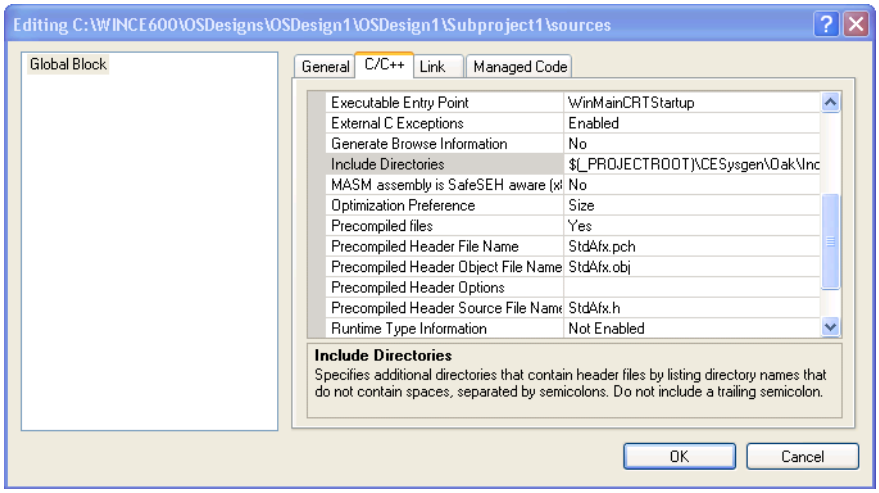


Abbildung 3.3 **Include Directories**-Eintrag für ein Applet in der Systemsteuerung

Aktivieren des Kioskmodus

Zahlreiche Windows Embedded CE-Geräte, beispielsweise medizinische Überwachungsgeräte, ATMs (Automated Teller Machines, d.h. Geldautomaten) oder Industriesteuergerätewerden für nur eine Aufgabe verwendet. Die Standardshell ist für diese Geräte nicht geeignet. Wenn Sie die Standardshell entfernen, wird der Zugriff auf die Konfigurationseinstellungen in der Systemsteuerung beschränkt und die Benutzer können keine anderen Anwendungen starten. Das Gerät wird im Kioskmodus ausgeführt, der eine spezielle Anwendung ohne Shellzugriff direkt auf dem Zielgerät öffnet.

Kioskanwendungen für Windows Embedded CE werden in nativem oder .NET-Code entwickelt. Die Anwendung wird anstatt der Standardshell (*Explorer.exe*) gestartet. Das System startet anschließend eine "schwarze Shell". Das heißt, auf dem Gerät wird keine Shellanwendung ausgeführt. Sie müssen die Registrierungseinträge unter dem Schlüssel *HKEY_LOCAL_MACHINE\Init* konfigurieren, um diese Konfiguration zu implementieren. Wie bereits erwähnt, ist *Launch50* der *LaunchXX*-Eintrag für

Explorer.exe. Ersetzen Sie *Explorer.exe* durch die benutzerdefinierte Kioskanwendung (siehe Tabelle 3.7). Beachten Sie, dass die benutzerdefinierte Kioskanwendung das *SignalStarted* API implementieren muss, damit der Kernel die Anwendungsabhängigkeiten korrekt verwaltet.

Tabelle 3.7 Registrierungsparameter für den Systemstart

Pfad	HKEY_LOCAL_MACHINE\INIT
Komponente	Benutzerdefinierte Kioskanwendung
Binär	Launch50="*myKioskApp.exe*"
Abhängigkeiten	Depend50=hex:14,00, 1e,00
Beschreibung	Um den Kioskmodus zu aktivieren, ersetzen Sie in der Geräteregistrierung den *Launch50*-Eintrag für *Explorer.exe* durch einen Eintrag, der auf eine Kioskanwendung verweist.

HINWEIS Kioskmodus für verwaltete Anwendungen

Um anstatt der Standardshell eine verwaltete Anwendung auszuführen, fügen Sie eine Binärdatei zum Run-Time Image hinzu und bearbeiten Sie die bib-Datei der verwalteten Anwendung. Sie müssen Binärdateien in einem FILES-Abschnitt für das System definieren, um die Anwendung in der CLR (Common Language Runtime) zu laden.

Zusammenfassung

Windows Embedded CE ist ein auf Komponenten basierendes Betriebssystem, das zahlreiche Elemente und anpassbare Features umfasst. Eines dieser Features ermöglicht das Konfigurieren des automatischen Anwendungsstarts beim Systemstart. Dies ist insbesondere für Installations- und Konfigurationstools nützlich. Sie können die Systemsteuerung anpassen, indem Sie in cpl-Dateien implementierte Applets hinzufügen, bei denen es sich um an das *CPlApplet* API gebundene DLLs handelt. Für Spezialgeräte, beispielsweise ATMs, Fahrkartenautomaten, medizinische Überwachungsgeräte, Flughafenterminals oder Industriesteuergeräte, können Sie die Benutzerumgebung anpassen, indem Sie die Standardshell durch eine Kioskanwendung ersetzen. Sie müssen die Codebasis oder den Startprozess des Betriebssystems Windows Embedded CE nicht ändern. Das Aktivieren des Kioskmodus umfasst lediglich das Ersetzen des *Launch50*-Standardeintrags durch einen benutzerdefinierten *Launch50*-Eintrag, der auf die in nativem oder .NET-Code entwickelte Anwendung verweist.

Lektion 3: Implementieren von Threads und Threadsynchronisierung

Windows Embedded CE ist ein Multithread-Betriebssystem. Das Verarbeitungsmodell unterscheidet sich von eingebetteten UNIX-Betriebssystemen insofern, dass Prozesse mehrere Threads umfassen können. Sie müssen mit dem Verwalten, Planen und Synchronisieren dieser Threads in einem Prozess und zwischen Prozessen vertraut sein, um Multithread-Anwendungen und -Treiber zu implementieren und zu debuggen sowie die optimale Systemleistung auf dem Zielgerät sicherzustellen.

Nach Abschluss dieser Lektion können Sie:

- Einen Thread erstellen und beenden.
- Die Threadprioritäten verwalten.
- Mehrere Threads synchronisieren.
- Probleme bei der Threadsynchronisierung debuggen.

Veranschlagte Zeit für die Lektion: 45 Minuten

Prozesse und Threads

Ein Prozess ist eine Instanz einer Anwendung. Der Verarbeitungskontext eines Prozesses kann einen virtuellen Adressraum, ausführbaren Code, Handles für Systemobjekte, einen Sicherheitskontext, eine eindeutige Prozess-ID und Umgebungsvariablen umfassen. Außerdem ist möglicherweise ein primärer Ausführungsthread vorhanden. Ein Thread ist die Basisausführungseinheit, die vom Scheduler verwaltet wird. In einem Windows-Prozess kann ein Thread weitere Threads erstellen. Es ist keine maximale Threadanzahl pro Prozess festgelegt. Die maximale Anzahl hängt von den verfügbaren Speicherressourcen ab, da der von einem Thread belegte Speicher auf der Plattform begrenzt ist. Die Anzahl der Prozesse in Windows Embedded CE ist auf maximal 32.000 beschränkt.

Thread Scheduling in Windows Embedded CE

Windows Embedded CE unterstützt das präemptive Multitasking, um mehrere Threads von verschiedenen Prozessen gleichzeitig auszuführen. Windows Embedded CE führt das Thread Scheduling basierend auf der Priorität aus. Jedem Thread im System ist eine Priorität zwischen 0 und 255 zugeordnet. 0 ist die höchste Priorität. Der Scheduler verwaltet eine Prioritätsliste und wählt den nächsten auszuführenden

Thread entsprechend der Priorität in einem Round Robin-Verfahren aus. Threads mit der gleichen Priorität werden willkürlich der Reihe nach ausgeführt. Beachten Sie, dass das Thread Scheduling von einem Zeitintervall-Algorithmus abhängt. Jeder Thread kann nur für eine begrenzte Zeitdauer ausgeführt werden. Das maximale Zeitintervall für die Threadausführung wird als *Quantum* bezeichnet. Nachdem das Quantum abgelaufen ist, unterbricht der Scheduler den Thread und fährt mit dem nächsten Thread in der Liste fort.

Eine Anwendung kann das Quantum auf Threadbasis festlegen, um das Thread Scheduling entsprechend ihrer Anforderungen anzupassen. Das Ändern des Quantums für einen Thread wirkt sich jedoch nicht auf die Threads mit einer höheren Priorität aus, da der Scheduler die Threads mit der höheren Priorität zuerst ausführt. Der Scheduler unterbricht Threads mit einer niedrigeren Priorität, sobald ein Thread mit einer höheren Priorität verfügbar ist.

Prozessverwaltungs-API

Windows Embedded CE umfasst mehrere Prozessverwaltungsfunktionen, die Bestandteil des Win32 APIs sind. In Tabelle 3.9 sind drei wichtige Funktionen zum Erstellen und Beenden von Prozessen aufgeführt.

Tabelle 3.8 Prozessverwaltungsfunktionen

Funktion	Beschreibung
CreateProcess	Startet einen neuen Prozess.
ExitProcess	Beendet einen Prozess, wobei die DLLs bereinigt und entladen werden.
TerminateProcess	Bricht einen Prozess ab, ohne die DLLs zu bereinigen und zu entladen.

WEITERE INFORMATIONEN Prozessverwaltungs-API

Weitere Informationen zu den Prozessverwaltungsfunktionen sowie die vollständige API-Dokumentation finden Sie in der *Core OS Reference für Windows Mobile® 6 und Windows Embedded CE 6.0* auf der Microsoft MSDN-Website unter *http://msdn2.microsoft.com/en-us/library/aa910709.aspx*.

Threadverwaltungs-API

Jeder Prozess umfasst mindestens einen primären Thread. Dieser Thread ist der Hauptthread des Prozesses. Das heißt, dass beim Beenden oder Abrechen dieses Threads auch der Prozess beendet wird. Der primäre Thread kann weitere Threads erstellen, beispielsweise Worker-Threads, um parallele Berechnungen oder andere Verarbeitungsaufgaben auszuführen. Diese zusätzlichen Threads können wiederum mit dem Win32 API weitere Threads erstellen. In Tabelle 3.9 sind die wichtigsten Funktionen für Anwendungen aufgeführt, die Threads verwenden.

Tabelle 3.9 Threadverwaltungsfunktionen

Funktion	Beschreibung
CreateThread	Erstellt einen neuen Thread.
ExitThread	Beendet einen Thread.
TerminateThread	Bricht einen Thread ab, ohne die Bereinigung oder anderen Code auszuführen. Verwenden Sie diese Funktion ausschließlich im Extremfall, da ein Thread Speicherobjekte zurücklassen und Speicherverlust verursachen kann.
GetExitCodeThread	Gibt den Threadbeendigungscode zurück.
CeSetThreadPriority	Legt die Threadpriorität fest.
CeGetThreadPriority	Ruft die aktuelle Threadpriorität ab.
SuspendThread	Unterbricht einen Thread.
ResumeThread	Setzt einen unterbrochenen Thread fort.
Sleep	Unterbricht einen Thread für eine bestimmte Zeitdauer.
SleepTillTick	Unterbricht einen Thread bis zum nächsten Systemtick.

WEITERE INFORMATIONEN Threadverwaltungs-API

Weitere Informationen zu den Threadverwaltungsfunktionen sowie die vollständige API-Dokumentation finden Sie in der *Core OS Reference für Windows Mobile 6 und Windows Embedded CE 6.0* auf der Microsoft MSDN-Website unter *http://msdn2.microsoft.com/en-us/library/aa910709.aspx*.

Erstellen, Beenden und Abbrechen von Threads

Die Funktion *CreateThread*, die einen neuen Thread erstellt, erwartet mehrere Parameter, die das Erstellen des Threads und die vom Thread ausgeführten Anweisungen steuern. Obwohl Sie die meisten dieser Parameter auf 0 festlegen können, müssen Sie mindestens einen Zeiger auf eine anwendungsdefinierte Funktion angeben, die der Thread ausführen soll. Diese Funktion definiert normalerweise die Hauptanweisungen für den Thread. Sie können jedoch andere Funktionen aus der Funktion aufrufen. Es ist wichtig, die Hauptfunktion als statische Referenz an *CreateThread* zu übergeben, da der Linker die Startadresse der Hauptfunktion während des Kompilierens bestimmen muss. Das Übergeben eines nicht statischen Funktionszeigers funktioniert nicht.

Der folgende Code wurde aus der Datei *Explorer.cpp* im Ordner *%_WINCEROOT%\Public\Shell\OAK\HPC\Explorer\Main* kopiert und veranschaulicht das Erstellen eines Threads.

```
void DoStartupTasks()
{
    HANDLE hThread = NULL;

    // Spin off the thread which registers and watches the font dirs
    hThread = CreateThread(NULL, NULL, FontThread, NULL, 0, NULL);
    if hThread)
    {
        CloseHandle(hThread);
    }

    // Launch all applications in the startup folder
    ProcessStartupFolder();
}
```

Der Code gibt *FontThread* als die Hauptfunktion des neuen Threads an. Das zurückgegebene Threadhandle wird umgehend geschlossen, da es vom aktuellen Thread nicht benötigt wird. Der neue Thread wird parallel zum aktuellen Thread ausgeführt und nach der Rückgabe von der Hauptfunktion implizit beendet. Dies ist die beste Methode zum Beenden von Threads, da die C++-Funktionsbereinigung aktiviert wird. Sie müssen *ExitThread* nicht explizit aufrufen.

Die Funktion *ExitThread* kann jedoch explizit in einer Threadroutine aufgerufen werden, um die Verarbeitung zu beenden, bevor das Ende der Hauptfunktion erreicht wird. *ExitThread* ruft den Einsprungspunkt aller zugehörigen DLLs mit einem Wert auf, der angibt, dass der aktuelle Thread getrennt wird, und hebt anschließend die Zuordnung des aktuellen Threadstacks auf, um den aktuellen Thread abzubrechen.

Der Anwendungsprozess wird beendet, wenn der aktuelle Thread der primäre Thread ist. Da *ExitThread* für den aktuellen Thread ausgeführt wird, müssen Sie keinen *Threadhandle* angeben. Sie müssen jedoch einen numerischen Beendigungscode übergeben, den andere Threads mit der Funktion *GetExitCodeThread* abrufen können. Dieser Prozess ist nützlich zum Erkennen der Fehler und Ursachen für die Beendigung des Threads. Wenn *ExitThread* nicht explizit aufgerufen wird, entspricht der Beendigungscode dem Rückgabewert der Threadfunktion. Wenn *GetExitCodeThread* den Wert *STILL_ACTIVE* zurückgibt, ist der Thread noch aktiv und wird ausgeführt.

Obwohl Sie dies vermeiden sollten, kann es in manchen Situationen erforderlich sein, dass Sie einen Thread durch den Aufruf der Funktion *TerminateThread* abbrechen müssen. Beispielsweise kann ein nicht funktionierender Thread, der Dateieinträge löscht, den Aufruf dieser Funktion erfordern. Möglicherweise müssen Sie *TerminateThread* auch beim Formatieren eines Dateisystems in Debugsitzungen aufrufen, wenn Sie das Erstellen des Codes noch nicht abgeschlossen haben. Sie müssen ein Handle an den zu beendenden Thread übergeben sowie einen Beendigungscode, den Sie zu einem späteren Zeitpunkt mit der Funktion *GetExitCodeThread* abrufen können. Die Funktion *TerminateThread* sollte nicht während der normalen Verarbeitung aufgerufen werden. Die Funktion hinterlässt den Threadstack und die zugehörigen DLLs, verlässt Critical Sections und Mutexe des abgebrochenen Threads und führt zu Speicherverlust sowie zu Instabilität. Verwenden Sie *TerminateThread* nicht als Bestandteil des Verfahrens für die Prozessbeendigung. Threads im Prozess können implizit oder explizit mit der Funktion *ExitThread* beendet werden.

Verwalten der Threadpriorität

Jedem Thread ist ein Prioritätswert zwischen 0 und 255 zugewiesen, der die Threadausführung in Bezug zu den anderen Threads im Prozess und zwischen Prozessen festlegt. Das Win32 API in Windows Embedded CE umfasst vier Funktionen für die Threadverwaltung, die die Priorität eines Threads wie folgt festlegen.

- **Basisprioritätsebenen** Verwenden Sie die Funktionen *SetThreadPriority* und *SetThreadPriority*, um die Threadpriorität auf Ebenen zu verwalten, die kompatibel mit den früheren Versionen von Windows Embedded CE sind (0–7).
- **Alle Prioritätsebenen** Verwenden Sie die Funktionen *CeSetThreadPriority* und *CeGetThreadPriority*, um die Threadpriorität auf allen Ebenen (0255) zu verwalten.

HINWEIS Basisprioritätsebenen

Die Basisprioritätsebenen 0 bis 7 der früheren Versionen von Windows Embedded CE sind nun den acht niedrigsten Prioritätsebenen (248 bis 255) der Funktion *CeSetThreadPriority* zugeordnet.

Beachten Sie, dass die Threadprioritäten eine Beziehung zwischen Threads definieren. Das Zuweisen einer höheren Threadpriorität kann sich nachteilig auf das System auswirken, wenn andere wichtige Threads mit einer niedrigeren Priorität ausgeführt werden. Sie können mit einem niedrigeren Prioritätswert möglicherweise ein besseres Anwendungsverhalten erzielen. Das Testen der Leistung mit verschiedenen Prioritätswerten ist eine zuverlässige Methode zum Ermitteln der besten Prioritätsebene für einen Thread in einer Anwendung oder einem Treiber. Es ist jedoch nicht effizient 256 verschiedene Prioritätswerte zu testen. Wählen Sie für Ihre Threads einen geeigneten Prioritätsbereich basierend auf dem Verwendungszweck des Treibers oder der Anwendung aus (siehe Tabelle 3.10).

Tabelle 3.10 Threadprioritätsbereiche

Bereich	Beschreibung
0 bis 96	Reserviert für Echtzeit-Treiber.
97 bis 152	Wird von den Standardgerätetreibern verwendet.
153 bis 247	Reserviert für die Echtzeitverarbeitung unterhalb der Priorität von Gerätetreibern.
248 bis 255	Wird Anwendungen ohne Echtzeitunterstützung zugeordnet.

Anhalten und Fortsetzen von Threads

Das Verzögern bestimmter konditionaler Aufgaben, die von zeitaufwendigen Initialisierungsroutinen und anderen Fakten abhängen, kann die Systemleistung verbessern. Es ist letzten Endes nicht effizient, 10.000 Tests auszuführen, um zu überprüfen, ob eine erforderliche Komponente einsatzbereit ist. Eine bessere Methode ist, den Workerthread für eine bestimmte Zeitdauer auszusetzen, beispielsweise 10 Millisekunden, anschließend die Abhängigkeiten zu überprüfen und den Thread erneut zu unterbrechen oder die Verarbeitung gegebenenfalls fortzusetzen. Verwenden Sie die *Sleep*-Funktion innerhalb des Threads, um einen Thread anzuhalten oder fortzusetzen. Sie können auch die Funktionen *SuspendThread* und *ResumeThread* verwenden, um einen Thread über einen anderen Thread zu steuern.

Die *Sleep*-Funktion akzeptiert einen numerischen Wert, der das *Sleep*-Intervall in Millisekunden angibt. Beachten Sie, dass das tatsächliche *Sleep*-Intervall diesen Wert wahrscheinlich überschreitet. Die *Sleep*-Funktion verwirft den Rest des Quantums des aktuellen Threads und der Scheduler weist dem Thread keine weitere Zeiteinheit zu, bis das angegebene Intervall verstrichen ist und keine anderen Threads mit einer höheren Priorität vorhanden sind. Beispielsweise legt der Funktionsaufruf *Sleep(0)* das *Sleep*-Intervall nicht auf 0 Millisekunden fest. Stattdessen überlässt *Sleep(0)* den Rest des aktuellen Quantums den anderen Threads. Der aktuelle Thread wird nur fortgesetzt, wenn keine anderen Threads mit der gleichen oder einer höheren Priorität in der Threadliste aufgeführt sind.

Ähnlich wie der *Sleep(0)*-Aufruf verwirft die Funktion *SleepTillTick* den Rest des Quantums des aktuellen Threads und setzt den Thread bis zum nächsten Systemtick aus. Dies ist nützlich zum Synchronisieren einer Aufgabe basierend auf Systemticks.

Die Funktionen *WaitForSingleObject* und *WaitForMultipleObjects* halten einen Thread an, bis ein anderer Thread oder ein Synchronisierungsobjekt signalisiert wird. Beispielsweise kann ein Thread warten, bis ein anderer Thread beendet wird, ohne die Funktionen *Sleep* und *GetExitCodeThread* aufrufen zu müssen, wenn die Funktion *WaitForSingleObject* aktiviert ist. Diese Methode resultiert in einer besseren Auslastung der Ressourcen und verbessert die Lesbarkeit des Codes. Es ist möglich, einen Timeoutwert in Millisekunden an die Funktionen *WaitForSingleObject* und *WaitForMultipleObjects* zu übergeben.

Beispielcode für die Threadverwaltung

Der folgende Codeausschnitt veranschaulicht das Erstellen eines Threads im angehaltenen Modus, die Angabe einer Threadfunktion und von Parametern, das Ändern der Threadpriorität, das Fortsetzen des Threads, das Warten auf die Threadbeendigung und das Beenden des Threads. Der letzte Schritt umfasst das Überprüfen des Fehlercodes, der von der Threadfunktion zurückgegeben wird.

```
// Structure used to pass parameters to the thread.
typedef struct
{
    BOOL bStop;
} THREAD_PARAM_T

// Thread function
DWORD WINAPI ThreadProc(LPVOID lpParameter)
{
    // Perform thread actions...
```

```
    // Exit the thread.
    return ERROR_SUCCESS;
}

BOOL bRet = FALSE;
THREAD_PARAM_T threadParams;
threadParams.bStop = FALSE;
DWORD dwExitCodeValue = 0;

// Create the thread in suspended mode.
HANDLE hThread = CreateThread(NULL, 0, ThreadProc,
                  (LPVOID) &threadParams,
                  CREATE_SUSPENDED, NULL);
if (hThread == NULL)
{
    // Manage the error...
}
else
{
    // Change the Thread priority.
    CeSetThreadPriority(hThread, 200);

    // Resume the thread, the new thread will run now.
    ResumeThread(hThread);

    // Perform parallel actions with the current thread...

    // Wait until the new thread exits.
    WaitForSingleObject(hThread, INFINITE);

    // Get the thread exit code
    // to identify the reason for the thread exiting
    // and potentially detect errors
    // if the return value is an error code value.
    bRet = GetExitCodeThread(hThread, &dwExitCodeValue);

    if (bRet && (ERROR_SUCCESS == dwExitCodeValue))
    {
       // Thread exited without errors.
    }
    else
    {
       // Thread exited with an error.
    }

    // Don't forget to close the thread handle
    CloseHandle(hThread);
}
```

Threadsynchronisierung

Die Kunst bei der Multithread-Programmierung liegt im Vermeiden von Deadlocks, dem Schützen des Zugriffs auf Ressourcen und dem Sicherstellen der Threadsynchronisierung. Windows Embedded CE umfasst mehrere Kernelobjekte zum Synchronisieren des Ressourcenzugriffs für Threads in Treibern oder Anwendungen, beispielsweise Critical Sections, Mutexe, Semaphores, Events und Interlock-Funktionen. Die Auswahl des Objekts hängt von der Aufgabe ab, die Sie ausführen möchten.

Critical Sections

Critical Sections sind Objekte, die in einem einzigen Prozess die Threads synchronisieren und den Zugriff auf Ressourcen schützen. Eine Critical Section kann nicht zwischen Prozessen freigegeben werden. Um auf eine durch einen kritischen Abschnitt geschützte Ressource zuzugreifen, ruft ein Thread die Funktion *EnterCriticalSection* auf. Diese Funktion blockiert den Thread bis die Critical Section verfügbar ist.

In einigen Fällen ist das Blockieren der Threadausführung jedoch möglicherweise nicht effizient. Wenn Sie beispielsweise eine optionale Ressource verwenden möchten, die möglicherweise nie verfügbar ist, blockiert der Aufruf der Funktion *EnterCriticalSection* den Thread und belegt Kernelressourcen, ohne die optionale Ressource zu verarbeiten. In diesem Fall ist es effizienter, eine Critical Section ohne Blockierung zu verwenden, indem Sie die Funktion *TryEnterCriticalSection* aufrufen. Diese Funktion versucht, die Critical Section zu übernehmen und gibt umgehend einen Wert zurück, wenn diese nicht verwendet werden kann. Der Thread kann mit einem alternativen Codepfad fortgesetzt werden, indem beispielsweise die Benutzereingabe angefordert oder ein fehlendes Gerät aktiviert wird.

Nachdem der Thread das Critical Section-Objekt über *EnterCriticalSection* oder *TryEnterCriticalSection* abgerufen hat, kann er exklusiv auf die Ressource zugreifen. Die anderen Threads können nicht auf die Ressource zugreifen, bis der aktuelle Thread die Funktion *LeaveCriticalSection* aufruft, um das Critical Section-Objekt freizugeben. Deshalb sollten Sie die Funktion *TerminateThread* nicht zum Abbrechen von Threads verwenden. *TerminateThread* führt die Bereinigung nicht aus. Wenn der abgebrochene Thread eine Critical Section verwendet, ist die geschützte Ressource erst dann verfügbar, nachdem der Benutzer die Anwendung gestartet hat.

In Tabelle 3.11 sind die wichtigsten Funktionen aufgeführt, die Sie im Zusammenhang mit Critical Section-Objekten für die Threadsynchronisierung verwenden können.

Tabelle 3.11 Kritisches Abschnitts-API

Funktion	Beschreibung
InitializeCriticalSection	Erstellt und initialisiert ein Critical Section-Objekt.
DeleteCriticalSection	Löscht ein kritisches Abschnittsobjekt.
EnterCriticalSection	Übernimmt ein kritisches Abschnittsobjekt.
TryEnterCriticalSection	Versucht, ein kritisches Abschnittsobjekt zu übernehmen.
LeaveCriticalSection	Gibt ein kritisches Abschnittsobjekt frei.

Mutexe

Im Gegensatz zu kritischen Abschnitten, die auf einen einzigen Prozess beschränkt sind, können Mutexe den unabhängigen Zugriff auf die von mehreren Prozessen verwendeten Ressourcen koordinieren. Ein Mutex ist ein Kernelobjekt, das die prozessübergreifende Synchronisierung unterstützt. Um einen Mutex zu erstellen, rufen Sie die Funktion *CreateMutex* auf. Der Thread kann während der Erstellung einen Namen für das Mutexobjekt angeben. Es ist jedoch möglich, einen Mutex ohne Namen zu erstellen. Die Threads in anderen Prozessen können *CreateMutex* ebenfalls aufrufen und den gleichen Namen angeben. Diese nachfolgenden Aufrufe erstellen jedoch keine neuen Kernelobjekte, sondern geben an den vorhandenen Mutex ein Handle zurück. Die Threads in separaten Prozessen können das Mutexobjekt nun zum Synchronisieren des Zugriffs auf die geschützte freigegebene Ressource verwenden.

Der Status eines Mutexobjekts wird signalisiert, wenn das Objekt von einem Thread übernommen wird. Wenn ein Thread das Mutexobjekt besitzt, wird der Status nicht signalisiert. Ein Thread muss eine der Wait-Funktionen, *WaitForSingleObject* oder *WaitForMultipleObjects*, verwenden, um den Besitz anzufordern. Sie können einen Timeoutwert festlegen, um die Threadverarbeitung im alternativen Codepfad fortzusetzen, wenn der Mutex während des Warteintervalls nicht verfügbar ist. Wenn der Mutex verfügbar ist und dem aktuellen Thread der Besitz gewährt wird, rufen Sie *ReleaseMutex* jedesmal auf, wenn der Mutex ein Warteintervall abgeschlossen hat, um

das Mutexobjekt für andere Threads freizugeben. Dies ist wichtig, da ein Thread eine Wait-Funktion mehrmals aufrufen kann, beispielsweise in einer Schleife, ohne seine eigene Ausführung zu blockieren. Das System blockiert den besitzenden Thread nicht, um einen Deadlock zu vermeiden. Der Thread muss *ReleaseMutex* jedoch genauso oft wie die Wait-Funktion aufrufen, um den Mutex freizugeben.

In Tabelle 3.12 sind die wichtigsten Funktionen aufgeführt, die Sie im Zusammenhang mit Mutexobjekten für die Threadsynchronisierung verwenden können.

Tabelle 3.12 Mutex API

Funktion	Beschreibung
CreateMutex	Erstellt und initialisiert ein benanntes oder unbenanntes Mutexobjekt. Um die von mehreren Prozessen verwendeten Ressourcen zu schützen, müssen Sie die Mutexobjekte benennen.
CloseHandle	Schließt ein Mutexhandle und löscht den Verweis auf das Mutexobjekt. Alle Verweise auf den Mutex müssen gelöscht werden, bevor der Kernel das Mutexobjekt löscht.
WaitForSingleObject	Wartet bis der Besitz eines Mutexobjekts gewährt wird.
WaitForMultipleObjects	Wartet bis der Besitz eines oder mehrerer Mutexobjekte gewährt wird.
ReleaseMutex	Gibt ein Mutexobjekt frei.

Semaphores

Abgesehen von Kernelobjekten, die den unabhängigen Zugriff auf Ressourcen in einem Prozess oder zwischen Prozessen ermöglichen, stellt Windows Embedded CE Semaphoreobjekte bereit, die den gleichzeitigen Zugriff auf eine Ressource durch einen oder mehrere Threads aktivieren. Diese Semaphoreobjekte unterstützen einen Indikator zwischen Null und einem maximalen Wert, um die Anzahl der Threads zu steuern, die auf die Ressource zugreifen. Der maximale Wert ist in der Funktion *CreateSemaphore* angegeben.

Der Semaphore-Indikator begrenzt die Anzahl der Threads, die gleichzeitig auf das Synchronisierungsobjekt zugreifen können. Das System verringert den Indikator jedesmal, wenn ein Thread ein Warteinverall für das Semaphoreobjekt abschließt, bis der Indikator den Wert 0 erreicht und in den nicht signalisierten Status übergeht. Der Indikator kann nicht über Null hinaus verringert werden. Kein anderer Thread kann auf die Ressource zugreifen, bis ein besitzender Thread den Semaphore durch Aufruf der Funktion *ReleaseSemaphore* freigibt, die den Indikator um einen angegebenen Wert erhöht und das Semaphoreobjekt wieder zurück in den signalisierten Status versetzt.

Ähnlich wie Mutexe können mehrere Prozesse Handles des gleichen Semaphoreobjekts öffnen, um auf die von mehreren Prozessen verwendeten Ressourcen zuzugreifen. Der erste Aufruf der Funktion *CreateSemaphore* erstellt das Semaphoreobjekt mit einem angegebenen Namen. Sie können auch nicht benannte Semaphores erstellen, aber diese Objekte sind nicht für die prozessübergreifende Synchronisierung verfügbar. Nachfolgende Aufrufe der Funktion *CreateSemaphore* mit dem gleichen Semaphorenamen erstellen keine neuen Objekte, sondern öffnen ein neues Handle des gleichen Semaphores.

In Tabelle 3.13 sind die wichtigsten Funktionen aufgeführt, die Sie im Zusammenhang mit Semaphoreobjekten für die Threadsynchronisierung verwenden können.

Tabelle 3.13 Semaphore API

Funktion	Beschreibung
CreateSemaphore	Erstellt und initialisiert ein benanntes oder unbenanntes Semaphoreobjekt mit einem Indikatorwert. Verwenden Sie benannte Semaphoreobjekte, um freigegebene Ressourcen zu schützen.
CloseHandle	Schließt ein Semaphorehandle und löscht den Verweis auf das Semaphoreobjekt. Alle Verweise auf den Semaphore müssen gelöscht werden, bevor der Kernel das Semaphoreobjekt löscht.
WaitForSingleObject	Wartet bis der Besitz eines Semaphoreobjekts gewährt wird.
WaitForMultipleObjects	Wartet bis der Besitz eines oder mehrerer Semaphoreobjekte gewährt wird.
ReleaseSemaphore	Gibt ein Semaphoreobjekt frei.

Event

Das Eventobjekt ist ein weiteres Kernelobjekt, das Threads synchronisiert. Dieses Objekt ermöglicht Anwendungen das Signalisieren anderer Threads, wenn eine Aufgabe beendet ist oder Daten für potenzielle Leser verfügbar sind. Jedes Event umfasst signalisierte oder nicht signalisierte Statusinformatinen, die vom API zum Ermitteln des Eventstatus verwendet werden. Die beiden Eventtypen für manuelle und automatisch zurückgesetzte Events werden entsprechend dem vom Event erwarteten Verhalten erstellt.

Der Thread gibt während der Erstellung einen Namen für das Eventobjekt an. Es ist jedoch möglich, ein Event ohne Namen zu erstellen. Die Threads in anderen Prozessen können *CreateMutex* aufrufen und den gleichen Namen angeben, aber diese nachfolgenden Aufrufe erstellen keine neuen Kernelobjekte.

In Tabelle 3.14 sind die wichtigsten Funktionen aufgeführt, die Sie im Zusammenhang mit Eventobjekten für die Threadsynchronisierung verwenden können.

Tabelle 3.14 Event-API

Funktion	Beschreibung
CreateEvent	Erstellt und initialisiert ein benanntes oder unbenanntes Eventobjekt.
SetEvent	Signalisiert ein Event (siehe unten).
PulseEvent	Löst ein Event aus und signalisiert das Event (siehe unten).
ResetEvent	Setzt ein signalisiertes Event zurück.
WaitForSingleObject	Wartet bis ein Event signalisiert wird.
WaitForMultipleObjects	Wartet auf die Signalisierung durch ein oder mehrere Eventobjekte.
CloseHandle	Gibt ein Eventobjekt frei.

Das Verhalten der Event-APIs variiert basierend auf dem entsprechenden Eventtyp. Wenn Sie *SetEvent* für ein manuelles Eventobjekt verwenden, wird das Event bis zum expliziten Aufruf von *ResetEvent* signalisiert. Automatisch zurückgesetzte Events werden signalisiert, bis ein wartender Thread freigegeben wird. Bei Verwendung der Funktion *PulseEvent* für automatisch zurückgesetzte Events wird maximal ein

wartender Thread freigegeben, bevor er umgehend zurück in den nicht signalisierten Status versetzt wird. Im Fall von manuellen Threads werden alle wartenden Threads freigegeben und sofort zurück in den nicht signalisierten Status versetzt.

Interlocked-Funktionen

In Multithread-Umgebungen können Threads jederzeit unterbrochen und zu einem späteren Zeitpunkt vom Scheduler fortgesetzt werden. Sie können Codeabschnitte oder Anwendungsressourcen unter Verwendung von Semaphores, Events oder kritischen Abschnitten schützen. In einigen Anwendungen ist die Verwendung dieser Systemobjekttypen möglicherweise zu zeitaufwendig, um nur eine Codezeile wie folgt zu schützen:

```
// Increment variable
dwMyVariable = dwMyVariable + 1;
```

Dieser Beispielquellcode ist in C eine einzeilige Anweisung, die in Assemblersprache ausgedrückt jedoch komplexer sein kann. In diesem Beispiel kann der Thread während der Ausführung angehalten und später fortgesetzt werden. Wenn jedoch ein anderer Thread die gleiche Variable verwendet, können potenzielle Fehler auftreten. Der Vorgang ist nicht atomar. In Windows Embedded CE 6.0 R2 ist es jedoch möglich, Werte in Multithread-sicheren, atomaren Vorgängen ohne Synchronisierungsobjekte zu verringern, zu erhöhen und hinzuzufügen. Dies wird mit Interlocked-Funktionen ausgeführt.

In Tabelle 3.15 sind die wichtigsten Interlocked-Funktionen aufgeführt, die zum atomaren Ändern von Variablen verwendet werden.

Tabelle 3.15 Interlock API

Funktion	Beschreibung
InterlockedIncrement	Erhöht den Wert einer 32-Bit-Variablen.
InterlockedDecrement	Verringert den Wert einer 32-Bit-Variablen.
InterlockedExchangeAdd	Fügt einen Wert atomar hinzu.

Beheben von Problemen bei der Threadsynchronisierung

Die Multithread-Programmierung ermöglicht das Strukturieren von Softwarelösungen basierend auf separaten Codeausführungseinheiten für die Interaktion mit der Benutzeroberfläche und Hintergrundaufgaben. Hierbei handelt es sich um eine fortgeschrittene Entwicklungsmethode, die das sorgfältige Implementieren von Threadsynchronisierungsmethoden erfordert. Deadlocks können insbesondere dann auftreten, wenn mehrere Synchronisierungsobjekte in Schleifen und Subroutinen verwendet werden. Wenn beispielsweise Thread 1 den Mutex A besitzt und vor der Freigabe von Mutex A auf Mutex B wartet, während Thread 2 vor der Freigabe von Mutex B auf Mutex A wartet, können beide Threads nicht fortgesetzt werden, da sie von einer Ressource abhängen, die vom jeweils anderen Thread freigegeben werden muss. Diese Probleme sind schwierig zu identifizieren und beheben, insbesondere dann, wenn Treads von mehreren Prozessen auf freigegebene Ressourcen zugreifen. Das Tool Remote Kernel Tracker stellt das Thread Scheduling dar und ermöglicht das Auffinden von Deadlocks.

Mit dem Remote Kernel Tracker können Sie alle Prozesse, Threads, Threadinteraktionen und andere Systemaktivitäten auf einem Zielgerät überwachen. Das Tool hängt vom CeLog-Eventüberwachungssystem ab, um Kernel- und andere Systemevents in der Datei *Celog.clg* im Verzeichnis *%_FLATRELEASEDIR%* zu protokollieren. Systemevents sind nach Zonen klassifiziert. Das CeLog-Eventüberwachungssystem kann für eine bestimmte Zone für die Datenprotokollierung konfiguriert werden.

Wenn KITL (Kernel Independent Transport Layer) auf dem Zielgerät aktiviert ist, visualisiert der Remote Kernel Tracker die CeLog-Daten und analysiert die Interaktionen zwischen Threads und Prozessen (siehe Abbildung 3.4). Obwohl KITL die Daten direkt an den Remote Kernel Tracker sendet, können die erfassten Daten offline analysiert werden.

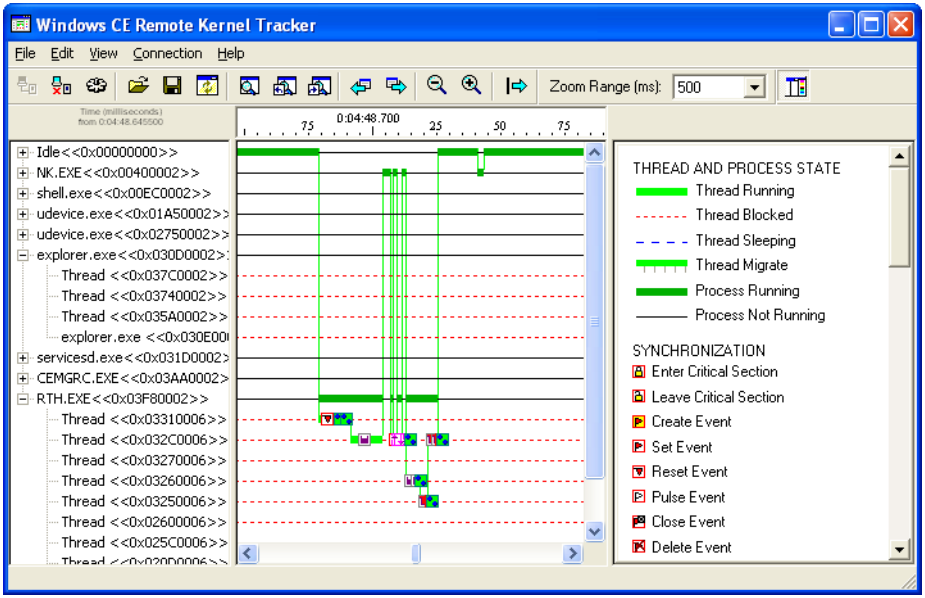


Abbildung 3.4 Das Tool Remote Kernel Tracker

WEITERE INFORMATIONEN CeLog-Eventüberwachung und Filterung

Weitere Informationen zur CeLog-Eventüberwachung und CeLog-Eventfilterung finden Sie im Abschnitt „CeLog Event Tracking Overview" in der Windows Embedded CE 6.0-Dokumentation auf der Microsoft MSDN-Website unter *http://msdn2.microsoft.com/en-us/library/aa935693.aspx*.

Zusammenfassung

Windows Embedded CE ist ein Multithread-Betriebssystem, das mehrere Prozessverwaltungsfunktionen umfasst, um Prozesse und Threads zu erstellen, Threadprioritäten von 0 bis 255 zuzuordnen, Threads anzuhalten und Threads fortzusetzen. Die *Sleep*-Funktion ist nützlich, um einen Thread für eine bestimmte Zeitdauer anzuhalten. Ein Thread kann jedoch auch mit den Funktionen *WaitForSingleObject* und *WaitForMultipleObjects* angehalten werden, bis ein anderer Thread oder ein Synchronisierungsobjekt signalisiert wird. Prozesse und Threads werden auf zwei Arten beendet: Mit und ohne Bereinigung. Verwenden Sie immer *ExitProcess* und *ExitThread*, damit das System die Bereinigung ausführen kann. Sie sollten *TerminateProcess* und *TerminateThread* nur dann verwenden, wenn Sie keine andere Wahl haben.

Bei der Arbeit mit mehreren Threads ist es von Vorteil die Threadsynchronisierung zu implementieren, um den Zugriff auf die in und zwischen Prozessen freigegebenen Ressourcen zu koordinieren. Windows Embedded CE umfasst für diesen Zweck mehrere Kernelobjekte, beispielsweise Critical Sections, Mutexe und Semaphores. Critical Sections überwachen den Zugriff auf Ressourcen in einem einzigen Prozess. Mutexe koordinieren den unabhängigen Zugriff auf die von mehreren Prozessen verwendeten Ressourcen. Semaphores implementieren den gleichzeitigen Zugriff mehrerer Threads auf Ressourcen in einem Prozess oder zwischen Prozessen. Events benachrichtigen die anderen Threads und verknüpfte Funktionen manipulieren Variablen auf eine Thread-sichere atomarische Art. Wenn in der Entwicklungsphase Probleme mit der Threadsynchronisierung auftreten, beispielsweise Deadlocks, verwenden Sie das CeLog-Eventüberwachungssystem und das Tool Remote Kernel Tracker, um die Threadinteraktionen auf dem Zielgerät zu analysieren.

PRÜFUNGSTIPP

Um die Zertifizierungsprüfung zu bestehen, stellen Sie sicher, dass Sie mit den verschiedenen Synchronisierungsobjekten in Windows Embedded CE 6.0 R2 vertraut sind.

Lektion 4: Implementieren der Ausnahmebehandlung

Zielgeräte, auf denen Windows Embedded CE ausgeführt wird, verwenden Ausnahmen als Bestandteil der System- und Anwendungsverarbeitung. Es ist schwierig, auf angemessene Art auf Ausnahmen zu reagieren. Die korrekte Behandlung von Ausnahmen stellt ein stabiles Betriebssystem und eine positive Benutzererfahrung sicher. Anstatt beispielsweise eine Videoanwendung abzubrechen, ist es möglicherweise sinnvoller, den Benutzer aufzufordern, eine USB-Kamera anzuschließen. Die Ausnahmebehandlung ist jedoch keine universelle Lösung. Unerwartetes Anwendungsverhalten kann das Ergebnis von böswilligen Codeänderungen in ausführbaren Dateien, DLLs, Speicherstrukturen und Daten sein. In diesem Fall ist das Abbrechen der fehlerhaften Komponente oder Anwendung die beste Vorgehensweise, um die Daten und das System zu schützen.

Nach Abschluss dieser Lektion können Sie:

- Die Ursachen von Ausnahmen verstehen.
- Ausnahmen abfangen und auslösen.

Veranschlagte Zeit für die Lektion: 30 Minuten

Übersicht der Ausnahmebehandlung

Ausnahmen sind Events, die aus Fehlerbedingungen resultieren. Diese Bedingungen können auftreten, wenn der Prozessor, das Betriebssystem oder die Anwendungen Anweisungen außerhalb des normalen Steuerungsflusses im Kernelmodus und Benutzermodus ausführen. Durch das Abfangen und Behandeln von Ausnahmen können Sie die Stabilität Ihrer Anwendungen verbessern und eine positive Benutzererfahrung sicherstellen. Sie müssen jedoch keine Ausnahmehandler in Ihrem Code implementieren, da die strukturierte Ausnahmebehandlung ein Bestandteil von Windows Embedded CE ist.

Das Betriebssystem fängt alle Ausnahmen ab und leitet diese an die Anwendungsprozesse weiter, die die Events ausgelöst haben. Wenn ein Prozess ein Ausnahmeevent nicht behandelt, leitet das System die Ausnahme an einen Postmortem-Debugger weiter und bricht den Prozess letztendlich ab, um das System vor fehlerhafter Hardware und Software zu schützen. Dr. Watson ist ein bekannter Postmortem-Debugger, der eine Speicherabbilddatei für Windows Embedded CE erstellt.

Ausnahmebehandlung und Kerneldebugging

Die Ausnahmebehandlung ist auch die Grundlage für das Kerneldebugging. Wenn Sie das Kerneldebugging in einem OS Design aktivieren, bezieht der Platform Builder das KdStub (Kernel Debugging Stub) in das Run-Time Image ein, um die Komponenten zu aktivieren, die Ausnahmen auslösen, um den Debugger aufzurufen. Sie können nun das Problem analysieren, den Code durchlaufen, die Verarbeitung fortsetzen oder den Anwendungsprozess manuell abbrechen. Sie benötigen jedoch eine KITL-Verbindung mit einem Entwicklungscomputer, um mit dem Zielgerät zu interagieren. Ohne KITL-Verbindung ignoriert der Debugger die Ausnahme und die Anwendung wird weiterhin fortgesetzt, damit das Betriebssystem einen anderen Ausnahmehandler verwenden kann, als ob kein Debugger verfügbar ist. Wenn die Anwendung die Ausnahme nicht behandelt, ermöglicht das Betriebssystem dem Kerneldebugger das Postmorten-Debugging auszuführen. In diesem Zusammenhang wird das Debuggen häufig als JIT-Debugging (Just In Time) bezeichnet. Der Debugger, der die Ausnahme nun akzeptieren muss, wartet auf eine KITL-Verbindung für die Debugausgabe. Windows Embedded CE wartet bis Sie die KITL-Verbindung hergestellt und das Debuggen des Zielgeräts gestartet haben. In der Entwicklerdokumentation werden häufig die Begriffe *First-Chance-* und *Second-Chance-*Ausnahme verwendet, da der Kerneldebugger zwei Chancen hat, eine Ausnahme zu behandeln. Diese Begriffe beziehen sich jedoch auf das gleiche Ausnahmeevent. Weitere Informationen zum Debuggen und Testen des Systems finden Sie in Kapitel 5.

Hardware- und Softwareausnahmen

Windows Embedded CE verwendet für alle Hardware- und Softwareausnahmen die gleiche strukturierte Ausnahmebehandlung (Structured Exception Handling, SEH). Die CPU (Central Processing Unit) kann Hardwareausnahmen als Reaktion auf ungültige Anweisungssequenzen auslösen, beispielsweise der Division durch 0 oder eine Zugriffsverletzung, die durch den Versuch auf eine ungültige Speicheradresse zuzugreifen, verursacht wird. Treiber, Systemanwendungen und Benutzeranwendungen können Softwareausnahmen auslösen, um mit der Funktion RaiseException die SEH-Methode des Betriebssystems aufzurufen. Beispielsweise können Sie eine Ausnahme auslösen, wenn ein erforderliches Gerät (z.B. eine USB-Kamera) nicht verfügbar ist oder der Benutzer einen ungültigen Befehlszeilenparameter eingibt. Eine Ausnahme kann aus jedem Grund ausgelöst werden, der das Ausführen spezieller Anweisungen außerhalb des normalen Codepfads erfordert. Sie können im Funktionsaufruf RaiseException mehrere Parameter für die Informationen angeben, die die Ausnahme beschreiben. Diese Spezifikation kann anschließend im Filterausdruck eines Ausnahmehandlers verwendet werden.

Ausnahmehandler-Syntax

Windows Embedded CE unterstützt die framebasierte strukturierte Ausnahmebehandlung. Es ist möglich, eine Codesequenz in Klammern ({}) einzuschließen und mit dem Schlüsselwort *__try* zu markieren, um anzuzeigen, dass alle während der Codeausführung ausgelösten Ausnahmen einen Ausnahmehandler in einem Abschnitt aufrufen, der mit dem Schlüsselwort *__except* gekennzeichnet ist. Der C/C++-Compiler in Microsoft Visual Studio unterstützt diese Schlüsselwörter und kompiliert die Codeblöcke mit zusätzlichen Anweisungen, die dem System ermöglichen, den Computerstatus wiederherzustellen und die Threadausführung an der Stelle fortzusetzen, an der die Ausnahme aufgetreten ist, oder die Steuerung an einen Ausnahmehandler zu übergeben und die Threadausführung im Aufrufstackframe fortzusetzen, in dem sich der Ausnahmehandler befindet.

Das folgende Codesegment veranschaulicht die Verwendung der Schlüsselwörter *__try* und *__except* für die strukturierte Ausnahmebehandlung:

```
__try
{
    // Place guarded code here.
}
__except (filter-expression)
{
    // Place exception-handler code here.
}
```

Das Schlüsselwort *__except* unterstützt einen Filterausdruck, bei dem es sich um einen einfachen Ausdruck oder eine Filterfunktion handeln kann. Der Filterausdruck kann auf einen der folgenden Werte festgelegt werden:

- **EXCEPTION_CONTINUE_EXECUTION** Das System nimmt an, dass die Ausnahme behoben ist und setzt die Threadausführung an der Stelle fort, an der die Ausnahme aufgetreten ist. Die Filterfunktionen geben diesen Wert normalerweise zurück, nachdem die Ausnahme behandelt wurde, um die Verarbeitung fortzusetzen.
- **EXCEPTION_CONTINUE_SEARCH** Das System setzt die Suche nach einem geeigneten Ausnahmehandler fort.
- **EXCEPTION_EXECUTE_HANDLER** Die Systemthreadausführung wird sequenziell vom Ausnahmehandler, anstatt an der Stelle, an der die Ausnahme aufgetreten ist, fortgesetzt.

HINWEIS Unterstützung der Ausnahmebehandlung

Die Ausnahmebehandlung ist eine Erweiterung der C-Programmiersprache, aber wird in C++ unterstützt.

Abbruchhandler-Syntax

Windows Embedded CE unterstützt die Abbruchbehandlung. Als eine Microsoft-Erweiterung der Programmiersprachen C und C++ ermöglicht Ihnen die Abbruchbehandlung, sicherzustellen, dass das System immer einen bestimmten Codeblock ausführt, unabhängig davon, ob der Codeblock geschützt ist. Dieser Codeabschnitt wird als Abbruchhandler bezeichnet, da er die Bereinigungsaufgaben auch dann ausführt, wenn im geschützten Code eine Ausnahme oder ein anderer Fehler auftritt. Beispielsweise können Sie einen Abbruchhandler verwenden, um nicht mehr benötigte Threadhandles zu schließen.

Das folgende Codesegment veranschaulicht die Verwendung der Schlüsselwörter *__try* und *__finally* für die strukturierte Ausnahmebehandlung:

```
__try
{
    // Place guarded code here.
}
__ finally
{
    // Place termination code here.
}
```

Die Abbruchbehandlung unterstützt das Schlüsselwort *__leave* im geschützten Abschnitt. Dieses Schlüsselwort beendet die Threadausführung an der aktuellen Stelle im geschützten Abschnitt und setzt die Threadausführung an der ersten Anweisung im Abbruchhandler fort, ohne den Aufrufstack abzuwickeln.

HINWEIS Verwenden von *__try*-, *__except*- und *__finally*-Blöcken

Ein *__try*-Block kann nicht gleichzeitig einen Ausnahmehandler und einen Abbruchhandler umfassen. Wenn Sie *__except* und *__finally* benötigen, verwenden Sie eine äußere *try-except*-Anweisung und eine innere *try-finally*-Anweisung.

Dynamische Arbeitsspeicherreservierung

Die dynamische Arbeitsspeicherreservierung ist eine Zuordnungsmethode, die von der strukturierten Ausnahmebehandlung abhängt, um die Gesamtanzahl der zugesicherten Speicherseiten im System zu reduzieren. Dies ist insbesondere für große Speicherreservierungen nützlich. Das vorzeitige Zusichern einer gesamten Zuordnung kann verursachen, dass nicht genügend zusicherbare Seiten im System verfügbar sind und die virtuelle Speicherreservierung fehlschlägt.

Die dynamische Speicherreservierung umfasst folgende Schritte:

1. Rufen Sie die Funktion *VirtualAlloc* mit der Basisadresse NULL auf, um einen Speicherblock zu reservieren. Das System reserviert den Speicherblock ohne die Seiten zuzusichern.
2. Versuchen Sie, auf eine Speicherseite zuzugreifen. Dies löst eine Ausnahme aus, da eine nicht zugesicherte Seite weder gelesen noch geschrieben werden kann. Dieser unzulässige Vorgang resultiert in einer Seitenfehlerausnahme. In den Abbildungen 3.5 und 3.6 ist ein nicht behandelter Seitenfehler in der Anwendung *PageFault.exe* dargestellt.
3. Implementieren Sie einen Ausnahmehandler basierend auf einer Filterfunktion. Sichern Sie in der Filterfunktion eine Seite aus dem reservierten Bereich zu. Wenn die Seitenzuordnung erfolgreich ist, geben Sie *EXCEPTION_CONTINUE_EXECUTION* zurück, um die Threadausführung im *__try*-Block an der Stelle fortzusetzen, an der die Ausnahme aufgetreten ist. Wenn die Seitenzuordnung fehlschlägt, geben Sie *EXCEPTION_EXECUTE_HANDLER* zurück, um den Ausnahmehandler im *__except*-Block aufzurufen und den gesamten Bereich der reservierten und zugesicherten Seiten freizugeben.

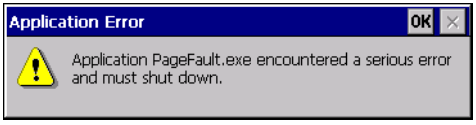


Abbildung 3.5 Meldung beim Auftreten einer nicht behandelten Seitenfehlerausnahme

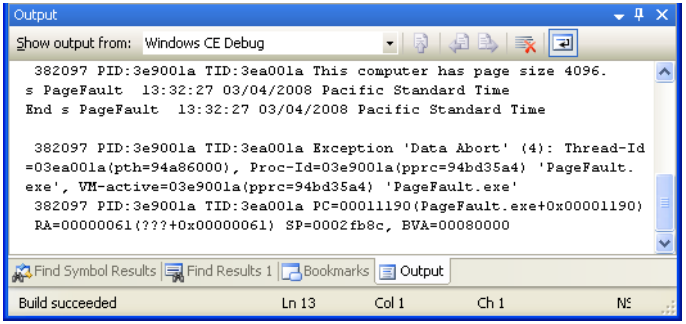


Abbildung 3.6 Die Debugausgabe von KITL in Visual Studio 2005 für eine nicht behandelte Seitenfehlerausnahme

Der folgende Codeabschnitt veranschaulicht die dynamische Speicherreservierung basierend auf der Behandlung einer Seitenfehlerausnahme:

```
#define PAGESTOTAL 42    // Max. number of pages

LPTSTR lpPage;           // Page to commit
DWORD dwPageSize;        // Page size, in bytes

INT ExceptionFilter(DWORD dwCode)
{
    LPVOID lpvPage;

    if (EXCEPTION_ACCESS_VIOLATION != dwCode)
    {
        // This is an unexpected exception!
        // Do not return EXCEPTION_EXECUTE_HANDLER
        // to handle this in the application process.
        // Instead, let the operating system handle it.
        return EXCEPTION_CONTINUE_SEARCH;
    }

    // Allocate page for read/write access.
    lpvPage = VirtualAlloc((LPVOID) lpPage,
                           dwPageSize, MEM_COMMIT,
                           PAGE_READWRITE);
    if (NULL == lpvPage)
    {
        // Continue thread execution
        // in __except block.
        return EXCEPTION_EXECUTE_HANDLER;
    }

    // Set lpPage to the next page.
    lpPage = (LPTSTR) ((PCHAR) lpPage + dwPageSize);

    // Continue thread execution in __try block.
```

```
    return EXCEPTION_CONTINUE_EXECUTION;
}

VOID DynamicVirtualAlloc()
{
    LPVOID lpvMem;
    LPTSTR lpPtr;
    DWORD i;
    BOOL bRet;

    // Get page size on computer.
    SYSTEM_INFO sSysInfo;
    GetSystemInfo(&sSysInfo);
    dwPageSize = sSysInfo.dwPageSize;

    // Reserve memory pages without committing.
    lpvMem = VirtualAlloc(NULL, PAGESTOTAL*dwPageSize,
                          MEM_RESERVE, PAGE_NOACCESS);

    lpPtr = lpPage = (LPTSTR) lpvMem;

    // Use structured exception handling when accessing the pages.
    for (i=0; i < PAGESTOTAL*dwPageSize; i++)
    {
        __try
        {   // Write to memory.
            lpPtr[i] = 'x';
        }
        __except (ExceptionFilter(GetExceptionCode()))
        {   // Filter function unsuccessful. Abort mission.
            ExitProcess( GetLastError() );
        }
    }

    // Release the memory.
    bRet = VirtualFree(lpvMem, 0, MEM_RELEASE);
}
```

Zusammenfassung

Windows Embedded CE unterstützt sowohl die Ausnahmebehandlung als auch die Abbruchbehandlung. Ausnahmen im Prozessor, im Betriebssystem und in Anwendungen werden als Reaktion auf unzulässige Anweisungssequenzen, den Zugriff auf eine nicht verfügbare Speicheradresse, nicht verfügbare Geräteressourcen, ungültige Parameter oder einen Vorgang ausgelöst, der die spezielle Verarbeitung erfordert, beispielsweise die dynamische Speicherreservierung. Sie können *try–except*-Anweisungen verwenden, um auf Fehlerbedingungen außerhalb des normalen Steuerungsflusses zu reagieren und *try–finally*-Anweisungen, um den Code

auszuführen, unabhängig davon, wie der Steuerungsfluss den geschützten __*try*-Codeblock beendet.

Die Ausnahmebehandlung unterstützt Filterausdrücke und Filterfunktionen, die Ihnen ermöglichen, auf ausgelöste Events zu reagieren. Es ist nicht empfehlenswert alle Ausnahmen abzufangen, da das unerwartete Anwendungsverhalten das Ergebnis von böswilligem Code sein kann. Behandeln Sie nur die Ausnahmen, die direkt behoben werden müssen, um das zuverlässige und stabile Anwendungsverhalten sicherzustellen. Das Betriebssystem kann nicht behandelte Ausnahmen an einen Postmortem-Debugger weiterleiten, um ein Speicherabbild zu erstellen und die Anwendung abzubrechen.

PRÜFUNGSTIPP

Um die Zertifizierungsprüfung zu bestehen, stellen Sie sicher, dass Sie mit der Ausnahmebehandlung und Abbruchbehandlung in Windows Embedded CE 6.0 R2 vertraut sind.

Lektion 5: Implementieren der Energieverwaltung

Die Energieverwaltung ist für Windows Embedded CE-Geräte unentbehrlich. Indem Sie den Energieverbrauch reduzieren, können Sie die Lebensdauer der Batterie verlängern und eine positive Benutzererfahrung sicherstellen. Dies ist das primäre Ziel der Energieverwaltung auf tragbaren Geräten. Feststehende Geräte profitieren ebenfalls von der Energieverwaltung. Unabhängig von der Gerätegröße können Sie die Betriebskosten, die Wärmeabgabe, die mechanische Abnutzung und den Geräuschpegel reduzieren, wenn Sie das Gerät nach einer bestimmten Inaktivitätszeitdauer in einen niedrigen Energiestatus versetzen. Außerdem hilft eine effektive Energieverwaltung die Umweltbelastung zu verringern.

Nach Abschluss dieser Lektion können Sie:

- Die Energieverwaltung auf einem Zielgerät aktivieren.
- Energieverwaltungsfeatures in Anwendungen implementieren.

Veranschlagte Zeit für die Lektion: 40 Minuten

Power Manager

Der Power Manager (*PM.dll*) in Windows Embedded CE ist eine Kernelkomponente, die mit dem Geräte-Manager (*Device.exe*) integriert ist und Energieverwaltungsfeatures implementiert. Der Power Manager agiert hauptsächlich als Vermittler zwischen dem Kernel, OAL (OEM Adaptation Layer) und den Treibern für Peripheriegeräte und Anwendungen. Indem der Kernel und OAL von den Treibern und Anwendungen getrennt werden, können diese ihren Energiestatus unabhängig vom Systemstatus verwalten. Die Treiber und Anwendungen erhalten Benachrichtigungen über Energieevents vom Power Manager. Der Power Manager kann den Systemenergiestatus basierend auf Events und Zeitgebern festlegen, den Treiberenergiestatus steuern und auf OAL-Events reagieren, die eine Energiestatusänderung erfordern, beispielweise wenn der Energiestatus der Batterie kritisch ist.

Power Manager-Komponenten und -Architektur

Der Power Manager umfasst eine Benachrichtigungsschnittstelle, eine Anwendungsschnittstelle und eine Geräteschnittstelle. Die Benachrichtigungsschnittstelle ermöglicht Anwendungen das Abrufen von Informationen zu Energieverwaltungsevents, beispielsweise wenn sich der Energiestatus des Systems oder eines Geräts ändert. Beim Auftreten dieser Events kommunizieren die für die Energieverwaltung

aktivierten Anwendungen über die Anwendungsschnittstelle mit dem Power Manager, um ihre Anforderungen an die Energieverwaltung zu übermitteln und den Systemenergiestatus zu ändern. Die Geräteschnittstelle bietet hingegen eine Methode zum Steuern des Energiestands der Gerätetreiber. Der Power Manager kann den Geräteenergiestatus unabhängig vom Systemenergiestatus festlegen. Gerätetreiber übermitteln ihre Energieanforderungen über die Treiberschnittstelle an den Power Manager. Der maßgebliche Punkt ist, dass der Power Manager und die Power Manager APIs die Energieverwaltung in Windows Embedded CE zentralisieren (siehe Abbildung 3.7).

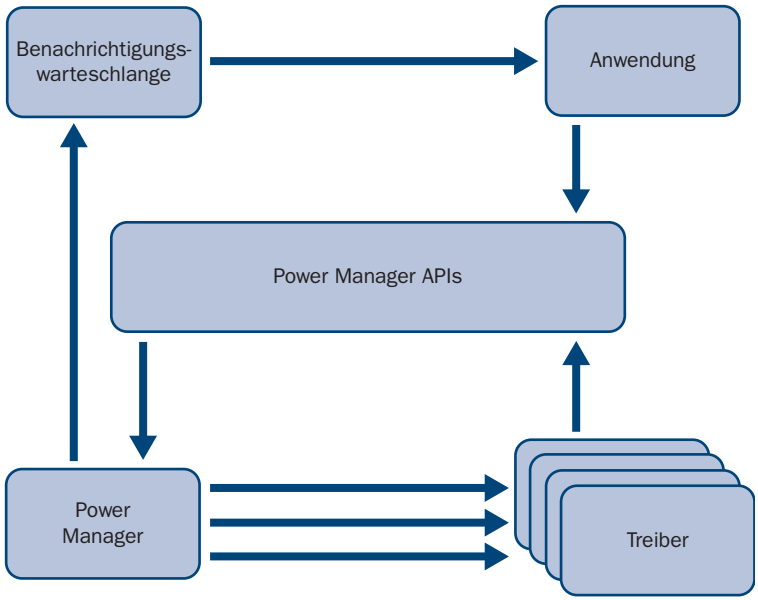


Abbildung 3.7 Interaktion zwischen dem Power Manager und der Energieverwaltung

Power Manager-Quellcode

Windows Embedded CE umfasst Quellcode für den Power Manager, der im Ordner *%_WINCEROOT%\Public\Common\Oak\Drivers\Pm* auf dem Entwicklungscomputer gespeichert ist. Durch Ändern des Codes können Sie die Energieverwaltungsmethoden auf einem Zielgerät anpassen. Beispielsweise kann ein OEM (Original Equipment Manufacturer) zusätzliche Logik implementieren, um bestimmte Komponenten vor dem Aufruf der Funktion *PowerOffSystem* zu deaktivieren. Die Methoden zum Klonen und Anpassen der Windows Embedded CE-Standardkomponenten sind in Kapitel 1 beschrieben.

Treiberenergiestatus

Anwendungen und Gerätetreiber können mit der Funktion *DevicePowerNotify* den Energiestatus von Peripheriegeräten steuern. Beispielsweise können Sie *DevicePowerNotify* aufrufen, um dem Power Manager mitzuteilen, dass Sie den Energiestand eines Treibers für die Hintergrundbeleuchtung, z.B. *BLK1*, ändern möchten. Der Power Manager erwartet, dass Sie einen der folgenden fünf Energiestände entsprechend der Geräteeigenschaften angeben:

- **D0** *Full On*. Das Gerät ist voll funktionsfähig.
- **D1** *Low On*. Das Gerät ist funktionsfähig, aber die Leistung ist geringer.
- **D2** *Standby*. Das Gerät ist teilweise mit Energie versorgt und wird auf Anforderung aktiviert.
- **D3** *Sleep*. Das Gerät ist teilweise mit Energie versorgt. In diesem Status kann das Gerät Interrupts auslösen, die die CPU aktivieren (vom Gerät initialisierte Aktivierung).
- **D4** *Off*. Das Gerät ist nicht mit Energie versorgt. In diesem Status sollte der Energieverbrauch des Geräts sehr gering sein.

HINWEIS Energiestatus von CE-Geräten

Die Werte für den Geräteenergiestatus (D0 bis D4) sind Richtlinien für OEMs für die Implementierung von Energieverwaltungsfunktionen auf Plattformen. Der Power Manager beschränkt den Energieverbrauch, die Reaktionsfähigkeit und Eigenschaften in diesen Energiezuständen nicht. Als allgemeine Regel gilt, dass ein Status mit einem höheren Wert weniger Energie verbraucht als ein Status mit einem niedrigeren Wert. Die Werte D0 und D1 werden für funktionsfähige Geräte festgelegt. Gerätetreiber, die den Energiestand eines physischen Geräts mit weniger Granularitäten verwalten, können eine Teilmenge der Energiezustände implementieren. D0 ist der einzige erforderliche Energiestatus.

Systemenergiestatus

Der Power Manager kann nicht nur Benachrichtigungen über Energiestatusänderungen auf Anforderung der Anwendungen und Gerätetreiber an die Gerätetreiber senden, sondern auch den Energiestatus des gesamten Systems aufgrund von Hardwareevents und Softwareanforderungen wechseln. Bei Hardwareevents reagiert der Power Manager auf niedrige oder kritische Batteriestände und wechselt vom Batteriebetrieb zu Netzstrom. Anwendungen reagieren auf Softwareanforderungen, indem sie durch Aufruf der Funktion *SetSystemPowerState* eine Änderung des Systemenergiestatus anfordern.

Die Power Manager-Standardimplementierung unterstützt die folgenden vier Systemenergiezustände:

- **On** Das System ist voll funktionsfähig und mit Energie versorgt.
- **UserIdle** Der Benutzer verwendet das Gerät passiv. Für eine konfigurierbare Zeitdauer erfolgt keine Benutzereingabe.
- **SystemIdle** Der Benutzer verwendet das Gerät nicht. Für eine konfigurierbare Zeitdauer erfolgt keine Systemaktivität.
- **Suspend** Das Gerät ist heruntergefahren, aber unterstützt die vom Gerät initialisierte Aktivierung.

Beachten Sie, dass die Systemenergiezustände von den Anforderungen und Eigenschaften des Zielgeräts abhängig sind. OEMs können zusätzliche Systemenergiezustände definieren, beispielsweise *InCradle* und *OutOfCradle*. In Windows Embedded CE ist die Anzahl der definierbaren Systemenergiezustände nicht begrenzt, aber alle Systemenergiezustände werden letztendlich in einen der Geräteenergiezustände umgewandelt, die bereits in dieser Lektion erwähnt wurden.

In Abbildung 3.8 ist die Beziehung zwischen den standardmäßigen Systemenergiezuständen und den Geräteenergiezuständen dargestellt.

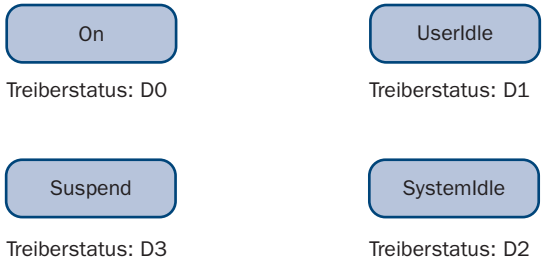


Abbildung 3.8 Standard-Systemenergiezustände und die zugehörigen Geräteenergiezustände

Aktivitätszeitgeber

Systemstatusübergänge basieren auf Aktivitätszeitgebern und den entsprechenden Events. Wenn das Gerät nicht verwendet wird, läuft der Zeitgeber ab und löst ein Inaktivitätsevent aus, das verursacht, dass der Power Manager das System in den Standby-Energiestatus versetzt. Sobald eine Benutzereingabe erfolgt, wird ein Aktivitätsevent ausgelöst und der Power Manager wechselt das System zurück in den *On*-Energiestatus. Dieses vereinfachte Modell berücksichtigt jedoch die langfristige Benutzeraktivität ohne Eingabe nicht, beispielsweise wenn der Benutzer einen

Videoclip auf einem PDA (Personal Digital Assistant) wiedergibt. Außerdem werden Zielgeräte ohne direkte Benutzereingabemethoden, beispielsweise Monitore, nicht berücksichtigt. Um diese Situationen zu unterstützen, unterscheidet die Power Manager-Standardimplementierung zwischen Benutzeraktivitäten und Systemaktivitäten und wechselt den Systemenergiestatus entsprechend (siehe Abbildung 3.9).

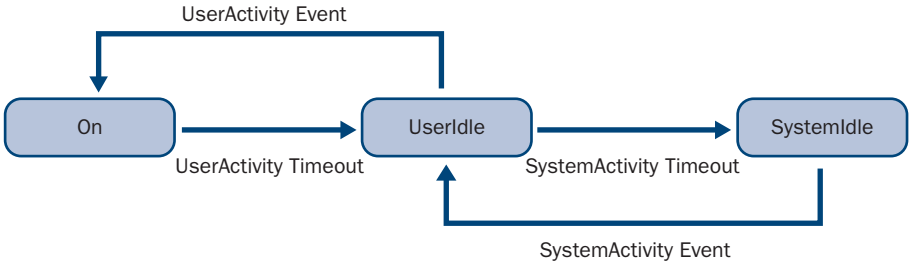


Abbildung 3.9 Übergang zwischen Aktivitätszeitgeber, Events und Systemenergiestatus

Über das Applet **Power Control Panel** können Sie Timeouts für die Systemaktivität und Benutzeraktivität konfigurieren. Außerdem können Sie zusätzliche Zeitgeber implementieren und die Timeouts festlegen, indem Sie die Registrierung direkt bearbeiten. In Windows Embedded CE ist die Anzahl der Zeitgeber, die Sie erstellen können, nicht begrenzt. Während des Systemstarts liest der Power Manager die Registrierungsschlüssel, listet die Aktivitätszeitgeber auf und erstellt die zugehörigen Events. In Tabelle 3.16 sind die Registrierungseinstellungen für den *SystemActivity*-Zeitgeber aufgeführt. OEMs können ähnliche Registrierungsschlüssel hinzufügen und die entsprechenden Werte für weitere Zeitgeber konfigurieren.

Tabelle 3.16 Registrierungseinstellungen für Aktivitätszeitgeber

Pfad	**HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Power\ActivityTimers\SystemActivity**	
Eintrag	Timeout	WakeSources
Typ	REG_DWORD	REG_MULTI_SZ
Wert	A (10 Minuten)	0x20

Tabelle 3.16 Registrierungseinstellungen für Aktivitätszeitgeber (Fortsetzung)

Pfad	HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Power\ActivityTimers\SystemActivity	
Beschreibung	Der Timeout-Registrierungseintrag definiert den Zeitgeberschwellwert in Minuten.	Der *WakeSources*-Registrierungseintrag ist optional und definiert eine Liste der IDs für mögliche Wake-Quellen. Während der vom Gerät initialisierten Aktivierung bestimmt der Power Manager mittels der *IOCTL_HAL_GET_WAKE_SOURCE*-Eingabe und dem IOCTL-Code (Output Control) die Aktivierungsquelle und legt die entsprechenden Aktivitätszeitgeber auf aktiv fest.

HINWEIS Aktivitätszeitgeber

Wenn Aktivitätszeitgeber festgelegt werden, erstellt der Power Manager benannte Events zum Zurücksetzen des Zeitgebers und zum Abrufen des Aktivitätsstatus. Weitere Informationen finden Sie im Abschnitt Activity Timers in der Windows Embedded CE 6.0-Dokumentation auf der Microsoft MSDN-Website unter *http://msdn2.microsoft.com/en-us/library/aa923909.aspx*.

Energieverwaltungs-API

Wie bereits erwähnt, umfasst der Power Manager die folgenden drei Schnittstellen, um Anwendungen und Laufwerke für die Energieverwaltung zu aktivieren: Benachrichtigungsschnittstelle, Treiberschnittstelle und Anwendungsschnittstelle.

Benachrichtigungsschnittstelle

Die Benachrichtigungsschnittstelle umfasst zwei Funktionen, die Anwendungen verwenden können, um sich über Meldungswarteschlangen für Energiebenachrichtigungen zu registrieren bzw. die Registrierung aufzuheben (siehe Tabelle 3.17). Beachten Sie, dass es sich bei Energiebenachrichtigungen um Multicastmeldungen handelt. Das heißt, der Power Manager sendet diese Benachrichtigungen ausschließlich an registrierte Prozesse. Auf diese Art können für die Energieverwal-

tung aktivierte Anwendungen nahtlos mit anderen Anwendungen, die das Energieverwaltungs-API nicht implementieren, in Windows Embedded CE koexistieren.

Tabelle 3.17 Benachrichtigungsschnittstelle der Energieverwaltung

Funktion	Beschreibung
RequestPowerNotifications	Registriert einen Anwendungsprozess mit dem Power Manager, um Energiebenachrichtigungen abzurufen. Der Power Manager sendet anschließend folgende Benachrichtigungen: ■ **PBT_RESUME** Das System wird aus dem Standby-Status reaktiviert. ■ **PBT_POWERSTATUSCHANGE** Das System wechselt zwischen Netzstrom und Batteriebetrieb. ■ **PBT_TRANSITION** Das System wechselt in einen neuen Energiestatus. ■ **PBT_POWERINFOCHANGE** Der Batteriestatus wird geändert. Diese Meldung ist nur gültig, wenn ein Batterietreiber geladen ist.
StopPowerNotifications	Hebt die Registrierung eines Anwendungsprozesses auf, der daraufhin keine Energiebenachrichtigungen mehr empfängt.

Der folgende Beispielcode veranschaulicht die Verwendung von Energiebenachrichtigungen:

```
// Size of a POWER_BROADCAST message.
DWORD cbPowerMsgSize =
    sizeof POWER_BROADCAST + (MAX_PATH * sizeof TCHAR);

// Initialize a MSGQUEUEOPTIONS structure.
MSGQUEUEOPTIONS mqo;
mqo.dwSize = sizeof(MSGQUEUEOPTIONS);
mqo.dwFlags = MSGQUEUE_NOPRECOMMIT;
mqo.dwMaxMessages = 4;
mqo.cbMaxMessage = cbPowerMsgSize;
mqo.bReadAccess = TRUE;

//Create a message queue to receive power notifications.
HANDLE hPowerMsgQ = CreateMsgQueue(NULL, &mqo);
```

```
if (NULL == hPowerMsgQ)
{
    RETAILMSG(1, (L"CreateMsgQueue failed: %x\n", GetLastError()));
    return ERROR;
}

// Request power notifications.
HANDLE hPowerNotifications = RequestPowerNotifications(hPowerMsgQ,
                                        PBT_TRANSITION |
                                        PBT_RESUME |
                                        PBT_POWERINFOCHANGE);

// Wait for a power notification or for the app to exit.
while(WaitForSingleObject(hPowerMsgQ, FALSE, INFINITE) == WAIT_OBJECT_0)
{
    DWORD cbRead;
    DWORD dwFlags;
    POWER_BROADCAST *ppb = (POWER_BROADCAST*) new BYTE[cbPowerMsgSize];

    // Loop through in case there is more than 1 msg.
    while(ReadMsgQueue(hPowerMsgQ, ppb, cbPowerMsgSize, &cbRead,
                       0, &dwFlags))
    {
      \\ Perform action according to the message type.
    }
}
```

Gerätetreiberschnittstelle

Für die Integration mit dem Power Manager müssen Gerätetreiber mehrere E/A-Steuerelemente (IOCTLs) unterstützen.. Der Power Manager verwendet diese, um die gerätespezifischen Energieeigenschaften abzufragen und den Energiestatus des Geräts festzulegen bzw. zu ändern (siehe Abbildung 3.10). Der Gerätetreiber sollte für das Hardwaregerät basierend auf den Power Manager IOCTLs eine entsprechende Energiekonfiguration auswählen.

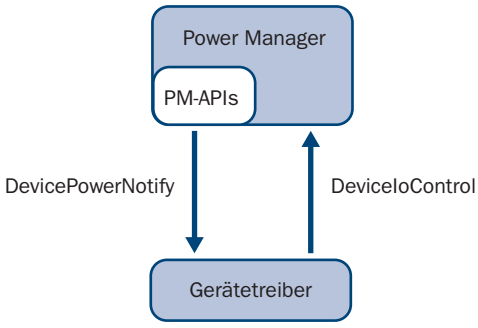


Abbildung 3.10 Interaktion zwischen dem Power Manager und dem Gerätetreiber

Der Power Manager verwendet folgende IOCTLs, um mit den Gerätetreibern zu interagieren:

- **IOCTL_POWER_CAPABILITIES** Der Power Manager überprüft die Energieverwaltungseigenschaften des Gerätetreibers. Die zurückgegebenen Informationen reflektieren die Eigenschaften der Hardware und der Treiber für das Hardwaregerät. Der Treiber darf nur einen unterstützten Dx-Status zurückgeben.
- **IOCTL_POWER_SET** Der Power Manager zwingt den Treiber in einen angegebenen Dx-Status zu wechseln. Der Treiber muss den Energiewechsel ausführen.
- **IOCTL_POWER_QUERY** Der Power Manger überprüft, ob der Treiber den Status des Geräts ändern kann.
- **IOCTL_POWER_GET** Der Power Manager möchte den aktuellen Energiestatus des Geräts bestimmen.
- **IOCTL_REGISTER_POWER_RELATIONSHIP** Der Power Manager benachrichtigt einen übergeordneten Treiber, um die untergeordneten Geräte zu registrieren. Der Power Manager sendet dieses IOCTL ausschließlich an Geräte mit dem Flag *POWER_CAP_PARENT* im Flags-Member der *POWER_CAPABILITIES*-Struktur.

HINWEIS Interne Energiestatusübergänge

Um die zuverlässige Energieverwaltung sicherzustellen, sollten die Gerätetreiber ihren internen Energiestatus nicht ohne den Power Manager wechseln. Wenn ein Treiber einen Energiestatusübergang erfordert, sollte er den Wechsel mit der Funktion *DevicePowerNotify* anfordern. Der Treiber kann anschließend seinen internen Energiestatus ändern, wenn der Power Manager eine Änderungsanforderung zurück an den Treiber sendet.

Anwendungsschnittstelle

Die Anwendungsschnittstelle umfasst Funktionen, mit denen Anwendungen den Energiestatus des Systems und einzelner Geräte über den Power Manager verwalten können. In Tabelle 3.18 sind diese Energieverwaltungsfunktionen aufgeführt.

Tabelle 3.18 Anwendungsschnittstelle

Funktion	Beschreibung
GetSystemPowerState	Ruft den aktuellen Systemenergiestatus ab.
SetSystemPowerState	Fordert eine Energiestatusänderung an. Beim Wechsel in den Standby-Modus wird die Funktion nach der Reaktivierung zurückgegeben, da der Standby für das System transparent ist. Sie können anschließend die Benachrichtigungsmeldung analysieren, um sicherzustellen, dass das System aus dem Standby-Modus reaktiviert wurde.
SetPowerRequirement	Fordert einen minimalen Energiestatus für das Gerät an.
ReleasePowerRequirement	Gibt eine zuvor mit der Funktion *SetPowerRequirement* festgelegte Energieanforderung frei und stellt den ursprünglichen Geräteenergiestatus wieder her.
GetDevicePower	Rufen den aktuellen Energiestatus eines Geräts ab.
SetDevicePower	Fordert einen Energiestatus für ein Gerät an.

Energiestatuskonfiguration

Wie in Abbildung 3.8 dargestellt, ordnet der Power Manager den Systemenergiestatus dem Geräteenergiestatus zu, um sicherzustellen, dass das System und die Geräte synchronisiert sind. Wenn der Power Manager nicht anders konfiguriert ist, erzwingt er die folgenden Systemstatus/Gerätestatus-Zuordnungen: *On = D0*, *UserIdle = D1*, *SystemIdle = D2* und *Suspend = D3*. Sie können die Zuordnung für einzelne Geräte und Geräteklassen mit expliziten Registrierungseinstellungen überschreiben.

Überschreiben der Energiestatuskonfiguration für ein Gerät

Die Power Manager-Standardimplementierung verwaltet die Systemstatus/Gerätestatus-Zuordnungen in der Registrierung unter dem Schlüssel *HKEY_LOCAL_MACHINE\System\CurrentControlSet\State*. Jeder Systemenergiestatus entspricht einem separaten Unterschlüssel. Sie können für den OEM-spezifischen Energiestatus weitere Unterschlüssel erstellen.

In Tabelle 3.19 ist eine Beispielkonfiguration für den Systemenergiestatus *On* aufgeführt. Mit dieser Konfiguration wechselt der Power Manager alle Geräte, außer den Treiber für die Hintergrundbeleuchtung *BLK1:*, in den **D0**-Geräteenergiestatus. Der Treiber *BLK1:* kann nur in den **D2**-Status versetzt werden.

Tabelle 3.19 Standardmäßige und treiberspezifische Energiestatusdefinitionen für den Systemenergiestatus On

Pfad	**HKEY_LOCAL_MACHINE\System\CurrentControlSet\State\On**		
Eintrag	Flags	Standard	BKL1:
Typ	REG_DWORD	REG_DWORD	REG_DWORD
Wert	0x00010000 (POWER_STATE_ON)	0 (D0)	2 (D2)
Beschreibung	Ermittelt den Systemenergiestatus, der diesem Registrierungs-schlüssel zugeordnet ist. Eine Liste der möglichen Flags finden Sie in der Datei *Pm.h* im Ordner *Public\Common\Sdk \Inc.*	Legt den Energiestatus für Treiber standardmäßig auf **D0** fest, wenn der Systemenergiestatus *On* ist.	Legt den Treiber für die Hintergrundbeleuchtung auf **D2** fest, wenn der Systemenergiestatus *On* ist.

Überschreiben der Energiestatuskonfiguration für Geräteklassen

Das Definieren des Geräteenergiestatus für mehrere Systemenergiezustände ist eine mühsame Aufgabe. Der Power Manager unterstützt die Konfiguration durch auf *IClass*-Werten basierenden Geräteklassen, die zum Definieren der Energieverwaltungsregeln verwendet werden können. Die folgenden drei Standardklassendefinitionen befinden sich unter dem Registrierungsschlüssel *HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Power\Interfaces*.

- **{A3292B7-920C-486b-B0E6-92A702A99B35}** Allgemeine für die Energieverwaltung aktivierte Geräte.
- **{8DD679CE-8AB4-43c8-A14A-EA4963FAA715}** Für die Energieverwaltung aktivierte Blockgeräte.
- **{98C5250D-C29A-4985-AE5F-AFE5367E5006}** Für die Energieverwaltung aktivierte NDIS-Miniportreiber (Network Driver Interface Specification).

In Tabelle 3.20 ist eine Beispielkonfiguration für die NDIS-Geräteklasse aufgeführt, die den D4-Status als maximalen Wert für NDIS-Treiber festlegt.

Tabelle 3.20 Tabelle 3.20 Beispieldefinition des Energiestatus für die NDIS-Geräteklasse

Pfad	**HKEY_LOCAL_MACHINE\System\ CurrentControlSet \Control \Power\State\On\{98C5250D-C29A-4985-AE5F-AFE5367E5006}**
Eintrag	Standard
Typ	REG_DWORD
Wert	4 (**D4**)
Beschreibung	Legt den Geräteenergiestatus für NDIS-Treiber auf **D4** fest, wenn der Systemenergiestatus *On* ist.

Prozessor-Leerlaufstatus

Neben den für die Energieverwaltung aktivierten Anwendungen und Gerätetreibern, trägt auch der Kernel zur Energieverwaltung bei. Der Kernel ruft die Funktion OEMIdle auf (Teil der OAL), wenn keine Threads zum Ausführen verfügbar sind. Diese Aktion versetzt den Prozessor in den Leerlaufstatus und umfasst das Speichern des aktuellen Kontextes, das Wechseln des Speichers in den Aktualisierungsstatus und das Anhalten der Systemuhr. Der Prozessor-Leerlaufstatus reduziert den Energieverbrauch auf das niedrigste Niveau, wobei der Prozessor jedoch schnell reaktiviert werden kann.

Beachten Sie, dass die Funktion OEMIdle den Power Manager nicht einbezieht. Der Kernel ruft die Funktion OEMIdle direkt auf und die Hardware wird von OAL in den geeigneten Leerlaufstatus oder Ruhezustand versetzt. Der Kernel übergibt einen DWORD-Wert (dwReschedTime) an *OEMIdle*, um die maximale Leerlaufzeit anzugeben. Wenn die Zeitdauer abgelaufen oder die maximale vom Hardwarezeitgeber unterstützte Verzögerung erreicht ist, wird der Prozessor reaktiviert, der vorherige Status wiederhergestellt und der Scheduler aufgerufen. Wenn immer noch kein Thread verfügbar ist, ruft der Kernel die Funktion *OEMIdle* umgehend erneut auf. Treiberevents, beispielsweise als Reaktion auf eine Benutzereingabe, können jederzeit auftreten und verursachen, dass das System aus dem Leerlaufstatus reaktiviert wird, bevor der Systemzeitgeber startet.

Der Scheduler basiert standardmäßig auf einem statischen Zeitgeber und Systemticks in Intervallen von 1 Millisekunde. Das System kann den Energieverbrauch jedoch unter Verwendung von dynamischen Zeitgebern und durch Festlegen des Systemzeitgebers auf den nächsten vom Scheduler ermittelten Timeout optimieren. Der Prozessor wechselt anschließend nicht bei jedem Tick wieder in den Leerlaufmodus. Stattdessen wird der Prozessor nur reaktiviert, nachdem der von *dwReschedTime* festgelegte Timeout abgelaufen oder ein Interrupt aufgetreten ist.

Zusammenfassung

Windows Embedded CE 6.0 R2 umfasst eine Power Manager-Standardimplementierung mit Energieverwaltungs-APIs, mit denen Sie den Energiestatus des Systems und der Geräte verwalten können. Die Funktion *OEMIdle* wird ausgeführt, wenn keine Threads geplant sind, um OEMs zu ermöglichen, das System für eine bestimmte Zeitdauer in den Leerlaufstatus zu versetzen.

Der Power Manager ist eine Kernelkomponente, die eine Benachrichtigungs-, eine Anwendungs- und eine Geräteschnittstelle umfasst. Der Power Manager agiert als Vermittler zwischen dem Kernel und OAL sowie zwischen den Gerätetreibern und Anwendungen. Anwendungen und Gerätetreiber können mit der Funktion *DevicePowerNotify* den Energiestatus von Peripheriegeräten auf fünf Ebenen steuern. Der Geräteenergiestatus kann einem Systemenergiestatus zugeordnet werden, damit das System und die Geräte synchronisiert sind. Der Power Manger kann basierend auf den Aktivitätszeiten und den entsprechenden Events den Systemstatus automatisch wechseln. Die vier Standardwerte für den Systemenergiestatus sind *On*, *UserIdle*, *SystemIdle* und *Suspend*. Die Systemstatus/Gerätestatus-Zuordnungen werden mit den Registrierungseinstellungen für Geräte und Geräteklassen angepasst.

Der Kernel unterstützt außerdem die Energieverwaltung mit der Funktion *OEMIdle*. Der Wechsel des Prozessors in den Leerlaufstatus reduziert den Energieverbrauch auf das niedrigste Niveau, wobei der Prozessor jedoch schnell reaktiviert werden kann. Der Prozessor wird regelmäßig oder bei einem Interrupt reaktiviert, beispielsweise als Reaktion auf die Benutzereingabe oder wenn ein Gerät für die Datenübertragung den Zugriff auf den Speicher anfordert.

Sie können den Energieverbrauch eines Geräts wesentlich reduzieren, wenn Sie die Energieverwaltung mit dem Power Manager und *OEMIdle* richtig implementieren und dadurch die Lebensdauer der Batterie und des Geräts verlängern sowie die Betriebskosten senken.

Lab 3: Kioskmodus, Threads und Energieverwaltung

In diesem Lab entwickeln Sie eine Kioskanwendung und konfigurieren ein Zielgerät zum Ausführen der Anwendung anstatt der Standardshell. Anschließend erweitern Sie die Anwendung, um im Anwendungsprozess mehrere Threads gleichzeitig auszuführen, und analysieren die Threadausführung mit dem Tool Remote Kernel Tracker. Zum Schluss aktivieren Sie die Anwendung für die Energieverwaltung.

HINWEIS Detailierte schrittweise Anleitungen

Um die Verfahren in diesem Lab erfolgreich auszuführen, lesen Sie das Dokument Detailed Step-by-Step Instructions for Lab 3 im Begleitmaterial.

▶ Erstellen eines Threads

1. Erstellen Sie mit dem **New Project Wizard** die neue WCE-Konsolenanwendung ***HelloWorld***. Verwenden Sie die Option **Typical Hello_World Application**.
2. Implementieren Sie vor der Funktion *_tmain* die Threadfunktion *ThreadProc*:

```
DWORD WINAPI ThreadProc( LPVOID lpParameter)
{
    RETAILMSG(1,(TEXT("Thread started")));

    // Suspend Thread execution for 3 seconds
    Sleep(3000);

    RETAILMSG(1,(TEXT("Thread Ended")));

    // Return code of the thread 0,
    // usually used to indicate no errors.
    return 0;
}
```

3. Starten Sie einen Thread mit der Funktion *CreateThread*:

```
HANDLE hThread = CreateThread( NULL, 0, ThreadProc, NULL, 0, NULL);
```

4. Überprüfen Sie den von *CreateThread* zurückgegebenen Wert, um sicherzustellen, dass der Thread erfolgreich erstellt wurde.
5. Warten Sie, bis der Thread das Ende der Threadfunktion erreicht und beenden Sie den Thread:

```
WaitForSingleObject(hThread, INFINITE);
```

6. Erstellen Sie das Run-Time Image und downloaden Sie dieses auf das Zielgerät.

7. Starten Sie das Tool Remote Kernel Tracker und analysieren Sie die Threadverwaltung.
8. Starten Sie die *HelloWorld*-Anwendung und überwachen Sie die Threadausführung im Fenster **Windows CE Remote Kernel Tracker** (siehe Abbildung 3.11).

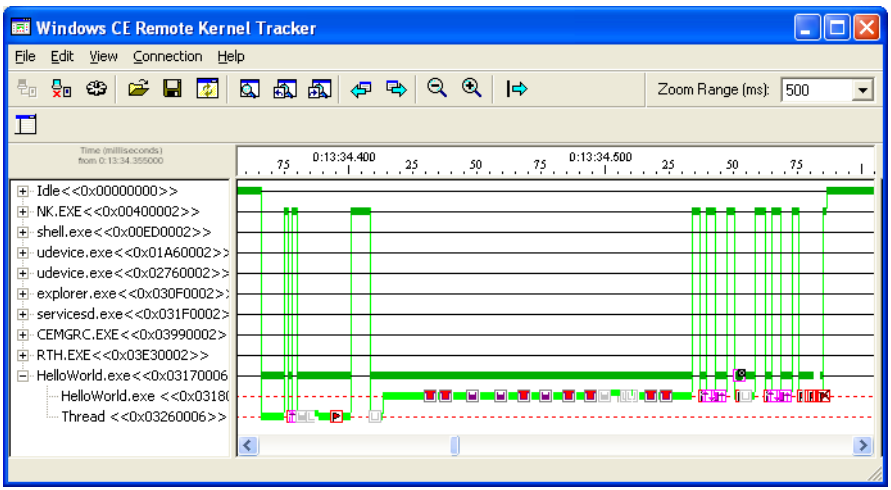


Abbildung 3.11 Überwachen der Threadausführung im Tool Remote Kernel Tracker

▶ Aktivieren von Benachrichtigungen der Energieverwaltung

1. Verwenden Sie die *HelloWorld*-Anwendung weiterhin in Visual Studio.
2. Generieren Sie die Benachrichtigungen in häufigeren Intervallen, indem Sie in den Registrierungseinstellungen für das Teilprojekt den Eintrag für *UserIdle* (*ACUserIdle*) auf fünf Sekunden festlegen:

```
[HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Power\Timeouts]
    "ACUserIdle"=dword:5                ; in seconds
```

3. Erstellen Sie in der Funktion *ThreadProc* ein Meldungswarteschlangenobjekt:

```
// Size of a POWER_BROADCAST message.
DWORD cbPowerMsgSize =
    sizeof POWER_BROADCAST + (MAX_PATH * sizeof TCHAR);

// Initialize our MSGQUEUEOPTIONS structure.
MSGQUEUEOPTIONS mqo;
mqo.dwSize = sizeof(MSGQUEUEOPTIONS);
mqo.dwFlags = MSGQUEUE_NOPRECOMMIT;
mqo.dwMaxMessages = 4;
mqo.cbMaxMessage = cbPowerMsgSize;
```

```
mqo.bReadAccess = TRUE;

//Create a message queue to receive power notifications.
HANDLE hPowerMsgQ = CreateMsgQueue(NULL, &mqo);
if (NULL == hPowerMsgQ)
{
    RETAILMSG(1, (L"CreateMsgQueue failed: %x\n", GetLastError()));
    return -1;
}
```

4. Fordern Sie Benachrichtigungen vom Power Manager an und überprüfen Sie die empfangenen Meldungen:

```
// Request power notifications
HANDLE hPowerNotifications = RequestPowerNotifications(hPowerMsgQ,
                                        PBT_TRANSITION |
                                        PBT_RESUME |
                                        PBT_POWERINFOCHANGE);
DWORD dwCounter = 20;

// Wait for a power notification or for the app to exit
while(dwCounter-- &&
      WaitForSinglObject(hPowerMsgQ, INFINITE) == WAIT_OBJECT_0)
{
    DWORD cbRead;
    DWORD dwFlags;
    POWER_BROADCAST *ppb =
      (POWER_BROADCAST*) new BYTE[cbPowerMsgSize];

    // loop through in case there is more than 1 msg.
    while(ReadMsgQueue(hPowerMsgQ, ppb, cbPowerMsgSize,
                       &cbRead, 0, &dwFlags))
    {
        switch(ppb->Message)
        {
            case PBT_TRANSITION:
            {
                RETAILMSG(1,(L"Notification: PBT_TRANSITION\n"));
                if(ppb->Length)
                {
                    RETAILMSG(1,(L"SystemPowerState: %s\n",
                                 ppb->SystemPowerState));
                }
                break;
            }
            case PBT_RESUME:
            {
                RETAILMSG(1,(L"Notification: PBT_RESUME\n"));
                break;
            }
            case PBT_POWERINFOCHANGE:
            {
```

```
                    RETAILMSG(1,(L"Notification: PBT_POWERINFOCHANGE\n"));
                    break;
                }
                default:
                    break;
            }
        }
        delete[] ppb;
    }
```

5. Erstellen Sie die Anwendung und das Run-Time Image neu.
6. Starten Sie das Run-Time Image.
7. Benutzeraktivität wird durch Bewegen des Mauscursors generiert. Der Power Manager sollte die Anwendung nach fünf Sekunden Inaktivität benachrichtigen (siehe Abbildung 3.12).

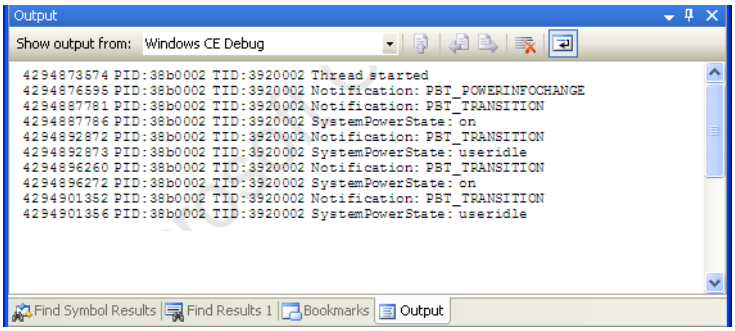


Abbildung 3.12 Empfangene Benachrichtigungen der Energieverwaltung

▶ Aktivieren des Kioskmodus

1. Erstellen Sie mit dem **Subproject Wizard** die WCE-Anwendung ***Subproject_Shell***. Verwenden Sie die Option **Typical Hello_World Application**.
2. Fügen Sie vor der ersten *LoadString*-Zeile eine *SignalStarted*-Anweisung ein.

```
// Initialization complete,
// call SignalStarted...
SignalStarted(_wtol(lpCmdLine));
```

3. Erstellen Sie die Anwendung.
4. Fügen Sie einen Registrierungsschlüssel in der Teilprojektdatei .reg hinzu, um die Anwendung beim Systemstart zu starten. Fügen Sie folgende Zeilen hinzu, die die entsprechenden *Launch99*- und *Depend99*-Einträge erstellen:

```
[HKEY_LOCAL_MACHINE\INIT]
"Launch99"="Subproject_Shell.exe"
"Depend99"=hex:14,00, 1e,00
```

5. Erstellen und starten Sie das Run-Time Image.
6. Überprüfen Sie, ob die Anwendung ***Subproject_Shell*** automatisch gestartet wird.
7. Ersetzen Sie den Verweis auf *Explorer.exe* im Registrierungsschlüssel *Launch50* durch einen Verweis auf die Anwendung **Subproject_Shell**:

```
[HKEY_LOCAL_MACHINE\INIT]
"Launch50"="Subproject_Shell.exe"
"Depend50"=hex:14,00, 1e,00
```

8. Erstellen und starten Sie das Run-Time Image.
9. Überprüfen Sie, dass auf dem Zielgerät die Anwendung ***Subproject_Shell*** anstatt der Standardshell ausgeführt wird (siehe Abbildung 3.13).

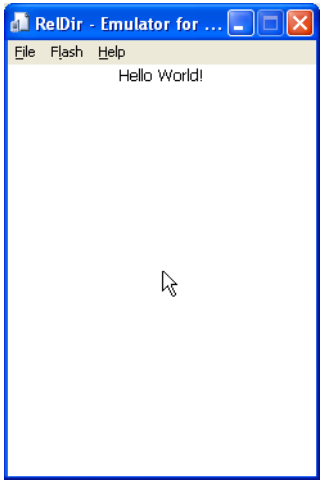


Abbildung 3.13 Ersetzen der Standardshell durch eine Subproject_Shell-Anwendung

Lernzielkontrolle

Windows Embedded CE umfasst zahlreiche Tools, Features und APIs, mit denen Sie die optimale Systemleistung und den geringsten Energieverbrauch auf dem Zielgerät sicherstellen können. Die Leistungstools, beispielsweise ILTiming, OSBench und Remote Performance Monitor, sind nützlich, um Leistungsprobleme in und zwischen Treibern, Anwendungen und dem OAL-Code zu identifizieren, beispielsweise Deadlocks oder Probleme mit der Threadsynchronisierung. Das Tool Remote Kernel Tracker ermöglicht das Überprüfen der Prozess- und Threadausführung abhängig von der strukturierten Ausnahmebehandlung, die Windows Embedded CE unterstützt.

Windows Embedded CE ist ein auf Komponenten basierendes Betriebssystem. Sie können Komponenten einbeziehen oder ausschließen und die Standardshell durch eine benutzerdefinierte Anwendung ersetzen. Das Ersetzen der Standardshell durch eine für den automatischen Start konfigurierte Anwendung ermöglicht eine Kioskkonfiguration. Windows Embedded CE wird in einer Kioskkonfiguration mit einer „schwarzen Shell“ ausgeführt. Das heißt, der Benutzer kann keine anderen Anwendungen auf dem Gerät starten oder verwenden.

Unabhängig von der Shell können Sie Energieverwaltungsfunktionen in den Gerätetreibern und Anwendungen implementieren, um den Energieverbrauch zu steuern. Die Power Manager-Standardimplementierung erfüllt zwar alle typischen Anforderungen, aber OEMs mit speziellen Anforderungen müssen eine benutzerdefinierte Logik hinzufügen. Windows Embedded CE umfasst den Power Manager-Quellcode. Das Energieverwaltungsframework ist flexibel und unterstützt mehrere benutzerdefinierte Systemenergiezustände, die den Geräteenergiestatus über Registrierungseinstellungen zuordnen können.

Schlüsselbegriffe

Kennen Sie die Bedeutung der folgenden wichtigen Begriffe? Sie können Ihre Antworten überprüfen, wenn Sie die Begriffe im Glossar am Ende dieses Buches nachschlagen.

- ILTiming
- Kioskmodus
- Synchronisierungsobjekte
- Power Manager
- RequestDeviceNotifications

Empfohlene Vorgehensweise

Um die in diesem Kapitel behandelten Prüfungsschwerpunkte erfolgreich zu beherrschen, sollten Sie die folgenden Aufgaben durcharbeiten.

Verwenden Sie die Tools ILTiming und OSBench

Mit den Iltiming und OSBench können Sie auf dem Emulatorgerät die Leistung des emulierten ARMV4-Prozessors überprüfen.

Implementieren Sie eine benutzerdefinierte Shell

Passen Sie das Design des Zielgeräts mit dem Task-Manager in Windows Embedded CE an, um die Shell zu ersetzen.

Experimentieren Sie mit Multithread-Anwendungen und Critical Sections

Schützen Sie den Zugriff auf eine globale Variable mit Critical Section-Objekten in einer Multithread-Anwendung. Führen Sie folgende Aufgaben aus:

1. Erstellen Sie im Hauptcode der Anwendungen zwei Threads und warten Sie in den Threadfunktionen zwei Sekunden (*Sleep(2000)*) und drei Sekunden (*Sleep(3000)*) in einer Endlosschleife. Der primäre Thread der Anwendung sollte warten, bis beide Threads mit der Funktion *WaitForMultipleObjects* beendet werden.
2. Erstellen Sie eine globale Variable und greifen Sie in beiden Threads auf die Variable zu. Ein Thread sollte in die Variable schreiben und der andere Thread sollte die Variable lesen. Durch den Zugriff auf die Variable vor und nach dem ersten Ruhezustand und das Anzeigen der Werte, sollten Sie den gleichzeitigen Zugriff visualisieren können.
3. Schützen Sie den Zugriff auf die Variable mit einem CriticalSection-Objekt, das von beiden Threads verwendet wird. Übernehmen Sie den kritischen Abschnitt am Anfang der Schleifen und geben Sie diesen am Schleifenende frei. Vergleichen Sie die Ergebnisse mit der vorherigen Ausgabe.

Kapitel 4

Debuggen und Testen des Systems

Debuggen und Testen sind wichtige Aufgaben bei der Softwareentwicklung, um Software- und Hardwareprobleme auf einem Zielgerät zu identifizieren. Beim Debuggen werden sowohl der Code durchlaufen als auch die Debugmeldungen während der Codeausführung analysiert, um die Ursache von Fehlern zu ermitteln. Das Debuggen ist außerdem ein effizientes Tool zum Überprüfen der Implementierung der Systemkomponenten und Anwendungen. Mit Systemtests wird die Qualität der endgültigen Systemkonfiguration in Bezug auf Leistung, Zuverlässigkeit, Sicherheit und andere relevante Aspekte überprüft. Im Gegensatz zum Debuggen, das der Erkennung und Behebung von Problemursachen dient, werden mit Systemtests Produktfehler und Probleme ermittelt, beispielsweise Speicherverlust, sowie Deadlocks und Hardwarekonflikte. Für viele Entwickler von Small Footprint- und Konsumgeräten ist das Erkennen und Beheben von Systemfehlern der schwierigste Teil der Softwareentwicklung, was sich negativ auf die Produktivität auswirkt. In diesem Kapitel sind die Debug- und Testtools in Microsoft® Visual Studio® 2005 mit Platform Builder für Microsoft Windows® Embedded CE 6.0 R2 und das Windows Embedded CE Test Kit (CETK) beschrieben, mit denen Sie diese Prozesse automatisieren und beschleunigen können, um Ihre Systeme schneller und mit weniger Fehlern auf den Markt zu bringen. Je besser Sie diese Tools beherrschen, desto mehr Zeit können Sie mit dem Programmieren anstatt dem Korrigieren von Code verbringen.

Prüfungsziele in diesem Kapitel

- Identifizieren der Anforderungen für das Debuggen des Run-Time Images
- Analysieren der Codeausführung unter Verwendung der Debuggerfunktionen
- Verstehen der Debugzonen, um die Ausgabe von Debugmeldungen zu verwalten
- Ausführen von Standardtests und benutzerdefinierten Tests mit dem CETK-Tool
- Debuggen des Boot Loaders und Betriebssystems

Bevor Sie beginnen

Damit Sie die Lektionen in diesem Kapitel durcharbeiten können, benötigen Sie:

- Mindestens Grundkenntnisse in der Windows Embedded CE-Softwareentwicklung und den Debugkonzepten.
- Grundkenntnisse der in Windows Embedded CE unterstützten Treiberarchitektur.
- Erfahrung mit dem OS Design und den Systemkonfigurationskonzepten.
- Einen Entwicklungscomputer, auf dem Microsoft Visual Studio 2005 Service Pack 1 und Platform Builder für Windows Embedded CE 6.0 installiert ist.

Lektion 1: Erkennen softwarebezogener Fehler

Softwarebezogene Fehler umfassen sowohl einfache Tippfehler, nicht initialisierte Variablen und Endlosschleifen als auch komplexe und schwerwiegende Probleme, beispielsweise kritische Racebedingungen und andere Threadsynchronisierungsprobleme. Die meisten Fehler können jedoch einfach behoben werden, wenn erst einmal deren Ursache festgestellt wurde. Die kosteneffektivste Methode zum Ermitteln dieser Fehler ist die Codeanalyse. Sie können auf Windows Embedded CE-Geräten zahlreiche Tools verwenden, um das Betriebssystem zu debuggen und Treiber sowie Anwendungen zu überprüfen. Sie sollten mit den Debugtools vertraut sein, um die Codeanalyse zu beschleunigen und Softwarefehler effizient zu beheben.

Nach Abschluss dieser Lektion können Sie:

- Die wichtigen Debugtools für Windows Embedded CE identifizieren.
- Debugmeldungen über die Debugzonen in Treibern und Anwendungen steuern.
- Die Target Control-Shell verwenden, um Speicherprobleme zu identifizieren.

Veranschlagte Zeit für die Lektion: 90 Minuten.

Debuggen und Zielgerätesteuerung

Das primäre Tool zum Debuggen und Steuern eines Windows Embedded CE-Zielgeräts auf dem Entwicklungscomputer ist Platform Builder (siehe Abbildung 4.1). Die Platform Builder IDE (Integrated Development Environment) umfasst zahlreiche Tools, beispielsweise einen Systemdebugger, die CESH (CE Target Control Shell) und das Debugmeldungsfeature (DbgMsg), mit dem Sie den Code beim Erreichen eines Breakpoints durchlaufen oder Informationen über den Speicher, Variablen und Prozesse anzeigen können. Außerdem umfasst die Platform Builder IDE eine Reihe von Remotetools, beispielsweise Heap Walker, Process Viewer und Kernel Tracker, um den Status des Zielgeräts zur Laufzeit zu analysieren.

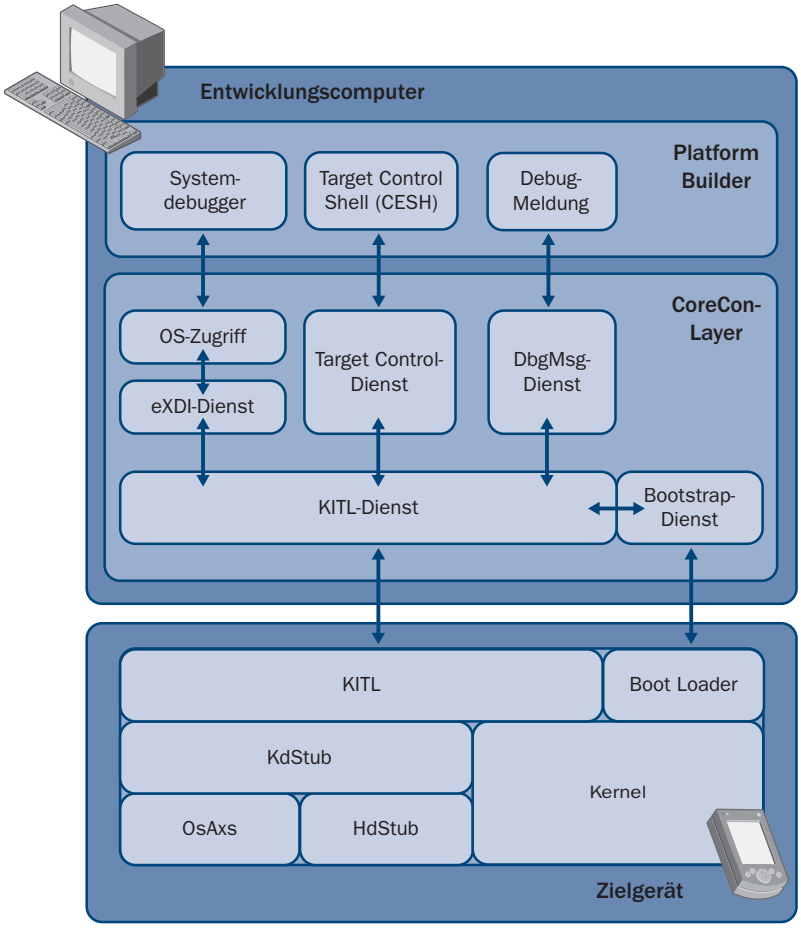


Abbildung 4.1 CE-Debuggen und Target Control-Architektur

Für die Kommunikation mit dem Zielgerät verwendet Platform Builder die CoreCon-Infrastruktur (Core Connectivity) und die Debugkomponenten, die auf dem Zielgerät als Bestandteil des Run-Time Images bereitgestellt sind. Die CoreCon-Infrastruktur umfasst OS Access (OsAxS), Target Control und DbgMsg-Dienste für Platform Builder sowie Schnittstellen mit dem Zielgerät über KITL (Kernel Independent Transport Layer) und den Bootstrap-Dienst. Auf dem Zielgerät hängt das Debuggen und die Target Control-Architektur für die Kommunikation von KITL und dem Boot Loader ab. Wenn das Run-Time Image Debugkomponenten umfasst, beispielsweise den KdStub (Kernel Debugger Stub), den HdStub (Hardware Debugger Stub) und die OsAxS-Bibliothek, können Sie die Kernel-Laufzeitinformationen mit Platform Builder abrufen und das JIT-Debuggen (Just-In-Time) ausführen. Platform Builder unterstützt außerdem das

hardwareunterstützte Debuggen über Extended Debugging Interface (eXDI), mit dem Sie die Zielgeräteroutinen vor dem Laden des Kernels debuggen können.

Kerneldebugger

Der Kerneldebugger ist das CE-Software-Debugmodul, mit dem Sie die Kernelkomponenten und CE-Anwendungen debuggen können. Auf dem Entwicklungscomputer arbeiten Sie direkt in Platform Builder, beispielsweise um im Quellcode Breakpoints einzufügen oder zu entfernen und die Anwendung auszuführen. Sie müssen jedoch die Unterstützung für KITL und die Debugbibliotheken (KdStub und OsAxS) in das Run-Time Image einbeziehen, damit Platform Builder die Debuginformationen erfassen und das Zielgerät steuern kann. Lektion 2 „Konfigurieren des Run-Time Images, um das Debuggen zu aktivieren" enthält detaillierte Informationen zu den Systemkonfigurationen für das Kerneldebuggen.

Die folgenden Komponenten auf dem Zielgerät sind für das Kerneldebuggen erforderlich:

- **KdStub** Erfasst Ausnahmen und Breakpoints, ruft die Kernelinformationen ab und führt Kernelvorgänge aus.
- **OsAxS** Ruft die Statusinformationen des Betriebssystems ab, beispielsweise über die Speicherzuordnung, aktive Prozesse, aktive Threads, Proxys und geladene DLLs.

HINWEIS Anwendungsdebuggen in Windows Embedded CE

Mit dem Kerneldebugger können Sie sowohl das gesamte Run-Time Image als auch einzelne Anwendungen steuern. KdStub ist jedoch eine Kernelkomponente, die Ausnahmen (*First-Chance* und *Second-Chance*) abfängt (siehe Kapitel 3 „Systemprogrammierung"). Wenn Sie den Kerneldebugger während einer Sitzung anhalten, ohne zuerst das KdStub-Modul auf dem Zielgerät zu beenden, und eine Ausnahme auftritt, reagiert das Run-Time Image nicht mehr, bis Sie den Debugger erneut starten, da der Kerneldebugger die Ausnahme behandeln muss, damit das Zielgerät weiter ausgeführt werden kann.

Debug Message-Dienst

Wenn Sie in Platform Builder ein KITL- und KdStub-aktiviertes Zielgerät starten, können Sie die Debuginformationen, die Platform Builder mit dem im DbgMsg-Dienst in der CoreCon-Infrastruktur vom Zielgerät abruft, im Fenster **Output** von Microsoft Visual Studio 2005 anzeigen. Debugmeldungen enthalten detaillierte Informationen zu den aktiven Prozessen, zeigen potenziell schwerwiegende

Probleme an, beispielsweise ungültige Eingaben, und weisen auf die Stelle eines Codefehlers hin, den Sie näher überprüfen können, indem Sie einen Breakpoint festlegen und den Code im Kerneldebugger durchlaufen. Eines der KdStub-Features ist die Unterstützung der dynamischen Verwaltung von Debugmeldungen, um das Konfigurieren der Ausführlichkeit der Meldungen ohne Quellcodeänderungen zu ermöglichen. Sie können unter anderem Zeitstempel, Prozess-IDs oder Thread-IDs ausschließen, wenn Sie die Debugmeldungsoptionen über das Menü **Target** in Visual Studio anzeigen. Außerdem können Sie die Debugausgabe in einer Datei speichern, um diese in einem anderen Tool zu analysieren. Auf dem Zielgerät werden alle Debugmeldungen direkt aus dem Standardausgabestream gesendet, der über die Funktion *NKDbgPrintf* verarbeitet wird.

HINWEIS Debugmeldungen mit und ohne KITL

Wenn Kerneldebugger und KITL aktiviert sind, werden die Debugmeldungen im Fenster Output von Visual Studio angezeigt. Wenn KITL nicht verfügbar ist, werden die Debuginformationen über einen seriellen Port, der von OEM OAL konfiguriert und verwendet wird, vom Zielgerät auf den Entwicklungscomputer übertragen.

Makros für Debugmeldungen

Zum Generieren von Debuginformationen stellt Windows Embedded CE mehrere Makros bereit, die in zwei Kategorien fallen: Debugmakros und Retailmakros. Debugmakros geben Informationen nur aus, wenn der Code in der Debug-Buildkonfiguration kompiliert wird (Umgebungsvariable **WINCEDEBUG=debug**). Retailmakros generieren Informationen in Debug- und Retail-Buildkonfigurationen (**WINCEDEBUG=retail**), außer Sie erstellen das Run-Time Image in der Ship-Konfiguration (**WINCESHIP=1**). In der Ship-Konfiguration sind alle Debugmakros deaktiviert.

In Tabelle 4.1 sind die Debugmakros aufgeführt, die Sie in den Code einfügen können, um Debuginformationen zu generieren.

Tabelle 4.1 Windows Embedded CE-Markos für Debugmeldungen

Makro	Description
DEBUGMSG	Gibt bedingt eine Debugmeldung vom Typ *printf* im Standard-ausgabestream (im Fenster **Output** in Visual Studio oder in einer Datei) aus, wenn das Run-Time Image in der Debug-Buildkonfiguration kompiliert wurde.

Tabelle 4.1 Windows Embedded CE-Markos für Debugmeldungen (Fortsetzung)

Makro	Description
RETAILMSG	Gibt bedingt eine Debugmeldung vom Typ *printf* im Standardausgabestream (im Fenster **Output** in Visual Studio oder in einer Datei) aus, wenn das Run-Time Image in der Debug- oder Release-Buildkonfiguration, aber noch nicht in der Ship-Buildkonfiguration, kompiliert wurde.
ERRORMSG	Gibt bedingt weitere Debuginformationen vom Typ *printf* im Standardausgabestream (im Fenster **Output** in Visual Studio oder in einer Datei) aus, wenn das Run-Time Image in der Debug- oder Release-Buildkonfiguration, aber noch nicht in der Ship-Buildkonfiguration, kompiliert wurde. Diese Fehlerinformationen umfassen den Namen der Quellcodedatei und die Zeilennummer, mittels der die Codezeile, die die Meldung generiert hat, schnell gefunden werden kann.
ASSERTMSG	Gibt bedingt eine Debugmeldung vom Typ *printf* im Standardausgabestream (im Fenster **Output** in Visual Studio oder in einer Datei) aus und ruft den Debugger auf, wenn das Run-Time Image in der Debug-Buildkonfiguration kompiliert wurde. *ASSERTMSG* ruft *DEBUGMSG* gefolgt von *DBGCHK* auf.
DEBUGLED	Übergibt bedingt einen WORD-Wert an die Funktion *WriteDebugLED*, wenn das Run-Time Image in der Debug-Buildkonfiguration kompiliert wurde. Dieses Makro ist auf Geräten mit LEDs nützlich, um den Systemstatus anzuzeigen. Für das Makro muss die Funktion *OEMWriteDebugLED* in OAL implementiert sein.
RETAILLED	Übergibt bedingt einen WORD-Wert an die Funktion *WriteDebugLED*, wenn das Run-Time Image in der Debug- oder Release-Buildkonfiguration kompiliert wurde. Dieses Makro ist auf Geräten mit LEDs nützlich, um den Systemstatus anzuzeigen. Für das Makro muss die Funktion *OEMWriteDebugLED* in OAL implementiert sein.

Debugzonen

Debugmeldungen sind ausgesprochen hilfreich, um Multithread-Prozesse zu analysieren, insbesondere die Threadsynchronisierung und andere Zeitprobleme, die beim Durchlaufen des Codes nur schwierig zu erkennen sind. Die Anzahl der auf einem Zielgerät generierten Debugmeldungen kann jedoch überwältigend sein, wenn Sie zahlreiche Debugmakros im Code verwenden. Um die generierten Informationen zu steuern, ermöglichen die Debugmakros die Angabe eines bedingten Ausdrucks. Der folgende Code gibt beispielsweise eine Fehlermeldung aus, wenn der Wert *dwCurrentIteration* größer als der maximal mögliche Wert ist.

```
ERRORMSG(dwCurrentIteration > dwMaxIteration,
         (TEXT("Iteration error: the counter reached %u, when max allowed is %u\r\n"),
          dwCurrentIteration, dwMaxIteration));
```

In diesem Beispiel gibt *ERRORMSG* die Debuginformationen aus, wenn *dwCurrentIteration* größer als *dwMaxIteration* ist. Sie können die Debugmeldungen jedoch steuern, indem Sie Debugzonen in der Bedingungsanweisung verwenden. Dies ist insbesondere hilfreich, um mit dem *DEBUGMSG*-Makro die Codeausführung in einem Modul (eine ausführbare Datei oder DLL) auf verschiedenen Ebenen zu überprüfen, ohne den Quellcode zu ändern und neu zu kompilieren. Sie müssen die Debugzonen in der ausführbaren Datei oder DLL aktivieren und die globale Variable *DBGPARAM* mit dem Debug Message-Dienst registrieren, um die aktiven Zonen festzulegen. Anschließend können Sie die aktuelle Standardzone programmatisch oder über die Registrierung auf dem Entwicklungscomputer oder dem Zielgerät festlegen. Über **CE Debug Zones** im Menü **Target** oder im Fenster **Target Control** können Sie die Debugzonen für ein Modul in Platform Builder auch dynamisch steuern.

TIPP Umgehen der Debugzonen

Sie können die Debugzonen in Treibern und Anwendungen umgehen, indem Sie eine boolesche Variable mit dem Wert Wert *TRUE* oder *FALSE* an die Makros *DEBUGMSG* und *RETAILMSG* übergeben, wenn Sie das Run-Time Image neu erstellen.

Zonenregistrierung

Um Debugzonen zu verwenden, müssen Sie die globale Variable *DBGPARAM* mit drei Feldern definieren, die den Modulnamen, die Namen der zu registrierenden Debugzonen und die aktuelle Zonenmaske angeben (siehe Tabelle 4.2).

Tabelle 4.2 DBGPARAM-Elemente

Feld	Description	Beispiel
lpszName	Definiert den Namen des Moduls mit einer maximalen Länge von 32 Zeichen.	`TEXT("ModuleName")`
rglpszZones	Definiert ein Array mit 16 Namen für die Debugzonen. Jeder Name kann bis zu 32 Zeichen lang sein. Platform Builder zeigt die Informationen an, wenn der Benutzer im Modul aktive Zonen auswählt.	`{ TEXT("Init"), TEXT("Deinit"), TEXT("On"), TEXT("Undefined"), TEXT("Undefined"), TEXT("Undefined"), TEXT("Undefined"), TEXT("Undefined"), TEXT("Undefined"), TEXT("Undefined"), TEXT("Undefined"), TEXT("Undefined"), TEXT("Undefined"), TEXT("Failure"), TEXT("Warning"), TEXT("Error") }`
ulZoneMask	Die aktuelle Zonenmaske im *DEBUGZONE*-Makro, um die ausgewählte Debugzone zu bestimmen.	`MASK_INIT | MASK_ON | MASK_ERROR`

HINWEIS Debugzonen

Windows Embedded CE unterstützt 16 benannte Debugzonen, die jedoch nicht alle definiert werden müssen. Jedes Modul verwendet andere Zonennamen, die den Verwendungszweck der implementierten Zonen reflektieren sollten.

Die Headerdatei *Dbgapi.h* definiert die *DBGPARAM*-Struktur und die Debugmakros. Da diese Makros eine vordefinierte *DBGPARAM*-Variable namens *dpCurSettings* verwenden, müssen Sie den gleichen Namen im Quellcode angeben:

```
#include <DBGAPI.H>

// A macro to increase the readability of zone mask definitions
```

```
#define DEBUGMASK(n)     (0x00000001<<n)

// Definition of zone masks supported in this module
#define MASK_INIT        DEBUGMASK(0)
#define MASK_DEINIT      DEBUGMASK(1)
#define MASK_ON          DEBUGMASK(2)
#define MASK_FAILURE     DEBUGMASK(13)
#define MASK_WARNING     DEBUGMASK(14)
#define MASK_ERROR       DEBUGMASK(15)

// Definition dpCurSettings variable with the initial debug zone state
// set to Failure, Warning, and Error.
DBGPARAM dpCurSettings =
{
  TEXT("ModuleName"), // Specify the actual module name for clarity!
  {
    TEXT("Init"), TEXT("Deinit"), TEXT("On"), TEXT("Undefined"),
    TEXT("Undefined"), TEXT("Undefined"), TEXT("Undefined"),
    TEXT("Undefined"), TEXT("Undefined"), TEXT("Undefined"),
    TEXT("Undefined"), TEXT("Undefined"), TEXT("Undefined"),
    TEXT("Failure"), TEXT("Warning"), TEXT("Error")
  },
  MASK_INIT | MASK_ON | MASK_ERROR
};

// Main entry point into DLL
BOOL APIENTRY DllMain( HANDLE hModule,
                       DWORD  ul_reason_for_call,
                       LPVOID lpReserved
                     )
{
    if(ul_reason_for_call == DLL_PROCESS_ATTACH)
    {
// Register with the Debug Message service each time
        // the DLL is loaded into the address space of a process.
        DEBUGREGISTER((HMODULE)hModule);
    }
    return TRUE;
}
```

Zonendefinitionen

Dieser Beispielcode registriert sechs Debugzonen für das Modul, die Sie zusammen mit Bedingungsanweisungen in Debugmakros verwenden können. Die folgende Codezeile veranschaulicht eine Methode:

```
DEBUGMSG(dpCurSettings.ulZoneMask & (0x00000001<<(15)),
        (TEXT("Error Information\r\n")));
```

Wenn die Debugzone auf *MASK_ERROR* festgelegt ist, wird der bedingte Ausdruck auf *TRUE* gesetzt und *DEBUGMSG* sendet die Informationen an den Debugausgabe-

stream. Um die Lesbarkeit des Codes zu verbessern, können Sie das in *Dbgapi.h* definierte *DEBUGZONE*-Makro verwenden, um Flags für die Zonen zu definieren: Die Methode vereinfacht unter anderem das Kombinieren der Debugzonen über logische UND- und ODER-Operatoren.

```
#include <DBGAPI.H>

// Definition of zone flags: TRUE or FALSE according to selected debug zone.
#define ZONE_INIT        DEBUGZONE(0)
#define ZONE_DEINIT      DEBUGZONE(1)
#define ZONE_ON          DEBUGZONE(2)
#define ZONE_FAILURE     DEBUGZONE(13)
#define ZONE_WARNING     DEBUGZONE(14)
#define ZONE_ERROR       DEBUGZONE(15)

DEBUGMSG(ZONE_FAILURE, (TEXT("Failure debug zone enabled.\r\n")));
DEBUGMSG(ZONE_FAILURE && ZONE_ WARNING,
        (TEXT("Failure and Warning debug zones enabled.\r\n")));
DEBUGMSG(ZONE_FAILURE || ZONE_ ERROR,
        (TEXT("Failure or Error debug zone enabled.\r\n")));
```

Aktiveren und Deaktivieren von Debugzonen

Das *DBGPARAM*-Feld *ulZoneMask* legt die aktuelle Debugzone für ein Modul fest. Sie können den *ulZoneMask*-Wert der globalen Variablen *dpCurSettings* direkt ändern. Außerdem können Sie den *ulZoneMask*-Wert im Debugger an einem Breakpoint im Fenster **Watch** ändern. Die Debugzone kann auch über den Aufruf der Funktion *SetDbgZone* festgelegt werden. Das Dialogfeld **Debug Zones** kann zur Laufzeit in Visual Studio mit Platform Builder über die Option **CE Debug Zones** im Menü **Target** angezeigt werden (siehe Abbildung 4.2).

In der Liste **Name** wird das auf dem Zielgerät ausgeführte Modul angezeigt, das Debugzonen unterstützt. Wenn das ausgewählte Modul mit dem Debug Message-Dienst registriert ist, wird die Liste mit den 16 Zonen unter **Debug Zones** angezeigt. Die Namen entsprechen der *dpCurSettings*-Definition des ausgewählten Moduls. Sie können die Zonen aktivieren oder deaktivieren. Standardmäßig sind die in der Variablen *dpCurSettings* definierten Zonen aktiviert und in der Liste **Debug Zones** markiert. Für Module, die nicht mit dem Debug Message-Dienst registriert sind, ist die Liste **Debug Zone** nicht aktiviert.

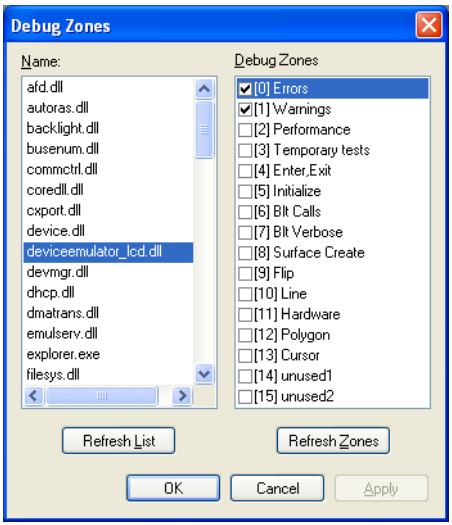


Abbildung 4.2 Festlegen der Debugzonen in Platform Builder

Überschreiben von Debugzonen beim Start

Windows Embedded CE aktiviert die in der Variablen *dpCurSettings* angegebenen Zonen, wenn Sie die Anwendung starten oder die DLL in einen Prozess laden. Sie können die Debugzone zu diesem Zeitpunkt nur ändern, wenn Sie einen Breakpoint festlegen und den *ulZoneMask*-Wert im Fenster **Watch** ändern. CE unterstützt jedoch eine praktischere Methode über Registrierungseinstellungen. Um ein Modul mit mehreren aktiven Debugzonen zu laden, erstellen Sie einen REG_DWORD-Wert mit einem Namen, der dem im *lpszName*-Feld der Variablen *dpCurSettings* angegebenen Modulnamen entspricht. Legen Sie diesen Wert auf die zusammengefassten Werte der zu aktivierenden Debugzonen fest. Dieser Wert kann auf dem Entwicklungscomputer oder auf dem Zielgerät konfiguriert werden. Das Konfigurieren des Werts auf dem Entwicklungscomputer ist jedoch im Allgemeinen vorzuziehen, da das Ändern der Registrierungseinstellungen auf dem Zielgerät erfordert, dass Sie das Run-Time Image neu erstellen. Wenn Sie hingegen die Registrierungseinstellungen auf dem Entwicklungscomputer ändern, müssen Sie lediglich die betroffenen Module neu starten.

In Tabelle 4.3 sind die Registrierungseinstellungen für das Beispielmodul *ModuleName* aufgeführt. Ersetzen Sie den Platzhalter durch den Namen der ausführbaren Datei oder der DLL.

Tabelle 4.3 Registrierungsparameter für den Systemstart

Pfad	Entwicklungscomputer	Zielgerät
Registrierungsschlüssel	HKEY_CURRENT_USER\Pegasus\Zones	HKEY_LOCAL_MACHINE\DebugZones
Eintrag	ModuleName	ModuleName
Typ	REG_DWORD	REG_DWORD
Wert	0x00000001 - 0x7FFFFFFF	0x00000001 - 0x7FFFFFFF
Kommentar	Das Debug Message-System verwendet den Wert für ein Modul auf dem Zielgerät nur, wenn der Entwicklungscomputer nicht verfügbar ist oder die Registrierung auf dem Entwicklungscomputer keinen Wert für das Modul enthält.	

HINWEIS Aktivieren aller Debugzonen

Windows Embedded CE verwendet die niedrigen 16 Bits des REG_DWORD-Werts, um für das Debuggen von Anwendungen benannte Debugzonen zu definieren. Die verbleibenden Bits sind für nicht benannte Debugzonen verfügbar (außer das höchste Bit, das für den Kernel reserviert ist). Sie sollten deshalb den Debugzonenwert eines Moduls nicht auf *0xFFFFFFFF* festlegen. Der maximale Wert *0x7FFFFFFF* aktiviert alle benannten und nicht benannten Debugzonen.

WEITERE INFORMATIONEN Pegasus-Registrierungsschlüssel

Der Name *Pegasus* bezieht sich auf den Codenamen der ersten Windows CE-Version für tragbare PCs und Unterhaltungselektronik, die 1996 von Microsoft auf den Markt gebracht wurde.

Bewährte Vorgehensweise

Beachten Sie bei der Arbeit mit Debugmeldungen, dass diese die Codeausführung verlangsamen. Da das System außerdem die Debugausgabevorgänge serialisiert, wird möglicherweise eine ungewollte Methode für die Threadsynchronisierung verwendet. Beispielsweise können mehrere nicht synchronisierte Threads Probleme in Release-Builds verursachen, die in den Debug-Builds nicht erkannt werden.

Berücksichtigen Sie bei der Arbeit mit Debugmeldungen und -zonen folgende Verfahren:

- **Bedingungsanweisungen** Verwenden Sie Debugmakros mit Bedingungsanweisungen basierend auf Debugzonen. Verwenden Sie *DEBUGMSG(TRUE)* nicht. Vermeiden Sie außerdem Retailmakros ohne Bedingungsanweisungen, beispielsweise *RETAILMSG(TRUE)*. Diese Methode ist jedoch für einige MDD (Model Device Driver)/PDD (Platform Dependent Driver) erforderlich.
- **Ausschließen des Debugcodes aus Release-Builds** Wenn Sie Debugzonen ausschließlich in Debug-Builds verwenden, beziehen Sie die globale Variable *dpCurSettings* und Zonenmaskendefinitionen in *#ifdef DEBUG #endif* Guards ein und beschränken Sie die Verwendung von Debugzonen auf Debugmakros (beispielsweise *DEBUGMSG*).
- **Retail-Makros in Release-Builds** Um Debugzonen in Release-Builds zu verwenden, beziehen Sie die globale Variable *dpCurSettings* und Zonenmaskendefinitionen in *#ifndef SHIP_BUILD #endif* -Bedingungen ein und ersetzen Sie den Aufruf von *DEBUGREGISTER* durch den Aufruf von *RETAILREGISTERZONES*.
- **Geben Sie den Modulnamen an** Wenn möglich, legen Sie den *dpCurSettings.lpszName*-Wert auf den Dateinamen des Moduls fest.
- **Einschränken der Verbosity** Legen Sie die Standardzonen für die Treiber nur auf *ZONE_ERROR* und *ZONE_WARNING* fest. Wenn Sie eine neue Plattform verwenden, aktivieren Sie *ZONE_INIT*.
- **Beschränken der Fehler-Debugzone auf nicht behebbare Probleme** Verwenden Sie *ZONE_ERROR* nur, wenn das Modul oder eine wichtige Funktion aufgrund einer falschen Konfigurationseinstellung oder eines anderen Problems nicht funktioniert. Verwenden Sie *ZONE_WARNING* für behebbare Probleme.
- **Eliminieren aller Fehlermeldungen und Warnungen** Sie sollten das Modul ohne *ZONE_ERROR*- oder *ZONE_WARNING*-Meldungen laden können.

Target Control-Befehle

Der Target Control-Dienst ermöglicht den Zugriff auf eine Befehlsshell für den Debugger, um Dateien auf das Zielgerät zu übertragen und Anwendungen zu debuggen. Sie können diese Shell in Visual Studio mit Platform Builder über die Option **Target Control** im Menü **Target** öffnen (siehe Abbildung 4.3). Beachten Sie

jedoch, dass die Target Control-Shell nur angezeigt wird, wenn die Platform Builder-Instanz über KITL mit einem Gerät verbunden ist.

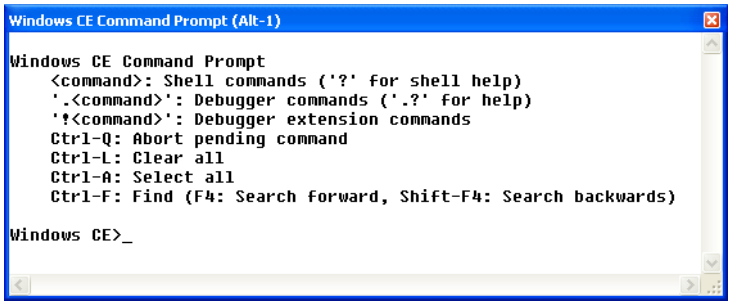


Abbildung 4.3 Die Target Control-Shell

Über die Target Control-Shell können Sie beispielsweise folgende Debugaktionen ausführen:

- Den Kerneldebugger aufrufen (Befehl **break**).
- Ein Speicherabbild an die Debugausgabe (Befehl **dd**) oder an eine Datei (Befehl **df**) senden.
- Die Speicherbelegung für den Kernel (Befehl **mi kernel**) oder das gesamte System (Befehl **mi full**) analysieren.
- Die Prozesse (Befehl **gi proc**), Threads (Befehl **gi thrd**) und Threadprioritäten (Befehl **tp**) sowie die im System geladenen Module (Befehl **gi mod**) auflisten.
- Prozesse starten (Befehl **s**) und beenden (Befehl **kp**).
- Den Prozessheap (Befehl **hp**) speichern.
- Den System Profiler (Befehl **prof**) aktivieren oder deaktivieren.

HINWEIS Target Control-Befehle

Eine vollständige Liste der Target Control-Befehle finden Sie im Abschnitt Target Control Debugging Commands in der Windows Embedded CE 6.0-Dokumentation auf der Microsoft MSDN®-Website unter *http://msdn2.microsoft.com/en-us/library/aa936032.aspx*.

Debugger-Erweiterungsbefehle (CEDebugX)

Zusätzlich zu den normalen Debugger-Befehlen stellt der Target Control-Dienst Erweiterungsbefehle (CEDebugX) bereit, um das Debuggen des Kernels und der Anwendungen zu optimieren. Die Erweiterung umfasst zusätzliche Features zum

Erkennen von Speicherverlust und Deadlocks sowie zum Analysieren der Systemintegrität. Sie können über die Target Control-Shell auf diese Befehle zugreifen, indem Sie ein Ausrufezeichen (!) eingeben.

Um die Debugger-Erweiterungsbefehle zu verwenden, führen Sie in der Target Control-Shell den Befehl **break** oder in Visual Studio über das Menü **Target** den Befehl **Break All** für den Kerneldebugger aus. Mit dem Befehl !**diagnose all** können Sie die Ursache eines Fehlers ermitteln, beispielsweise eine Heapbeschädigung, Deadlocks oder Speicherprobleme. Auf einem ordnungsgemäß funktionierenden System sollte CEDebugX keine Fehler finden (siehe Abbildung 4.4).

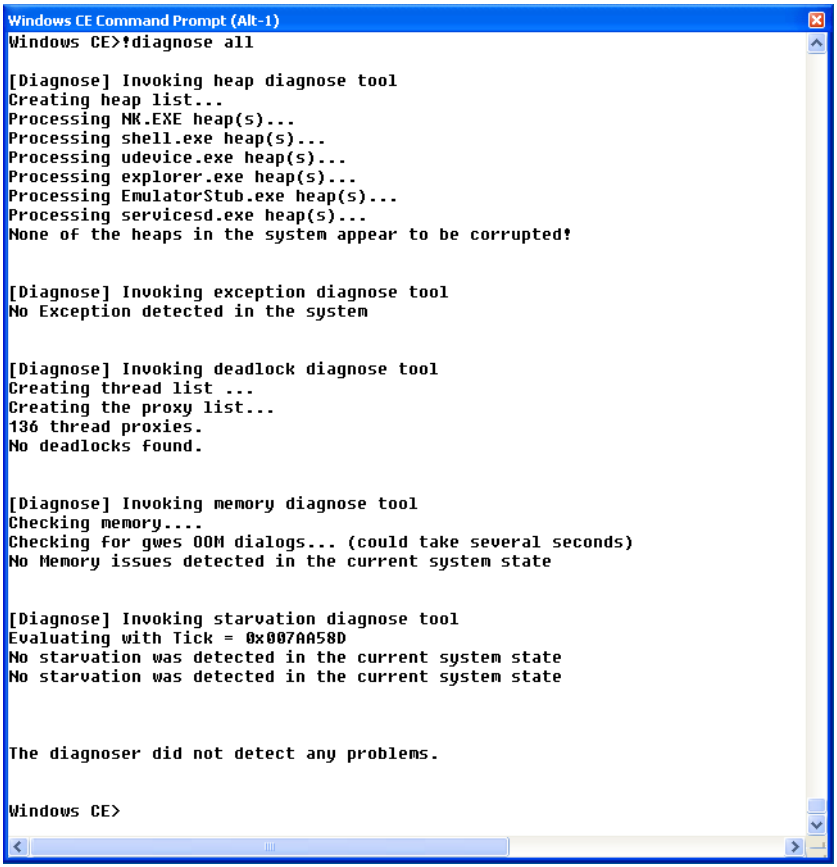


Abbildung 4.4 Analysieren eines Run-Time Images mit CEDebugX

Der Befehl **!diagnose all** führt folgende Diagnosevorgänge aus:

- **Heap** Analysiert alle Heapobjekte der auf dem System ausgeführten Prozesse, um potenzielle Inhaltsbeschädigungen zu identifizieren.
- **Exception** Analysiert eine auf dem System aufgetretene Ausnahme und zeigt detaillierte Informationen an, beispielsweise die Prozess-ID, die Thread-ID und die PC-Adresse zum Zeitpunkt der Ausnahme.
- **Memory** Analysiert den Systemspeicher, um potenzielle Speicherbeschädigungen und zu geringen Arbeitsspeicher zu erkennen.
- **Deadlock** Analysiert den Status der Threads und Systemobjekte (siehe Kapitel 3). Zeigt außerdem eine Liste der Systemobjekte und Thread-IDs an, die den Deadlock generiert haben.
- **Starvation** Analysiert Threads und Systemobjekte, um eine potenzielle Thread-Starvation zu identifizieren. Eine Starvation tritt auf, wenn ein Thread auf dem System nicht vom Scheduler aktiviert wird, da dieser mit Threads einer höheren Priorität ausgelastet ist.

Erweiterte Debuggertools

Die Target Control-Shell und CEDebugX-Befehle ermöglichen die eingehende Analyse eines Systems oder einer CE-Abbilddatei (wenn Sie den CE Dump File Reader als Debugger für das Postmortem-Debuggen auswählen). Sie sind jedoch nicht auf die Befehlszeile beschränkt. Platform Builder umfasst mehrere grafische Tools, um die Effizienz des Debuggens zu erhöhen. Sie können in Visual Studio über die Option **Windows** im Menü **Debug** auf die erweiterten Debuggertools zugreifen.

Die Platform Builder IDE umfasst die folgenden erweiterten Debuggertools:

- **Breakpoints** Listet die auf dem System aktivierten Breakpoints auf und ermöglicht den Zugriff auf die Breakpointeigenschaften.
- **Watch** Ermöglicht den Lese- und Schreibzugriff auf lokale und globale Variablen.
- **Autos** Ermöglicht (ähnlich wie das Watch-Fenster) den Zugriff auf Variablen. Der Debugger erstellt die Variablenliste jedoch dynamisch, wohingegen das Watch-Fenster alle manuell zugefügten Variablen auflistet, unabhängig davon, ob auf diese zugegriffen werden kann. Das Autos-Fenster ist nützlich, um die an eine Funktionen übergebenen Parameterwerte zu überprüfen.

- **Call Stack** Dieses Tool ist nur verfügbar, wenn sich das System im Break-Status befindet (die Codeausführung wurde an einem Breakpoint angehalten). In diesem Fenster sind alle auf dem System aktivierten Prozesse und gehosteten Threads aufgelistet.
- **Threads** Listet die in den Systemprozessen ausgeführten Threads auf. Diese Informationen werden dynamisch abgerufen und können jederzeit aktualisiert werden.
- **Modules** Listet die geladenen und entladenen Module auf und zeigt die entsprechende Speicheradresse an. Dieses Feature ist nützlich, um zu ermitteln, ob eine Treiber-DLL geladen ist.
- **Processes** Dieses Fenster listet, ähnlich wie das Fenster Threads, die auf dem System ausgeführten Prozesse auf. Sie können hier Prozesse beispielsweise abbrechen.
- **Memory** Ermöglicht den direkten Zugriff auf den Gerätespeicher. Sie können den gewünschten Speicherinhalt mittels Speicheradressen oder Variablennamen abrufen.
- **Disassembly** Zeigt den Assemblycode der aktuellen Codezeile an.
- **Registers** Ermöglicht den Zugriff auf die CPU-Registrierungswerte, wenn eine bestimmte Codezeile ausgeführt wird.
- **Advanced Memory** Durchsucht den Gerätespeicher, verschiebt Speicherabschnitte in andere Bereiche und füllt Speicherbereiche mit Inhaltsmustern auf.
- **List Nearest Symbols** Legt eine Speicheradresse für die nächsten in den Binärdateien verfügbaren Symbole fest. Außerdem wird der vollständige Pfad zur Datei angezeigt, die die Symbole enthält. Dieses Tool ist nützlich, um den Namen einer Funktionen abzurufen, die eine Ausnahme generiert hat.

ACHTUNG Speicherbeschädigung

Die Tools Memory und Advanced Memory können den Speicherinhalt ändern. Die unsachgemäße Verwendung dieser Tools kann auf dem Zielgerät zu Systemfehlern und zur Beschädigung des Betriebsystems führen.

Das Tool Application Verifier

Ein weiteres nützliches Tool zum Erkennen potenzieller Kompatiblitäts- und Stabilitätsprobleme von Anwendungen sowie für die Problembehebung auf Quellcodeebene ist das Tool Application Verifier im CETK. Das Tool kann auf eine Anwendung oder DLL zugreifen, um Probleme zu analysieren, die auf eigenständigen Geräten schwierig nachzuverfolgen sind. Das Application Verifier-Tool erfordert keine Geräteverbindung mit einem Entwicklungscomputer und kann beim Systemstart ausgeführt werden, um die Treiber und Systemanwendungen zu überprüfen. Sie können das Tool auch über die CETK-Benutzeroberfläche oder manuell auf dem Zielgerät starten. Um das Application Verifier-Tool außerhalb des CETK zu verwenden, kopieren Sie mit der Datei *Getappverif_cetk.bat* alle erforderlichen Dateien in das *Release*-Verzeichnis.

HINWEIS Application Verifier-Dokumentation

Weitere Informationen zum Application Verifier-Tool, einschließlich den Shimerweiterungs-DLLs zum Ausführen von benutzerdefiniertem Code oder Ändern des Funktionsverhaltens während Anwendungstests, finden Sie im Abschnitt Application Verifier Tool in der Windows Embedded CE 6.0-Dokumentation auf der Microsoft MSDN-Website unter *http://msdn2.microsoft.com/en-us/library/aa934321.aspx*.

CELog Event Tracking und Verarbeitung

Windows Embedded CE umfasst ein erweiterbares Eventüberwachungssystem, das Sie in ein Run-Time Image einbeziehen können, um Leistungsprobleme zu analysieren. Das CELog Event Tracking-System protokolliert vordefinierte Kernel- und Coredll-Events, die sich auf Mutexe, Events, die Speicherzuordnung und andere Kernelobjekte beziehen. Die erweiterbare Architektur des CELog Event Tracking-Systems ermöglicht außerdem das Implementieren angepasster Filter, mit denen benutzerdefinierte Events überwacht werden können. Für Plattformen die über KITL mit einem Entwicklungscomputer verbunden sind, kann das CELog Event Tracking-System die Events selektiv basierend auf den im *ZoneCE*-Registrierungseintrag angegebenen Zonen protokollieren (siehe Tabelle 4.4).

Tabelle 4.4 CELog-Registrierungsparameter für Event-Protokollierungszonen

Pfad	HKEY_LOCAL_MACHINE\System\CELog
Registrierungseintrag	ZoneCE
Typ	REG_DWORD
Wert	*<Zone IDs>*
Description	Standardmäßig werden alle Zonen protokolliert. Eine vollständige Liste der Zonen-IDs finden Sie im Abschnitt CELog Zones in der Windows Embedded CE 6.0-Dokumentation auf der Microsoft MSDN®-Website unter *http://msdn2.microsoft.com/en-us/library/aa909194.aspx*.

Mit dem CELog Event Tracking-System können Sie Daten zusammenstellen, die CELog in einem Puffer im RAM auf dem Zielgerät speichert. Sie können diese Daten anschließend mit Leistungstools, beispielsweise Remote Kernel Tracker und Readlog, verarbeiten. Außerdem können Sie mit dem CELogFlush-Tool die Daten regelmäßig in eine Datei übertragen.

HINWEIS CELog und Ship-Builds

Beziehen Sie das CELog Event Tracking-System nicht in den endgültigen Build ein, um Leistungs- und Speichereinbußen aufgrund von CELog-Aktivitäten zu verhindern und die Angriffsfläche für Systemattacken zu verringern.

Remote Kernel Tracker

Das Tool Remote Kernel Tracker ermöglicht das Überwachen der Systemaktivitäten auf dem Zielgerät basierend auf Prozessen und Threads. Das Tool kann über KITL die Informationen des Zielgeräts in Echtzeit anzeigen. Sie können den Remote Kernel Tracker basierend auf den CELog-Datendateien auch offline verwenden. Weitere Informationen zum Tool Remote Kernel Tracker finden Sie in Kapitel 3 „Systemprogrammierung."

In Abbildung 4.5 ist das Zusammenstellen der Informationen über Threadaktivitäten mit dem Tool Remote Kernel Tracker dargestellt.

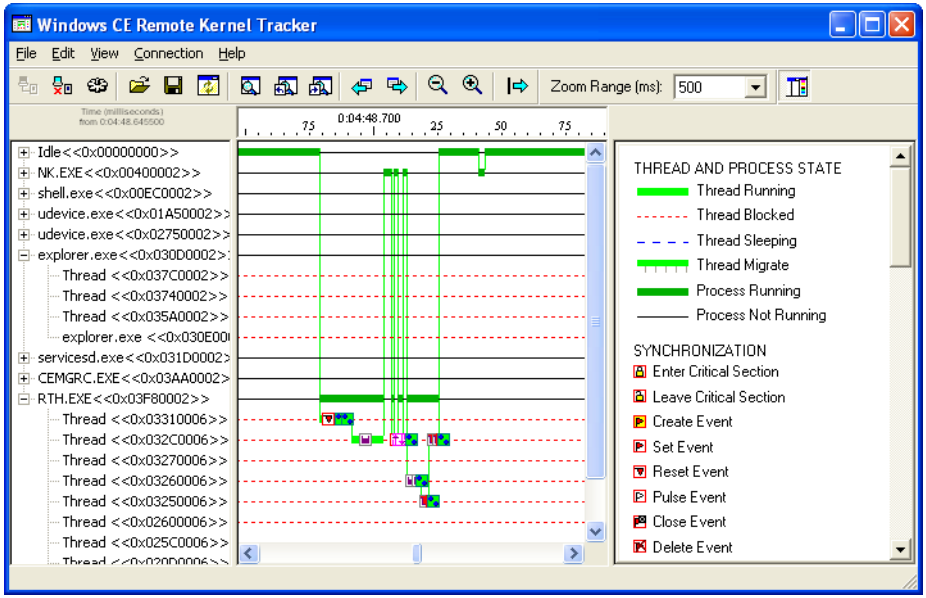


Abbildung 4.5 Threadinformationen im Kernel Tracker

Das Tool CELogFlush

Mit dem Tool CELogFlush können Sie CELog-Datendateien erstellen, um die CELog-Eventdaten, die im RAM gepuffert sind, in einer .clg-Datei zu speichern. Diese Datei kann im RAM-Dateisystem, im permanenten Speicher oder im *Release*-Dateisystem auf dem Entwicklungscomputer gespeichert werden. Um den Datenverlust aufgrund von Pufferüberläufen möglichst gering zu halten, legen Sie einen größeren RAM-Puffer fest und erhöhen Sie das Intervall zum Löschen des Puffers. Sie können die Leistung optimieren, indem Sie die Datei geöffnet lassen, um wiederholte Öffnungs- und Schließvorgänge zu vermeiden, und die Datei im RAM-Dateisystem, anstatt auf einem langsameren permanenten Speichermedium, zu speichern.

HINWEIS CELogFlush-Konfiguration

Weitere Informationen zum Tool CELogFlush, einschließlich des Konfigurierens des Tools über Registrierungseinstellungen, finden Sie im Abschnitt CELogFlush Registry Settings in der Windows Embedded CE 6.0-Dokumentation auf der Microsoft MSDN-Website unter *http://msdn2.microsoft.com/en-us/library/aa935267.aspx*.

Das Tool Readlog

Zusätzlich zum grafischen Remote Kernel Tracker-Tool können Sie CELog-Datendateien mit dem Readlog-Tool verarbeiten, das im Ordner *%_WINCEROOT%\Public\Common\Oak\Bin\i386* gespeichert ist. Readlog ist ein Befehlszeilentool zum Verarbeiten und Anzeigen von Informationen, die im Remote Kernel Tracker nicht angezeigt werden, beispielsweise Debugmeldungen und Startevents. Es ist oft hilfreich die Systemaktivitäten zuerst im Remote Kernel Tracker und anschließend einen bestimmten Prozess oder Thread mit dem Readlog-Tool zu analysieren. Die vom Tool CELogFlush in die .clg-Datei geschriebenen Rohdaten sind nach Zonen sortiert, damit bestimmte Informationen einfacher gefunden und extrahiert werden können. Sie können die Daten filtern und die Filterfunktionen basierend auf Erweiterungs-DLLs erweitern, um die vom benutzerdefinierte Events Collector erfassten Dateien zu verarbeiten.

Eine der hilfreichsten Funktionen von Readlog ist das Ersetzen der Thread-Startadressen (die Funktion wird im *CreateThread*-Aufruf übergeben) in den CELog-Datendateien durch die Namen der tatsächlichen Threadfunktionen, um die Systemanalyse im Remote Kernel Tracker zu unterstützen. Sie müssen hierzu Readlog mit dem Parameter **-fixthreads** (**readlog -fixthreads**) starten. Readlog ermittelt die .map-Symboldateien im *Release*-Verzeichnis, um die Threadfunktionen basierend auf den Startadressen zu identifizieren, und generiert neue Protokolle mit den entsprechenden Referenzen.

In Abbildung 4.6 sind die CELog-Daten im Remote Kernel Tracker dargestellt, die über das CELog Event Tracking-System erfasst, mit dem Tool CELogFlush in eine .clg-Datei übertragen und mit Readlog mit dem Parameter **-fixthreads** übersichtlicher angezeigt werden.

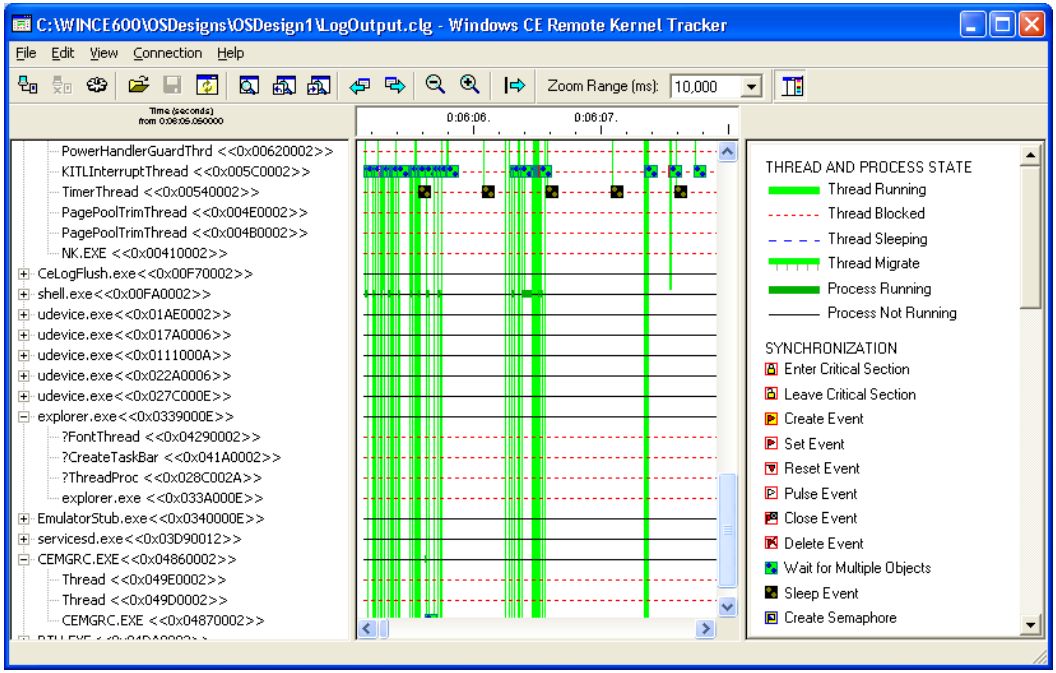


Abbildung 4.6 Eine mit **readlog -fixthreads** vorbereitete CELog-Datendatei im Remote Kernel Tracker

HINWEIS Verbessern der Zuordnung von Referenznamen

Das CELog Event Tracking-System kann den Kernelprofiler verwenden, um Threadfunktionsnamen für *CreateThread*-Events basierend auf Startadressen zu suchen, wenn Sie den Kernelprofiler explizit aktivieren und Profilsymbole zum Run-Time Image hinzufügen, indem Sie das Image mit der Umgebungsvariablen *IMGPROFILER* neu erstellen. CELog kann jedoch ausschließlich die Profilsymbole ermitteln, die im Run-Time Image integriert sind. Symbole von Anwendungen, die mit einem Software Development Kit (SDK) entwickelt wurden, sind normalerweise für das CELog Event Tracking-System nicht verfügbar.

Zusammenfassung

Für das Debuggen von Betriebssystemen und Anwendungen müssen Sie mit dem CE-System und den Debugtools in Platform Builder und CETK vertraut sein. Die wichtigsten Debugtools sind der System Debugger, das Debug Message-Feature und die CE Target Control-Shell. Der System Debugger ermöglicht das Festlegen von Breakpoints und Durchlaufen des Kernel- und Anwendungscodes. Das Debug Message-Feature analysiert die Systemkomponenten und Anwendungen, ohne die

Codeausführung zu unterbrechen. Zum Ausgeben der Debuginformationen von Zielgeräten mit oder ohne Anzeigekomponente sind zahlreiche Debug- und Retail-Makros verfügbar. Da Systeme und Anwendungen eine große Anzahl an Debugmeldungen generieren können, sollten Sie die Ausgabe der Debuginformationen unter Verwendung von Debugzonen steuern. Der Vorteil von Debugzonen ist, dass Sie die Ausführlichkeit der Debuginformationen dynamisch ändern können, ohne das Run-Time Image neu erstellen zu müssen. Die Target Control-Shell ermöglicht das Senden von Befehlen an das Zielgerät, beispielsweise **break** mit dem Parameter **!diagnose all**, um den Debugger aufzurufen und eine CEDebugX-Überprüfung der Systemintegrität auszuführen, einschließlich Speicherverlust, Ausnahmen und Deadlocks.

Zusätzlich zu diesen Debugtools können Sie die CE-Tools für die Konfiguration und Problembehandlung verwenden, beispielsweise Application Verifier, um potenzielle Probleme mit der Anwendungskompatibilität und -stabilität zu identifizieren, und den Remote Kernel Tracker, um Prozesse, Threads und die Systemleistung zu analysieren. Remote Kernel Tracker hängt vom CELog Event Tracking-System ab, das die protokollierten Daten im Speicher auf dem Zielgerät beibehält. Sie können diese Daten mit dem Tool CELogFlush in eine Datei übertragen. Wenn Symboldateien für die zu analysierenden Module verfügbar sind, können Sie mit dem Tool Readlog die Threadstartadressen durch die Funktionsnamen ersetzen und neue CELog-Datendateien für die Offline-Analyse im Remote Kernel Tracker generieren.

Lektion 2: Konfigurieren des Run-Time Images, um das Debuggen zu aktivieren

Die Debugfeatures von Windows Embedded CE hängen von den Komponenten des Entwicklungscomputers und des Zielgeräts ab, die bestimmte Einstellungen und Hardwareunterstützung erfordern. Zum Austauschen der Debuginformationen und anderer Anforderungen zwischen dem Entwicklungscomputer und dem Zielgerät müssen diese verbunden sein. Wenn die Kommunikation unterbrochen wird (da beispielsweise der Debugger auf dem Entwicklungscomputer angehalten wurde, ohne zuerst den Debugstub auf dem Zielgerät zu entladen), reagiert das Run-Time Image möglicherweise nicht mehr auf Benutzereingaben, sondern wartet, dass der Debugger die Codeausführung fortsetzt, nachdem eine Ausnahme aufgetreten ist.

Nach Abschluss dieser Lektion können Sie:

- Den Kerneldebugger für das Run-Time Image aktivieren.
- Die KITL-Anforderungen identifizieren.
- Den Kerneldebugger im Debugkontext verwenden.

Veranschlagte Zeit für die Lektion: 20 Minuten.

Aktivieren des Kerneldebuggers

Wie bereits in Lektion 1 erwähnt, umfasst die Windows Embedded CE 6.0-Entwicklungsumgebung einen Kerneldebugger, der dem Entwickler das Durchlaufen und Interagieren mit dem auf einem CE-Zielgerät ausgeführten Code ermöglicht. Der Debugger erfordert, dass Sie die Kerneloptionen und einen Kommunikationslayer zwischen dem Zielgerät und dem Entwicklungscomputer festlegen.

OS Design-Einstellungen

Um ein OS-Design zum Debuggen zu aktivieren, müssen Sie die Umgebungsvariablen *IMGNODEBUGGER* and *IMGNOKITL* entfernen, damit Platform Builder die KdStub-Bibliothek einbezieht und den KITL-Kommunikationslayer im Board Support Package (BSP) aktiviert, wenn das Run-Time Image erstellt wird. Platform Builder stellt für diese Aufgabe eine praktische Methode bereit. Klicken Sie mit der rechten Maustaste in Visual Studio auf das **OS Design**-Projekt und wählen Sie **Properties** aus, um das Dialogfeld **OS Design Property Pages** zu öffnen, und aktivieren Sie unter Build **Options** die Kontrollkästchen **Enable Kernel Debugger** und **Enable KITL**. In

Kapitel 1 „Anpassen des OS-Designs“ wurde das Dialogfeld **OS Design Property Pages** ausführlich beschrieben.

Auswählen eines Debuggers

Nachdem Sie KdStub und KITL für ein Run-Time Image aktiviert haben, können Sie einen Debugger auswählen, um das System auf dem Zielgerät zu analysieren. Um die Parameter für die Analyse zu konfigurieren, öffnen Sie in Visual Studio das Dialogfeld **Target Device Connectivity Options**, indem Sie im Menü **Target** die Option **Connectivity Options** auswählen (siehe Kapitel 2 „Erstellen und Bereitstellen eines Run-Time Images“).

In den Verbindungsoptionen ist standardmäßig kein Debugger ausgewählt. Die folgenden Optionen sind verfügbar:

- **KdStub** Der Softwaredebugger für den Kernel und Anwendungen zum Debuggen der Systemkomponenten, Treiber und Anwendungen, die auf dem Zielgerät ausgeführt werden. KdStub erfordert KITL für die Kommunikation mit Platform Builder.
- **CE Dump File Reader** Platform Builder umfasst eine Option zum Erstellen von Abbilddateien, die Sie mit dem CE Dump File Reader öffnen können. Mit den Abbilddateien können Sie den Status eines Systems zu einem bestimmten Zeitpunkt überprüfen.
- **Sample Device Emulator eXDI 2 Driver** KdStub kann weder Routinen, die das System vor dem Laden des Kernels ausführt, noch ISRs (Interrupt Service Routines) debuggen, da diese Debugbibliothek von Softwarebreakpoints abhängt. Für das hardwareunterstützte Debuggen stellt Platform Builder einen eXDI-Beispieltreiber bereit, den Sie zusammen mit einem JTAG-Test (Joint Test Action Group) verwenden können. Der JTAG-Test ermöglicht das Festlegen von Hardwarebreakpoints, die vom Prozessor verarbeitet werden.

HINWEIS Hardwareunterstütztes Debuggen

Weitere Informationen zum hardwareunterstützten Debuggen finden Sie im Abschnitt Hardware-assisted Debugging in der Windows Embedded CE 6.0-Dokumentation auf der Microsoft MSDN-Website unter *http://msdn2.microsoft.com/en-us/library/aa935824.aspx*.

KITL

KITL ist ein erforderlicher Kommunikationslayer zwischen dem Entwicklungscomputer und dem Zielgerät und muss für den Kerneldebugger aktiviert werden (siehe Abbildung 4.1). KITL ist hardwareunabhängig und funktioniert über Netzwerkverbindungen, serielle Kabel, USB (Universal Serial Bus) und andere unterstützte Kommunikationsmethoden, beispielsweise DMA (Direct Memory Access). Die einzige Voraussetzung ist, dass der Entwicklungscomputer und das Zielgerät die gleiche Schnittstelle unterstützen und verwenden. Die bekannteste und schnellste KITL-Schnittstelle für den Geräteemulator ist DMA (siehe Abbildung 4.7). Für Zielgeräte mit einem unterstützten Ethernet-Chip sollten Sie eine Netzwerkschnittstelle verwenden.

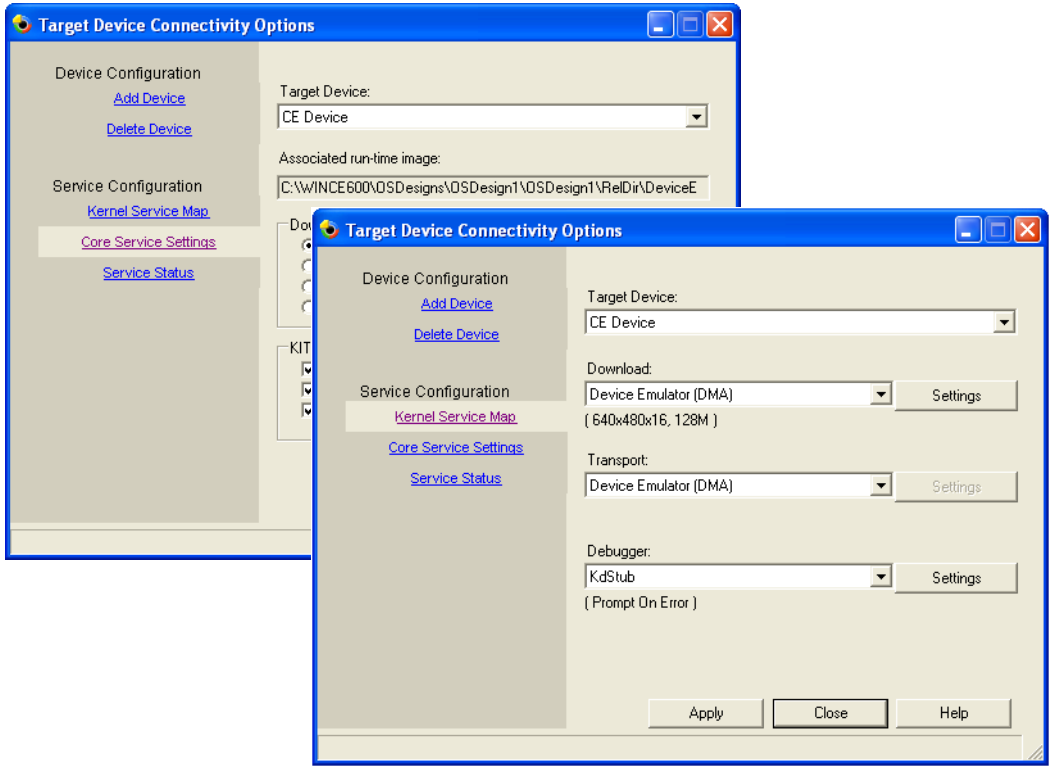


Abbildung 4.7 Konfigurieren der KITL-Kommunikationsschnittstelle

KITL unterstützt die folgenden beiden Modi:

- **Aktiver Modus** Platform Builder konfiguriert KITL standardmäßig für die Verbindung mit dem Entwicklungscomputer während des Startprozesses. Diese

Einstellung eignet sich am besten für das Debuggen des Kernels und der Anwendungen während der Softwareentwicklung.

- **Passiver Modus** Wenn Sie das Kontrollkästchen **Enable KITL on Device Boot** deaktivieren, können Sie KITL für den passiven Modus konfigurieren, damit Windows Embedded CE die KITL-Schnittstelle initialisiert. KITL stellt die Verbindung jedoch nicht während des Startprozesses her. Beim Auftreten einer Ausnahme versucht KITL die Verbindung mit dem Entwicklungscomputer herzustellen, um das JIT-Debuggen auszuführen. Der passive Modus eignet sich am besten für die Arbeit mit mobilen Geräten, die beim Start keine physische Verbindung mit dem Entwicklungscomputer haben.

HINWEIS KITL-Modi und Startargumente

Die Einstellung Enable KITL on Device Boot entspricht einem Startargument (BootArgs), das Platform Builder für den Boot Loader konfiguriert. Weitere Informationen zum Thema Boot Loader und deren Vorteile für den BSP-Entwicklungsprozess finden Sie im Abschnitt Boot Loaders in der Windows Embedded CE 6.0-Dokumentation auf der Microsoft MSDN-Website unter *http://msdn2.microsoft.com/en-us/library/aa917791.aspx*.

Debuggen eines Zielgeräts

Beachten Sie, dass die Debuggerkomponenten für den Entwicklungscomputer und das Zielgerät unabhängig voneinander ausgeführt werden. Beispielsweise können Sie den Kerneldebugger in Visual Studio 2005 mit Platform Builder ausführen, auch wenn kein aktives Zielgerät verfügbar ist. Wenn Sie im Menü **Debug** auf **Start** klicken oder die *F5*-Taste drücken, wird der Kerneldebugger gestartet und eine Meldung angezeigt, dass der Debugger auf eine Verbindung mit dem Zielgerät wartet. Wenn Sie ein für das Debuggen aktiviertes Run-Time Image ohne aktive KITL-Verbindung mit einem Debugger starten, reagiert das Image nicht, da das System auf Anforderungen vom Debugger wartet. Deshalb wird der Debugger normalerweise automatisch gestartet, wenn er mit einem für das Debuggen aktivierten Zielgerät verbunden ist. Wählen Sie im Menü **Target** die Option **Attach Device** aus, anstatt die *F5*-Taste zu drücken.

Aktivieren und Verwalten von Breakpoints

Die Debugfeatures von Platform Builder umfassen die meisten Funktionen, die auch in anderen Debuggern für Windows-Desktopanwendungen verfügbar sind. Sie können Breakpoints festlegen, den Code zeilenweise durchlaufen und Variablenwerte oder Objekteigenschaften im Fenster **Watch** anzeigen und ändern (siehe Abbildung 4.8).

Beachten Sie jedoch, dass zum Festlegen von Breakpoints die KdStub-Bibliothek im Run-Time Image vorhanden sein muss.

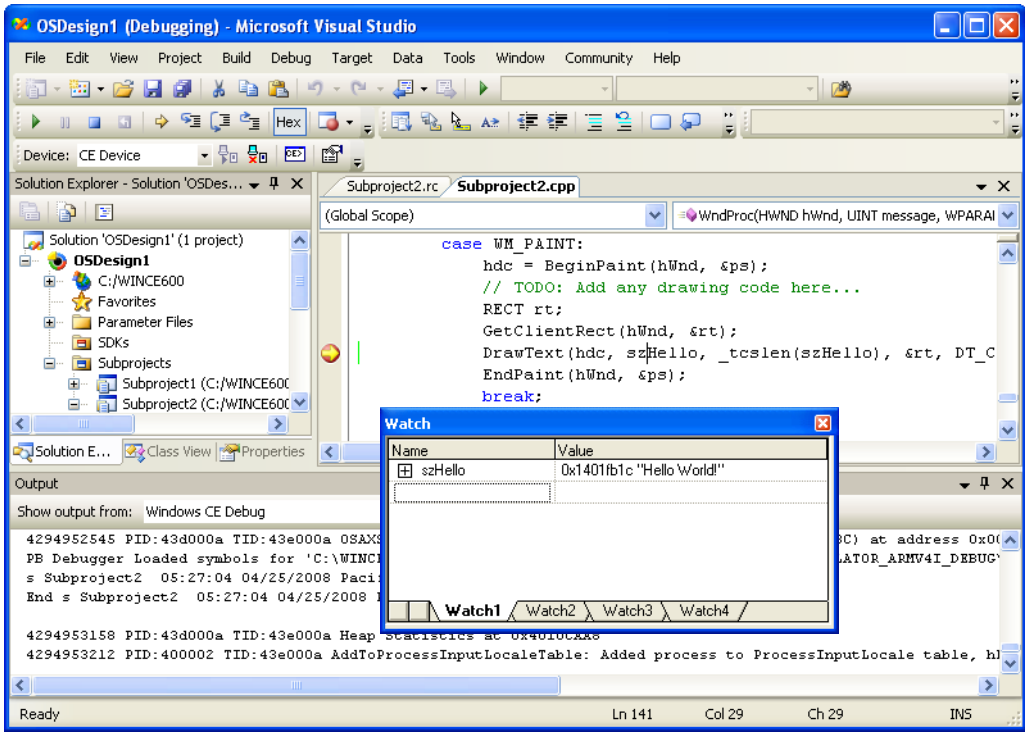


Abbildung 4.8 Debuggen der Hello World-Anwendung

Um einen Breakpoint festzulegen, wählen Sie in Visual Studio im Menü **Debug** die Option **Toggle Breakpoint** aus. Um einen Breakpoint in der aktuellen Zeile festzulegen, drücken Sie die *F9*-Taste oder klicken Sie auf den linken Randbereich der Codezeile. Platform Builder zeigt den Breakpoint mit einem roten Punkt oder einem roten Kreis an, abhängig davon, ob der Debugger den Breakpoint initialisieren kann. Der rote Kreis zeigt einen nicht initialisierten Breakpoint an. Nicht initialisierte Breakpoints treten auf, wenn die Visual Studio-Instanz nicht mit dem Zielcode verknüpft ist, der Breakpoint festgelegt, aber nicht geladen wurde, der Debugger nicht aktiviert ist oder der Debugger ausgeführt wird, aber die Codeausführung nicht angehalten wurde. Wenn Sie einen Breakpoint festlegen, während der Debugger ausgeführt wird, muss das Gerät den Debugger aufrufen, bevor dieser den Breakpoint initialisieren kann.

Zum Verwalten von Breakpoints in Visual Studio mit Platform Builder stehen Ihnen folgende Optionen zur Verfügung:

- **Die Fenster Source Code, Call Stack und Disassembly** Sie können einen Breakpoint festlegen, entfernen, aktivieren oder deaktivieren, indem Sie die *F9*-Taste drücken, im Menü **Debug** die Option **Toggle Breakpoint** auswählen oder im Kontextmenü auf **Insert/Remove Breakpoint** klicken.
- **Dialogfeld New Breakpoint** Sie können dieses Dialogfeld über die Untermenüs der Option **New Breakpoint** im Menü **Debug** öffnen. Im Dialogfeld **New Breakpoint** können Sie Breakpoints basierend auf dem Pfad und Bedingungen festlegen. Der Debugger wird nur dann an einem bedingten Breakpoint angehalten, wenn die Bedingung *TRUE* ist (beispielsweise wenn ein Schleifenzähler oder eine andere Variable einen bestimmten Wert hat).
- **Das Fenster Breakpoints** Um das Fenster **Breakpoints** zu öffnen, klicken Sie im Untermenü **Windows** des Menüs **Debug** auf **Breakpoints** oder drücken Sie *Alt+F9*. Im Fenster **Breakpoints**, in dem alle festgelegten Breakpoints aufgelistet sind, können Sie die Breakpointeigenschaften konfigurieren. Anstatt Informationen manuell im Dialogfeld **New Breakpoint** anzugeben, können Sie den gewünschten Breakpoint direkt im Quellcode festlegen und seine Eigenschaften im Fenster **Breakpoints** anzeigen, um Bedingungsparameter zu konfigurieren.

TIPP Zu viele Breakpoints

Verwenden Sie Breakpoints sparsam. Wenn Sie zu viele Breakpoints festlegen und ständig Resume auswählen, wird das Debuggen beeinträchtigt und es ist schwierig, einen bestimmten Aspekt des Systems zu überprüfen. Sie sollten Breakpoints deaktiveren und bei Bedarf erneut aktivieren.

Breakpoint-Einschränkungen

Wenn Sie die Eigenschaften eines Breakpoints im Dialgofeld **New Breakpoint** oder im Fenster **Breakpoints** konfigurieren, klicken Sie auf **Hardware**, um den Breakpoint als Hardware- oder Softwarebreakpoint zu konfigurieren. In OAL-Code oder Interrupt-Handlern können keine Softwarebreakpoints verwendet werden, da der Breakpoint die Systemausführung nicht beenden darf. Die KITL-Verbindung muss zusammen mit anderen Systemprozessen aktiv sein, um die Kommunikation mit dem Debugger auf dem Entwicklungscomputer sicherzustellen. KITL ist mit der OAL verknüpft und verwendet die auf Interrupts basierenden Kommunikationsmethoden

des Kernels. Wenn Sie in einer Interrupthandler-Funktion einen Breakpoint festlegen, kann das System bei Erreichen des Breakpoints nicht mehr kommunizieren, da die Interruptverarbeitung eine Singlethread-Funktion ist, die nicht unterbrochen werden kann.

Zum Debuggen von Interrupthandlern können Sie Debugmeldungen oder Hardwarebreakpoints verwenden. Hardwarebreakpoints erfordern jedoch einen eXDI-kompatiblen Debugger (beispielsweise einen JTAG-Test), um den Interrupt in der Debugregistrierung des Prozessors zu registrieren. Für einen Prozessor kann nur ein Hardwaredebugger aktiviert werden, obwohl JTAG mehrere Debugger verwalten kann. Die KdStub-Bibliothek kann nicht für das hardwareunterstützte Debuggen verwendet werden.

Um einen Hardwarebreakpoint zu konfigurieren, führen Sie folgende Schritte aus:

1. Öffnen Sie das Fenster **Breakpoint** über das Menü **Debug** und klicken Sie auf **Breakpoint**.
2. Klicken Sie mit der rechten Maustaste auf den Breakpoint in der Liste.
3. Klicken Sie auf **Breakpoint Properties**, um das Dialogfeld **Breakpoint Properties** zu öffnen, und anschließend auf **Hardware**.
4. Aktivieren Sie die Option **Hardware** und klicken Sie zweimal auf **OK**, um die Dialogfelder zu schließen.

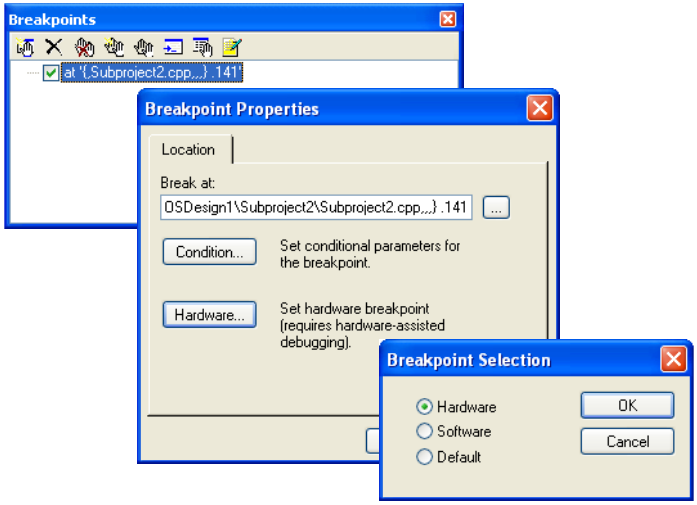


Abbildung 4.9 Festlegen eines Hardwarebreakpoints

Zusammenfassung

Das Aktivieren des Debuggers in der Platform Builder IDE ist ein unkompliziertes Verfahren, wenn Sie KITL und die Debuggerbibliotheken in das Run-Time Image einbeziehen. Im Dialogfeld **Target Device Connectivity Options** können Sie einen geeigneten Transport und Debugger auswählen. Der Transport ist normalerweise DMA oder Ethernet. Sie können den Entwicklungscomputer und das Zielgerät jedoch auch mit einem USB- oder seriellen Kabel miteinander verbinden.

Die Debugfeatures von Platform Builder umfassen die meisten Funktionen, die auch in anderen Debuggern für Windows-Desktopanwendungen verfügbar sind. Sie können Breakpoints festlegen, den Code zeilenweise durchlaufen und die Variablenwert und Objekteigenschaften im Fenster **Watch** anzeigen und ändern. Platform Builder unterstützt bedingte Breakpoints, um die Codeausführung basierend auf bestimmten Kriterien anzuhalten. Der Standarddebugger für Software ist KdStub, obwohl Sie mit Platform Builder auch einen eXDI-Treiber für das hardwareunterstützte Debuggen basierend auf einem JTAG-Test oder einem anderen Hardwaredebugger verwenden können. Das hardwareunterstützte Debuggen ermöglicht das Analysieren der Systemroutinen, die vor dem Laden des Kernels, der OAL-Komponenten und der Interrupthandler-Funktionen ausgeführt werden, für die Sie keine Softwarebreakpoints festlegen können.

Lektion 3: Testen des Systems mit CETK

Automatisierte Softwaretests sind wesentlich zum Verbessern der Produktqualität sowie zum Reduzieren der Entwicklungs- und Supportkosten. Diese Tests sind insbesondere wichtig, wenn Sie ein benutzerdefiniertes BSP für ein Zielgerät erstellen, neue Gerätetreiber hinzufügen oder benutzerdefinierten OAL-Code implementieren. Bevor Sie ein neues System für die Produktion freigeben, müssen Sie unter anderem funktionelle Tests, Komponententests und Stresstests ausführen, um alle Bestandteile des Systems zu überprüfen und sicherzustellen, dass das Zielgerät unter normalen Bedingungen zuverlässig funktioniert. Das Beheben von Problemen nachdem ein neues Produkt auf den Markt gebracht wurde, kann kostspielig werden. Im Allgemeinen ist es billiger, Testtools und Skripts zu erstellen, die den Betrieb des Zielgeräts simulieren, und Fehler zu beheben, während sich das System noch in der Entwicklungsphase befindet. Systemtests sollten nicht als zweitrangig behandelt werden. Mit dem CETK können Sie während der Softwareentwicklung effiziente Systemtests ausführen.

Nach Abschluss dieser Lektion können Sie:

- Typische Nutzungsszenarios für CETK-Testtools beschreiben.
- Benutzerdefinierte CETK-Tests erstellen.
- CETK-Tests auf einem Zielgerät ausführen.

Veranschlagte Zeit für die Lektion: 30 Minuten.

Windows Embedded CE Test Kit

Das CETK ist eine separate in Platform Builder für Windows Embedded CE integrierte Testanwendung, die die Stabilität von Anwendungen und Gerätetreibern basierend auf automatisierten Tests überprüft, die in einem CE-Testkatalog organisiert sind. Das CETK umfasst zahlreiche Standardtests für mehrere Treiberkategorien für Peripheriegerät. Sie können auch benutzerdefinierte Tests erstellen, um spezielle Anforderungen zu erfüllen.

HINWEIS CETK-Tests

Eine vollständige Liste der CETK-Standardtests CETK finden Sie im Abschnitt CETK Tests in der Windows Embedded CE 6.0-Dokumentation auf der Microsoft MSDN-Website unter *http://msdn2.microsoft.com/en-us/library/aa917791.aspx*.

CETK-Architektur

Die CETK-Anwendung ist eine Client/Server-Lösung mit Komponenten, die auf dem Entwicklungscomputer und auf dem Zielgerät ausgeführt werden (siehe Abbildung 4.10). Auf dem Entwicklungscomputer wird *CETest.exe* (die Serveranwendung) und auf dem Zielgerät wird *Clientside.exe* (die Clientanwendung), *Tux.exe* (das Testmodul) und *Kato.exe* (Testergebnisprotokollierung) ausgeführt. Diese Architektur ermöglicht Ihnen unter anderem mehrere Tests auf unterschiedlichen Geräten von einem Entwicklungscomputer aus gleichzeitig auszuführen. Die Server- und Clientanwendungen kommunizieren über KITL, ActiveSync® oder eine Windows Sockets-Verbindung (Winsock).

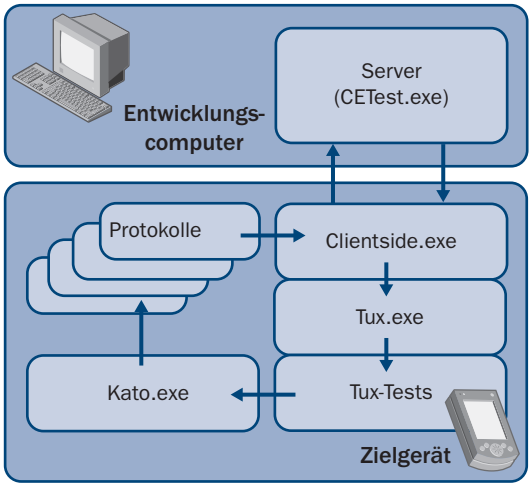


Abbildung 4.10 Die CETK Client/Server-Architektur

Die CETK-Anwendung umfasst folgende Komponenten:

- **Entwicklungsserver** *CETest.exe* stellt eine Benutzeroberfläche zum Ausführen und Verwalten der CETK-Tests bereit. Diese Anwendung ermöglicht außerdem das Konfigurieren der Servereinstellungen und Verbindungsparameter sowie die Verbindung mit dem Zielgerät. Nachdem die Geräteverbindung hergestellt wurde, kann der Server die Clientanwendung automatisch herunterladen und starten, Testanforderungen übermitteln und die Testergebnisse basierend auf den Protokollen in Echtzeit kompilieren.
- **Clientanwendung** *Clientside.exe* steuert das Testmodul und gibt die Testergebnisse an die Serveranwendung zurück. Wenn Clientside.exe auf dem Zielgerät nicht verfügbar ist, kann der Server nicht mit dem Zielgerät kommunizieren.

- **Testmodul** CETK-Tests werden in DLLs implementiert, die *Tux.exe* lädt und auf dem Zielgerät ausführt. Das Testmodul wird normalerweise remote über den Server und die Clientanwendung gestartet. Sie können *Tux.exe* jedoch auch lokal als eigenständige Anwendung starten.
- **Testergebnisprotokollierung** *Kato.exe* protokolliert die Ergebnisse der CETK-Tests in Protokolldateien. Tux DLLs verwenden die Protokollierung, um zusätzliche Informationen über einen Test bereitzustellen. Die Ausgabe kann an mehrere Geräte umgeleitet werden. Da alle CETK-Tests die gleiche Protokollierung und das gleiche Format verwenden, können Sie einen Standarddateiparser auswählen oder einen benutzerdefinierten Protokolldateiparser für die automatische Ergebnisverarbeitung implementieren, die auf bestimmten Anforderungen basiert.

HINWEIS CETK für verwalteten Code

Zum Überprüfen von systemeigenem und .NET-Code ist eine .NET-Version des CETK verfügbar. Weitere Informationen zur .NET-Version finden Sie im Abschnitt „Tux.Net Test Harness" in der Windows Embedded CE 6.0-Dokumentation auf der Microsoft MSDN-Website unter *http://msdn2.microsoft.com/en-us/library/aa934705.aspx*.

Verwenden des CETK

Sie können CETK-Tests, abhängig von den Verbindungsoptionen auf dem Zielgerät, auf mehrere Arten ausführen. Über KITL, Microsoft ActiveSync oder TCP/IP können Sie die Verbindung mit dem Zielgerät herstellen, die CETK-Komponenten herunterladen, die gewünschten Tests ausführen und die Ergebnisse auf dem Entwicklungscomputer anzeigen. Wenn das Zielgerät diese Verbindungsoptionen jedoch nicht unterstützt, müssen Sie die Tests lokal mit den entsprechenden Befehlszeilenparametern ausführen.

Verwenden der CETK-Serveranwendung

Um mit der Serveranwendung zu arbeiten, klicken Sie auf dem Entwicklungscomputer in der Programmgruppe **Windows Embedded CE 6.0** auf **Windows Embedded CE 6.0 Test Kit** und wählen Sie im Menü **Connection** den Befehl **Start Client** aus. Klicken Sie anschließend auf **Settings**, um den Transport zu konfigurieren. Wenn das Zielgerät eingeschaltet und mit dem Entwicklungscomputer verbunden ist, klicken Sie auf **Connect**, wählen Sie das gewünschte Zielgerät aus und klicken Sie auf **OK**, um den Kommunikationskanal zu aktivieren und die

erforderlichen Binärdateien bereitzustellen. Die CETK-Anwendung kann nun Tests auf dem Zielgerät ausführen.

Die CETK-Anwendung erkennt die auf dem Zielgerät verfügbaren Gerätetreiber automatisch und bietet eine praktische Methode zum Ausführen der Tests (siehe Abbildung 4.11). Klicken Sie im Menü **Tests** unter **Start/Stop Test** auf den Gerätenamen, um alle erkannten Komponenten zu testen. Sie können auch mit der rechten Maustaste auf den Knoten **Test Catalog** klicken und den Befehl **Start Tests** auswählen. Außerdem können Sie die einzelnen Container erweitern, mit der rechten Maustaste auf einen Gerätetest klicken und **Quick Start** auswählen, um nur eine Komponente zu testen. Die Serveranwendung ermöglicht den Zugriff auf die Tools Application Verifier, CPU Monitor, Resource Consume und Windows Embedded CE Stress, wenn Sie mit der rechten Maustaste auf den Geräteknoten klicken und das Untermenü **Tools** öffnen.

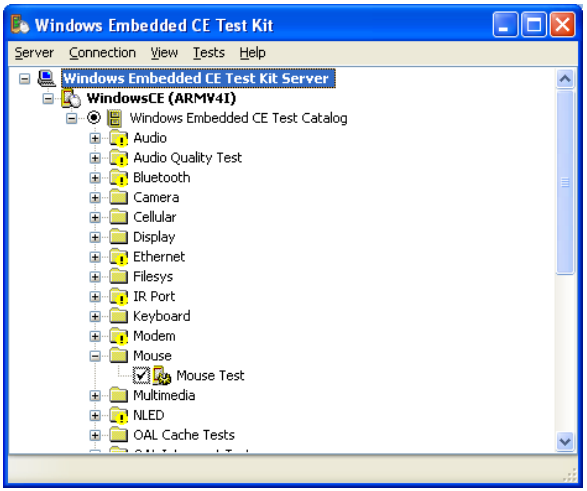


Abbildung 4.11 Die Benutzeroberfläche der CETK-Anwendung

Erstellen einer Testsuite

Anstatt alle Tests gleichzeitig oder einzelne Schnelltests auszuführen, können Sie Testsuites erstellen, die benutzerdefinierte Tests umfassen, die wiederholt während der Softwareentwicklung ausgeführt werden. Um eine neue Testsuite zu erstellen, verwenden Sie den Test Suite Editor, der über das Menü **Tests** in der Serveranwendung verfügbar ist. Über die Test Suite Editor-Benutzeroberfläche können Sie die Tests in einer Suite auswählen. Sie können die Testsuitedefinitionen in .tks-Dateien (Test Kit Suite) exportieren, die Sie wiederum auf anderen Entwicklungs-

computern importieren können, um sicherzustellen, dass alle Serveranwendungen die gleichen Tests ausführen. Die .tks-Dateien stellen die Basis für Testdefinitionsarchive dar.

Anpassen der Standardtests

Die Benutzeroberfläche ermöglicht das Anpassen der Befehlszeilen, die die Serveranwendung an das Testmodul (*Tux.exe*) sendet, um die Tests auszuführen. Um die Parameter eines Tests zu ändern, klicken Sie mit der rechten Maustaste unter **Test Catalog** auf den Test und wählen Sie die Option **Edit Command Line** aus. Beispielsweise analysiert der Storage Device Block Driver Benchmark Test die Leistung eines Speichergeräts, indem Daten in allen Gerätesektoren gelesen und in diese geschrieben werden. Das bedeutet, dass alle Daten auf dem Speichergerät gelöscht werden. Um den versehentlichen Datenverlust zu verhindern, wird der Storage Device Block Driver Benchmark Test standardmäßig übersprungen. Um den Storage Device Block Driver Benchmark Test erfolgreich auszuführen, müssen Sie den Parameter **-zorch** explizit in der Befehlszeile angeben.

Die unterstützten Befehlszeilenparameter hängen von der jeweiligen CETK-Testimplementierung ab. Ein Test unterstützt oder erfordert möglicherweise mehrere Konfigurationsparameter, beispielsweise eine Indexnummer, um den zu testenden Gerätetreiber zu identifizieren, oder andere Informationen, um den Test auszuführen.

HINWEIS Befehlszeilenparameter für CETK-Tests

Eine vollständige Liste der CETK-Standardtests mit Links zu weiteren Informationen, beispielsweise den Befehlszeilenparametern, finden Sie im Abschnitt CETK Tests in der Windows Embedded CE 6.0-Dokumentation auf der Microsoft MSDN-Website unter *http://msdn2.microsoft.com/en-us/library/ms893193.aspx*.

Manuelles Ausführen von Clientside.exe

Wenn Sie das **Windows Embedded CE Test Kit**-Katalogelement in das Run-Time Image einbeziehen, die CETK-Komponenten mit der Serveranwendung herunterladen oder die Komponenten mit dem Tool File Viewer vom Entwicklungscomputer auf das Zielgerät exportieren, können Sie *Clientside.exe* auf dem Zielgerät starten und die Verbindung mit einem Server manuell herstellen. Sollte auf dem Zielgerät das Dialogfeld **Run** nicht verfügbar sein, klicken Sie in der Platform Builder IDE im Menü **Target** auf **Run Programs**, wählen Sie **Clientside.exe** aus und klicken Sie auf **Run**.

Clientside.exe unterstützt die folgenden Befehlszeilenparameter, die Sie angeben können, um auf eine bestimmte Serveranwendung zuzugreifen, installierte Treiber zu erkennen und Tests automatisch auszuführen:

```
Clientside.exe [/i=<Server IP Address> | /n=<Server Name>] [/p=<Server Port Number>] [/a] [/
s] [/d] [/x]
```

Beachten Sie, dass Sie diese Parameter auch in der Datei *Wcetk.txt* oder im Registrierungsschlüssel *HKEY_LOCAL_MACHINE/Software/Microsoft/CETT* auf dem Zielgerät definieren können, um *Clientside.exe* ohne Befehlszeilenparameter zu starten. In diesem Fall sucht *Clientside.exe* die Datei *Wcetk.txt* zuerst im Stammverzeichnis und anschließend im *Windows*-Verzeichnis auf dem Zielgerät sowie im *Release*-Verzeichnis auf dem Entwicklungscomputer. Wenn *Wcetk.txt* in diesen Verzeichnissen nicht vorhanden ist, überprüft *Clientside.exe* den *CETT*-Registrierungsschlüssel. In Tabelle 4.5 sind die Parameter für *Clientside.exe* aufgeführt.

Tabelle 4.5 Startparameter für Clientside.exe

Befehls-zeile	Wcetk.txt	CETT-Registrierungs-schlüssel	Description
/n	SERVERNAME	ServerName (REG_SZ)	Gibt den Namen des Hostservers an. Dieser Parameter kann nicht zusammen mit **/i** verwendet werden und erfordert DNS (Domain Name System) für die Namensauflösung.
/i	SERVERIP	ServerIP (REG_SZ)	Gibt die IP-Adresse des Hosts an. Kann nicht zusammen mit **/n** verwendet werden.

Tabelle 4.5 Startparameter für Clientside.exe (Fortsetzung)

Befehlszeile	Wcetk.txt	CETT-Registrierungsschlüssel	Description
/p	PORTNUMBER	PortNumber (REG_DWORD)	Gibt die Serverportnummer an, die über die Serverschnittstelle konfiguriert werden kann.
/a	AUTORUN	Autorun (REG_SZ)	Der Wert 1 gibt an, dass das Gerät den Test automatisch startet, nachdem die Verbindung hergestellt wurde.
/s	DEFAULTSUITE	DefaultSuite (REG_SZ)	Gibt den Namen der auszuführenden Standardtestsuite an.
/x	AUTOEXIT	Autoexit (REG_SZ)	Der Wert 1 gibt an, dass die Anwendung automatisch beendet wird, nachdem die Tests abgeschlossen wurden.
/d	DRIVERDETECT	DriverDetect (REG_SZ)	Der Wert 0 gibt an, dass die Erkennung der Gerätetreiber deaktiviert ist.

Ausführen von CETK-Tests im Standalone-Modus

Clientside.exe ist mit *CETest.exe* auf dem Entwicklungscomputer verbunden. Sie können CETK-Tests jedoch auch ohne Verbindung ausführen. Wenn Sie das Windows Embedded CE Test Kit-Katalogelement in das Run-Time Image einbeziehen, können Sie das Testmodul (*Tux.exe*) direkt starten, das das Kato-Protokollmodul (*Kato.exe*) lädt, um die Testergebnisse in den Protokolldateien

nachzuverfolgen. Um beispielsweise Maustests (*Mousetest.dll*) auszuführen und die Ergebnisse in der Datei *Test_results.log* zu speichern, führen Sie folgenden Befehl aus:

```
Tux.exe -o -d mousetest -f test_results.log
```

HINWEIS Tux-Befehlszeilenparameter

Eine vollständige Liste der Befehlszeilenparameter für Tux.exe finden Sie im Abschnitt Tux Command-Line Parameters in der Windows Embedded CE 6.0-Dokumentation auf der Microsoft MSDN®-Website unter *http://msdn2.microsoft.com/en-us/library/aa934656.aspx*.

Erstellen einer benutzerdefinierten CETK-Testlösung

CETK umfasst zahlreiche Tests, die jedoch nicht alle Testanforderungen erfüllen (wenn Sie beispielsweise benutzerdefinierte Gerätetreiber zu einem BSP hinzugefügt haben). Zum Implementieren benutzerdefinierter Tests für angepasste Treiber, hängt CETK vom Tux-Framework ab. Mit der **WCE TUX DLL**-Vorlage in Platform Builder können Sie mit nur einigen Mausklicks ein Tux-Rahmenmodul erstellen. Bevor Sie die Logik für einen Treiber implementieren, sollten Sie den Quellcode auf vorhandene Testimplementierungen überprüfen. Der CETK-Quellcode kann mit dem Setup Wizard für Windows Embedded CE als Teil der Windows Embedded CE Shared Source installiert werden. Das Standardverzeichnis ist *%_WINCEROOT%\Private\Test*.

Erstellen eines benutzerdefinierten Tux-Moduls

Um eine mit dem Tux-Framework kompatible Testbibliothek zu erstellen, starten Sie den **Windows Embedded CE Subproject Wizard**, indem Sie ein Teilprojekt zum OS Design des Run-Time Images hinzufügen und die **WCE TUX DLL**-Vorlage auswählen. Der Tux Wizard erstellt ein Rahmenmodul, das Sie an die Treiberanforderungen anpassen können.

Sie müssen die folgenden Dateien im Teilprojekt bearbeiten, um das Tux-Rahmenmodul anzupassen:

- **Headerdatei *Ft.h*** Definiert die TUX Function Table (TFT), einschließlich einem Header und Einträgen. Die Funktionstabelleneinträge weisen den Funktionen, die die Testlogik enthalten, Test-IDs zu.
- **Quellcodedatei *Test.cpp*** Enthält die Testfunktionen. Das Tux-Rahmenmodul umfasst eine TestProc-Function, die Sie als Referenz verwenden können, wenn Sie benutzerdefinierte Tests zur Tux DLL hinzufügen. Sie können den Beispielcode ersetzen, um den benutzerdefinierten Treiber zu laden und zu

überprüfen, Aktivitäten über Kato protokollieren und nach Abschluss der Tests den entsprechenden Statuscode an das Tux-Modul zurückgeben.

Definieren eines benutzerdefinierten Tests in der CETK-Testanwendung

Da das Tux-Rahmenmodul voll funktionstüchtig ist, können Sie die Lösung kompilieren und das Run-Time Image ohne Codeänderungen erstellen. Um die neue Testfunktion auf einem Zielgerät auszuführen, müssen Sie mit der CETK-Serveranwendung einen benutzerdefinierten Test konfigurieren. Sie können hierzu den **User-Defined Test Wizard** in CETK starten, indem Sie im Menü **Tests** auf **User Defined** klicken. In Abbildung 4.12 ist der **User-Defined Test Wizard** mit den Konfigurationsparametern zum Ausführen eines Tux-Rahmenmoduls dargestellt.

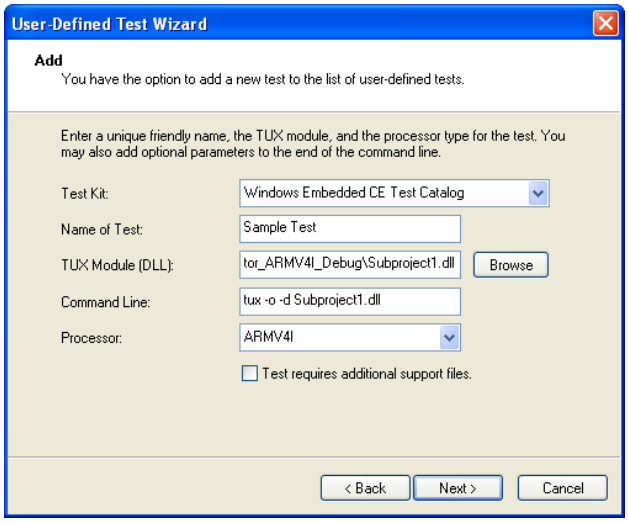


Abbildung 4.12 Konfigurieren eines benutzerdefinierten Tests im **User-Defined Test Wizard**

Debuggen eines benutzerdefinierten Tests

Da Tux-Tests vom Code und der Logik abhängen, die in den Tux DLLs implementiert sind, müssen Sie den Testcode möglicherweise debuggen. Sie können Breakpoints in den Testroutinen festlegen. Wenn jedoch die Codeausführung an diesen Breakpoints angehalten wird, wird die Verbindung zwischen der Clientanwendung (*Clientside.exe*) und der Serveranwendung (*CETest.exe*) getrennt. Deshalb sollten Sie Debugmeldungen anstatt Breakpoints verwenden. Wenn Breakpoints für das umfassende Debuggen erforderlich sind, führen Sie *Tux.exe* im Standalone-Modus direkt auf dem Zielgerät aus. Um die erforderliche Befehlszeile in der

Serveranwendung anzuzeigen, klicken Sie mit der rechten Maustaste auf den Test und wählen Sie **Edit Command Line** aus.

Analysieren der CETK-Testergebnisse

CETK-Tests sollten die Kato-Protokolltestergebnisse verwenden, die im Tux-Rahmenmodul veranschaulicht sind:

```
g_pKato->Log(LOG_COMMENT, TEXT("This test is not yet implemented."));
```

Die Serveranwendung ruft die Protokolle automatisch über *Clientside.exe* ab und speichert die Protokolle auf dem Entwicklungscomputer. Sie können auch über andere Tools auf die Protokolldateien zugreifen. Wenn Sie CETK im Standalone-Modus verwenden, können Sie die Protokolldateien mit dem Tool File Viewer auf den Entwicklungscomputer importieren.

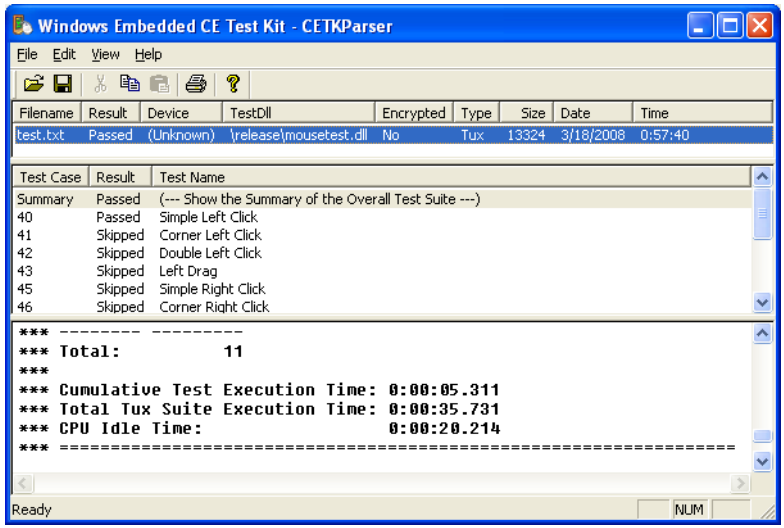


Abbildung 4.13 Analysieren der CETK-Testergebnisse

CETK umfasst einen allgemeinen CETK-Parser (*Cetkpar.exe*), der im Ordner *C:\Program Files\Microsoft Platform Builder\6.00\Cepb\Wcetk* gespeichert ist und die importierten Protokolldateien anzeigt (siehe Abbildung 4.13). Um den Parser zu laden, klicken Sie mit der rechten Maustaste in der Serveranwendung auf einen abgeschlossenen Test und wählen Sie **View Results** aus. Sie können *Cetkpar.exe* auch direkt starten. Einige Tests, beispielsweise auf *PerfLog.dll* basierende Leistungstests, können in CSV-Dateien (Comma-Separated Values) geparst und in einem Arbeitsblatt geöffnet werden, um die Leistungsdaten zusammenzufassen. Zu diesem Zweck

umfasst CETK den Parser PerfToCsv. Sie können für bestimmte Analysen benutzerdefinierte Parser entwickeln. Kato-Protokolldateien haben ein einfaches Textformat.

Zusammenfassung

Das erweiterbare Windows Embedded CE Test Kit ermöglicht das Testen von Treibern und Anwendungen auf einem Zielgerät im verbundenen Modus sowie im Standalone-Modus. Das Ausführen der CETK-Tools im Standalone-Modus ist nützlich, wenn das Zielgerät die Verbindung über KITL, ActiveSync oder TCP/IP nicht unterstützt. Entwickler verwenden das CETK normalerweise zum Testen der Gerätetreiber, die zum BSP eines Zielgeräts hinzugefügt wurden.

CETK hängt vom Tux-Testmodul ab, das ein allgemeines Framework für alle Test-DLLs bereitstellt. Die Tux DLLs enthalten die Testlogik und werden auf dem Zielgerät ausgeführt, um die Treiber zu laden und zu überprüfen. Tux DLLs verwenden Kato, um die Testergebnisse in den Protokolldateien nachzuverfolgen. Sie können in der CETK-Testanwendung oder mit anderen Tools, beispielsweise benutzerdefinierten Parsern und Arbeitsblättern, direkt auf die Dateien zugreifen.

Lektion 4: Testen des Boot Loaders

Die Aufgabe des Boot Loaders ist das Laden des Kernels in den Speicher und das Aufrufen der OS-Startroutine, nachdem das Gerät eingeschaltet wurde. In Windows Embedded CE ist der Boot Loader ein Bestandteil des Board Support Package (BSP) und initialisiert die Kernhardwareplattform, lädt das Run-Time Image herunter und startet den Kernel. Auch wenn Sie nicht planen, im fertigen Produkt einen Boot Loader zu implementieren und das Run-Time Image direkt zu laden, kann ein Boot Loader während der Entwicklungsphase nützlich sein. Ein Boot Loader kann unter anderem die Entwicklung des Run-Time Images vereinfachen. Das Herunterladen des Run-Time Images über Ethernet-Verbindungen, serielle Kabel, DMA oder USB-Verbindungen von einem Entwicklungscomputer ist ein praktisches Feature, das Ihnen Zeit bei der Entwicklung sparen kann. Basierend auf dem Quellcode in Platform Builder für Windows Embedded CE 6.0 können Sie einen benutzerdefinierten Boot Loader entwickeln, um neue Hardware oder Features zu unterstützen. Beispielsweise können Sie einen Boot Loader verwenden, um das Run-Time Image aus dem RAM in den Flashspeicher zu kopieren, ohne dass ein Flashspeicher-Programmierer oder Institute of Electrical and Electronic Engineers (IEEE) 1149.1-kompatibler Testport und Boundary-Scanning-Technologie erforderlich ist. Das Debuggen und Testen eines Boot Loaders ist jedoch ein komplexes Verfahren, da Sie mit Code arbeiten, der vor dem Laden des Kernels ausgeführt wird.

Nach Abschluss dieser Lektion können Sie:

- Die CE Boot Loader-Architektur beschreiben.
- Allgemeine Debugmethoden für Boot Loader auflisten.

Veranschlagte Zeit für die Lektion: 15 Minuten.

CE Boot Loader-Architektur

Ein Boot Loader lädt ein kleines Programm mit Vorstartroutinen im linearen, permanenten Speicher, auf den durch die CPU zugegriffen werden kann. Nachdem das ursprüngliche Boot Loader-Image an der Speicheradresse, bei der die CPU beginnt, den Code über ein integriertes Überwachungsprogramm (vom Boardhersteller oder einem JTAG-Test) abzurufen, auf das Zielgerät kopiert wurde, wird der Boot Loader bei jedem Start oder Neustart des Systems ausgeführt. Die in dieser Phase ausgeführten Boot Loader-Aktionen umfassen sowohl das Initialisieren

der CPU (Central Processing Unit), des Speichercontrollers, der Systemuhr, der Universal Asynchronous Receiver/Transmitters (UARTs), der Ethernet-Controller und anderer Hardwarekomponenten als auch das Herunterladen und Kopieren des Run-Time Images in den RAM basierend auf dem BIB-Layout (Binary Image Builder) und das Wechseln zur *StartUp*-Funktion. Der letzte Eintrag des Run-Time Images enthält die Startadresse dieser Funktion. Die *StartUp*-Funktion setzt den Startprozess durch Aufrufen der Routinen für die Kernelinitialisierung fort.

Obwohl die Komplexität und der Verwendungszweck der Boot Loader-Implementierungen unterschiedlich sind, sind über die statischen Bibliotheken in Windows Embedded CE gemeinsame Eigenschaften verfügbar, um die Boot Loader-Entwicklung zu unterstützen (siehe Abbildung 4.14). Die Boot Loader-Architektur beeinflusst das Debuggen des Boot Loader-Codes. In Kapitel 5 „Anpassen eines BSP" ist die Boot Loader-Entwicklung ausführlich beschrieben.

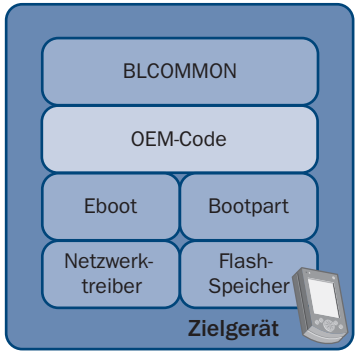


Abbildung 4.14 Boot Loader-Architektur in Windows Embedded CE

Die Boot Loader-Architektur in The Windows Embedded CE basiert auf folgenden Codesegmenten und Bibliotheken:

- **BLCOMMON** Implementiert das Boot Loader-Basisframework zum Kopieren des Boot Loaders aus dem Flashspeicher in den RAM, um die Ausführung, das Decodieren des Image-Dateiinhalts, das Überprüfen der Prüfsummen und das Überwachen des Ladestatus zu beschleunigen. BLCOMMON ruft über den Prozess zum Verarbeiten hardwarespezifischer Anpassungen eindeutig definierte OEM-Funktionen auf.
- **OEM-Code** Der OEM muss diesen Code für seine Hardwareplattformen implementieren, um die BLCOMMON-Bibliothek zu unterstützen.

- **Eboot** Stellt Dynamic Host Configuration Protocol (DHCP), Trivial File Transfer Protocol (TFTP) und User Datagram Protocol (UDP) bereit, um Run-Time Images über Ethernet-Verbindungen zu übertragen.
- **Bootpart** Stellt Routinen für die Speicherpartitionierung bereit, damit der Boot Loader jeweils eine Partion für das binäre ROM-Imagedateisystem (BinFS) und ein anderes Dateisystem auf dem gleichen Speichergerät erstellen kann. Bootpart kann außerdem eine Startpartition erstellen, auf der die Startparameter gespeichert werden.
- **Netzwerktreiber** Kapselt die Basisinitialisierung und die Zugriffsstammfunktionen für zahlreiche Netzwerkcontrollergeräte ein. Die generische Schnittstelle für die Bibliotheken wird vom Boot Loader und dem Betriebssystem verwendet. Der Boot Loader verwendet die Schnittstelle zum Übertragen von Run-Time Images und das Betriebssystem implementiert über die Schnittstelle eine KITL-Verbindung mit Platform Builder.

Debugmethoden für Boot Loader

Das Boot Loader-Design besteht normalerweise aus mindestens zwei Teilen. Der erste Teil ist in der Assemblersprache geschrieben und initialisiert das System bevor der zweite Teil aktiviert wird, der in C geschrieben ist. Wenn Sie eine auf BLCOMMON basierende Architektur verwenden, müssen Sie den Assemblercode möglicherweise nicht debuggen (siehe Abbildung 4.14). Wenn das Gerät mit einem UART ausgestattet ist, können Sie mit dem *RETAILMSG*-Makro in C-Code die Daten über eine serielle Schnittstelle ausgeben.

Abhängig davon, ob Sie den Assembler- oder C-Code debuggen, sind folgende Debugmethoden verfügbar:

- **Assemblercode** Die Standarddebugmethoden für den initialen Startcode basieren auf LEDs, beispielsweise auf einem Debugboard mit Sieben-Segment-LEDs und UARTs für eine serielle Kommunikationsschnittstelle, da der Zugriff auf die GPIO-Register und das Ändern der I/O-Ports relativ unkompliziert ist.
- **C-Code** Das Debuggen auf C-Codeebene ist einfacher, da Sie auf erweiterte Kommunikationsschnittstellen und Debugmakros zugreifen können.
- **Assembler und C-Code** Wenn ein Hardwaredebugger (JTAG-Test) verfügbar ist, können Sie Platform Builder zusammen mit einem eXDI-Treiber verwenden, um den Boot Loader zu debuggen.

PRÜFUNGSTIPP

Um die Zertifizierungsprüfung zu bestehen, müssen Sie mit den verschiedenen Methoden zum Debuggen des Boot Loaders, des Kernels, der Gerätetreiber und der Anwendungen vertraut sein.

Zusammenfassung

Das Debuggen des Boot Loaders ist ein komplexes Verfahren, das solide Kenntnisse der Hardwareplattform voraussetzt. Wenn ein Hardwaredebugger verfügbar ist, können Sie Platform Builder zusammen mit einem eXDI-Treiber für das hardwareunterstützte Debuggen verwenden. Anderenfalls debuggen Sie den Assemblercode unter Verwendung eines LED-Boards und verwenden Sie C-Makros, um die Debugmeldungen in C-Code über eine serielle Kommunikationsschnittstelle auszugeben.

Lab 4: Debuggen und Testen des Systems basierend auf KITL, Debugzonen und den CETK-Tools

In diesem Lab debuggen Sie eine Konsolenanwendung, die als Teilprojekt zu einem OS Design basierend auf dem Geräteemulator-BSP hinzugefügt wurde. Um das Debuggen zu aktivieren, beziehen Sie KdStub und KITL in das Run-Time Image ein und konfigurieren die entsprechenden Verbindungsoptionen für das Zielgerät. Sie ändern den Quellcode der Konsolenanwendung, um die Unterstützung für Debugzonen zu implementieren, legen die ursprünglichen aktiven Debugzonen im Pegasus-Registrierungsschlüssel fest und verbinden das Zielgerät mit dem Kerneldebugger, um die Debugmeldungen in Visual Studio im Fenster Output anzuzeigen. Anschließend verwenden Sie das CETK, um den Maustreiber im Run-Time Image zu testen. Um das ursprüngliche OS Design in Visual Studio zu erstellen, verwenden Sie die in Kapitel 1 beschriebenen Verfahren.

HINWEIS Detailierte schrittweise Anleitungen

Um die Verfahren in diesem Lab erfolgreich auszuführen, lesen Sie das Dokument Detailed Step-by-Step Instructions for Lab 4 im Begleitmaterial.

▶ Aktivieren von KITL und Verwenden von Debugzonen

1. Öffnen Sie das OS Design-Projekt, das Sie in Lab 1 in Visual Studio erstellt haben, klicken Sie mit der rechten Maustaste auf den OS Design-Namen und wählen Sie **Properties** aus, um die OS Design-Eigenschaften zu bearbeiten. Wählen Sie anschließend **Configuration Properties** und **Build Options** aus, und aktivieren Sie das Kontrollkästchen **Enable KITL**.
2. Aktivieren Sie im Dialogfeld mit dem OS Design-Eigenschaften das Feature **Kernel Debugger**, übernehmen Sie die Änderungen und schließen Sie das Dialogfeld.
3. Stellen Sie sicher, dass Sie in der Debug-Buildkonfiguration arbeiten, um ein Image zu erstellen, dass KITL und die Kerneldebuggerkomponenten umfasst, die in den vorherigen Schritten aktiviert wurden.
4. Erstellen Sie das OS Design, indem Sie im Menü **Build** unter **Advanced Build Commands** den Befehl **Rebuild Current BSP and Subprojects** auswählen (wenn während der nachfolgenden Schritte Fehler auftreten, führen Sie **Clean Sysgen** aus).
5. Klicken Sie im Menü **Target** auf **Connectivity Options**, um das Dialogfeld **Target Device Connectivity Options** zu öffnen. Konfigurieren Sie folgende Einstellungen und klicken Sie auf **OK**:

Table 4-6 Device connectivity Einstellungen

Konfigurationsparameter	Einstellung
Download	Device Emulator (DMA)
Transport	Device Emulator (DMA)
Debugger	KdStub

6. Fügen Sie ein Teilprojekt zum OS Design hinzu und wählen Sie die Vorlage **WCE Console Application** aus. Geben Sie dem Projekt den Namen ***TestDbgZones*** und wählen Sie im **CE Subproject Wizard** die Option **A Typical Hello World Application** aus.
7. Fügen Sie die neue Headerdatei *DbgZone.h* zum Teilprojekt hinzu und definieren Sie folgende Zonen:

```
#include <DBGAPI.H>

#define DEBUGMASK(n)        (0x00000001<<n)
#define MASK_INIT           DEBUGMASK(0)
#define MASK_DEINIT         DEBUGMASK(1)
#define MASK_ON             DEBUGMASK(2)
#define MASK_ZONE3          DEBUGMASK(3)
#define MASK_ZONE4          DEBUGMASK(4)
#define MASK_ZONE5          DEBUGMASK(5)
#define MASK_ZONE6          DEBUGMASK(6)
#define MASK_ZONE7          DEBUGMASK(7)
#define MASK_ZONE8          DEBUGMASK(8)
#define MASK_ZONE9          DEBUGMASK(9)
#define MASK_ZONE10         DEBUGMASK(10)
#define MASK_ZONE11         DEBUGMASK(11)
#define MASK_ZONE12         DEBUGMASK(12)
#define MASK_FAILURE        DEBUGMASK(13)
#define MASK_WARNING        DEBUGMASK(14)
#define MASK_ERROR          DEBUGMASK(15)

#define ZONE_INIT           DEBUGZONE(0)
#define ZONE_DEINIT         DEBUGZONE(1)
#define ZONE_ON             DEBUGZONE(2)
#define ZONE_3              DEBUGZONE(3)
#define ZONE_4              DEBUGZONE(4)
#define ZONE_5              DEBUGZONE(5)
#define ZONE_6              DEBUGZONE(6)
#define ZONE_7              DEBUGZONE(7)
#define ZONE_8              DEBUGZONE(8)
#define ZONE_9              DEBUGZONE(9)
#define ZONE_10             DEBUGZONE(10)
#define ZONE_11             DEBUGZONE(11)
```

```
#define ZONE_12             DEBUGZONE(12)
#define ZONE_FAILURE        DEBUGZONE(13)
#define ZONE_WARNING        DEBUGZONE(14)
#define ZONE_ERROR          DEBUGZONE(15)
```

8. Fügen Sie eine *Include*-Anweisung für die Headerdatei *DbgZone.h* zur Datei *TestDbgZones.c* hinzu:

```
#include "DbgZone.h"
```

9. Definieren Sie die Variable *dpCurSettings* für die Debugzonen bevor der Funktion *_tmain*:

```
DBGPARAM dpCurSettings =
{
  TEXT("TestDbgZone"),
  {
    TEXT("Init"), TEXT("Deinit"), TEXT("On"), TEXT("n/a"),
    TEXT("n/a"), TEXT("n/a"), TEXT("n/a"), TEXT("n/a"),
    TEXT("n/a"), TEXT("n/a"), TEXT("n/a"), TEXT("n/a"),
    TEXT("n/a"), TEXT("Failure"), TEXT("Warning"), TEXT("Error")
  },
  MASK_INIT | MASK_ON | MASK_ERROR
};
```

10. Registrieren Sie die Debugzonen des Moduls im der ersten Zeile der Funktion *_tmain*:

```
DEBUGREGISTER(NULL);
```

11. Verwenden Sie die Makros *RETAILMSG* und *DEBUGMSG*, um Debugmeldungen anzuzeigen und Debugzonen zuzuordnen:

```
DEBUGMSG(ZONE_INIT,
        (TEXT("Message : ZONE_INIT")));
RETAILMSG(ZONE_FAILURE || ZONE_WARNING,
         (TEXT("Message : ZONE_FAILURE || ZONE_WARNING")));
DEBUGMSG(ZONE_DEINIT && ZONE_ON,
        (TEXT("Message : ZONE_DEINIT && ZONE_ON")));
```

12. Erstellen Sie die Anwendung, stellen Sie die Verbindung mit dem Zielgerät her und starten Sie die Anwendung über das Fenster **Target Control**.

13. Beachten Sie, dass im Fenster **Output** nur die erste Debugmeldung angezeigt wird:

```
4294890680 PID:3c50002 TID:3c60002 Message : ZONE_INIT
```

14. Öffnen Sie den Registrierungs-Editor (*Regedit.exe*) auf dem Entwicklungscomputer, um die übrigen Debugzonen standardmäßig zu aktivieren.

15. Öffnen Sie den Schlüssel *HKEY_CURRENT_USER\Pegasus\Zones* und erstellen Sie einen REG_DWORD-Wert namens *TestDbgZone* (basierend auf dem Namen des in der Variablen *dpCurSettings* definierten Moduls).

16. Legen Sie den Wert auf *0xFFFF* fest, um alle 16 benannten Zonen zu aktivieren, die den niedrigen 16-Bits im 32-Bit DWORD-Wert entsprechen (siehe Abbildung 4.15).

17. Starten Sie die Anwendung erneut in Visual Studio, um folgende Ausgabe zu generieren:

```
4294911331 PID:2270006 TID:2280006 Message : ZONE_INIT
4294911336 PID:2270006 TID:2280006 Message : ZONE_FAILURE || ZONE_WARNING
4294911336 PID:2270006 TID:2280006 Message : ZONE_DEINIT && ZONE_ON
```

18. Ändern Sie den TestDbgZone-Wert in der Registrierung, um andere Debugzonen zu aktivieren und zu deaktivieren, und die Ergebnisse im Fenster **Output** in Visual Studio anzuzeigen.

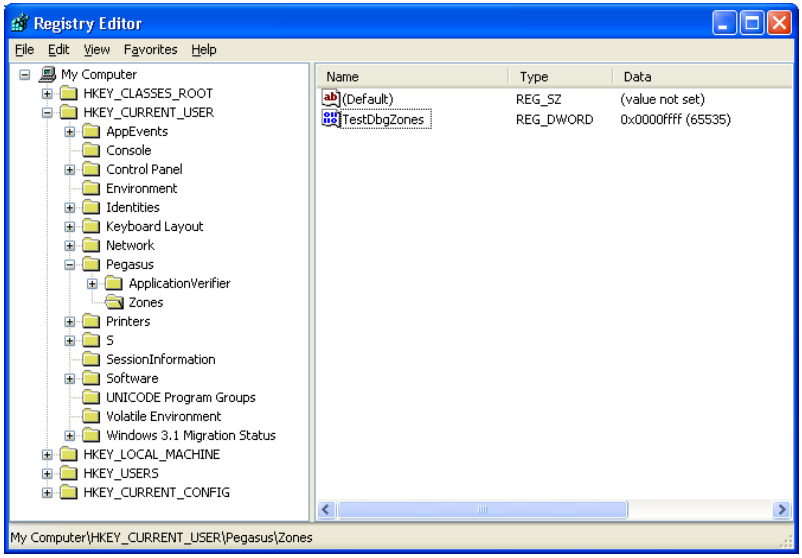


Abbildung 4.15 HKEY_CURRENT_USER\Pegasus\Zones: "TestDbgZone"=dword:FFFF

HINWEIS Aktivieren und Deaktivieren von Debugzonen in Platform Builder

Sie können die Debugzonen für das *TestDbgZone*-Modul in Platform Builder nicht steuern, da der Anwendungsprozess beendet wird, bevor Sie die aktive Zone für das Modul öffnen und ändern können. Sie können nur die Debugzonen für geladene Module in Platform Builder verwalten, beispielsweise für grafische Anwendungen und DLLs.

▶ Testen des Maustreibers mit CETK

1. Öffnen Sie die Windows CE Test Kit-Anwendung im Startmenü auf dem Entwicklungscomputer (klicken Sie im Windows Embedded CE 6.0-Menü auf Windows Embedded CE Test Kit).
2. Klicken Sie im Fenster **Windows Embedded CE Test Kit** im Menü **Connection** auf **Start Client**, um die Verbindung mit dem Zielgerät herzustellen.
3. Klicken Sie auf **Connect** und wählen Sie das Gerät im Fenster **Connection Manager** aus.
4. Überprüfen Sie, ob die Serveranwendung mit dem Gerät verbunden ist, die erforderlichen CETK-Binärdateien bereitstellt, die verfügbaren Gerätetreiber erkennt und alle Komponenten hierarchisch anzeigt (siehe Abbildung 4.16).
5. Klicken Sie mit der rechten Maustaste auf den Knoten **Windows CE Test Catalog** und wählen Sie **Deselect All Tests** aus.
6. Öffnen Sie alle Knoten in der Liste und aktivieren Sie das Kontrollkästchen **Mouse Test**.
7. Klicken Sie im Menü **Test** auf **Start/Stop Test**, um einen Maustest auszuführen.
8. Führen Sie auf dem Zielgerät die erforderlichen Mausaktionen aus.
9. Zeigen Sie den Testbericht nach Abschluss des Tests an, indem Sie mit der rechten Maustaste auf den Testeintrag klicken und **View Results** auswählen.
10. Überprüfen Sie die Ergebnisse im CETK-Parser, der erfolgreiche, übersprungene und fehlgeschlagene Testverfahren anzeigt.

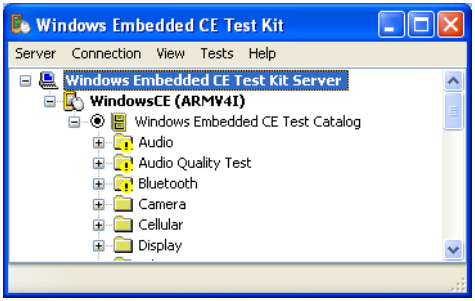


Abbildung 4.16 Gerätekategorien im Fenster Windows Embedded CE Test Kit

Lernzielkontrolle

Platform Builder für Windows Embedded CE enthält zahlreiche Debug- und Testtools zum Analysieren und Beheben von Fehlern sowie zum Überprüfen der endgültigen Systemkonfiguration, bevor das System für die Produktion freigegeben wird. Die Debugtools werden in Visual Studio integriert und kommunizieren über KITL-Verbindungen mit dem Zielgerät. Außerdem können Sie Speicherabbilder und den CE Dump File Reader verwenden, um das System im Offline-Modus zu debuggen, was insbesondere für das Postmortem-Debuggen nützlich ist. Die Debugumgebung ist mittels eXDI-Treibern erweiterbar, um das hardwareunterstützte Debuggen auszuführen, das über die Funktionen des Standard-Kerneldebuggers hinausgeht.

Der Kerneldebugger ist ein Hybriddebugger für Kernelkomponenten und Anwendungen. Das Debuggen wird automatisch gestartet, wenn Sie die Verbindung mit einem Zielgerät herstellen und KdStub und KITL aktiviert sind. Im Fenster Target Control können Sie Anwendungen zum Debuggen starten und Systemtests basierend auf CEDebugX-Befehlen ausführen. Beachten Sie jedoch, dass Sie in Interrupthandlern oder OAL-Modulen keine Breakpoints festlegen können, da der Kernel auf diesen Ebenen im Single-Thread-Modus funktioniert und die Kommunikation mit dem Entwicklungscomputer beendet, wenn die Codeausführung angehalten wird. Um Interrupthandler zu debuggen, verwenden Sie einen Hardwaredebugger oder Debugmeldungen. Das Feature für Debugmeldungen unterstützt Debugzonen, die die Informationsausgabe steuern, ohne dass das Run-Time Image neu erstellt werden muss. Mit den Debugmeldungen können Sie außerdem das C-Codesegment eines Boot Loaders debuggen. Für das Assembly-Codesegment müssen Sie jedoch einen Hardwaredebugger oder ein LED-Board verwenden.

KITL ist eine Voraussetzung zum Zentralisieren der Systemtests basierend auf der CETK-Testanwendung, obwohl Sie die CETK-Tests auch im Standalone-Modus ausführen können. Wenn Sie ein benutzerdefiniertes BSP für ein Zielgerät entwickeln, können Sie mit CETK automatisierte oder teilweise automatisierte Komponententests basierend auf angepassten Tux DLLs ausführen. Platform Builder umfasst eine **WCE TUX DLL**-Vorlage zum Erstellen eines Tux-Rahmenmoduls, das Sie an Ihre Testanforderungen anpassen können. Sie können die benutzerdefinierte Tux DLL in der CETK-Testanwendung integrieren und die Tests einzeln oder als Teil einer Testsuite ausführen. Da alle CETK-Tests die gleiche Protokollierungs-Engine und das gleiche Protokolldateiformat verwenden, können Sie die Ergebnisse der Standardtests und benutzerdefinierten Tests mit dem gleichen Parsertool analysieren.

Schlüsselbegriffe

Kennen Sie die Bedeutung der folgenden wichtigen Begriffe? Sie können Ihre Antworten überprüfen, wenn Sie die Begriffe im Glossar am Ende dieses Buches nachschlagen.

- Debugzonen
- KITL
- Hardwaredebugger
- dpCurSettings
- DebugX
- Target Control
- Tux
- Kato

Empfohlene Vorgehensweise

Um die in diesem Kapitel behandelten Prüfungsschwerpunkte erfolgreich zu beherrschen, sollten Sie die folgenden Aufgaben durcharbeiten.

Erkennen von Speicherverlusten

Fügen Sie ein Teilprojekt für eine Konsolenanwendung im OS Design hinzu, das Speicherverluste generiert, indem Speicherblöcke zugeordnet, aber nicht freigegeben, werden. Verwenden Sie die in diesem Kapitel beschriebenen Tools, um Probleme zu erkennen und zu beheben.

Anpassen eines CETK-Tests

Fügen Sie ein Teilprojekt für eine **WCE TUX DLL** zum OS Design hinzu. Erstellen Sie die Tux DLL und registrieren Sie diese in der Windows Embedded CE Test Kit-Anwendung. Führen Sie einen CETK-Test aus und überprüfen Sie die Testergebnisse. Legen Sie Breakpoints in der Tux DLL fest und debuggen Sie den Code, indem Sie einen CETK-Test im Standalone-Modus ausführen.

Kapitel 5

Anpassen eines Board Support Package

Anwendungsentwickler müssen nur selten ein BSP (Board Support Package) erstellen. OEMs (Original Equipment Manufacturers) benötigen jedoch ein BSP, um Microsoft® Windows® Embedded CE 6.0 R2 auf eine neue Hardwareplattform zu portieren. Windows Embedded CE umfasst eine PQOAL-Architektur (Production Quality OEM Adaptation Layer), die die Wiederverwendung von Code basierend auf OAL-Bibliotheken unterstützt, die nach Prozessormodell und OAL-Funktion organisiert sind. Microsoft empfiehlt OEM-Entwicklern, ein vorhandenes BSP zu klonen und an ihre Anforderungen anzupassen, um die getesteten und bewährten Features für die Energieverwaltung, Leistungsoptimierung und IOCTL (Input/Output Controls) zu nutzen. In diesem Kapitel ist die PQOAL-Architektur beschrieben, einschließlich das Klonen von BSPs und die Funktionen, die der OEM-Entwickler implementieren muss, um Windows Embedded CE an neue Hardwarearchitekturen und Modelle anzupassen. Sie sollten mit den verschiedenen Aspekten der BSP-Anpassung vertraut sein, auch wenn Sie keine BSPs entwickeln. Die BSP-Anpassung umfasst das Ändern des Startprozesses und das Implementieren der Kernelinitialisierungsroutinen, um Gerätetreiber, Energieverwaltungsfunktionen und die Unterstützung für die Leistungsoptimierung hinzuzufügen.

Prüfungsziele in diesem Kapitel

- Verstehen der BSP-Architektur von Windows Embedded CE
- Ändern und Anpassen eines BSP und Boot Loaders für bestimmte Zielgeräte
- Verstehen der Speicherverwaltung und des Speicherlayouts
- Aktivieren der Energieverwaltung in einem BSP

Bevor Sie beginnen

Damit Sie die Lektionen in diesem Kapitel durcharbeiten können, benötigen Sie:

- Mindestens Grundkenntnisse in der Windows Embedded CE-Softwareentwicklung.
- Umfassende Kenntnisse in den Hardwarearchitekturen für eingebettete Geräte.
- Grundkenntnisse in der Energieverwaltung und der Implementierung der Energieverwaltung in Treibern und Anwendungen.
- Einen Entwicklungscomputer, auf dem Microsoft Visual Studio® 2005 Service Pack 1 und Platform Builder für Windows Embedded CE 6.0 installiert ist.

Lektion 1: Anpassen und Konfigurieren eines Board Support Package

Die Entwicklung eines BSP für eine neue Hardwareplattform beginnt normalerweise nach den funktionellen Hardwaretests unter Verwendung eines ROM-Monitors und umfasst das Klonen eines entsprechenden Referenz-BSP sowie das Implementieren eines Boot Loaders und der OAL-Basisfunktionen, um den Kernel zu unterstützen. Das Ziel ist, mit nur geringen Codeanpassungen ein startbares System zu erstellen. Anschließend können Sie Gerätetreiber zum BSP hinzufügen, um integrierte Hardwarekomponenten und Peripheriegeräte zu unterstützen sowie das System durch Implementieren der Energieverwaltung und anderer OS-Features zu erweitern.

Nach Abschluss dieser Lektion können Sie:

- Den Inhalt eines PQOAL-basierenden BSP identifizieren und ermitteln.
- Hardwarespezifische und allgemeine Codebibliotheken identifizieren.
- Ein BSP klonen.
- Einen Boot Loader, OAL und Gerätetreiber anpassen.

Veranschlagte Zeit für die Lektion: 40 Minuten.

Board Support Package - Übersicht

Ein BSP enthält den Quellcode für den Boot Loader, den OAL und die Gerätetreiber für eine Plattform. Zusätzlich zu diesen Komponenten umfasst das BSP auch Build- und Systemkonfigurationsdateien (siehe Abbildung 5.1). Die Konfigurationsdateien sind nicht im Run-Time Image enthalten, obwohl sie als Teil des BSP unter anderem die Quellcodedateien, das Speicherlayout und die Registrierungseinstellungen angeben, die erforderlich sind, um das Run-Time Image zu kompilieren (siehe Kapitel 2).

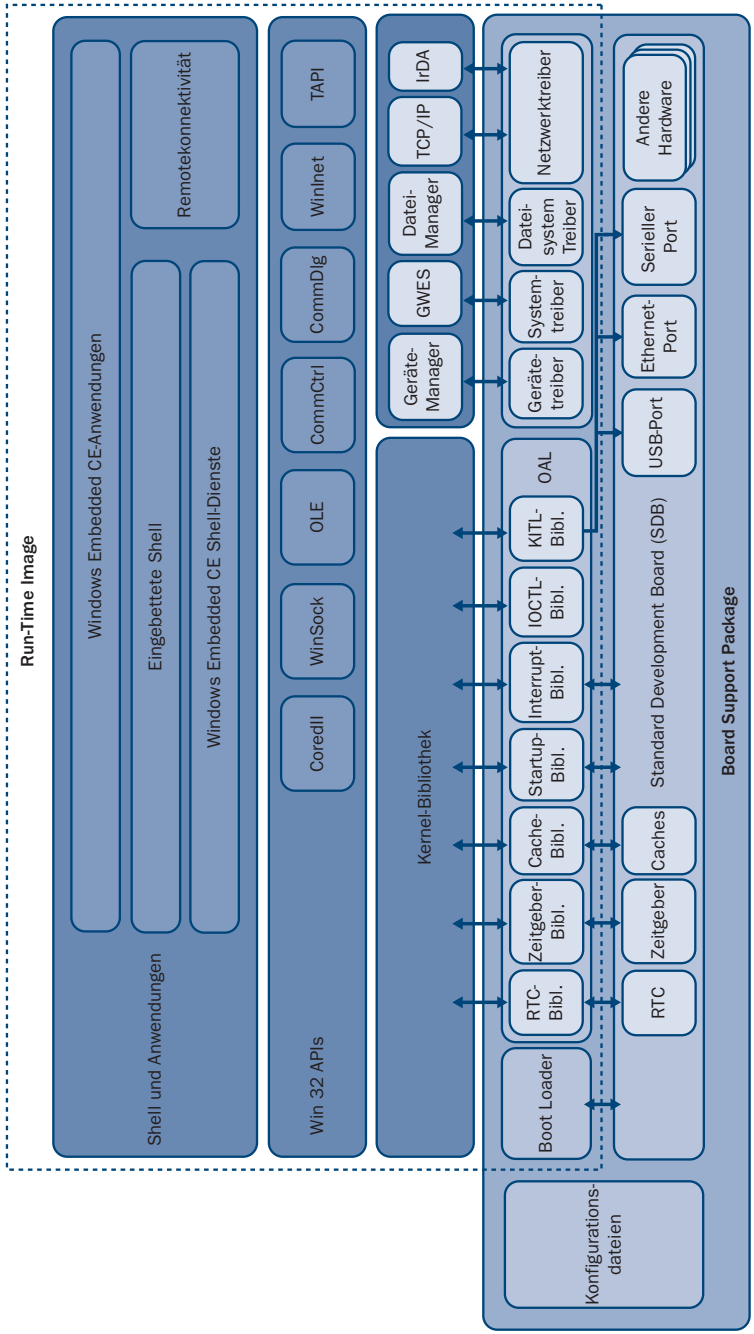


Abbildung 5.1 Komponenten eines BSP in Beziehung zu den übrigen Elementen von Windows Embedded CE 6.0

Ein BSP besteht aus den folgenden Komponenten:

- **Boot Loader** Wird beim Starten oder Zurücksetzen des Geräts ausgeführt. Der Boot Loader ist für das Initialisieren der Hardwareplattform und des Betriebssystems verantwortlich.
- **OEM Adaptation Layer (OAL)** Repräsentiert das Herzstück des BSP und die Schnittstelle zwischen dem Kernel und der Hardware. Da das BSP direkt mit dem Kernel gelinkt ist, ist es in einem CE Run-Time Image Teil des Kernels. Einige der Basiskernelkomponenten hängen für die Hardwareinitialisierung vom OAL ab, beispielsweise für die Interrupt- und Zeitgeberverarbeitung für den Threadscheduler.
- **Gerätetreiber** Verwalten die Funktionen eines Peripheriegeräts und bilden die Schnittstelle zwischen der Gerätehardware und dem Betriebssystem. Windows Embedded CE unterstützt mehrere Gerätearchitekturen basierend auf den vorhandenen Schnittstellen (siehe Kapitel 6 „Entwickeln von Gerätetreibern“).
- **Konfigurationsdateien** Enthalten die zum Steuern des Buildprozesses und für das Design des Betriebssystems erforderlichen Informationen. Typische Konfigurationsdateien in einem BSP sind *Sources-*, *Dirs-*, *Config.bib-*, *Platform.bib-*, *Platform.reg-*, *Platform.db-*, *Platform.dat-* und Katalogdateien (*.pbcxml).

Anpassen eines Board Support Package

Sie können die BSP-Entwicklung beschleunigen, indem Sie ein vorhandenes Referenz-BSP klonen, anstatt ein neues BSP zu erstellen. Auch wenn Sie ein BSP für eine neue Plattform mit einer neuen CPU erstellen müssen, sollten Sie ein BSP basierend auf einer ähnlichen Prozessorarchitektur klonen. Auf diese Art können Sie die Entwicklungszeit für das BSP reduzieren, da der hardwareunabhängige Code des vorhandenen BSPs wiederverwendet wird. Außerdem wird die künftige Migration zu neuen Windows Embedded-Versionen verkürzt. Das Migrieren eines proprietären BSP-Designs ist im allgemeinen schwieriger als das Migrieren eines auf PQOAL basierenden Designs, da das proprietäre BSP die PQOAL-Codesegmente nicht nutzen kann, die Microsoft als Teil einer neuen Betriebssystemversion implizit migriert und testet.

Das Anpassen eines BSP umfasst folgende Schritte:

1. Das Klonen eines Referenz-BSP.
2. Das Implementieren eines Boot Loaders.
3. Das Anpassen der OAL-Funktionen.

4. Das Ändern der Run-Time Image-Konfigurationsdateien.
5. Das Entwickeln der Gerätetreiber.

Klonen eines Referenz-BSP

Platform Builder umfasst einen Wizard zum Klonen eines Referenz-BSP. Dieser Wizard kopiert den gesamten Quellcode des ausgewählten Referenz-BSP in eine neue Ordnerstruktur, damit Sie das BSP für die neue Hardware anpasssen können, ohne dass das Referenz-BSP oder andere BSPs in der *%_WINCEROOT%*-Ordnerhierarchie beeinflusst werden. In Abbildung 5.2 ist das Starten des BSP Cloning Wizard in Microsoft Visual Studio 2005 mit Platform Builder für Windows Embedded CE 6.0 dargestellt.

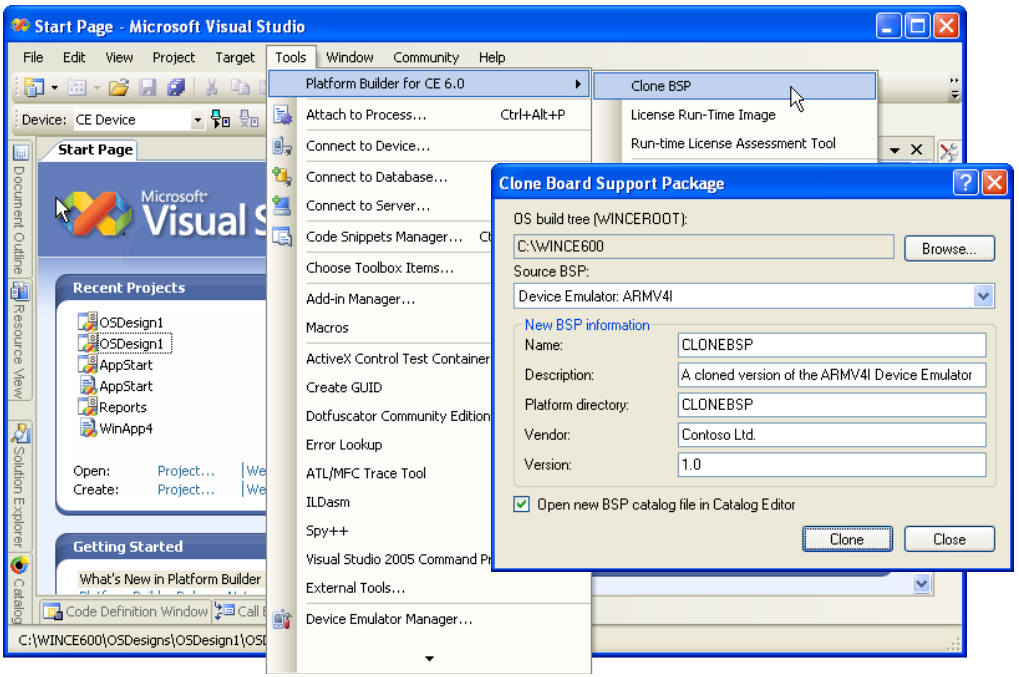


Abbildung 5.2 Klonen eines BSP in Visual Studio 2005

HINWEIS BSP-Namen

Wenn Sie ein BSP klonen, müssen Sie einen neuen Namen angeben. Der für die Plattform ausgewählte Name muss mit dem Namen des Ordners auf der Festplatte übereinstimmen. Wie bereits im vorherigen Kapitel erklärt, basiert das Buildmodul auf einem Befehlszeilenskript, das Leerzeichen in Ordnernamen nicht erkennt. Geben Sie deshalb im BSP-Namen statt einem Leerzeichen einen Unterstrich (_) ein.

BSP-Ordnerstruktur

Um die Wiederverwendbarkeit des Codes sicherzustellen, unterstützen PQOAL-basierte BSPs eine allgemeine Architektur und Ordnerstruktur, die für alle Prozessorfamilien konsistent ist. Aufgrund dieser allgemeinen Architektur können große Quellcodeabschnitte unabhängig von den hardwarespezifischen BSP-Anforderungen wiederverwendet werden. In Abbildung 5.3 ist die typische BSP-Ordnerstruktur dargestellt und in Tabelle 5.1 sind die wichtigsten BSP-Ordner aufgeführt.

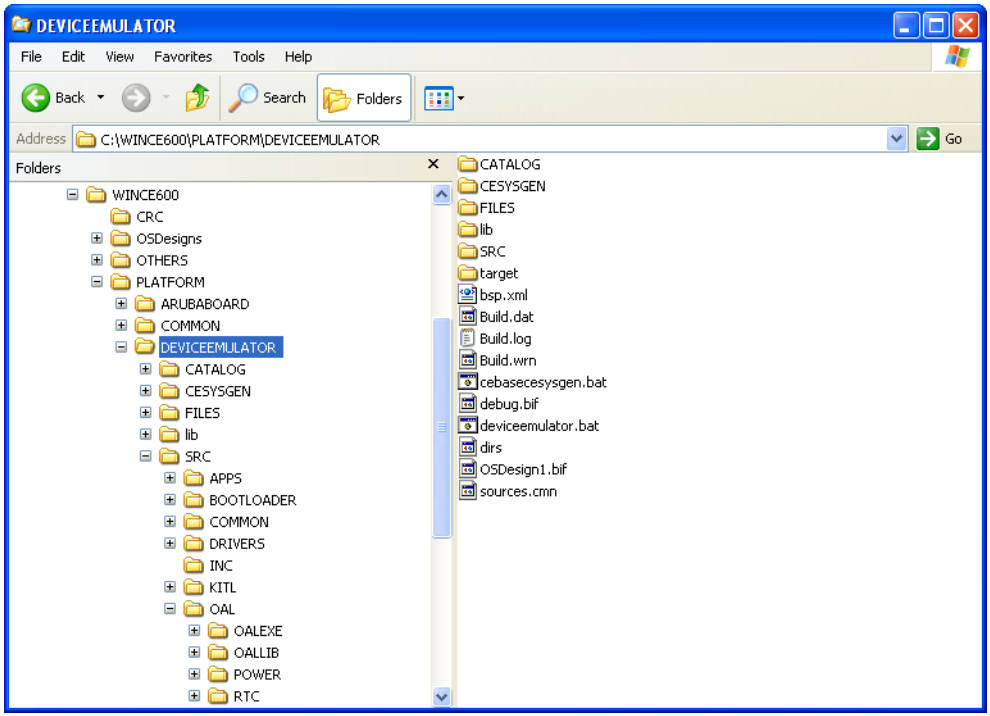


Abbildung 5.3 Ordnerstruktur eines typischen BSP

TIPP %_TARGETPLATROOT%

Mit der Umgebungsvariablen %_TARGETPLATROOT% können Sie im Buildfenster den Pfad des BSP ermitteln, das im aktuellen OS Design verwendet wird (*Release*-Verzeichnis unter Open Build Window im Menü Build in Visual Studio).

Tabelle 5.1 Wichtige BSP-Ordner

Ordner	Beschreibung
Stammordner	Enthält die Konfigurations- und Batchdateien. Die beiden wichtigsten Dateien für Entwickler sind: ■ **Sources.cmn** Enthält die Makrodefinitionen für das BSP. ■ ***<BSP Name>.bat*** Legt die BSP-Standardumgebungsvariablen fest.
CATALOG	Enthält die BSP-Katalogdatei, in der die BSP-Komponenten definiert sind. Diese Datei wird in der OS Designphase verwendet, um BSP-Features hinzuzufügen oder zu entfernen. In Kapitel 1 „Anpassen des OS Designs“ ist das Verwalten der Katalogelemente beschrieben.
CESYSGEN	Enthält die *Makefile*-Datei für das Sysgen-Tool. Dieses Verzeichnis muss beim Konfigurieren eines BSP nicht geändert werden.
FILES	Enthält die Buildkonfigurationsdateien, beispielsweise die .bib-, .reg-, .db- und .dat-Dateien.
SRC	Enthält den plattformspezifischen Quellcode, der entsprechend dem PQOAL-Modell angepasst werden muss, das den Code zwischen plattformspezifischen und allgemeinen Komponenten basierend auf dem CPU-Typ aufteilt.
COMMON	Dieses Unterverzeichnis des Platform-Verzeichnisses enthält den meisten BSP-Quellcode und allgemeine prozessorspezifische Komponenten. Das BSP ist mit den Bibliotheken in diesem Ordner gelinkt, die während des Buildprozesses generiert werden. Die Bibliotheken werden für prozessorbasierte Peripheriegeräte und prozessorspezifische OAL-Komponenten erstellt. Wenn die Hardware die CPU einer unterstützten Prozessorfamilie verwendet, können die Bibliotheken ohne Änderungen übernommen werden.

Plattformspezifischer Quellcode

Der wichtigste plattformspezifische Quellcode, den Sie für das BSP anpassen müssen, ist der Quellcode für den Boot Loader, den OAL und die Gerätetreiber. Der entsprechende Quellcode ist im Src-Ordner in folgenden Unterverzeichnissen gespeichert:

- ***Src\Boot loader*** Enthält den Boot Loader-Code. Wenn der Boot Loader von BLCOMMON und den entsprechenden Bibliotheken abhängig ist, ist nur der hardwarespezifische Teil des Boot Loaders in diesem Verzeichnis gespeichert. Der wiederverwendbare Boot Loader-Code ist im Ordner *Public* (*%_WINCEROOT%\Public\Common\Oak\Drivers\Ethdbg*) verfügbar und als Bibliotheken mit dem BSP-Abschnitt gelinkt. In Kapitel 4 „Debuggen und Testen des Systems“ sind die statischen Bibliotheken beschrieben, die die Boot Loader-Entwicklung unterstützen.
- ***Src\Oal*** Enthält nur den Mindestcode für die Hardwareplattform. Der meiste OAL-Code, der sich in *%_WINCEROOT%\Platform\Common* befindet, ist in hardwareunabhängige Gruppen sowie in Gruppen spezifisch für die Prozessorfamilie, Chipsets und Plattformen aufgeteilt. Diese Codegruppen umfassen die meisten OAL-Funktionen und sind mit den plattformspezifischen Komponenten als Bibliotheken gelinkt.
- ***Src\Common und Src\Drivers*** Enthält den Treiberquellcode, der in unterschiedlichen Kategorien organisiert ist, um die Verwaltung und Portabilität zu unterstützen. Die Kategorien sind normalerweise prozessor- und plattformspezifisch. Die prozessorspezifische Komponente ist im Verzeichnis *Src\Common* gespeichert und muss für die Anpassung an neue Hardware, die auf der gleichen Prozessorfamilie basiert, nicht geändert werden.

Implementieren eines Boot Loaders aus vorhandenen Bibliotheken

Wenn Sie einen Boot Loader für eine neue Plattform anpassen, müssen Sie mehrere Aspekte berücksichtigen:

- Änderungen in der Prozessorarchitektur.
- Position des Boot Loader-Codes auf dem Zielgerät.
- Die Speicherarchitektur der Plattform.
- Die Aufgaben, die während des Startprozesses ausgeführt werden müssen.

- Die für das Herunterladen des Run-Time Images unterstützten Transportmechanismen.
- Zusätzliche Features, die unterstützt werden müssen.

Speicherzuordnungen

Die erste Aufgabe bei der Anpassung bezieht sich auf die Definition der Speicherzuordnungen für den Boot Loader. Die Standard-BSPs in Windows Embedded CE definieren die Speicherkonfiguration in einer .bib-Datei im Boot Loader-Unterverzeichnis, beispielsweise *%_WINCEROOT%\Platform\Arubaboard\Src\Boot loader\Eboot\Eboot.bib*. Sie können die folgende *Eboot.bib*-Beispieldatei an Ihre Anforderungen anpassen.

```
MEMORY

;   Name     Start     Size      Type
;   -------  --------  --------  ----

    ; Reserve some RAM before Eboot.
    ; This memory will be used later.

    DRV_GLB  A0008000  00001000  RESERVED   ; Driver globals; 4 KB is sufficient.

    EBOOT    A0030000  00020000  RAMIMAGE   ; Set aside 128 KB for loader; finalize later.
    RAM      A0050000  00010000  RAM        ; Free RAM; finalize later.

CONFIG
    COMPRESSION=OFF
    PROFILE=OFF
    KERNELFIXUPS=ON

    ; These configuration options cause the .nb0 file to be created.
    ; An .nb0 file may be directly written to flash memory and then
    ; booted. Because the loader is linked to execute from RAM,
    ; the following configuration options
    ; must match the RAMIMAGE section.
    ROMSTART=A0030000
    ROMWIDTH=32
    ROMSIZE=20000

MODULES
;   Name         Path                                          Memory Type
;   -----------  --------------------------------------------- -----------
    nk.exe       $(_TARGETPLATROOT)\target\$(_TGTCPU)\$(WINCEDEBUG)\EBOOT.exe   EBOOT
```

Driver Globals

Sie können die Datei *Eboot.bib* unter anderem zum Reservieren eines Speicherabschnitts für den Boot Loader verwenden, um während des Startprozesses Informationen an das Betriebssystem zu übergeben. Diese Informationen können sich beispielsweise auf den aktuellen Status der initialisierten Hardware, die Kommunikationsfunktionen (wenn der Boot Loader Downloads über das Ethernet unterstützt) sowie die Benutzer- und Systemflags für das Betriebssystem beziehen (z.B. zum Aktivieren von KITL). Um die Kommunikation zu aktivieren, müssen der Boot Loader und das Betriebssystem einen gemeinsamen physischen Speicher verwenden, der als *Driver Globals* bezeichnet wird (*DRV_GLB*). Die *Eboot.bib*-Auflistung umfasst eine *DRV_GLB*-Zuordnung. Die Daten, die der Boot Loader im *DRV_GLB*-Bereich an das Betriebssystem übergibt, müssen einer *BOOT_ARGS*-Struktur entsprechen, die Sie an Ihre Anforderungen anpassen können.

Das folgende Verfahren veranschaulicht die Übergabe der Ethernet- und IP-Konfigurationsinformationen des Boot Loaders über einen *DRV_GLB*-Bereich an das Betriebssystem. Erstellen Sie im Ordner *%_WINCEROOT%\Platform\<BSP Name>\Src\Inc* eine Headerdatei, beispielsweise *Drv_glob.h*, mit folgendem Inhalt:

```
#include <halether.h>

// Debug Ethernet parameters.
typedef struct _ETH_HARDWARE_SETTINGS
{
    EDBG_ADAPTER    Adapter;            // The NIC to communicate with Platform Builder.
    UCHAR           ucEdbgAdapterType;  // Type of debug Ethernet adapter.
    UCHAR           ucEdbgIRQ;          // IRQ line to use for debug Ethernet adapter.
    DWORD           dwEdbgBaseAddr;     // Base I/O address for debug Ethernet adapter.
    DWORD           dwEdbgDebugZone;    // EDBG debug zones to be enabled.

    // Base for creating a device name.
    // This will be combined with the EDBG MAC address
    // to generate a unique device name to identify
    // the device to Platform Builder.
    char szPlatformString[EDBG_MAX_DEV_NAMELEN];

    UCHAR           ucCpuId;            // Type of CPU.
} ETH_HARDWARE_SETTINGS, *PETH_HARDWARE_SETTINGS;


// BootArgs - Parameters passed from the boot loader to the OS.
#define BOOTARG_SIG  0x544F4F42 // "BOOT"

typedef struct BOOT_ARGS
{
    DWORD   dwSig;
```

```
    DWORD   dwLen;              // Total length of BootArgs struct.
    UCHAR   ucLoaderFlags;      // Flags set by boot loader.
    UCHAR   ucEshellFlags;      // Flags from Eshell.
    DWORD   dwEdbgDebugZone;    // Which debug messages are enabled?

    // The following addresses are only valid if LDRFL_JUMPIMG is set and
    // the corresponding bit in ucEshellFlags is set (configured by Eshell, bit
    // definitions in Ethdbg.h).
    EDBG_ADDR EshellHostAddr;   // IP/Ethernet addr and UDP port of host
                                // running Eshell.
    EDBG_ADDR DbgHostAddr;      // IP/Ethernet address and UDP port of host
                                // receiving debug messages.
    EDBG_ADDR CeshHostAddr;     // IP/Ethernet addr and UDP port of host
                                // running Ethernet text shell.
    EDBG_ADDR KdbgHostAddr;     // IP/Ethernet addr and UDP port of host
                                // running kernel debugger.

    ETH_HARDWARE_SETTINGS Edbg; // The debug Ethernet controller.

} BOOT_ARGS, *PBOOT_ARGS;

// Definitions for flags set by the boot loader.
#define    LDRFL_USE_EDBG    0x0001  // Set to attempt to use debug Ethernet.

// The following two flags are only looked at if LDRFL_USE_EDBG is set.
#define    LDRFL_ADDR_VALID  0x0002  // Set if EdbgAddr member is valid.
#define    LDRFL_JUMPIMG     0x0004  // If set, do not communicate with Eshell
                                     // to get configuration information,
                                     // use ucEshellFlags member instead.
typedef struct _DRIVER_GLOBALS
{
    //
    // TODO: Later, fill in this area with shared information between
    // drivers and the OS.
    //
    BOOT_ARGS        bootargs;
} DRIVER_GLOBALS, *PDRIVER_GLOBALS;
```

StartUp-Einsprungspunkt und Main-Funktion

Der *StartUp*-Einsprungspunkt des Boot Loaders muss sich im linearen Speicher an der Adresse befinden, an der die CPU beginnt, den auszuführenden Code abzurufen, da diese Routine die Hardware initialisiert. Wenn die Anpassung auf einem Referenz-BSP für den gleichen Prozessor-Chipset basiert, müssen Sie den meisten Code für die CPU und den Speichercontroller nicht ändern. Wenn Sie jedoch eine andere CPU-Architektur verwenden, müssen Sie die Startroutine anpassen, um folgende Aufgaben auszuführen:

1. Die CPU in den richtigen Modus versetzen.

2. Alle Interrupts deaktivieren.
3. Den Speichercontroller initialisieren.
4. Cache-Speicher, Translation Lookaside Buffers (TLBs) und Memory Management Unit (MMU) konfigurieren.
5. Den Boot Loader für die schnellere Ausführung aus dem Flashspeicher in den RAM kopieren.
6. Zum C-Code in der Main-Funktion wechseln.

Die StartUp-Routine ruft die Main-Funktion des Boot Loaders auf. Wenn der Boot Loader auf BLCOMMON basiert, ruft diese Funktion die Funktion *BootLoaderMain* auf, die den Downloadtransport initialisiert, indem die OEM-Plattformfunktionen aufgerufen werden. Der Vorteil der Standardbibliotheken von Microsoft ist, dass die zum Anpassen eines BSP für eine neue Hardwareplattform erforderlichen Änderungen auf Komponenten basieren und isoliert sowie minimiert sind.

Serielle Debugausgabe

Der nächste Schritt bei der Anpassung des Boot Loaders ist das Initialisieren der seriellen Debugausgabe. Die Debugausgabe ist ein wichtiger Teil des Startprozesses, da diese dem Benutzer das Interagieren mit dem Boot Loader und dem Entwickler das Analysieren der Debugmeldungen ermöglicht (siehe Kapitel 4).

In Tabelle 5.2 sind die OEM-Plattformfunktionen aufgeführt, die die serielle Debugausgabe im Boot Loader unterstützen.

Tabelle 5.2 Serielle Debugausgabefunktionen

Funktion	Beschreibung
OEMDebugInit	Initialisiert die UART auf der Plattform.
OEMWriteDebugString	Schreibt eine Zeichenfolge in die Debug UART.
OEMWriteDebugByte	Schreibt ein Byte in die Debug UART, die von OEMWriteDebugString verwendet wird.
OEMReadDebugByte	Liest ein Byte aus der Debug UART.

Initialisieren der Plattform

Nachdem die CPU und die serielle Debugausgabe initialisiert wurden, können Sie die anderen Aufgaben für die Hardwareinitialisierung ausführen. Die *OEMPlatformInit*-Routine führt folgende Aufgaben aus:

- Initialisieren der Echtzeit-Uhr.
- Konfigurieren des externen Speichers, beispielsweise des Flashspeichers.
- Initialisieren des Netzwerkcontrollers.

Download über Ethernet

Wenn die Hardwareplattform einen Netzwerkcontroller umfasst, kann der Boot Loader das Run-Time Image über Ethernet herunterladen. In Tabelle 5.3 sind die Funktionen aufgeführt, die Sie implementieren müssen, um die auf Ethernet basierende Kommunikation zu unterstützen.

Tabelle 5.3 Ethernet-Unterstützungsfunktionen

Funktion	Beschreibung
OEMReadData	Liest die Daten aus dem Transport für den Download.
OEMEthGetFrame	Liest die Daten unter Verwendung des Funktionszeigers *pfnEDbgGetFrame* aus der NIC.
OEMEthSendFrame	Übergibt die Daten unter Verwendung des Funktionszeigers *pfnEDbfSendFrame* an die NIC.
OEMEthGetSecs	Gibt die Anzahl der vergangenen Sekunden relativ zur festgelegten Zeitdauer zurück.

Die Ethernet-Unterstützungsfunktionen verwenden Callbacks in Netzwerkcontroller-spezifischen Routinen. Das heißt, Sie müssen zusätzliche Routinen implementieren und in der Funktion *OEMPlatformInit* die entsprechenden Funktionszeiger konfigurieren, wenn Sie einen anderen Netzwerkcontroller unterstützen möchten:

```
cAdaptType=pBootArgs->ucEdbgAdapterType;

// Set up EDBG driver callbacks based on
// Ethernet controller type.
switch (cAdaptType)
{
```

```
case EDBG_ADAPTER_NE2000:
    pfnEDbgInit           = NE2000Init;
    pfnEDbgInitDMABuffer  = NULL;
    pfnEDbgGetFrame       = NE2000GetFrame;
    pfnEDbgSendFrame      = NE2000SendFrame;
    break;

case EDBG_ADAPTER_DP83815:
    pfnEDbgInit   = DP83815Init;
    pfnEDbgInitDMABuffer  = DP83815InitDMABuffer;
    pfnEDbgGetFrame       = DP83815GetFrame;
    ...
}
```

Flashspeicher-Unterstützung

Nachdem Sie die Funktionen für die Netzwerkkommunikation implementiert haben, müssen Sie den Boot Loader aktivieren, um das Run-Time Image auf die neue Hardwareplattform herunterzuladen.

Sie können das Run-Time Image auch im Flashspeicher speichern. In Tabelle 5.4 sind die Funktionen für die Download- und Flashspeicherunterstützung aufgeführt, die Sie implementieren müssen, wenn der Boot Loader des Referenz-BSPs diese Features nicht bereits unterstützt.

Tabelle 5.4 Funktionen für die Download- und Flashspeicherunterstützung

Funktion	Beschreibung
OEMPreDownload	Konfiguriert das erforderliche Downloadprotokoll, das von Platform Builder unterstützt wird.
OEMIsFlashAddr	Überprüft, ob das Image für Flash oder RAM konfiguriert ist.
OEMMapMemAddr	Ordnet das Image vorübergehend dem RAM zu.
OEMStartEraseFlash	Bereitet das teilweise Löschen des Flashspeichers für das OS Image vor.
OEMContinueEraseFlash	Setzt das Löschen des Flashspeichers basierend auf dem Downloadstatus fort.
OEMFinishEraseFlash	Beendet das Löschen des Flashspeichers, nachdem der Download abgeschlossen ist.
OEMWriteFlash	Schreibt das OS Image in den Flashspeicher.

Benutzerinteraktion

Ein Boot Loader ermöglicht die Benutzerinteraktion basierend auf einem Menü mit Startoptionen für die Plattform. Dies ist während des Entwicklungsprozesses sowie für die Verwaltung und Softwareupdates nützlich. In Abbildung 5.4 ist ein Boot Loader-Standardmenü dargestellt. Die Datei *Menu.c* im Verzeichnis *Src\Boot loader\Eboot* eines Referenz-BSP oder im Ordner *%_WINCEROOT%\Platform \Common\Src\Common\Boot\Blmenu* enthält Quellcodebeispiele.

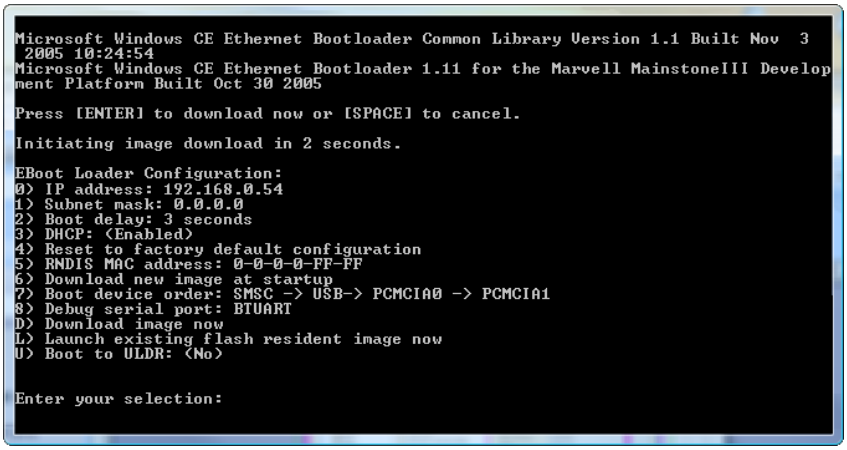


Abbildung 5.4 Ein Boot Loader-Menü

Zusätzliche Features

Neben den Basisfunktionen können Sie weitere Features hinzufügen, beispielsweise eine Download-Statusanzeige und die Unterstützung für das Herunterladen mehrerer .bin-Dateien während der gleichen Downloadsitzung oder ausschließlich vertrauenswürdiger Images. Außerdem können Sie die Unterstützung für das Herunterladen von Run-Time Images direkt aus Platform Builder implementieren. Hierzu muss der Boot Loader ein *BOOTME*-Paket vorbereiten, das Informationen über das Zielgerät enthält, und das Paket über den zugrundeliegenden Transport senden. Wenn Ethernet als Transport verwendet wird, wird das Paket über das Netzwerk übertragen. Die Bibliotheken von Microsoft unterstützen diese Features, die Sie an Ihre Anforderungen anpassen können.

HINWEIS OEM Boot Loader-Funktionen

Weitere Informationen zu den erforderlichen und optionalen Boot Loader-Funktionen und den Boot Loader-Strukturen finden Sie im Abschnitt „Boot Loader Reference" in der Windows Embedded CE 6.0-Dokumentation auf der Microsoft MSDN®-Website unter *http://msdn2.microsoft.com/en-us/library/aa908395.aspx*.

Anpassen eines OAL

Die plattformspezifische Konfiguration des OAL ist ein wichtiger Aspekt der BSP-Anpassung. Wenn die neue Plattform eine nicht unterstützte CPU verwendet, müssen Sie den OAL-Code ändern, um die neue Prozessorarchitektur zu unterstützen. Sollte die neue Hardware jedoch der Plattform des Referenz-BSP ähnlich sein, können Sie möglicherweise den meisten vorhandenen Code wiederverwenden.

OEM-Adresstabelle

Der Kernel, der spezielle Aufgaben ausführt, beispielsweise das Initialisieren des virtuellen Speichers, darf nicht von einem Boot Loader abhängig sein. Ansonsten hängt das Betriebssystem von einem Boot Loader ab und das Run-Time Image kann nicht direkt geladen werden. Um über die MMU (Memory Management Unit) virtuelle und physische Adressen zuzuordnen, muss der Kernel das Speicherlayout der zugrundeliegenden Hardwareplattform ermitteln können. Um diese Informationen abzurufen, verwendet der Kernel eine importierte Tabelle namens *OEMAddressTable* (or *g_oalAddressTable*), die statische virtuelle Speicherbereiche definiert. Der OAL umfasst eine Deklaration der Tabelle *OEMAddressTable* als schreibgeschützten Abschnitt, der vom Kernel gelesen wird. Der Kernel erstellt die entsprechenden virtuellen Speicherzuordnungstabellen und wechselt zur virtuellen Speicheradresse, an der der Kernel den Code ausführen kann. Der Kernel kann die physische Adresse der Tabelle *OEMAddressTable* im linearen Speicher basierend auf den Adressinformationen im Run-Time Image bestimmen.

Sie müssen die an der Speicherkonfiguration einer neuen Hardwareplattform vorgenommenen Änderungen in der Tabelle *OEMAddressTable* angeben. Der folgende Beispielcode veranschaulicht die Deklaration des *OEMAddressTable*-Abschnitts.

```
;------------------------------------------------------------------
public    _OEMAddressTable

    _OEMAddressTable:

    ; OEMAddressTable defines the mapping between Physical and Virtual Address
```

```
; o MUST be in a READONLY Section
; o First Entry MUST be RAM, mapping from 0x80000000 -> 0x00000000
; o each entry is of the format ( VA, PA, cbSize )
; o cbSize must be multiple of 4M
; o last entry must be (0, 0, 0)
; o must have at least one non-zero entry
; RAM 0x80000000 -> 0x00000000, size 64M
dd  80000000h,    0,   04000000h
; FLASH and other memory, if any
; dd  FlashVA,      FlashPA,    FlashSize
; Last entry, all zeros
dd  0   0   0
```

StartUp-Einsprungspunkt

Ähnlich wie der Boot Loader enthält der OAL einen StartUp-Einsprungspunkt für den Boot Loader oder das System, um die Kernelausführung zu starten und das System zu initialisieren. Beispielsweise ist der Assemblycode, der den Prozessor in den korrekten Status versetzt, normalerweise mit dem in Boot Loader verwendeten Code identisch. Die Verwendung des gleichen Codes im Boot Loader und im OAL ist üblich, um die Codeduplizierung im BSP zu verringern. Es wird jedoch nicht der gesamte Code zweimal ausgeführt. Auf Hardwareplattformen, die von einem Boot Loader gestartet werden, springt StartUp direkt zur *KernelStart*-Funktion, da der Boot Loader die ursprüngliche Initialisierung bereits ausgeführt hat.

Die *KernelStart*-Funktion initialisiert die Speicherzuordnungstabellen und lädt die Kernelbibliothek, um den Microsoft-Kernelcode auszuführen. Der Microsoft-Kernelcode ruft die Funktion *OEMInitGlobals* auf, um einen Zeiger auf die statische *NKGLOBALS*-Struktur an den OAL zu übergeben und einen Zeiger auf eine *OEMGLOBALS*-Struktur in Form eines Rückgabewerts vom OAL abzurufen. *NKGLOBALS* enthält Zeiger auf alle Funktionen und Variablen, die von KITL verwendet werden, sowie den Microsoft-Kernelcode. *OEMGLOBALS* umfasst Zeiger auf alle Funktionen und Variablen, die für das BSP im OAL implementiert sind. Durch den Austausch der Zeiger auf die globalen Strukturen können *Oal.exe* und *Kernel.dll* auf die Funktionen und Daten der jeweils anderen Datei zugreifen und die architektur- sowie die plattformspezifischen Startvorgänge fortsetzen.

Die architekturspezifischen Vorgänge umfassen das Konfigurieren von Seitentabellen und Cacheinformationen, das Übertragen von TLBs (Transition Lookaside Buffers), das Initialisieren architekturspezifischer Busse und Komponenten, das Konfigurieren des Interlocked API-Codes, das Laden von KITL, um die Kernelkommunikation für das Debuggen zu unterstützen, und das Initialisieren der Kerneldebugausgabe. Der

Kernel ruft die *OEMInit*-Funktion über den Funktionszeiger in der *OEMGLOBALS*-Struktur auf, um die plattformspezifische Initialisierung auszuführen.

In Tabelle 5.5 sind die plattformspezifischen Funktionen aufgeführt, die *Kernel.dll* aufruft und im BSP ändert, um Windows Embedded CE auf einer neuen Hardwareplattform auszuführen.

Tabelle 5.5 Kernel-Startfunktionen

Funktion	Beschreibung
OEMInitGlobals	Tauscht globale Zeiger zwischen *Oal.exe* und *Kernel.dll* aus.
OEMInit	Initialisiert die Hardwareschnittstellen für die Plattform.
OEMGetExtensionDRAM	Stellt gegebenenfalls Informationen über zusätzlichen RAM bereit.
OEMGetRealTime	Ruft die Zeit aus RTC ab.
OEMSetAlarmTime	Legt den RTC-Alarm fest.
OEMSetRealTime	Legt die Zeit im RTC fest.
OEMIdle	Versetzt die CPU in den Leerlaufstatus, wenn keine Threads ausgeführt werden.
OEMInterruptDisable	Deaktiviert bestimmte Hardwareinterrupts.
OEMInterruptEnable	Aktiviert bestimmte Hardwareinterrupts.
OEMInterruptDone	Zeigt den Abschluss der Interruptverarbeitung an.
OEMInterruptHandler	Verarbeitet Interrupts (nicht für SHx-Prozessoren).
OEMInterruptHandler	Verarbeitet FIQ (spezifisch für ARM-Prozessoren).
OEMIoControl	IO-Steuerungscode für OEM-Informationen.
OEMNMI	Unterstützt einen nicht maskierbaren Interrupt (spezifisch für SHx-Prozessoren).
OEMNMIHandler	Handler für nicht maskierbare Interrupts (spezifisch für SHx-Prozessoren).

Tabelle 5.5 Kernel-Startfunktionen (Fortsetzung)

Funktion	Beschreibung
OEMPowerOff	Versetzt die CPU in den Standby-Modus und führt die Energiesparvorgänge aus.

Kernel Independent Transport Layer (KITL)

Die *OEMInit*-Funktion ist die OAL-Hauptroutine, die boardspezifische Peripheriegeräte initialisiert, die Kernelvariablen konfiguriert und KITL startet, indem KITL IOCTL an den Kernel übergeben wird. Wenn Sie KITL zum Run-Time Image hinzugefügt und aktiviert haben, startet der Kernel KITL für das Debuggen über verschiedene Transportlayer (siehe Kapitel 4 „Debuggen und Testen des Systems“).

In Tabelle 5.6 sind die Funktionen aufgeführt, die OAL einbeziehen muss, um die KITL-Unterstützung auf einer neuen Plattform zu aktivieren.

Tabelle 5.6 KITL-Unterstützungsfunktionen

Funktion	Beschreibung
OEMKitlInit	Initialisiert KITL.
OEMKitlGetSecs	Gibt die aktuelle Zeit in Sekunden zurück.
TransportDecode	Decodiert die empfangenen Frames.
TransportEnableInt	Aktiviert oder deaktiviert den KITL-Interrupt.
TransportEncode	Codiert die Daten entsprechend der für den Transport erforderlichen Framestruktur.
TransportGetDevCfg	Ruft die KITL-Transportkonfiguration des Geräts ab.
TransportReceive	Empfängt einen Frame vom Transport.
TransportSend	Sendet einen Frame über den Transport.
KitlInit	Initialisiert das KITL-System.
KitlSendRawData	Sendet Rohdaten über den Transport und umgeht das Protokoll.

Tabelle 5.6 KITL-Unterstützungsfunktionen (Fortsetzung)

Funktion	Beschreibung
KitlSetTimerCallback	Registriert einen Callback, der nach einer bestimmten Zeitdauer aufgerufen wird.
KitlStopTimerCallback	Deaktiviert einen Zeitgeber, der von der Routine verwendet wird.

Unterstützung des Profilzeitgebers

Der OAL des Betriebssystems überprüft die Leistung des Systems und unterstützt die Leistungsoptimierung. Sie können das Tool ILTiming (Interrupt Latency Timing) verwenden, um die zum Starten einer Interrupt Service Routine (ISR) erforderliche Zeitdauer nach einem Interrupt (ISR-Latenz) und die Zeitdauer zwischen dem Beenden der ISR und dem Starten des Interrupt Service Thread (IST) (IST-Latenz) zu messen. Dieses Tool erfordert jedoch einen Systemhardware-Tickzeitgeber oder einen alternativen Zeitgeber mit hoher Auflösung, der nicht auf allen Hardwareplattformen verfügbar ist. Wenn die neue Hardwareplattform einen Hardwarezeitgeber mit hoher Auflösung unterstützt, können Sie ILTiming und ähnliche Tools unterstützen, indem Sie die in Tabelle 5.7 aufgeführten Funktionen implementieren.

Tabelle 5.7 Profilzeitgeber-Unterstützungsfunktionen

Funktion	Beschreibung
OEMProfileTimerEnable	Aktiviert einen Profiler-Zeitgeber.
OEMProfileTimerDisable	Deaktiviert einen Profiler-Zeitgeber.

HINWEIS Threadplanung und Interruptverarbeitung

Der OAL muss die Interruptverarbeitung und den Kernelscheduler unterstützen. Der Scheduler ist unabhängig vom Prozessortyp, aber die Interruptverarbeitung muss für die unterschiedlichen Prozessoren optimiert werden.

Integrieren neuer Gerätetreiber

Neben den Basissystemfunktionen enthält das BSP die Gerätetreiber für Peripheriegeräte. Bei den Peripheriegeräten kann es sich um Komponenten des Prozessorchips oder externe Komponenten handeln. Obwohl diese Komponenten

vom Prozessor getrennt sind, sind sie ein integraler Bestandteil der Hardwareplattform.

Verzeichnisse für den Gerätetreibercode

In Tabelle 5.8 sind die Quellcodeverzeichnisse für die Gerätetreiber entsprechend dem PQOAL-Modell aufgeführt. Wenn das BSP auf dem gleichen Prozessor wie das Referenz-BSP basiert, müssen Sie für die Anpassung der Gerätetreiber den Quellcode im Ordner *%TGTPLATROOT%* ändern. Sie können auch neue Treiber zum BSP hinzufügen, wenn die neue Plattform Peripheriegeräte umfasst, die in der Referenzplattform nicht vorhanden sind. Weitere Informationen zum Entwickeln von Gerätetreibern finden Sie in Kapitel 6 „Entwickeln von Gerätetreibern."

Tabelle 5.8 Quellcodeordner für Gerätetreiber

Ordner	Beschreibung
%_WINCEROOT%\Platform\%_TGTPLAT%	Enthält die plattformunabhängigen Treiber.
%_WINCEROOT%\Platform\Common\Src\Soc	Enthält Treiber für prozessoreigene Peripheriegeräte.
%_WINCEROOT%\Public\Common\Oak\Drivers	Enthält Treiber für nicht systemeigene Peripheriegeräte, einschließlich externe Controller.

Ändern der Konfigurationsdateien

Wenn Sie das BSP aus einem vorhandenen BSP geklont haben, sind bereits alle Konfigurationsdateien vorhanden. Sie müssen jedoch das Speicherlayout in der Datei *Config.bib* überprüfen (siehe Lektion 2). Die anderen Konfigurationsdateien müssen nur geändert werden, wenn Sie neue Treiber hinzufügen oder Komponenten im BSP geändert haben (siehe Kapitel 2 „Erstellen und Bereitstellen eines Run-Time Images").

Zusammenfassung

Es ist von Vorteil den BSP-Entwicklungsprozess mit dem Klonen eines geeigneten Referenz-BSP zu beginnen. Idealerweise sollte das BSP auf der gleichen oder einer ähnlich Hardwareplattform basieren, um die getesteten und bewährten Produktionsfeatures zu nutzen. Windows Embedded CE umfasst eine PQOAL-Architektur und Platform Builder-Tools, die den Klonprozess unterstützen. Das Ziel ist, mit minimalen Anpassungen ein startbares System zu erstellen und zusätzliche Features hinzuzufügen sowie gegebenenfalls Peripheriegeräte zu unterstützen.

Die BSP-Komponente, die zuerst angepasst werden muss, ist der Boot Loader, der die Hardwareplattform initialisiert und die Ausführung an den Kernel übergibt. Die nächste Komponente ist der OAL, der den plattformspezifischen Code enthält, den der Kernel für die Hardwareinitialisierung, die Interruptverarbeitung, die Zeitgeberverarbeitung für den Threadscheduler, KITL und die Kerneldebugausgabe benötigt. Außerdem müssen Sie die Gerätetreiber für Peripheriegeräte sowie die Konfigurationsdateien anpassen, die den Buildprozess steuern, das Speicherlayout bestimmen und die Systemkonfigurationseinstellungen festlegen. Wenn die BSP-Anpassung auf einem Referenz-BSP für die gleiche Prozessorarchitektur basiert, müssen Sie den meisten Code für die CPU und den Speichercontroller nicht ändern. Sie müssen nur die plattformspezifischen Codesegmente für die Hardware bearbeiten, anstatt die für das BSP erforderlichen Konfigurationseinstellungen vorzunehmen.

Lektion 2: Konfigurieren der Speicherzuordnung eines BSP

Die Speicherverwaltung in Windows Embedded CE unterscheidet sich wesentlich von früheren Versionen. In früheren Versionen haben alle Prozesse den gleichen 4 GB-Adressbereich verwendet. In CE 6.0 besitzt jeder Prozess einen separaten Adressbereich. Das neue Verwaltungssystem für virtuellen Speicher ermöglicht CE 6.0 das Ausführen von bis zu 32.000 Prozessen im Gegensatz zu den 32 Prozessen in früheren Versionen. In dieser Lektion werden die neue Speicherarchitektur und Verwaltung ausführlich beschrieben, damit Sie die virtuellen Speicherbereiche den korrekten physischen Speicheradressen zuordnen können.

Nach Abschluss dieser Lektion können Sie:

- Die Verwaltung des virtuellen Speichers in Windows Embedded CE beschreiben.
- Statische Speicherzuordnungen für eine Hardwareplattform konfigurieren.
- Nicht zusammenhängenden physischen Speicher zum virtuellen Speicher zuordnen.
- Ressourcen zwischen OAL und Gerätetreibern freigeben.

Veranschlagte Zeit für die Lektion: 15 Minuten.

Systemspeicherzuordnung

Windows Embedded CE verwendet ein in Seiten aufgeteiltes Speicherverwaltungssystem mit einem virtuellen 32-Bit-Adressbereich, der dem physischen Speicher unter Verwendung der MMU zugeordnet wird. Mit 32 Bits kann das System insgesamt 4 GB virtuellen Speicher verwenden, der von CE 6.0 in die folgenden beiden Bereiche aufgeteilt wird (siehe Abbildung 5.5):

- **Kernelbereich** Der Kernelbereich in den oberen 2 GB des virtuellen Speichers wird von allen Anwendungsprozessen auf dem Zielgerät gemeinsam genutzt.
- **Benutzerbereich** Der Benutzerbereich in den niedrigen 2 GB des virtuellen Speichers wird ausschließlich von einzelnen Prozessen verwendet. Jedem Prozess ist ein separater Adressbereich zugeordnet. Der Kernel verwaltet die Zuordnung des Prozessadressbereichs bei einem Prozesswechsel. Prozesse können nicht direkt auf den Kerneladressbereich zugreifen.

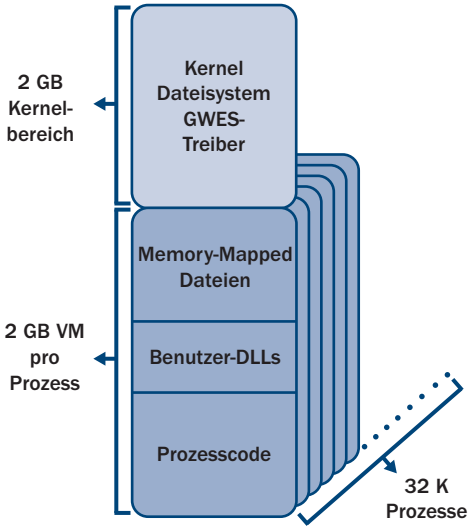


Abbildung 5.5 Virtueller Speicher in Windows Embedded CE 6.0

Kerneladressbereich

Windows Embedded CE 6.0 teilt den Kerneladressbereich für bestimmte Zwecke in mehrere Bereiche auf (siehe Abbildung 5.6). Die niedrigen beiden 512 MB Bereiche ordnen den physischen Speicher statisch in virtuellen zwischengespeicherten („cached“) und nicht zwischengespeicherten („non-cached“) Speicher zu. Die mittleren beiden Bereiche für die Kernel XIP DLLs (eXecute In Place) und den Objektspeicher sind wichtig für das OS Design. Der übrige Bereich ist für die Kernelmodule und die CPU bestimmt.

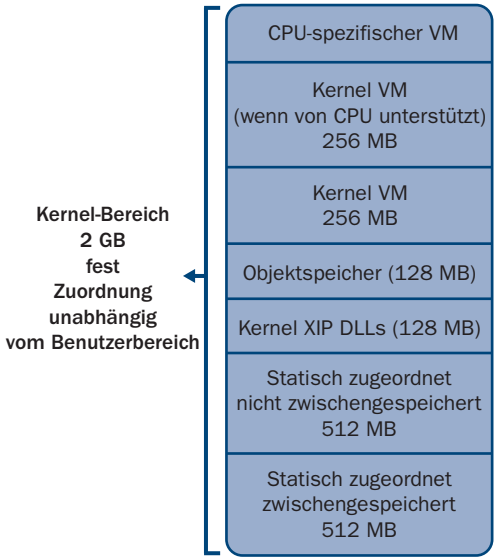


Abbildung 5.6 Kernelbereich in Windows Embedded CE 6.0

In Tabelle 5.9 sind die virtuellen Speicherbereiche für den Kernel mit den Start- und Endadressen aufgeführt.

Tabelle 5.9 Kernelspeicherbereiche

Startadresse	Endadresse	Beschreibung
0xF0000000	0xFFFFFFFF	Wird für CPU-spezifisches Systemtrap- und Kerneldatenseiten verwendet.
0xE0000000	0xEFFFFFFF	Virtuelles CPU-abhängiges Kernelmodul (nicht für SHx verfügbar).
0xD0000000	0xDFFFFFFF	Wird für alle Kernelmodusmodule des Betriebssystems verwendet.
0xC8000000	0xCFFFFFFF	Objektspeicher für das RAM-Dateisystem, die Datenbank und die Registrierung.
0xC0000000	0xC7FFFFFF	XIP DLLs.
0xA0000000	0xBFFFFFFF	Non-cached-Zuordnung des physischen Speichers.
0x80000000	0x9FFFFFFF	Cached-Zuordnung des physischen Speichers.

Prozessadressbereich

Der Prozessadressbereich, der im Bereich von 0x00000000 bis 0x7FFFFFFF liegt, ist in einen CPU-abhängigen Kerneldatenbereich, vier Hauptprozessbereiche und einen 1 MB Puffer zwischen dem Benutzer- und Kernelbereich aufgeteilt. In Abbildung 5.7 sind die Hauptbereiche dargestellt. Der erste 1 GB große Prozessbereich ist für den Anwendungscode und Daten bestimmt. Der zweite 512 GB große Prozessbereich ist für die DLLs und schreibgeschützten Daten bestimmt. Die nächsten beiden Bereiche, die 256 MB und 255 MB groß sind, sind für Memory-Mapped-Objekte und den gemeinsam genutzten („shared“) Systemheap reserviert. Der gemeinsam genutzte („shared“) Systemheap kann vom Anwendungsprozess nur gelesen, aber vom Kernel gelesen und geschrieben werden.

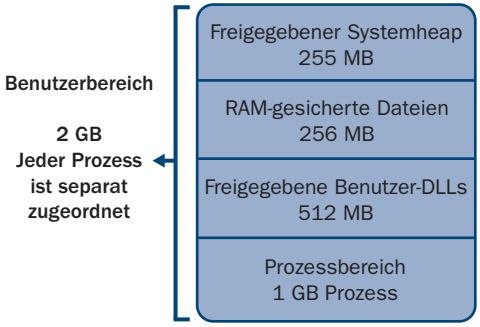


Abbildung 5.7 Prozessbereich in Windows Embedded CE 6.0

In Tabelle 5.10 sind die virtuellen Speicherbereiche im Benutzerbereich mit den Start- und Endadressen aufgeführt.

Tabelle 5.10 Prozessspeicherbereiche

Startadresse	Endadresse	Beschreibung
0x7FF00000	0x7FFFFFFF	Nicht zugeordneter Puffer zwischen dem Benutzer- und Kernelbereich.
0x70000000	0x7FEFFFFF	Shared Heap zwischen dem Kernel und den Prozessen.

Tabelle 5.10 Prozessspeicherbereiche (Fortsetzung)

Startadresse	Endadresse	Beschreibung
0x60000000	0x6FFFFFFF	Memory-Mapped- Dateiobjekte, die keiner physischen Datei entsprechen und hauptsächlich für die Abwärtskompatibilität mit Anwendungen bestimmt sind, die im RAM gesicherte Zuordnungsdateien für die prozessübergreifende Kommunikation verwenden.
0x40000000	0x5FFFFFFF	In den Prozess geladene DLLs und schreibgeschützte Daten.
0x00010000	0x3FFFFFFF	Anwendungscode und Daten.
0x00000000	0x00010000	CPU-abhängige Benutzerkerneldaten (schreibgeschützt für Benutzerprozesse).

Memory Management Unit

Windows Embedded CE 6.0 erfordert, dass der Prozessor eine Methode für die Speicherzuordnung bereitstellt, um physischen Speicher dem virtuellem Speicher zuzuordnen (bis zu 512 MB zugeordneter physischer Speicher). In Abbildung 5.8 sind 32 MB Flashspeicher und 64 MB RAM in cached und non-cached statische Kernelbereiche zugeordnet. Auf ARM- und x86-basierten Plattformen hängt die Speicherzuordnung von einer benutzerdefinierten OEMAddressTable-Tabelle ab. Auf SHx- und MIPS-basierten Plattformen wird die Zuordnung direkt von der CPU definiert. Die MMU (Memory Management Unit) ist für die Zuordnung der physischen in virtuelle Adressen verantwortlich.

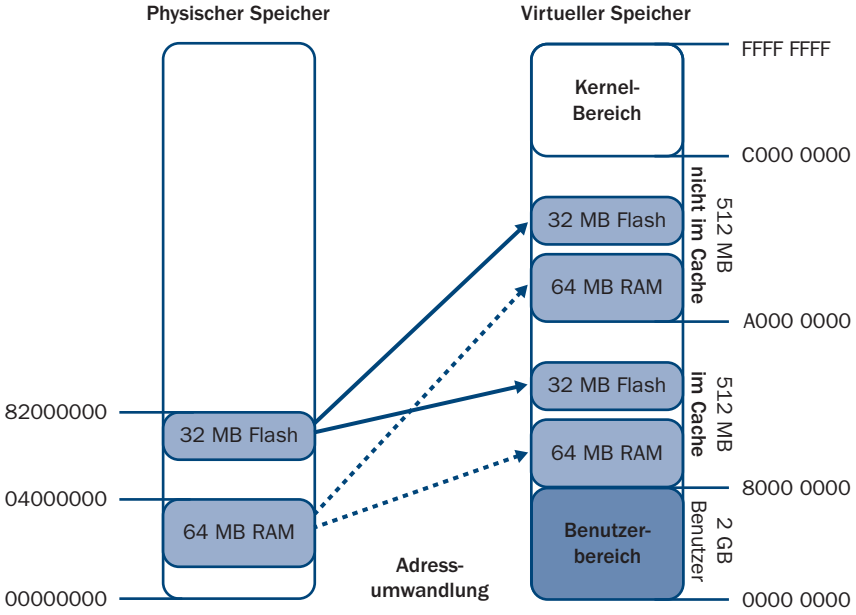


Abbildung 5.8 Zuordnung des physischen Speichers in virtuellen Speicher

HINWEIS MMU-Initialisierung

Der Kernel initialisiert die MMU und erstellt die erforderlichen Seitentabellen während des Systemstarts. Der prozessorspezifische Teil des Kernels hängt von der Architektur der Hardwareplattform ab. Weitere Informationen zur Implementierung finden Sie im privaten Windows Embedded CE-Code in den Unterverzeichnissen der Prozessortypen unter *%_PRIVATEROOT%\Winceos\Coreos\Kernel*.

Statisch zugeordnete virtuelle Adressen

Die in Abbildung 5.8 dargestellten virtuellen Speicherbereiche sind statisch zugeordnete virtuelle Adressen, die zur Startzeit definiert werden und die Zuordnung nicht ändern. Statisch zugeordnete virtuelle Adressen sind im Kernelmodus immer verfügbar. Beachten Sie jedoch, dass Windows Embedded CE die statische Zuordnung zur Laufzeit mittels *CreateStaticMapping* and *NKCreateStaticMapping* APIs unterstützt. Diese Funktionen geben eine nicht zwischengespeicherte („non-cached“) virtuelle Adresse zurück, die der entsprechenden physischen Adresse zugeordnet ist.

Dynamisch zugeordnete virtuelle Adressen

Der Kernel kann die Zuordnung der physischen zu virtuellen Adressen dynamisch verwalten. Diese Methode wird für nicht statische Zuordnungen verwendet. Ein Treiber oder eine DLL, die über *LoadKernelLibrary* in den Kernel geladen werden, können im Kerneladressbereich einen virtuellen Speicherbereich reservieren, indem die Funktion *VirtualAlloc* aufgerufen wird. Die virtuelle Adresse wird einer physischen Adresse zugeordnet, indem die Funktion *VirtualCopy* aufgerufen wird, um einen neuen Seitentabelleneintrag zu erstellen. Diese Methode ordnet den Registrierungen oder Framepuffern von Peripheriegeräten virtuelle Adressen zu, um E/A-Vorgänge auszuführen. Wenn der zugeordnete Puffer nicht mehr benötigt wird, kann der Gerätetreiber oder die DLL die Funktion *VirtualFree* aufrufen, um den Seitentabelleneintrag zu entfernen und den reservierten virtuellen Speicher freizugeben.

Speicherzuordnung und das BSP

Sie müssen die folgenden beiden Elemente anpassen, um die Informationen über die statischen Speicherzuordnungen in einem BSP einzubeziehen:

- ***Config.bib*** Stellt Informationen zur beabsichtigten Verwendung der verschiedenen Speicherbereiche auf der Plattform bereit. Beispielsweise können Sie angeben, wieviel Speicher für das Betriebssystem, als RAM und für bestimmte Zwecke verfügbar ist.
- ***OEMAddressTable*** Stellt Informationen zum Speicherlayout der Plattform bereit (siehe Lektion 1). Der in der Datei *Config.bib* angegebene Speicher sollte auch in der Tabelle *OEMAddressTable* zugeordnet werden.

Zuordnen nicht zusammenhängender physischer Speicherblöcke

Wie in Kapitel 2 „Erstellen und Bereitstellen eines Run-Time Images“ erklärt, müssen Sie im RAMIMAGE-Speicherbereich einen zusammenhängenden Bereich für das Betriebssystem im MEMORY-Abschnitt der Datei *Config.bib* definieren. Das System verwendet diese Definition, um das Kernelimage und die in den Abschnitten MODULES und FILES angegebenen Module zu laden. Sie können nicht mehrere RAMIMAGE-Bereiche definieren. OAL kann jedoch den RAMIMAGE-Bereich erweitern und zusätzliche nicht zusammenhängende Speicherblöcke zur Laufzeit bereitstellen.

In Tabelle 5.11 sind die Variablen und Funktionen zum Erweitern des RAM-Bereichs aufgeführt.

Tabelle 5.11 Variablen und Funktionen zum Erweitern des RAM-Bereichs

Variable/Funktion	Beschreibung
MainMemoryEndAddress	Diese Variable zeigt das Ende des RAM-Bereichs an. Der Kernel legt die Variable ursprünglich auf die in der Datei *Config.bib* für das Betriebssystem reservierte Größe fest. Die OAL-Funktion *OEMInit* aktualisiert die Variable, wenn zusätzlicher zusammenhängender Speicher verfügbar ist.
OEMGetExtensionDRAM	Der OAL verwendet die Funktion, um einen zusätzlichen nicht zusammenhängenden Speicherbereich für den Kernel anzuzeigen. *OEMGetExtensionDRAM* gibt die Startadresse und die Länge des zweiten Speicherbereichs zurück.
pNKEnumExtensionDRAM	Der OAL verwendet diese Funktion, um mehr als einen zusätzlichen Speicherbereich für den Kernel anzuzeigen. Diese Methode unterstützt bis zu 15 nicht zusammenhängende Speicherbereiche. Wenn Sie den *pNKEnumExtensionDRAM*-Funktionszeiger implementieren, wird *OEMGetExtensionDRAM* nicht während des Startprozesses aufgerufen.

Aktivieren der Ressourcenfreigabe zwischen Treibern und dem OAL

Gerätetreiber müssen häufig auf physische Ressourcen zugreifen, beispielsweise im Speicher zugeordnete Registrierungen oder DMA-Puffer. Die Treiber können jedoch nicht direkt auf den physischen Speicher zugreifen, da das System nur virtuelle Adressen verwendet. Damit die Gerätetreiber auf den physischen Speicher zugreifen können, müssen die physischen Adressen virtuellen Adressen zugeordnet werden.

Dynamischer Zugriff auf den physischen Speicher

Wenn ein Treiber zusammenhängenden physischen Speicher benötigt, kann der Treiber mit der Funktion *AllocPhysMem* zusammenhängenden physischen Speicher zuordnen. Nachdem der Speicher zugeordnet wurde, gibt *AllocPhysMem* einen Zeiger

auf die virtuelle Adresse zurück, die der angegebenen physischen Adresse entspricht. Da das System den Speicher zuordnet, muss der Speicher durch Aufruf der Funktion freigegeben werden, wenn er nicht mehr benötigt wird.

Sollte ein Treiber jedoch den nicht auf Seiten basierenden Zugriff auf einen physischen Speicherbereich erfordern, der in der Datei *Config.bib* festgelegt ist, können Sie die Funktion *MmMapIoSpace* verwenden. *MmMapIoSpace* gibt eine nicht ausgelagerte virtuelle Adresse zurück, die der entsprechenden physischen Adresse direkt zugeordnet wird. Diese Funktion wird normalerweise verwendet, um auf Geräteregister zuzugreifen.

Statische Reservierung des physischen Speichers

Es kann gelegentlich vorkommen, dass Sie einen allgemeinen physischen Speicherbereich zur gemeinsamen Nutzung für Treiber oder zwischen einem Treiber und dem OAL freigeben müssen (beispielsweise zwischen einem IST und einem ISR). Ähnlich wie bei der gemeinsamen Nutzung eines Speicherbereichs für Startargumente zwischen dem Boot Loader und dem Kernel, können Sie in der Datei *Config.bib* einen gemeinsam genutzten Speicherbereich für die Treiberkommunikation reservieren. Eine Standardmethode ist die Verwendung der *DRIVER_GLOBALS*-Struktur, die in *Drv_glob.h* definiert ist (siehe Lektion 1).

Kommunikation zwischen den Treibern und dem OAL

Zusätzlich zum vom Kernel benötigten Standardsatz von IOCTLs, können Treiber über benutzerdefinierte IOCTLs, die in *OEMIoControl* implementiert sind, mit dem OAL kommunizieren. Die Kernelmodustreiber rufen *OEMIoControl* indirekt über *KernelIoControl* durch Übergabe in der benutzerdefinierten IOCTL auf. Der Kernel führt außer der Übergabe der Parameter an *OEMIoControl* keine weiteren Verarbeitungsvorgänge aus. Die Benutzermodustreiber können benutzerdefinierte OAL IOCTLs standardmäßig nicht direkt aufrufen. Die *KernelIOControl*-Aufrufe von Benutzermodustreibern oder Prozessen werden über eine Kernelmoduskomponente (*Oalioctl.dll*) an *OEMIoControl* übergeben, die eine Liste der OAL IOCTL-Codesegmente enthält, auf die der Benutzer zugreifen kann. Der Aufruf wird abgelehnt, wenn der angeforderte IOCTL-Code nicht im Modul aufgelistet ist. Sie können die Liste anpassen, indem Sie die Datei *Oalioctl.cpp* im Ordner *%_WINCEROOT%\Public\Common\Oak\Oalioctl* bearbeiten.

Zusammenfassung

Ein CE-Entwickler muss über umfassende Kenntnisse der Windows Embedded CE 6.0-Speicherarchitektur verfügen. BSP-Entwickler müssen insbesondere mit der Zuordnung des physischen Speichers in virtuelle Speicheradressbereiche vertraut sein. Der Speicherzugriff über OAL, Kernelmodusmodule, Benutzermodustreiber und Anwendungen erfordert umfassende Kenntnisse der statischen und dynamischen Zuordnungsmethoden, die im Kernelmodus oder Benutzermodus verfügbar sind. Weitere Informationen zur Kommunikation zwischen Kernelmodus- und Benutzermodus-Komponenten finden Sie in Kapitel 6 „Entwickeln von Gerätetreibern“.

Lektion 3: Hinzufügen der Unterstützung für die Energieverwaltung zu einem OAL

Wie in Kapitel 3 „Systemprogrammierung“ erklärt, umfasst Windows Embedded CE 6.0 zahlreiche Features für die Energieverwaltung basierend auf der Power Manager-Komponente, die OEM-Entwickler anpassen können, um Definitionen für den Systemenergiestatus ihrer Hardwareplattformen zu implementieren. In Bezug auf den OAL umfasst das Implementieren der Energieverwaltungsfunktionen zwei Schritte. Sie müssen sicherstellen, dass das Betriebssystem den Energiestatus der Hardwarekomponenten steuern und die Hardwareplattform das Betriebssystem über Änderungen des Energiestatus informieren kann. Für die meisten eingebetteten Geräte muss mindestens die Standardenergieverwaltung unterstützt werden, um den Energieverbrauch zu verringern und die Batterielebensdauer zu verlängern.

Nach Abschluss dieser Lektion können Sie:

- Den Energieverbrauch des Prozessors reduzieren.
- Den Übergang zum Standbymodus und das Reaktivieren des Systems beschreiben.

Veranschlagte Zeit für die Lektion: 15 Minuten.

Energiestatusübergänge

Eingebettete Geräte, die nicht ständig verwendet werden, beispielsweise PDAs (Personal Digital Assistant), befinden sich lange Zeit im Leerlauf und können in den Standbymodus oder einen anderen Energiesparmodus versetzt werden. Die meisten Prozessoren, die heute erhältlich sind, unterstützen diese Übergänge (siehe Abbildung 5.9).

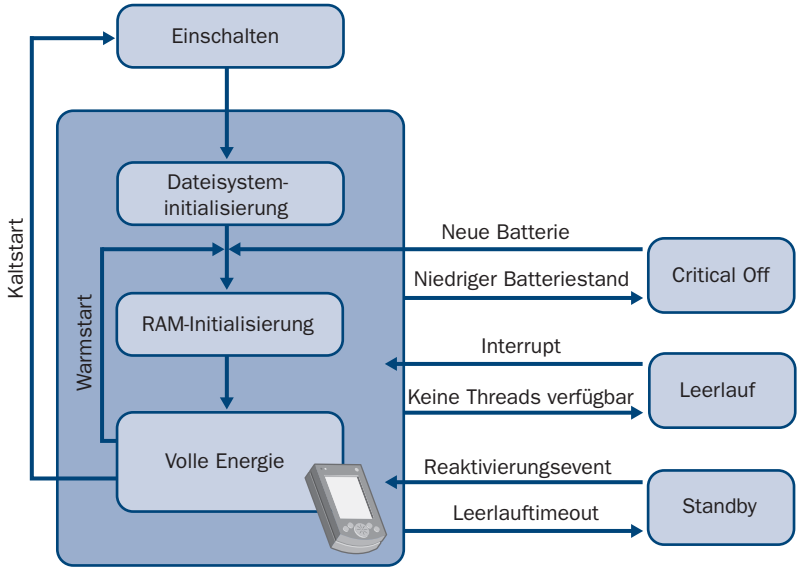


Abbildung 5.9 Energiestatusübergänge

Windows Embedded CE reagiert auf energiebezogene Events wie folgt:

- **Batteriestand sehr niedrig** Das System wechselt in den Critical Off-Status, wenn ein Spannungskomparator auf dem Board einen nicht maskierbaren Interrupt (NMI) auslöst, damit der Benutzer die Batterie ersetzen kann.
- **Leerlauf** Das System schaltet die CPU in den reduzierten Energiemodus, wenn die CPU keine Workerthreads ausführt, und wird durch einen Interrupt reaktiviert.
- **Standby („Suspend“)** Das System wechselt das Gerät in den Standby-Modus, wenn der Benutzer die Off-Taste drückt oder ein Leerlauftimeout auftritt. Das System wird durch ein Reaktivierungsevent wieder aktiviert, beispielsweise wenn der Benutzer erneut die Einschalttaste drückt. Auf einigen eingebetteten Geräten entspricht der Standby-Modus dem Aus-Status und das System wird mit einem Kaltstart reaktiviert.

Reduzieren des Energieverbrauchs im Leerlaufmodus

Um das Gerät in den reduzierten Energiemodus zu versetzen, verwendet Windows Embedded CE die Funktion *OEMIdle*, die der Kernel aufruft, wenn der Scheduler keine Threads ausführt. Die Funktion *OEMIdle* ist eine hardwarespezifische Routine, die von den Plattformfunktionen abhängt. Wenn der Systemzeitgeber beispielsweise

ein festes Intervall verwendet, kann die Funktion *OEMIdle* die erwarteten Energiesparfunktionen nicht bereitstellen, da das System bei jedem Zeitgeberinterrupt reaktiviert wird. Sie können jedoch die Kernelvariable *dwReschedTime* verwenden, um die Zeitdauer im reduzierten Energiemodus festzulegen, wenn der Prozessor programmierbare Intervallzeitgeber unterstützt.

Das System muss nach der Reaktivierung aus dem reduzieren Energiemodus die globalen Kernelvariablen aktualisieren, die vom Scheduler verwendet werden. Dies ist insbesondere für die Variable *CurMSec* wichtig, mit der das System die Anzahl der Millisekunden seit dem letzten Systemstart nachverfolgt. Bei der Reaktivierungsquelle kann es sich um den Systemzeitgeber oder einen anderen Interrupt handeln. Wenn die Reaktivierungsquelle der Systemzeitgeber ist, wurde die Variable *CurMSec* bereits vor der Übergabe der Ausführung an die Funktion *OEMIdle* aktualisiert. Ansonsten enthält *CurMSec* keinen aktualisierten Wert. Weitere Informationen zur Implementierung von *OEMIdle* finden Sie in der Quellcodedatei *Idle.c* im Ordner *%_WINCEROOT%\Platform\Common\Src\Common\Timer\Idle*.

HINWEIS Globale Kernelvariablen

Weitere Informationen zu globalen Variablen, die der Kernel für den Scheduler exportiert, finden Sie im Abschnitt „Kernel Global Variables for Scheduling" in der Windows Embedded CE 6.0-Dokumentation auf der Microsoft MSDN-Website unter *http://msdn.microsoft.com/en-us/library/aa915099.aspx*.

Aus-Modus und Standby-Modus des Systems

Ein Windows Embedded CE-Gerät kann sowohl den Aus-Modus als auch den Standby-Modus unterstützen. Das System ruft direkt die Funktion *GwesPowerOffSystem* oder die Funktion *SetSystemPowerState* auf, um das Gerät in den Standby-Modus zu versetzen. Beide Funktionen rufen die Routine *OEMPowerOff* auf.

Die Routine *OEMPowerOff*, die Teil des OAL ist, versetzt die CPU in den Standby-Modus. *OEMPowerOff* versetzt außerdem den RAM in den Selbstaktualisierungs-Modus, wenn der Prozessor beim Aktivieren des Standby-Modus die Aktualisierung nicht automatisch ausführt. Sie können die Interrupts festlegen, die das Gerät reaktivieren. Für tragbare Geräte wird normalerweise der Netzschalter-Interrupt festgelegt, aber Sie können ein beliebiges Reaktivierungsevent auswählen, das für die Zielplattform geeignet ist.

Aktivieren des Standby-Modus

Beim Aktivieren des Standby-Modus führt Windows Embedded CE folgende Vorgänge aus:

1. GWES informiert die Taskleiste über das Energiesparevent.
2. Das System bricht die Kalibrierung ab, falls es sich im Kalibrierungsfenster befindet.
3. Das System hält die Windows Message Queues an. Anschließend geht das System in den Singlethread-Modus über, der Funktionsaufrufe verhindert, die von blockierenden Operationen abhängen.
4. Das System überprüft, ob die Start-Benutzeroberfläche bei der Reaktivierung angezeigt werden muss.
5. Das System speichert den Videospeicher im RAM.
6. Das System ruft *SetSystemPowerState (NULL, POWER_STATE_SUSPEND, POWER_FORCE)* auf.
7. Power Manager:
   a. Ruft *FileSystemPowerFunction* auf, um die Treiber für das Dateisystem zu deaktivieren.
   b. Ruft *PowerOffSystem* auf, damit der Kernel den Standby-Modus aktiviert.
   c. Ruft *Sleep(0)* auf, um den Scheduler zu aktivieren.

HINWEIS FileSystemPowerFunction und PowerOffSystem

Wenn das OS Design keinen Power Manager oder GWES umfasst, muss der OEM die Funktionen *FileSystemPowerFunction* und *PowerOffSystem* explizit aufrufen.

8. Kernel:
   a. Entlädt den GWES-Prozess.
   b. Entlädt *Filesys.exe*.
   c. Ruft *OEMPowerOff* auf.
9. *OEMPowerOff* konfiguriert die Interrupts und versetzt die CPU in den Standby-Modus.

Reaktivieren aus dem Standby-Modus

Wenn ein konfigurierter Interrupt das System reaktiviert, wird der entsprechende ISR ausgeführt und an die Routine *OEMPowerOff* übergeben. Anschließend durchläuft das System die Reaktivierungssequenz, die folgende Vorgänge umfasst:

1. *OEMPowerOff* legt den ursprünglichen Interruptstatus erneut fest.
2. Kernel:
   a. Ruft *InitClock* auf, um den Systemzeitgeber neu zu initialisieren.
   b. Startet *Filesys.exe* mit einer Aktivierungsbenachrichtigung.
   c. Startet GWES mit einer Aktivierungsbenachrichtigung.
   d. Initialisiert den KITL-Interrupt neu, wenn dieser verwendet wurde.
3. Power Manager ruft *FileSystemPowerFunction* mit einer Aktivierungsbenachrichtigung auf.
4. GWES:
   a. Stellt den Videospeicher aus dem RAM wieder her.
   b. Aktiviert den Windows Manager.
   c. Legt den Anzeigekontrast fest.
   d. Zeigt gegebenenfalls die Start-Benutzeroberfläche an.
   e. Informiert die Taskleiste über die Reaktivierung.
   f. Informiert das Benutzer-Subsystem.
   g. Startet die erforderlichen Anwendungen.

HINWEIS Registrieren der Reaktivierungsquellen

Wenn der OAL den Kernel *IOCTL_HAL_ENABLE_WAKE* unterstützt, können die Anwendungen Reaktivierungsquellen registrieren. Weitere Informationen finden Sie im Abschnitt „IOCTL_HAL_ENABLE_WAKE" in der Windows Embedded CE 6.0-Dokumentation auf der Microsoft MSDN-Website unter *http://msdn2.microsoft.com/en-us/library/aa914884.aspx*.

Der Critical Off-Status

Auf Hardwareplattformen, die mit einem Spannungskomparator ausgestattet sind, der NMI auslöst, können Sie die Unterstützung für den Critical Off-Status implementieren, um Datenverlust bei niedrigem Batteriestand zu verhindern. Auf x86-Hardware exportiert der Kernel die Funktion *OEMNMIHandler*, um kritische

Systemevents zu erfassen. Auf anderen Systemen müssen Sie möglicherweise einen benutzerdefinierten IST implementieren, der *SetSystemPowerState* aufruft, um das System mit dem Power Manager auszuschalten. Der Critical Off-Status entspricht normalerweise dem Standby-Status und die dynamische RAM-Aktualisierung ist aktiviert.

HINWEIS Batteriestand erreicht Null

Stellen Sie beim Implementieren der Unterstützung für den Critical Off-Status sicher, dass der NMI zu einem Zeitpunkt ausgelöst wird, zu dem das System noch genügend Zeit hat, alle erforderlichen Vorgänge auszuführen, beispielsweise das Ausschalten der Peripheriegeräte, das Versetzen des RAM in den Selbstaktivierungs-Modus und das Aussetzen der CPU.

Zusammenfassung

Die Energieverwaltung ist ein wichtiges Windows Embedded CE-Feature, das den effizienten Energieverbrauch auf dem Zielgerät sicherstellt. OEMs sollten die Energieverwaltungsfeatures im OAL implementieren, um den Übergang in den Leerlauf- oder Standby-Modus sowie in den Critical Off-Status für batteriebetriebene Geräte zu ermöglichen. Das Implementieren der Energieverwaltung umfasst das erneute Synchronisieren der Kernelvariablen für den Zeitgeber, das Herunterfahren der Peripheriegeräte, das Versetzen des RAM in den Selbstaktualisierungs-Modus, das Festlegen der Reaktivierungsbedingungen und das Aussetzen der CPU. Microsoft stellt Referenzcode in den Beispiel-BSPs mit ausführlichen Informationen zur Implementierung bereit.

Lab 5: Anpassen eines Board Support Package

In diesem Lab klonen Sie ein Referenz-BSP in Visual Studio 2005 mit Platform Builder und verwenden dieses, um ein Run-Time Image zu erstellen. Als Plattform wird der Geräteemulator verwendet, da dieser auf dem Windows Embedded CE-Entwicklungscomputer ausgeführt werden kann. Microsoft stellt das Geräteemulator-BSP in Platform Builder als Referenz-BSP bereit.

HINWEIS Detaillierte schrittweise Anleitungen

Um die Verfahren in diesem Lab erfolgreich auszuführen, lesen Sie das Dokument „Detailed Step-by-Step Instructions for Lab 5" im Begleitmaterial.

▶ Klonen eines BSP

1. Klicken Sie in Visual Studio 2005 im Menü **Tools** auf **Platform Builder for CE 6.0**, und wählen Sie **Clone BSP** aus.
2. Wählen Sie im Fender **Clone Board Support Package** unter **Source BSP** die Option **Device Emulator: ARMV4I** in der Dropdown-Liste aus.
3. Geben Sie unter **New BSP Information** die in Tabelle 5.12 aufgeführten Informationen ein (siehe Abbildung 5.10):

Tabelle 5.12 Neue BSP-Informationen

Parameter	Wert
Name	DeviceEmulatorClone
Description	Klon des Geräteemulator-BSP
Platform Directory	DeviceEmulatorClone
Vendor	Contoso Ltd.
Version	0.0

4. Aktivieren Sie das Kontrollkästchen **Open New BSP Catalog File In Catalog Editor** und klicken Sie auf **Clone**.
5. Stellen Sie sicher, dass Platform Builder das Geräteemulator-BSP klont, und klicken Sie im Dialogfeld **Clone BSP** auf **OK**.
6. Überprüfen Sie, ob Visual Studio die Katalogdatei *DeviceEmulatorClone.pbcxml* automatisch öffnet. Schließen Sie den Katalog-Editor ohne Änderungen vorzunehmen.

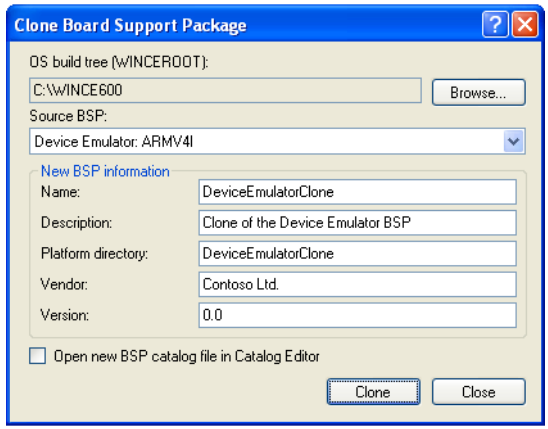


Abbildung 5.10 BSP-Kloninformationen

▶ Erstellen eines Run-Time Images

1. Um das geklonte BSP zu überprüfen, erstellen Sie ein neues OS Design basierend auf dem BSP ***DeviceEmulatorClone***. Rufen Sie das OS Design ***DeviceEmulatorCloneTest*** auf (siehe Abbildung 5.11). Weitere Informationen finden Sie in Lab 1 in Chapter 1.
2. Wählen Sie **Industrial Device** in den Designvorlagen und **Industrial Controller** in den Designvorlagenvarianten aus. Übernehmen Sie die Standardoptionen in den nachfolgenden Schritten.
3. Nachdem Platform Builder das Projekt ***DeviceEmulatorCloneTest*** generiert hat, überprüfen Sie das OS Design mittels der Katalogelemente in der **Catalog Items View**.
4. Stellen Sie sicher, dass die Debug-Buildkonfiguration aktiviert ist, indem Sie den **Configuration Manager** im Menü **Build** öffnen und überprüfen, ob im Listenfeld **Active Solution Configuration** die Option **DeviceEmulatorClone ARMV4I Debug** angezeigt wird.
5. Klicken Sie im Menü **Build** auf **Build Solution**.
6. Nachdem der Build abgeschlossen ist, konfigurieren Sie die Verbindungsoptionen, um den Geräteemulator zu verwenden.
7. Klicken Sie im Menü **Target** auf **Attach Device**, um das Run-Time Image auf den Geräteemulator zu übertragen, und starten Sie Windows Embedded CE. Beachten Sie die Debugmeldungen im Ausgabefenster von Visual Studio 2005. Warten Sie bis das Gerät vollständig gestartet wurde.

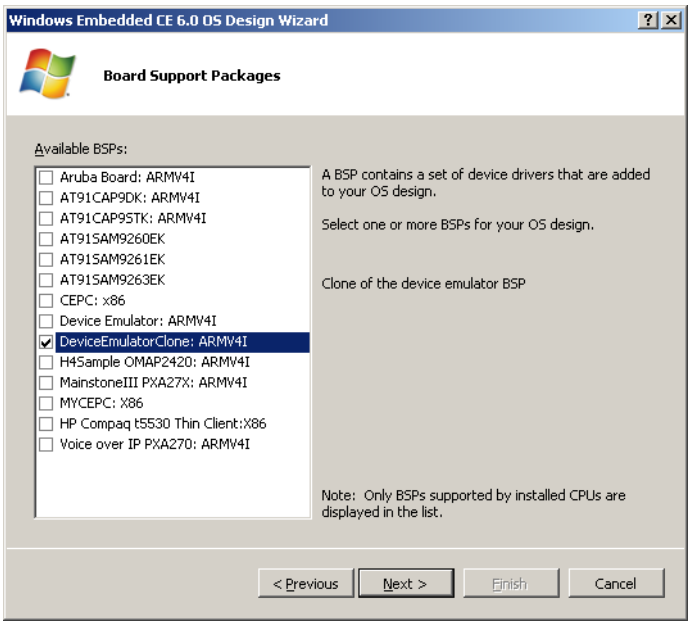


Abbildung 5.11 Ein neues OS Design basierend auf dem ***DeviceEmulatorClone*** BSP

HINWEIS BSP-Anpassung

Der Geräteemulator emuliert für das Referenz-BSP und das geklonte BSP die gleiche Hardwareplattform. Deshalb kann das neue Run-Time Image auf dem Geräteemulator ohne weitere Änderungen ausgeführt werden. Da die Hardware in den meisten Fällen jedoch unterschiedlich ist, muss das BSP angepasst werden, um CE zu starten.

▶ Anpassen des BSP

1. Trennen Sie die Verbindung mit dem Zielgerät und schließen Sie den Geräteemulator.
2. Öffnen Sie in Visual Studio die Quellcodedatei *init.c* im Ordner *%_PLATFORMROOT%\DeviceEmulatorClone\Src\Oal\Oallib* (siehe Abbildung 5.12).
3. Suchen Sie die OAL-Funktion *OEMGetExtensionDRAM* und fügen Sie folgende Codezeile hinzu, um während dem Systemstart im Ausgabefenster von Visual Studio eine Debugmeldung auszugeben.

```
BOOL
OEMGetExtensionDRAM(
   LPDWORD lpMemStart,
```

```
    LPDWORD lpMemLen
    )
{

...

    OALMSG(OAL_FUNC, (L"++OEMGetExtensionDRAM\r\n"));

    // Test message to confirm that our modifications are part of run-time image.
    OALMSG(1,(TEXT("This modification is part of the run-time image.\r\n")));

...

}
```

4. Erstellen Sie das Run-Time Image neu, um die Änderungen einzubeziehen, und stellen Sie die Verbindung mit dem Gerät erneut her, um das neue Run-Time Image herunterzuladen und im Geräteemulator zu starten. Überprüfen Sie, ob Windows Embedded CE die Debugmeldung im Ausgabefenster anzeigt.

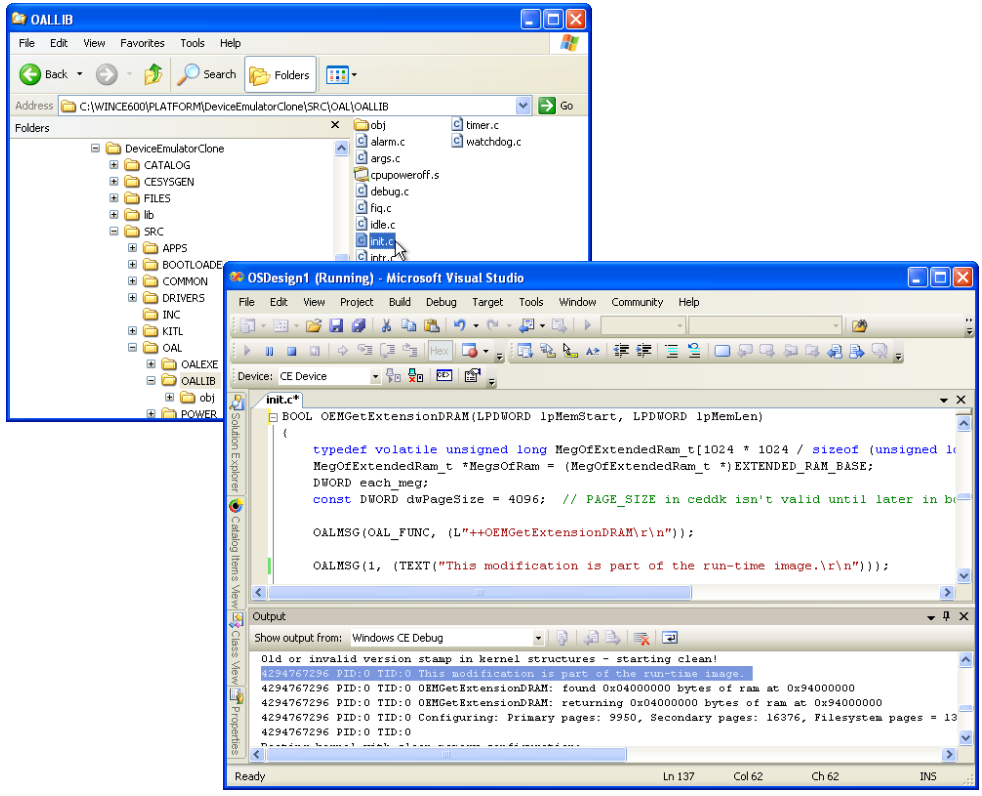


Abbildung 5.12 DeviceEmulatorClone BSP-Anpassung

Lernzielkontrolle

Das Anpassen eines Board Support Packages ist eine der schwierigsten und wichtigsten Aufgaben bei der Portierung von Windows Embedded CE 6.0 auf eine neue Hardwareplatform. Microsoft stellt hierzu Referenz-BSPs in Platform Builder bereit und empfiehlt OEMs, den Entwicklungsprozess mit dem Klonen eines geeigneten BSP zu beginnen. Die PQOAL-basierten BSPs folgen einer organisierten Ordner- und Dateistruktur, um den plattformagnostischen und plattformspezifischen Code nach Prozessortyp und OAL-Funktion zu trennen, damit sich der OEM auf die plattformspezifischen Implementierungsdetails konzentrieren kann, ohne Zeit mit den allgemeinen Aspekten des Kernels oder des Betriebssystems verbringen zu müssen.

OEM-Entwickler sollten folgende Empfehlungen berücksichtigen, um ein BSP erfolgreich anzupassen:

- **Überprüfen der Windows Embedded CE Referenz-BSPs** Windows Embedded CE BSPs folgen einer organisierten Architektur mit engen Beziehungen zum Kernel. Deshalb müssen zahlreiche APIs implementiert werden, die der Kernel benötigt, um das Betriebssystem auszuführen. Der Entwickler muss mit diesen APIs und deren Verwendungszweck vertraut sein. Die PQOAL-basierte Architektur entwickelt sich kontinuierlich weiter.
- **Klonen eines BSP** Sie sollten ein BSP nicht vollkommen neu erstellen. Klonen Sie stattdessesn ein vorhandenes BSP. Indem Sie möglichst viel Code aus einem Referenz-BSP verwenden, können Sie nicht nur die Entwicklungszeit verringern, sondern auch die Qualität Ihrer Lösung verbessern und eine solide Grundlage für die effiziente Verarbeitung künftiger Updates bereitstellen.
- **Boot Loader und BLCOMMON** Verwenden Sie BLCOMMON und die entsprechenden Bibliotheken zum Implementieren eines Boot Loaders, da die Bibliotheken nützliche hardwareunabhängige Features zum Herunterladen von Run-Time Images bereitstellen und dem Benutzer ermöglichen während des Startprozesses mit dem Zielgerät zu kommunizieren.
- **Speicher und BSPs** Sie müssen über umfassende Kenntnisse bezüglich des physischen und virtuellen Speichers in Windows Embedded CE 6.0 verfügen. Konfigurieren Sie die Dateien <Boot loader>.*bib* und *Config.bib*, um Informationen über den verfügbaren Speicher bereitzustellen, und gegebenenfalls die Einträge in der Tabelle *OEMAddressTable* zu ändern. Beachten Sie, dass Sie in Windows Embedded CE nicht direkt auf den physischen

Speicher zugreifen können. Verwenden Sie die korrekten APIs für die Speicherzuordnung, um die physischen Speicheradressen zu virtuellen Speicheradressen zuzuordnen.

- **Implementieren der Energieverwaltung** Implementieren Sie die *OEMIdle*-Funktion, damit das System die CPU in den Leerlaufmodus versetzen kann. Sie sollten außerdem *OEMPowerOff* implementieren, wenn die Plattform den Übergang in den Standby-Modus aufgrund von Benutzeraktionen oder kritischen Batterieständen unterstützt.

Schlüsselbegriffe

Kennen Sie die Bedeutung der folgenden wichtigen Begriffe? Sie können Ihre Antworten überprüfen, wenn Sie die Begriffe im Glossar am Ende dieses Buches nachschlagen.

- PQOAL
- Boot Loader
- KernelIoControl
- Driver Globals

Empfohlene Vorgehensweise

Um die in diesem Kapitel behandelten Prüfungsschwerpunkte erfolgreich zu beherrschen, sollten Sie die folgenden Aufgaben durcharbeiten.

Zugriff auf die Hardwareregister eines Peripheriegeräts

Implementieren Sie einen Gerätetreiber für die Peripheriegeräte und greifen Sie über das *MmMapIoSpace* API auf die Hardwareregister zu, um mit dem Gerät zu interagieren. Beachten Sie, dass Sie *MmMapIoSpace* nicht aus einer Anwendung aufrufen können.

HINWEIS Emulatoreinschränkungen

Da der Geräteemulator einen ARM-Prozessor in der Software emuliert, können Sie nicht auf Hardwaregeräte zugreifen. Sie müssen eine Originalhardwareplattform verwenden, um diese Übung auszuführen.

Neuorganisieren der Plattformspeicherzuordnungen

Wenn Sie die Datei *Config.bib* des geklonten Geräteemulator-BSB ändern, wird der verfügbare RAM verringert. Überprüfen Sie die Auswirkungen bezüglich des verfügbaren Speichers auf das System unter Verwendung der Speicher-APIs oder der Platform Builder-Tools.

Kapitel 6

Entwickeln von Gerätetreibern

Gerätetreiber sind Komponenten, die dem Betriebssystem und den Benutzeranwendungen ermöglichen mit den Peripheriegeräten eines Zielgeräts zu kommunizieren, beispielsweise mit dem PCI-Bus (Peripheral Component Interconnect), der Tastatur, der Maus, den seriellen Ports, dem Bildschirm, einem Netzwerkadapter und Speichergeräten. Anstatt direkt auf die Hardware zuzugreifen, lädt das Betriebssystem die entsprechenden Gerätetreiber und spricht die Geräte mittels Funktionen und E/A-Diensten an. Auf diese Art bleibt die Microsoft® Windows® Embedded CE 6.0 R2-Architektur flexibel, erweiterbar und unabhängig von der Hardware. Die Gerätetreiber enthalten den hardwarespezifischen Code. Es können zusätzlich zu den Standardtreibern in CE benutzerdefinierte Treiber implementiert werden, um weitere Peripheriegeräte zu unterstützen. Die Gerätetreiber bilden den größten Teil eines Board Support Packages (BSP) für ein OS Design. Beachten Sie jedoch, dass falsch implementierte Treiber ein ansonsten zuverlässiges System destabilisieren können. Sie müssen sich beim Entwickeln von Gerätetreibern an strikte Programmierverfahren für den Code halten und die Komponenten umfassend in verschiedenen Systemkonfigurationen testen. In diesem Kapitel werden die bewährten Verfahren zum Programmieren von Gerätetreibern erklärt, einschließlich bewährter Codestrukturen, dem Entwickeln einer sicheren und geeigneten Benutzeroberfläche, dem Sicherstellen der langfristigen Zuverlässigkeit und der Unterstützung mehrerer Energieverwaltungsfeatures.

Prüfungsziele in diesem Kapitel

- Laden und Verwenden von Gerätetreibern in Windows Embedded CE
- Verwalten der Systeminterrupts
- Verstehen des Speicherzugriffs und der Speicherverwaltung
- Erweitern der Treiberportabilität und Systemintegration

Bevor Sie beginnen

Damit Sie die Lektionen in diesem Kapitel durcharbeiten können, benötigen Sie:

- Mindestens Grundkenntnisse in der Windows Embedded CE-Softwareentwicklung, einschließlich der Basiskonzepte für die Treiberentwicklung, beispielsweise IOCTL (I/O Control) und DMA (Direct Memory Access).
- Kenntnisse in der Interruptverarbeitung und der Behandlung von Interrupts in einem Gerätetreiber.
- Kenntnisse in der Speicherverwaltung in C und C++ sowie im Vermeiden von Speicherverlust.
- Einen Entwicklungscomputer, auf dem Microsoft Visual Studio® 2005 Service Pack 1 und Platform Builder für Windows Embedded CE 6.0 installiert ist.

Lektion 1: Gerätetreiber-Grundlagen

Ein Gerätetreiber in Windows Embedded CE ist eine DLL (Dynamic-Link Library), die eine Abstraktionsschicht zwischen der Hardware und dem Betriebssystem sowie den auf dem Zielgerät ausgeführten Anwendungen bereitstellt. Die Treiber umfassen Funktionen und Programmlogik zum Initialisieren und Kommunizieren mit der Hardware. Softwareentwickler verwenden wiederum die Funktionen der Treiber in ihren Anwendungen, um auf die Hardware zuzugreifen. Wenn ein Gerätetreiber ein bekanntes API verwendet, beispielsweise Device Driver Interface (DDI), können Sie den Treiber als Bestandteil des Betriebssystems laden (z.B. einen Bildschirmtreiber oder den Treiber für ein Speichergerät). 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. Sie können Peripheriegeräte unterstützen, indem Sie Treiber zum OS Design hinzufügen, ohne die Anwendungen neu programmieren zu müssen.

Nach Abschluss dieser Lektion können Sie:

- Die systemeigenen Treiber und Streamtreiber unterscheiden.
- Die Vorteile und Nachteile monolithischer und mehrschichtiger Treiberarchitekturen beschreiben.

Veranschlagte Zeit für die Lektion: 15 Minuten.

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 („Pageing“). Das Betriebssystem lagert mittels *LoadDriver* geladene Treiber jedoch nicht aus.

Obwohl alle Treiber den *DllMain*-Einsprungspunkt verwenden, unterstützt Windows Embedded CE zwei Treibertypen: Systemeigene Treiber und Streamtreiber. Zu den systemeigenen CE-Treibern zählen Eingabe- und Ausgabetreiber, Tastaturtreiber und Touchscreentreiber. GWES (Graphics, Windowing and Events Subsystem) lädt und verwaltet diese Treiber direkt. Systemeigene Treiber implementieren Funktionen entsprechend dem Verwendungszweck, die GWES durch den Aufruf des APIs *GetProcAddress* bestimmt. *GetProcAddress* gibt einen Zeiger auf die gewünschte Funktion oder NULL zurück, wenn der Treiber die Funktion nicht unterstützt.

Hingegen verwenden Streamtreiber wohldefinierte Funktionen. Diese Treiber werden über den Geräte-Manager geladen und verwaltet. Damit der Geräte-Manager mit einem Streamtreiber interagieren kann, muss der Treiber die Funktionen *Init*, *Deinit*, *Open*, *Close*, *Read*, *Write*, *Seek* und *IOControl* implementieren. In vielen Streamtreibern ermöglichen die Funktionen *Read*, *Write* und *Seek* den Zugriff auf den Streaminhalt, obwohl nicht alle Peripheriegeräte Streamgeräte sind. Wenn die Anforderungen des Geräts über *Read*, *Write* und *Seek* hinausgehen, können Sie die Funktion *IOControl* verwenden, um die erforderlichen Funktionen zu implementieren. Die universelle Funktion *IOControl* kann die besonderen Anforderungen eines Streamtreibers erfüllen. Beispielsweise können Sie die Treiberfunktionen erweitern, indem Sie benutzerdefinierten IOCTL-Befehlscode sowie Eingabe- und Ausgabepuffer übergeben.

HINWEIS Systemeigene Treiberschnittstelle

Systemeigene Treiber müssen abhängig von der Treiberfunktionalität unterschiedliche Schnittstellentypen implementieren. Weitere Informationen zu den unterstützten Treibertypen finden Sie im Abschnitt Windows Embedded CE Drivers in der Windows Embedded CE 6.0-Dokumentation auf der Microsoft MSDN®-Website unter *http://msdn2.microsoft.com/en-us/library/aa930800.aspx*.

Monolithische und mehrschichtige Treiberarchitektur

Systemeigene Treiber und Streamtreiber unterscheiden sich ausschließlich in den bereitgestellten APIs. Sie können beide Treibertypen während des Systemstarts oder bei Bedarf laden. Beide Treibertypen können ein monolithisches oder mehrschichtiges Design verwenden (siehe Abbildung 6.1).

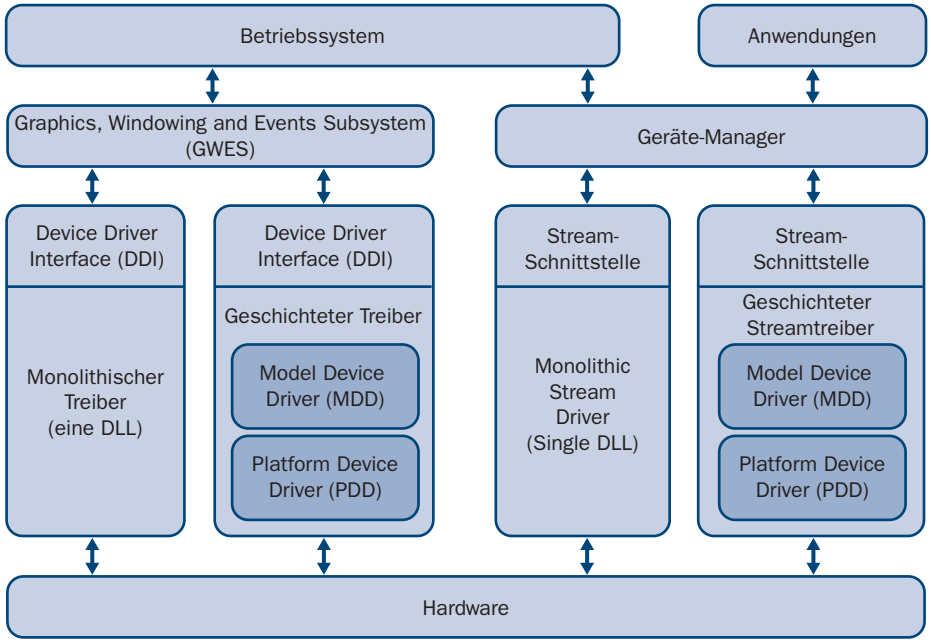


Abbildung 6.1 Monolithische und mehrschichtige Treiberarchitektur

Monolithische Treiber

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 Hardwarekomponenten besser geeignet. Wenn für ein Treiberprojekt kein mehrschichtiger Treibercode vorhanden ist, können Sie einen Treiber in einer monolithischen Architektur implementieren. Dies trifft insbesondere dann zu, wenn monolithischer Quellcode verfügbar ist.

MehrschichtigeTreiber

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 bereit. 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 für neue Hardware müssen Sie den Code im MDD-Layer normalerweise nicht ändern. Außerdem können Sie einen mehrschichtigen Treiber problemlos duplizieren und Funktionen hinzufügen oder entfernen, anstatt einen neuen Treiber zu erstellen. Viele der in Windows Embedded CE integrierten Treiber nutzen die Vorteile der mehrschichtigen Treiberarchitektur.

HINWEIS MDD/PDD-Architektur und Treiberupdates

Die MDD/PDD-Architektur spart Zeit bei der Entwicklung von Treiberupdates, um beispielsweise QFEs bereitzustellen. Das Beschränken von Änderungen auf den PDD-Layer erhöht die Entwicklungseffizienz.

Zusammenfassung

Windows Embedded CE unterstützt systemeigene Treiber und Streamtreiber. Systemeigene Treiber eignen sich am besten für Geräte, die keine Streamgeräte sind. Beispielsweise muss ein systemeigener Bildschirmtreiber die Daten eigenständig verarbeiten können. Für andere Geräte, beispielsweise Speicherhardware und serielle Ports, sollten Sie Streamtreiber verwenden, da diese Geräte die Daten hauptsächlich als geordnete Bytestreams verarbeiten. Systemeigene Treiber und Streamtreiber können das monolithische oder mehrschichtige Treiberdesign verwenden. Normalerweise ist die mehrschichtige Architektur basierend auf MDD und PDD besser geeignet, da sie die Wiederverwendung des Codes und die Entwicklung von Treiberupdates unterstützt. Verwenden Sie monolithische Treiber, wenn Sie aus Leistungsgründen zusätzliche Funktionsaufrufe zwischen MDD und PDD vermeiden möchten.

Lektion 2: Implementieren eines Streamschnittstellentreibers

In Windows Embedded CE ist ein Streamtreiber ein Gerätetreiber, der das Streamschnittstellen-API implementiert. Unabhängig von den Hardwareanforderungen stellen alle CE-Streamtreiber die Streamschnittstellenfunktionen für das Betriebssystem bereit, damit der Geräte-Manager 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 Hardwarekomponenten und Peripheriegeräte. Ein Streamtreiber kann jedoch auch auf andere Treiber zugreifen, um Anwendungen den Zugriff auf die Hardware zu ermöglichen. Sie müssen mit den Streamschnittstellenfunktionen und deren Implementierung vertraut sein, um funktionelle und zuverlässige Streamtreiber zu entwickeln.

Nach Abschluss dieser Lektion können Sie:

- Den Verwendungszweck des Geräte-Managers beschreiben.
- Die Streamtreiber-Anforderungen identifizieren.
- Einen Streamtreiber implementieren und verwenden.

Veranschlagte Zeit für die Lektion: 40 Minuten.

Geräte-Manager

Der Geräte-Manager 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 Geräte-Manager während des Startprozesses. Der Kernel lädt die Geräte-Manager-Shell (*Device.dll*), die wiederum den Geräte-Manager-Basiscode (*Devmgr.dll*) lädt, der die Streamtreiber lädt und entlädt (siehe Abbildung 6.2).

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 Geräte-Manager über das Dateisystem bereitstellt, werden von den Anwendungen als reguläre Dateiressourcen mit speziellen Dateinamen behandelt. Die Funktion *DeviceIoControl* ermöglicht Anwendungen das Ausführen direkter E/A-

Vorgänge. Die Anwendungen interagieren jedoch in beiden Fällen über den Geräte-Manager mit den Streamtreibern.

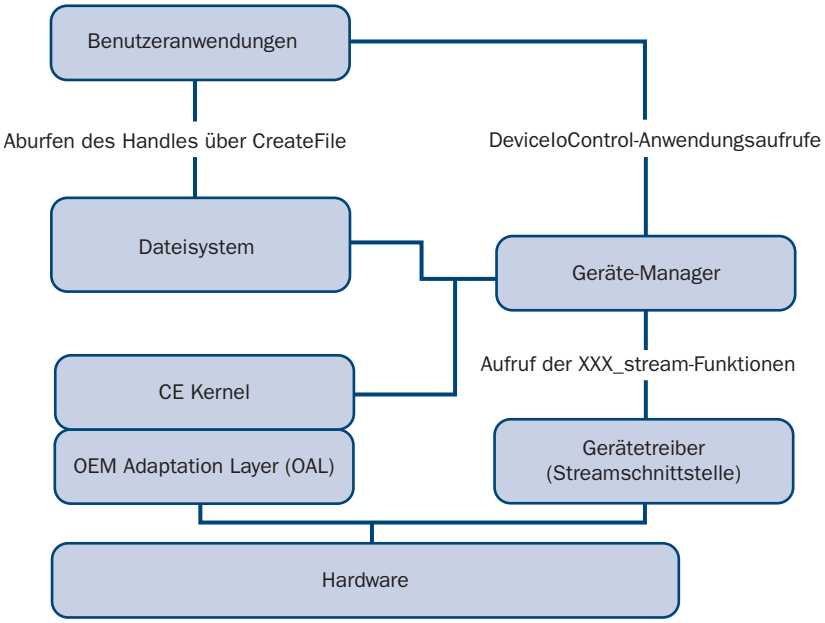


Abbildung 6.2 Der Geräte-Manager in Windows Embedded CE 6.0

Namenskonventionen für Treiber

Damit eine Anwendung einen Streamtreiber über das Dateisystem verwenden kann, muss der Streamtreiber als Dateiressource dargestellt werden. Die Anwendung kann die Gerätedatei in einem *CreateFile*-Aufruf angeben, um ein Gerätehandle abzurufen. Nachdem das Handle abrufen wurde, führt die Anwendung über *ReadFile* oder *WriteFile* E/A-Vorgänge aus, die der Geräte-Manager in die entsprechenden Aufrufe umwandelt, um die erforderlichen Lese- und Schreibvorgänge auszuführen. Damit Windows Embedded CE 6.0 die Streamgeräteressourcen 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.

Windows Embedded CE unterstützt folgende Namenskonventionen für Streamtreiber:

- **Legacynamen** Die klassische Namenskonvention für Streamtreiber besteht aus drei Großbuchstaben, einer Ziffer und einem Doppelpunkt. (*XXX*[0–9]:). *XXX*

steht für den aus drei Buchstaben bestehenden Treibernamen und [*0–9*] ist der Index des Treibers in den Registrierungseinstellungen (siehe Lektion 3). Da der Treiberindex aus nur einer Ziffer besteht, unterstützen Legacynamen nur bis zu zehn Instanzen eines Streamtreibers. Die erste Instanz entspricht dem Index 1, die neunte Instanz dem Index 9 und die zehnte Instanz dem Index 0. Beispielsweise greift *CreateFile(L"COM1:"...)* auf den ersten seriellen Port unter Verwendung des Legacynamens *COM1:* für den Streamtreiber zu.

HINWEIS Legacynameneinschränkung

Die Legacynamenskonvention unterstützt nur zehn Instanzen pro Streamtreiber.

- **Gerätenamen** Um auf einen Streamtreiber mit einem höheren Index zuzugreifen, können Sie den Gerätenamen anstatt des Legacynamens verwenden. Der Gerätename hat das Format *\$device\XXX[index]*. *\$device* ist ein Namespace, der anzeigt, dass es sich um einen Gerätenamen handelt. *XXX* ist der dreibuchstabige Treibername und [*index*] ist der Index des Treibers. Der Index kann aus mehreren Ziffern bestehen. Beispielsweise greift *CreateFile(L"\$device\COM11"...)* auf den Streamtreiber für den 11. seriellen Port zu. Auf Streamtreiber mit einem Legacynamen kann ebenfalls zugegriffen werden, beispielsweise *CreateFile(L"\$device\COM1"...)*.

HINWEIS Zugriff auf Legacynamen und Gerätenamen

Obwohl Legacynamen und Gerätenamen unterschiedliche Formate haben und andere Treiberinstanzen unterstützen, gibt *CreateFile* das gleiche Handle für den gleichen Streamtreiber zurück.

- **Busname** Streamtreiber für Geräte auf einem Bus, beispielsweisePersonal Computer Memory Card International Association (PCMCIA) oder Universal Serial Bus (USB), entsprechen den Busnamen, die der relevante Bustreiber an den Geräte-Manager übergibt, wenn die auf dem Bus verfügbaren Treiber aufgelistet werden. Busnamen beziehen sich auf die zugrundeliegende Struktur. Das übliche Format ist *\$bus\BUSNAME_[bus number]_[device number]_[function number]*. *\$bus* ist ein Namespace, der anzeigt, dass es sich um einen Busnamen handelt. *BUSNAME* bezieht sich auf den Namen oder Typ des Busses und [*bus number*], [*device number*] und [*function number*] sind busspezifische IDs. Beispielweise greift *CreateFile(L"\$ bus\PCMCIA_0_0_0"...)* auf das Gerät 0 und die Funktion 0 auf dem PCMCIA-Bus 0 zu, was einem seriellen Port entsprechen könnte.

HINWEIS Busnamenzugriff

Busnamen werden hauptsächlich zum Abrufen eines Handles zum Entladen und erneuten Laden von Bustreibern und für die Energieverwaltung verwendet, aber nicht für Lese- und Schreibvorgänge.

Streamschnittstellen-API

Damit der Geräte-Manager einen Streamtreiber laden und verwalten kann, muss der Treiber eine Schnittstelle exportieren, die als Streamschnittstelle bezeichnet wird. Die Streamschnittstelle besteht aus 12 Funktionen, die das Gerät initialisieren und öffnen, Daten lesen und schreiben, das Gerät ein- oder ausschalten oder das Gerät deaktivieren (siehe Tabelle 6.2).

Tabelle 6.1 Streamschnittstellenfunktionen

Funktion	Beschreibung
XXX_Init	Der Geräte-Manager 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 Geräte-Manager 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 Geräte-Manager 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 Geräte-Manager ruft diese Funktion auf, wenn eine Anwendung *CreateFile* aufruft, um den Zugriff auf das Gerät für Lese- oder Schreibvorgänge anzufordern.

Tabelle 6.1 Streamschnittstellenfunktionen (Fortsetzung)

Funktion	Beschreibung
XXX_PreClose	Der Geräte-Manager 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 Geräte-Manager 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 Geräte-Manager 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 für die Kompatibilität mit dem Geräte-Manager implementieren.
XXX_Write	Der Geräte-Manager ruft diese Funktion nach einem *WriteFile*-Aufruf auf, um die Daten an das Gerät zu übergeben. Wie *XXX_Read* ist *XXX_Write* erforderlich, aber kann einen leeren Wert haben, wenn das Gerät, beispielsweise ein Kommunikationseingangsport, Schreibvorgänge nicht unterstützt.
XXX_Seek	Der Geräte-Manager 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, aber muss für die Kompatibilität mit dem Geräte-Manager exportiert werden.

Tabelle 6.1 Streamschnittstellenfunktionen (Fortsetzung)

Funktion	Beschreibung
XXX_IOControl	Der Geräte-Manager ruft diese Funktion nach einem *DeviceIoControl*-Aufruf auf, um gerätespezifische Steuerungsaufgaben auszuführen. Beispielsweise hängen die Energieverwaltungsfeatures von *DeviceIoControl*-Aufrufen ab, wenn der Treiber eine Energieverwaltungsschnittstelle ankündigt, z.B. zum Abfragen der Energiefunktionen und zum Verwalten des Energiestatus über IOCTLs *IOCTL_POWER_CAPABILITIES*, *IOCTL_POWER_QUERY* und *IOCTL_POWER_SET*. 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 Geräte-Manager 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 Singlethread-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 Geräte-Manager 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. Microsoft empfiehlt, dass Treiber die Energieverwaltung basierend auf Power Manager und Energieverwaltungs-IOCTLs implementieren.

HINWEIS Das Präfix XXX_

In Funktionsnamen ist das Präfix *XXX* ein Platzhalter für den dreibuchstabigen Treibernamen. Sie müssen das Präfix durch den Namen im Treibercode ersetzen, beispielsweise *COM_Init* für einen Treiber namens *COM* oder *SPI_Init* für einen *SPI*-Treiber (Serial Peripheral Interface).

Gerätetreiberkontext

Der Geräte-Manager unterstützt die Kontextverwaltung basierend auf dem Gerätekontext und offenen Kontextparametern, die mit jedem Funktionsaufruf als DWORD-Werte an den Streamtreiber übergeben werden. Die Kontextverwaltung ist erforderlich, wenn ein Treiber instanzenspezifische Ressourcen zuordnen bzw. neu zuordnen muss, beispielsweise als Speicherblöcke. Beachten Sie, dass die Gerätetreiber DLLs sind und globale Variablen sowie andere im Treiber definierte oder zugeordnete Speicherstrukturen daher von allen Treiberinstanzen verwendet werden. Das erneut Zuordnen der falschen Ressourcen in Beantwortung eines *XXX_Close*- oder *XXX_Deinit*-Aufrufs kann zu Speicherverlust, Anwendungsfehlern und zur Instabilität des Systems führen.

Streamtreiber können Kontextinformationen pro Gerätetreiberinstanz basierend auf den folgenden beiden Ebenen verwalten:

1. **Gerätekontext** Der Treiber initialisiert den Kontext in der *XXX_Init*-Funktion. Dieser Kontext wird deshalb auch als Init-Kontext bezeichnet. Der Hauptverwendungszweck ist, den Treiber beim Verwalten der Ressourcen für den Hardwarezugriff zu unterstützen. Der Geräte-Manager übergibt die Kontextinformationen an die Funktionen *XXX_Init*, *XXX_Open*, *XXX_PowerUp*, *XXX_PowerDown*, *XXX_PreDeinit* und *XXX_Deinit*.
2. **Offener Kontext** Der Treiber initialisiert diesen Kontext in der Funktion *XXX_Open*. Wenn eine Anwendung die Funktion *CreateFile* für einen Streamtreiber aufruft, erstellt der Streamtreiber einen neuen offenen Kontext. Der offene Kontext ermöglicht dem Streamtreiber das Zuordnen von Datenzeigern und anderen Ressourcen für jede geöffnete Treiberinstanz. Der Geräte-Manager übergibt den Gerätekontext an den Streamtreiber in der Funktion *XXX_Open*, damit der Treiber eine Referenz auf den Gerätekontext im offenen Kontext speichern kann. Auf diese Art kann der Treiber in nachfolgenden Aufrufen, beispielsweise von *XXX_Read*, *XXX_Write*, *XXX_Seek*, *XXX_IOControl*, *XXX_PreClose* und *XXX_Close*, auf die

Gerätekontextinformationen zugreifen. Der Geräte-Manager übergibt den offenen Kontext ausschließlich als DWORD-Parameter an diese Funktionen.

Das folgende Codesegment veranschaulicht das Initialisieren eines Gerätekontexts für einen Beispieltreiber mit dem Namen SMP (SMP1:):

```
DWORD SMP_Init(LPCTSTR pContext, LPCVOID lpvBusContext)
{
    T_DRIVERINIT_STRUCTURE *pDeviceContext = (T_DRIVERINIT_STRUCTURE *)
        LocalAlloc(LMEM_ZEROINIT|LMEM_FIXED, sizeof(T_DRIVERINIT_STRUCTURE));

    if (pDeviceContext == NULL)
    {
        DEBUGMSG(ZONE_ERROR,(L" SMP: ERROR: Cannot allocate memory "

                             + "for sample driver's device context.\r\n"));

        // Return 0 if the driver failed to initialize.

        return 0;
    }

    // Perform system intialization...

    pDeviceContext->dwOpenCount = 0;

    DEBUGMSG(ZONE_INIT,(L"SMP: Sample driver initialized.\r\n"));

    return (DWORD)pDeviceContext;
}
```

Erstellen eines Gerätetreibers

Um einen Gerätetreiber zu erstellen, können Sie 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. Weitere Informationen zum Konfigurieren von Windows Embedded CE-Teilprojekten finden Sie in Kapitel 1 „Anpassen des OS Designs."

Ein guter Ausgangspunkt für einen Gerätetreiber ist die Vorlage **A Simple Windows Embedded CE DLL Subproject**, die Sie auf der Seite **Auto-Generated Subproject Files** im **Windows Embedded CE Subproject Wizard** auswählen können. Die Quellcodedatei mit einer Definition des *DllMain*-Einsprungspunkt 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.

Weitere Informationen zu Parameterdateien und der Sources-Datei finden Sie in Kapitel 2 „Erstellen und Bereitstellen eines Run-Time Images."

Implementieren der Streamfunktionen

Nachdem Sie das DLL-Teilprojekt erstellt haben, können Sie die Quellcodedatei in Visual Studio öffnen und die erforderlichen Funktionen hinzufügen, um die Streamschnittstelle und Treiberfunktionen zu implementieren. Das folgende Codesegment veranschaulicht die Definition der Streamschnittstellenfunktionen:

```
// SampleDriver.cpp : Defines the entry point for the DLL application.
//

#include "stdafx.h"

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

DWORD SMP_Init(LPCTSTR pContext, LPCVOID lpvBusContext)
{
    // Implement device context initialization code here.
    return 0x1;
}

BOOL SMP_Deinit(DWORD hDeviceContext)
{
    // Implement code to close the device context here.
    return TRUE;
}

DWORD SMP_Open(DWORD hDeviceContext, DWORD AccessCode, DWORD ShareMode)
{
    // Implement open context initialization code here.
    return 0x2;
}

BOOL SMP_Close(DWORD hOpenContext)
{
    // Implement code to close the open context here.
    return TRUE;
}

DWORD SMP_Write(DWORD hOpenContext, LPCVOID pBuffer, DWORD Count)
{
    // Implement the code to write to the stream device here.
    return Count;
```

```
}

DWORD SMP_Read(DWORD hOpenContext, LPVOID pBuffer, DWORD Count)
{
    // Implement the code to read from the stream device here.
    return Count;
}

BOOL SMP_IOControl(DWORD hOpenContext, DWORD dwCode,

                   PBYTE pBufIn, DWORD dwLenIn, PBYTE pBufOut,

                   DWORD dwLenOut, PDWORD pdwActualOut)
{
    // Implement code to handle advanced driver actions here.
    return TRUE;
}

void SMP_PowerUp(DWORD hDeviceContext)
{
    // Implement power management code here or use IO Control.
    return;
}

void SMP_PowerDown(DWORD hDeviceContext)
{
    // Implement power management code here or use IO Control.
    return;
}
```

Exportieren von Streamfunktionen

Um die Streamfunktionen in der Treiber-DLL für exterene Anwendungen bereitzustellen, muss der Linker die Funktionen während des Buildprozesses exportieren. C++ umfasst hierzu mehrere Optionen. Für mit dem Geräte-Manager kompatible Treiber-DLLs müssen Sie die Funktionen exportieren, indem Sie diese in der .def-Datei des DLL-Teilprojekts definieren. Der Linker bestimmt mittels der .def-Datei, welche Funktionen exportiert werden. Für einen Standardstreamtreiber müssen Sie die Streamschnittstellenfunktionen mit dem Präfix exportieren, der in der Quelldatei des Treibers und in den Registrierungseinstellungen angegeben ist. In Abbildung 6.3 ist eine .def-Datei für die Streamschnittstellenfunktionen dargestellt.

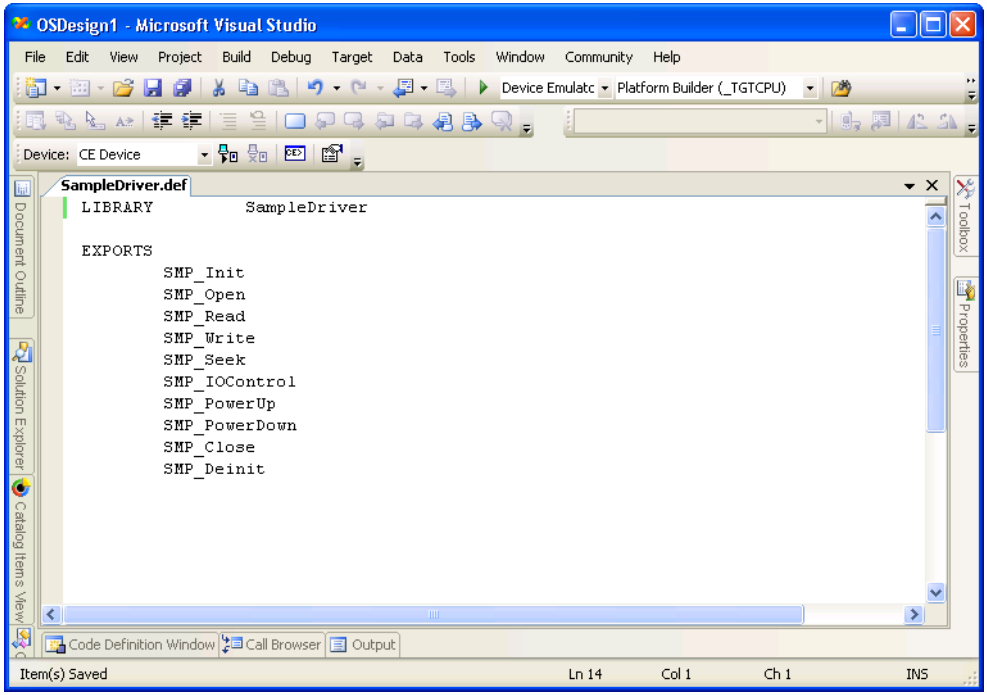


Abbildung 6.3 Eine .def-Datei für einen Streamtreiber

Sources-Datei

Bevor Sie den neuen Streamtreiber erstellen, sollten Sie die Sources-Datei im Stammordner des DLL-Teilprojekts überprüfen, um sicherzustellen, dass diese alle für den Buildprozess erforderlichen Dateien umfasst. Die Sources-Datei konfiguriert den Compiler und den Linker, um die erforderlichen Binärdateien zu erstellen (siehe Kapitel 2). In Tabelle 6.2 sind die wichtigsten Sources-Dateidirektiven für Gerätetreiber aufgeführt.

Tabelle 6.2 Wichtige Sources-Dateidirektiven für Gerätetreiber

Direktive	Beschreibung
WINCEOEM=1	Stellt sicher, dass zusätzliche Headerdateien und Importbibliotheken aus der *%_WINCEROOT%\Public*-Struktur einbezogen werden, die dem Treiber das Aufrufen plattformabhängiger Funktionen ermöglichen, beispielweise der Funktionen *KernelIoControl*, *InterruptInitialize* und *InterruptDone*.
TARGETTYPE=DYNLINK	Weist das Buildtool an, eine DLL zu erstellen.
DEFFILE=<*Driver Def File Name*>.def	Referenziert die Moduldefinitionsdatei, die die exportierten DLL-Funktionen definiert.
DLLENTRY=<*DLL Main Entry Point*>	Gibt die Funktion an, die aufgerufen wird, wenn Prozesse und Threads an die Treiber-DLL angehängt bzw. von dieser entfernt werden (Process Attach, Process Detach, Thread Attach und Thread Detach).

Öffnen und Schließen eines Streamtreibers mit dem Datei-API

Für den Zugriff auf einen Streamtreiber kann eine Anwendung die Funktion *CreateFile* verwenden und den gewünschten Gerätenamen angeben. Das folgende Beispiel veranschaulicht das Öffnen des Treibers *SMP1:* für Lese- und Schreibvorgänge. Beachten Sie jedoch, dass der Geräte-Manager den Treiber bereits geladen haben muss, beispielsweise während des Startprozesses. Lektion 3 enthält detaillierte Informationen zum Konfigurieren und Laden von Gerätetreibern.

```
// Open the driver, which results in a call to the SMP_Open function
hSampleDriver = CreateFile(L"SMP1:",
    GENERIC_READ | GENERIC_WRITE,
    FILE_SHARE_READ | FILE_SHARE_WRITE,
    NULL,
    OPEN_EXISTING,
    FILE_ATTRIBUTE_NORMAL,
```

```
    NULL);

if (hSampleDriver == INVALID_HANDLE_VALUE )
{
    ERRORMSG(1,(TEXT("Unable to open the driver.\r\n"));
    return FALSE;
}

// Access the driver and perform read,
// write, and seek operations as required.

// Close the driver
CloseHandle(hSampleDriver);
```

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 Geräte-Manager den Streamtreiber lädt und die *XXX_Init*-Funktion des Treibers aufruft. *ActivateDevice* ruft *ActivateDeviceEx* auf. Beachten Sie, dass *ActivateDeviceEx* den Zugriff auf einen bereits geladenen Treiber nicht ermöglicht. Die Funktion *ActivateDeviceEx* wird in erster Linie verwendet, um den im Funktionsaufruf angegebenen treiberspezifischen Registrierungsschlüssel zu lesen, um den DLL-Namen, das Gerätepräfix und andere Werte zu bestimmen, die relevanten Werte zur aktiven Geräteliste hinzuzufügen und den Gerätetreiber im Prozessbereich des Geräte-Managers zu laden. Der Funktionsaufruf gibt ein Handle zurück, das die Anwendung später zum Entladen des Treibers im Aufruf der Funktion *DeactivateDevice* verwenden kann.

ActivateDeviceEx ersetzt die Funktion *RegisterDevice* als Methode zum Laden eines erforderlichen Treibers:

```
// Ask Device Manager to load the driver for which the definition
// is located at HKLM\Drivers\ Sample in the registry.
hActiveDriver = ActivateDeviceEx(L"\\Drivers\\Sample", NULL, 0, NULL);
if (hActiveDriver == INVALID_HANDLE_VALUE)
{
    ERRORMSG(1, (L"Unable to load driver"));
    return -1;
}

// Once the driver is lodaded, applications can open the driver
hDriver = CreateFile (L"SMP1:",
                      GENERIC_READ| GENERIC_WRITE,
                      FILE_SHARE_READ | FILE_SHARE_WRITE,
```

```
                    NULL,
                    OPEN_EXISTING,
                    FILE_ATTRIBUTE_NORMAL,
                    NULL);

if (hDriver == INVALID_HANDLE_VALUE)
{
    ERRORMSG(1, (TEXT("Unable to open Sample (SMP) driver")));
    return 0;
}

//Insert code that uses the driver here

// Close the driver when access is no longer needed
if (hDriver != INVALID_HANDLE_VALUE)
{
    bRet = CloseHandle(hDriver);
    if (bRet == FALSE)
    {
        ERRORMSG(1, (TEXT("Unable to close SMP driver")));
    }
}

// Manually unload the driver from the system using Device Manager
if (hActiveDriver != INVALID_HANDLE_VALUE)
{
    bRet = DeactivateDevice(hActiveDriver);
    if (bRet == FALSE)
    {
        ERRORMSG(1, (TEXT("Unable to unload SMP driver ")));
    }
}
```

HINWEIS Automatisches und dynamisches Laden von Treibern

Das Laden eines Treibers durch den Aufruf der Funktion *ActivateDeviceEx* hat das gleiche Ergebnis wie das automatische Laden des Treibers während des Startprozesses über die im Schlüssel *HKEY_LOCAL_MACHINE\Drivers\BuiltIn* definierten Parameter. Der *BuiltIn*-Registrierungsschlüssel wird in Lektion 3 beschrieben.

Zusammenfassung

Streamtreiber sind Windows Embedded CE-Treiber, die das Streamschnittstellen-API implementieren. Die Streamschnittstelle ermöglicht dem Geräte-Manager das Laden und Verwalten dieser Treiber. Anwendungen können die Standardfunktionen des Dateisystems verwenden, um auf die Treiber zuzugreifen und E/A-Vorgänge auszuführen. Um einen Streamtreiber als Dateiressource über einen *CreateFile*-Aufruf darzustellen, muss der Name des Streamtreibers eine Namenskonvention einhalten,

die die Geräteressource von einer normalen Datei unterscheidet. Legacynamen (z.B. *COM1:*) sind auf zehn Instanzen pro Treiber beschränkt, da die Instanzen-ID aus nur einer Ziffer besteht. Wenn auf einem Zielgerät mehr als zehn Treiberinstanzen unterstützt werden müssen, verwenden Sie einen Gerätenamen (z.B. *\$device\COM1*).

Da der Geräte-Manager einen Treiber mehrmals laden kann, um die Anforderungen von Prozessen und Threads zu erfüllen, müssen die Streamtreiber die Kontextverwaltung implementieren. Windows Embedded CE kann für Gerätetreiber zwei Kontextebenen (Gerätekontext und offener Kontext) verwenden, die das Betriebssystem in den entsprechenden Funktionsaufrufen an den Treiber übergibt, damit der Treiber die internen Ressourcen und erforderlichen Speicherbereiche zuordnen kann.

Die Streamschnittstelle besteht aus 12 Funktionen: *XXX_Init*, *XXX_Open*, *XXX_Read*, *XXX_Write*, *XXX_Seek*, *XXX_IOControl*, *XXX_PowerUp*, *XXX_PowerDown*, *XXX_PreClose*, *XXX_Close*, *XXX_PreDeinit* und *XXX_Deinit*. Nicht alle Funktionen sind erforderlich (beispielsweise *XXX_PreClose* und *XXX_PreDeinit*), aber jede Funktion, die vom Streamgerätetreiber implentiert wird, muss von der Treiber-DLL an den Geräte-Manager übergeben werden. Um diese Funktionen zu exportieren, definieren Sie diese in der .def-Datei des DLL-Teilprojekts. Passen Sie außerdem die Sources-Datei des DLL-Teilprojekts an, um sicherzustellen, dass die Treiber-DLL von der Plattform unabhängige Funktionen aufrufen kann.

Lektion 3: Konfigurieren und Laden eines Treibers

In Windows Embedded CE 6.0 stehen Ihnen zum Laden eines Streamtreibers zwei Methoden zur Verfügung. Sie können den Geräte-Manager anweisen, den Treiber während des Startprozesses automatisch zu laden, indem Sie die Treibereinstellungen im Registrierungsschlüssel *HKEY_LOCAL_MACHINE\Drivers\BuiltIn* konfigurieren oder den Treiber dynamisch mit der Funktion *ActivateDeviceEx* laden. Der Geräte-Manager kann den Gerätetreiber immer mit den gleichen Registrierungsflags und Einstellungen laden. *ActivateDeviceEx* gibt jedoch ein Treiberhandle zurück, das Sie im Aufruf der Funktion *DeactivateDevice* verwenden können. Das dynamische Laden eines Treibers über *ActivateDeviceEx* ist insbesondere während der Entwicklungsphase vorteilhaft, da Sie den Treiber entladen, eine aktualisierte Version installieren und den Treiber erneut laden können, ohne das Betriebssystem neu starten zu müssen. Sie können *DeactivateDevice* auch zum Entladen eines Treibers verwenden, der automatisch basierend auf den Einträgen im *BuiltIn*-Registrierungsschlüssel geladen wurde. Zum erneuten Laden des Treibers müssen Sie jedoch *ActivateDeviceEx* direkt aufrufen.

Nach Abschluss dieser Lektion können Sie:

- Die erforderlichen Registrierungseinstellungen für einen Gerätetreiber identifizieren.
- In einem Treiber auf die Registrierungseinstellungen zugreifen.
- Einen Treiber beim Start oder bei Bedarf in einer Anwendung laden.
- Einen Treiber im Benutzer- oder Kernelbereich laden.

Veranschlagte Zeit für die Lektion: 25 Minuten.

Ladeprozedur für Gerätetreiber

Unabhängig davon, ob Sie einen Gerätetreiber statisch oder dynamisch laden, müssen Sie die Funktion *ActivateDeviceEx* verwenden. Der dedizierte Treiber namens Busenumerator (BusEnum) ruft *ActivateDeviceEx* für jeden Treiber auf, der unter *HKEY_LOCAL_MACHINE\Drivers\BuiltIn* registriert ist. Sie können *ActivateDeviceEx* auch direkt aufrufen, beispielsweise um einen alternativen Registrierungspfad für die Treibereinstellungen im Parameter *lpszDevKey* zu übergeben.

Der Geräte-Manager geht wie folgt vor, um Gerätetreiber beim Start zu laden:

1. Der Geräte-Manager liest den Eintrag *HKEY_LOCAL_MACHINE\Drivers\RootKey*, um den Pfad der Gerätetreibereinträge in der Registrierung zu bestimmen. Der Standardwert des *RootKey*-Eintrags ist *Drivers\BuiltIn*.
2. Der Geräte-Manager liest den Dll-Registrierungswert an der in *RootKey* angegebenen Stelle (*HKEY_LOCAL_MACHINE\Drivers\BuiltIn*), um die Enumerator-DLL zu laden. Standardmäßig ist dies der Busenumerator (*BusEnum.dll*). Der Busenumerator ist ein Streamtreiber, der die *Init*- und *Deinit*-Funktionen exportiert.
3. Der Busenumerator wird beim Start ausgeführt, um den *RootKey*-Registrierungspfad nach Unterschlüsseln zu durchsuchen, die auf weitere Busse und Geräte verweisen. Der Busenumerator kann später erneut mit einem anderen *RootKey*-Wert ausgeführt werden, um weitere Treiber zu laden. Der Busenumerator überprüft den *Order*-Wert in allen Unterschlüsseln, um die Ladereihenfolge zu bestimmen.
4. Der Busenumerator durchläuft die Unterschlüssel beginnend mit den niedrigsten *Order*-Werten und ruft *ActivateDeviceEx* auf, indem der aktuelle Registrierungspfad des Treiber (*HKEY_LOCAL_MACHINE\Drivers\BuiltIn\<DriverName>*) übergeben wird.
5. *ActivateDeviceEx* lädt die im Unterschlüssel des Treibers registrierte Treiber-DLL und erstellt einen Unterschlüssel für den Treiber unter dem Registrierungsschlüssel *HKEY_LOCAL_MACHINE\Drivers\Active*, um alle geladenen Treiber nachzuverfolgen.

In Abbildung 6.4 ist die Registrierung für einen Audiogerätetreiber unter dem Schlüssel *HKEY_LOCAL_MACHINE\Drivers\BuiltIn* dargestellt.

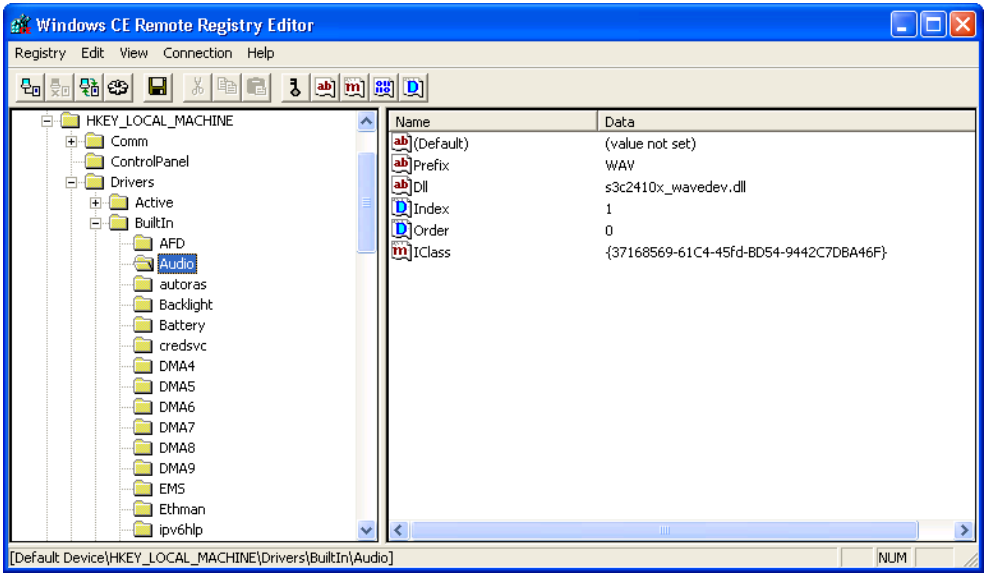


Abbildung 6.4 Registrierung eines Audiogerätetreibers

Registrierungseinstellungen zum Laden von Gerätetreibern

Wenn Sie den Treiber dynamisch über *ActivateDeviceEx* laden, müssen Sie die Registrierungseinstellungen des Treibers nicht in einen Unterschlüssel von *HKEY_LOCAL_MACHINE\Drivers\BuiltIn* hinzufügen. Sie können einen beliebigen Pfad verwenden, beispielsweise *HKEY_LOCAL_MACHINE\SampleDriver*. Die Registrierungswerte für den Treiber sind jedoch immer identisch. In Tabelle 6.3 sind die allgemeinen Registrierungseinträge für einen Gerätetreiber aufgeführt (siehe Abbildung 6.4 für Beispielwerte).

Tabelle 6.3 Allgemeine Registrierungseinträge für Gerätetreiber

Registrierungs-eintrag	Typ	Beschreibung
Prefix	REG_SZ	Ein Zeichenfolgenwert, der den dreibuchstabigen Namen des Treibers enthält. Dieser Wert ersetzt *XXX* in den Streamschnittstellenfunktionen. Anwendungen verwenden dieses Präfix außerdem zum Öffnen eines Treiberkontexts über CreateFile.

Tabelle 6.3 Allgemeine Registrierungseinträge für Gerätetreiber (Fortsetzung)

Registrierungs-eintrag	Typ	Beschreibung
Dll	REG_SZ	Der Name der DLL, die der Geräte-Manager zum Laden des Treibers verwendet. Beachten Sie, dass dies der einzige erforderliche Registrierungseintrag für einen Treiber ist.
Index	REG_DWORD	Die Zahl, die an den Treiberpräfix angehängt wird, um den Dateinamen des Treibers zu erstellen. Wenn der Wert 1 ist, können Anwendungen über die Funktion *CreateFile(L"XXX1:"...)* oder *CreateFile(L"\$device\XXX1"...)* auf den Treiber zugreifen. Beachten Sie, dass dieser Wert optional ist. Wenn Sie keinen Wert angeben, weist der Geräte-Manager dem Treiber den nächsten verfügbaren Indexwert zu.

Tabelle 6.3 Allgemeine Registrierungseinträge für Gerätetreiber (Fortsetzung)

Registrierungs-eintrag	Typ	Beschreibung
Order	REG_DWORD	Die Reihenfolge, in der die Treiber vom Geräte-Manager geladen werden. Wenn kein Wert angegeben ist, wird der Treiber zuletzt mit den anderen Treibern geladen. Treiber mit dem gleichen *Order*-Wert werden gleichzeitig geladen. Sie sollten diesen Wert nur konfigurieren, um eine sequentielle Reihenfolge zu erzwingen. Beispielsweise benötigt ein GPS-Treiber (Global Positioning System) möglicherweise einen UART-Treiber (Universal Asynchronous Receiver/Transmitter), um über einen seriellen Port auf die GPS-Datei zuzugreifen. In diesem Fall ist es wichtig, dem UART-Treiber einen niedrigeren Order-Wert als dem GPS-Treiber zuzuweisen, damit der UART-Treiber zuerst gestartet wird. Dies ermöglicht dem GPS-Treiber den Zugriff auf den UART-Treiber während der Initialisierung.
IClass	REG_MULTI_SZ	Dieser Wert kann vordefinierte Geräteschnittstellen-GUIDs (Globally Unique Identifiers) angeben. Um einen Schnittstelle zum Geräte-Manager anzukündigen, beispielsweise um das Plug & Play-System und Energieverwaltungs-funktionen zu unterstützen, fügen Sie die entsprechenden Schnittstellen-GUIDs zum *IClass*-Wert hinzu oder rufen Sie AdvertiseInterface im Treiber auf.

Tabelle 6.3 Allgemeine Registrierungseinträge für Gerätetreiber (Fortsetzung)

Registrierungseintrag	Typ	Beschreibung
Flags	REG_DWORD	Dieser Wert kann folgende Flags umfassen: ■ **DEVFLAGS_UNLOAD (0x0000 0001)** Der Treiber wird nach einem *XXX_Init*-Aufruf entladen. ■ **DEVFLAGS_NOLOAD (0x0000 0004)** Der Treiber kann nicht geladen werden. ■ **DEVFLAGS_NAKEDENTRIES (0x0000 0008)** Die Einsprungspunkte des Treibers sind beispielsweise *Init*, *Open* und *IOControl* ohne Präfixe. ■ **DEVFLAGS_BOOTPHASE_1 (0x0000 1000)** Der Treiber wird während der Systemphase 1 für Systeme mit mehreren Startphasen geladen. Dies verhindert, dass der Treiber während des Startprozesses mehrmals geladen wird. ■ **DEVFLAGS_IRQ_EXCLUSIVE (0x0000 0100)** Der Bustreiber lädt den Treiber nur, wenn er exklusiven Zugriff auf den IRQ hat, der vom IRQ-Wert angegeben wird. ■ **DEVFLAGS_LOAD_AS_USERPROC (0x0000 0010)** Lädt den Treiber im Benutzermodus.

Tabelle 6.3 Allgemeine Registrierungseinträge für Gerätetreiber (Fortsetzung)

Registrierungs-eintrag	Typ	Beschreibung
UserProcGroup	REG_DWORD	Lädt einen mit dem Flag *DEVFLAGS_LOAD_AS_USERPROC (0x0000 0010)* markierten Treiber im Benutzermodus mit Hostprozessen für Benutzermodustreiber. Benutzermodustreiber der gleichen Gruppe werden vom Geräte-Manager in der gleichen Hostprozessinstanz geladen. Wenn dieser Registrierungseintrag nicht vorhanden ist, lädt der Geräte-Manager den Benutzermodustreiber in einer neuen Hostprozessinstanz.

HINWEIS Flags

Weitere Informationen zum Flags-Registrierungswert finden Sie im Abschnitt „ActivateDeviceEx" in der Windows Embedded CE 6.0-Dokumentation auf der Microsoft MSDN-Website unter *http://msdn2.microsoft.com/en-us/library/aa929596.aspx*.

Registrierungseinstellungen für geladene Gerätetreiber

Neben den konfigurierbaren Registrierungseinträgen in treiberspezifischen Unterschlüsseln verwaltet der Geräte-Manager auch dynamische Registrierungsinformationen in den Unterschlüsseln für geladene Treiber unter dem Schlüssel *HKEY_LOCAL_MACHINE\Drivers\Active*. Die Unterschlüssel entsprechen nummerischen Werten, die das Betriebssystem dynamisch zuweist und für jeden Treiber inkrementell erhöht, bis das System neu gestartet wurde. Die Zahl gibt keinen bestimmten Treiber an. Wenn Sie beispielsweise einen Gerätetreiber laden oder entladen, weist das Betriebssystem dem Treiber die nächste Zahl zu und verwendet den vorherigen Unterschlüssel nicht erneut. Da Sie die zuverlässige Zuweisung zwischen dem Wert des Unterschlüssels und einem bestimmten Gerätetreiber nicht sicherstellen können, sollten Sie die Treibereinträge im Schlüssel *HKEY_LOCAL_MACHINE\Drivers\Active* nicht manuell bearbeiten. Sie können die treiberspezifischen Registrierungsschlüssel beim Laden in der *XXX_Init*-Funktion

des Treibers erstellen, lesen und schreiben, da der Geräte-Manager den Pfad zum aktuellen Drivers\Active-Unterschlüssel als den ersten Parameter an den Streamtreiber übergibt. Der Treiber kann den Registrierungsschlüssel mit der Funktion *OpenDeviceKey* öffnen.

In Tabelle 6.4 sind die Einträge in den Unterschlüsseln von *Drivers\Active* aufgeführt.

Tabelle 6.4 Registrierungseinträge für Gerätetreiber unter dem Schlüssel HKEY_LOCAL_MACHINE\Drivers\Active

Registrierungs-eintrag	Typ	Beschreibung
Hnd	REG_DWORD	Der Handlewert für den geladenen Gerätetreiber. Sie können den DWORD-Wert aus der Registrierung abrufen und in einem Aufruf an DeactivateDevice übergeben, um den Treiber zu entladen.
BusDriver	REG_SZ	Der Name des Treiberbusses.
BusName	REG_SZ	Der Name des Gerätebusses.
DevID		Eine eindeutige Geräte-ID vom Geräte-Manager.
FullName	REG_SZ	Der Name des Geräts, wenn der $device-Namespace verwendet wird.
Name	REG_SZ	Der Legacyname des Treibers, einschließlich Index, wenn ein Präfix angegeben ist (für Treiber ohne Präfix nicht vorhanden).
Order	REG_DWORD	Der gleiche *Order*-Wert wie im Registrierungsschlüssel des Treibers.
Key	REG_SZ	Der Pfad zum Registrierungsschlüssel des Treibers.
PnpId	REG_SZ	Die Plug & Play-ID für PCMCIA-Treiber.
Sckt	REG_DWORD	Beschreibt für PCMCIA-Treiber die aktuelle Socket und Funktion der PC-Karte.

HINWEIS Überprüfen des Active-Schlüssels

Wenn Sie die Funktion *RequestDeviceNotifications* mit der Geräteschnittstelle GUID von *DEVCLASS_STREAM_GUID* aufrufen, kann eine Anwendung Meldungen vom Geräte-Manager empfangen, um die Streamtreiber programmatisch zu identifizieren. *RequestDeviceNotifications* ersetzt *EnumDevices*.

Kernelmodustreiber und Benutzermodustreiber

Treiber können im Kernel- oder im Benutzerspeicherbereich ausgeführt werden. Im Kernelmodus können Treiber auf die Hardware und den Kernelspeicher zugreifen. Funktionsaufrufe sind jedoch normalerweise auf Kernel-APIs beschränkt. Windows Embedded CE 6.0 führt Treiber standardmäßig im Kernelmodus aus. Treiber im Benutzermodus können nicht direkt auf den Kernelspeicher zugreifen und die Leistung im Benutzermodus ist geringer. Der Benutzermodus hat jedoch den Vorteil, dass sich ein Treiberfehler nur auf den aktuellen Prozess auswirkt, wohingegen ein Treiberfehler im Kernelmodus das gesamte Betriebssystem beeinträchtigt kann. Ein Fehler in einem Benutzermodustreiber kann im Allgemeinen einfacher behoben werden.

HINWEIS Einschränkungen für Kerneltreiber

Kerneltreiber können Elemente der Benutzeroberfläche in CE 6.0 R2 nicht direkt anzeigen. Um Elemente einer Benutzeroberfläche zu verwenden, muss der Entwickler eine DLL erstellen, die im Benutzermodus geladen wird, und die DLL anschließend mit *CeCallUserProc* aufrufen. Weitere Informationen zu CeCallUserProc finden Sie auf der MSDN-Website unter *http://msdn2.microsoft.com/en-us/library/aa915093.aspx*.

Benutzermodustreiber und der Reflector-Dienst

Um mit der Hardware zu kommunizieren und bestimmte Aufgaben auszuführen, müssen Benutzermodustreiber auf den Systemspeicher und APIs zugreifen können, die für Standardprozesse im Benutzermodus nicht verfügbar sind. Windows Embedded CE 6.0 stellt deshalb den Reflector-Dienst bereit, der im Kernelmodus ausgeführt wird, das Puffermarshalling ausführt und Speicherverwaltungs-APIs für die Benutzermodustreiber aufruft. Der Reflector-Dienst ist transparent, damit die Benutzermodustreiber ähnlich wie Kernelmodustreiber ohne Änderungen funktionieren. Eine Ausnahme ist ein Treiber, der Kernel-APIs verwendet, die im Benutzermodus nicht verfügbar sind. Dieser Treiber kann nicht im Benutzermodus ausgeführt werden.

Wenn eine Anwendung *ActivateDeviceEx* aufruft, lädt der Geräte-Manager den Treiber direkt im Kernelbereich oder übergibt die Anforderung an den Reflector-Dienst, der einen Hostprozess für Benutzermodustreiber (*Udevice.exe*) über einen *CreateProcess*-Aufruf startet. Der *Flags*-Eintrag im Registrierungsschlüssel des Treibers legt fest, ob ein Treiber im Benutzermodus ausgeführt wird (*DEVFLAGS_LOAD_AS_USERPROC*-Flag). Nachdem die erforderliche Instanz von *Udevice.exe* und der Benutzermodustreiber gestartet wurden, leitet der Reflector-Dienst den *XXX_Init*-Aufruf des Geräte-Managers an den Benutzermodustreiber weiter und gibt den Code vom Benutzermodustreiber an den Geräte-Manager zurück (siehe Abbildung 6.5). Dies trifft auch auf alle anderen Streamfunktionen zu.

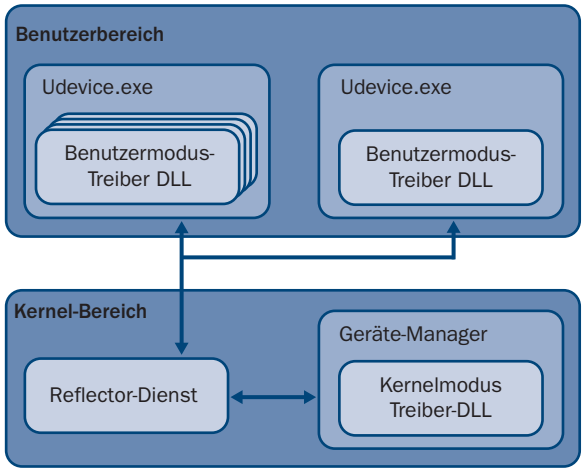


Abbildung 6.5 Benutzermodustreiber, Kernelmodustreiber und der Reflector-Dienst

Registrierungseinstellungen für Benutzermodustreiber

In Windows Embedded CE 6.0 können Sie mehrere Benutzermodustreiber in einem einzigen Hostprozess ausführen und mehrere Hostprozesse aktivieren. Die in einer Instanz von *Udevice.exe* zusammengefassten Treiber verwenden den gleichen Prozessbereich. Dies ist insbesondere für Treiber nützlich, die voneinander abhängig sind. Treiber im gleichen Prozessbereich können die Stabilität der anderen Treiber im gleichen Bereich beeinträchtigen. Wenn beispielsweise ein Benutzermodustreiber das Fehlschlagen des Hostprozesses verursacht, schlagen alle Treiber in diesem Hostprozess fehl. Das System funktioniert bis auf die betroffenen Treiber und Anwendungen weiterhin. Sie können dieses Problem beheben, indem Sie die Treiber neu laden. Wenn Sie einen wichtigen Treiber in einem separaten Hostprozess für Benutzermodustreiber isolieren, können Sie die Systemstabilität verbessern. Unter

Verwendung der in Tabelle 6.5 aufgeführten Registrierungseinträge können Sie individuelle Hostprozessgruppen definieren.

Tabelle 6.5 Registrierungseinträge für Treiberhostprozesse im Benutzermodus

Registrierungseintrag	Typ	Beschreibung
HKEY_LOCAL_MACHINE\Drivers\ProcGroup_###	REG_KEY	Definiert eine aus drei Ziffern bestehende Gruppen-ID (###) für den Hostprozess eines Benutzermodustreibers, beispielsweise *ProcGroup_003*, die Sie im *UserProcGroup*-Eintrag im Registrierungsschlüssel eines Treibers angeben können, beispielsweise *UserProcGroup =3*.
ProcName	REG_SZ	Der Prozess, der vom Reflector-Dienst gestartet wird, um den Benutzermodustreiber zu hosten, beispielsweise *ProcName= Udevice.exe*.
ProcVolPrefix	REG_SZ	Gibt das Dateisystemlaufwerk an, das vom Reflector-Dienst für den Hostprozess des Benutzermodustreibers bereitgestellt wird, beispielsweise *ProcVolPrefix = $udevice*. Das angegebene ProcVolPrefix ersetzt das *$device*-Laufwerk in Treibergerätenamen.

Nachdem Sie die gewünschten Hostprozessgruppen definiert haben, können Sie alle Benutzermodustreiber einer bestimmten Gruppe zuordnen, indem Sie den *UserProcGroup*-Registrierungseintrag zum Unterschlüssel des Gerätetreibers hinzufügen (siehe Tabelle 6.3). Der Registrierungseintrag *UserProcGroup* ist nicht standardmäßig vorhanden. Dies entspricht einer Konfiguration, in der der Geräte-Manager alle Benutzermodustreiber in separaten Hostprozessinstanzen lädt.

Binary Image Builder-Konfiguration

Der Windows Embedded CE-Buildprozess verwendet .bib-Dateien, um den Inhalt des Run-Time Images zu generieren und das Speicherlayout des Geräts zu definieren (siehe Kapitel 2). Sie können unter anderem die Flags für die Moduldefinition eines Treibers festlegen. Wenn die Einstellungen in der .bib-Datei nicht mit den Registrierungseinträgen für einen Gerätetreiber übereinstimmen, können Probleme auftreten. Wenn Sie beispielsweise das *K*-Flag für ein Gerätetreibermodul in einer .bib-Datei und das Flag *DEVFLAGS_LOAD_AS_USERPROC* im Unterschlüssel des Treibers festlegen, um den Treiber in den Hostprozess für Benutzermodustreiber zu laden, kann der Treiber nicht geladen werden, da das *K*-Flag *Romimage.exe* anweist, das Modul im Kernelbereich über der Speicheradresse 0x80000000 zu laden. Um einen Treiber im Benutzermodus zu laden, laden Sie das Modul im Benutzerbereich unter 0x80000000, beispielsweise im *NK*-Speicherbereich, der in der Datei *Config.bib* für das BSP definiert ist.

Der folgende Eintrag in der .bib-Datei veranschaulicht das Laden eines Benutzermodustreibers in den *NK*-Speicherbereich:

```
driver.dll      $(_FLATRELEASEDIR)\driver.dll    NK    SHQ
```

Die Flags *S* und *H* zeigen an, dass die Datei *Driver.dll* eine ausgeblendete Systemdatei im flachen *Release*-Verzeichnis ist. Das *Q*-Flag gibt an, dass das System das Modul im Kernelbereich und im Benutzerbereich laden kann. Dieses Flag fügt zwei Kopien der DLL zum Run-Time Image hinzu (eine Kopie mit und eine Kopie ohne das *K*-Flag), was die ROM- und RAM-Speicheranforderungen für den Treiber verdoppelt. Setzen Sie deshalb das *Q*-Flag mit Bedacht ein.

Das *Q*-Flag entspricht folgenden Einträgen:

```
driver.dll      $(_FLATRELEASEDIR)\driver.dll    NK    SH
driver.dll      $(_FLATRELEASEDIR)\driver.dll    NK    SHK
```

Zusammenfassung

Windows Embedded CE kann Treiber im Kernel- oder im Benutzerbereich laden. Im Kernelbereich ausgeführte Treiber können auf System-APIs und den Kernelspeicher zugreifen sowie die Systemstabilität beim Auftreten von Fehlern beeinflussen. Die Leistung richtig implementierter Kernelmodustreiber ist besser als die von Benutzermodustreibern, da der Kontextwechsel zwischem dem Kernelmodus und dem Benutzermodus reduziert wird. Der Vorteil von Benutzermodustreibern ist, dass Fehler hauptsächlich den aktuellen Benutzermodusprozess beeinträchtigen.

Beachten Sie außerdem, dass Benutzermodustreiber weniger privilegiert sind, was ein wichtiger Aspekt beim Einsatz nicht vertrauenswürdiger Treiber von Drittanbietern sein kann.

Um einen Benutzermodustreiber mit dem Geräte-Manager im Kernelmodus zu integrieren, verwendet der Geräte-Manager den Reflector-Dienst, der den Treiber in einem Hostprozess für Benutzermodustreiber lädt und die Streamfunktionsaufrufe und Rückgabewerte zwischen dem Treiber und dem Geräte-Manager weiterleitet. Auf diese Art können Anwendungen weiterhin über bekannte Dateisystem-APIs auf den Treiber zugreifen, und der Treiber erfordert keine Codeänderungen bezüglich des Streamschnittstellen-APIs für die Kompatibilität mit dem Geräte-Manager. Standardmäßig werden Benutzermodustreiber in separaten Hostprozessen ausgeführt. Sie können jedoch Hostprozessgruppen konfigurieren und diesen Gruppen Treiber zuweisen, indem Sie den entsprechenden *UserProcGroup*-Registrierungseintrag zum Unterschlüssel eines Treibers hinzufügen. Die Unterschlüssel für Treiber befinden sich in verschiedenen Registrierungseinträgen. Um Treiber während des Starts automatisch zu laden, müssen Sie die Unterschlüssel im *RootKey* des Geräte-Managers einfügen (standardmäßig *HKEY_LOCAL_MACHINE\Drivers\BuiltIn*). Treiber, deren Unterschlüssel in verschiedenen Registrierungseinträgen gespeichert sind, können mit der Funktion *ActivateDeviceEx* bei Bedarf geladen werden.

Lektion 4: Implementieren einer Interruptmethode in einem Gerätetreiber

Interrupts sind Benachrichtigungen, die von der Hardware oder Software generiert werden, um die CPU zu informieren, dass ein Event aufgetreten ist, das sofort beachtet werden muss (beispielsweise ein Zeitgeber- oder Tastaturevent). Beim Auftreten eines Interrupts hält die CPU die Ausführung des aktuellen Threads an, springt zu einem Traphandler im Kernel, um auf das Event zu reagieren, und setzt die Ausführung des ursprünglichen Threads fort, nachdem der Interrupt verarbeitet wurde. Die integrierten Hardwarekomponenten und Peripheriegeräte, beispielsweise die Systemuhr, serielle Ports, Netzwerkadapter, Tastaturen, die Maus und Touchscreen-Monitore, werden von der CPU beachtet. Der Kernelausnahmehandler führt den entsprechenden Code in ISRs im Kernel oder in den zugeordneten Gerätetreibern aus. Um die Interruptverarbeitung in einem Gerätetreiber zu implementieren, müssen Sie mit den Interruptverarbeitungsmethoden in Windows Embedded CE 6.0 vertraut sein, einschließlich des Registrierens von ISRs im Kernel und dem Ausführen von ISTs im Geräte-Manager.

Nach Abschluss dieser Lektion können Sie:

- Einen Interrupthandler im OEM Adaptation Layer (OAL) implementieren.
- Interrupts in einem Gerätetreiber-IST (Interrupt Service Thread) registrieren und behandeln.

Veranschlagte Zeit für die Lektion: 40 Minuten.

Architektur für die Interruptverarbeitung

Windows Embedded CE 6.0 ist ein portables Betriebssystem, das aufgrund der flexiblen Interruptbehandlungsarchitektur unterschiedliche CPU-Typen mit verschiedenen Interruptschemas unterstützt. Die Interruptverarbeitungsarchitektur nutzt die Interrupt-Synchronisierungsfunktionen im OEM Adaptation Layer (OAL) und die Thread-Synchronisierungsfunktionen von Windows Embedded CE, um die Interruptverarbeitung in ISRs und ISTs aufzuteilen (siehe Abbildung 6.6).

Die Interruptverarbeitung in Windows Embedded CE 6.0 basiert auf folgenden Konzepten:

1. Der Kernel ruft während des Startprozesses die Funktion *OEMInit* im OAL auf, um alle verfügbaren im Kernel integrierten ISRs basierend auf den IRQ-Werten

mit den entsprechenden Hardwareinterrupts zu registrieren. Die IRQ-Werte identifizieren die Quelle des Interrupts in den Registern der Prozessor-Interruptcontroller.

2. Die Gerätetreiber rufen die Funktion *LoadIntChainHandler* auf, um die in den ISR DLLs implementierten ISRs dynamisch zu installieren. *LoadIntChainHandler* lädt die ISR DLL in den Kernelspeicher und registriert die angegebene ISR mit dem IRQ-Wert in der Interrupt-Verteilertabelle des Kernels.
3. Ein Interrupt benachrichtigt die CPU, dass der aktuelle Thread aufgrund eines Events angehalten und die Steuerung an eine andere Routine übergeben werden muss.
4. Wenn ein Interrupt auftritt, hält die CPU die Ausführung des aktuellen Threads an und verwendet den Kernel-Ausnahmehandler als primäres Ziel der Interrupts.
5. Der Ausnahmehandler maskiert alle Interrupts mit der gleichen oder einer niedrigeren Priorität und ruft anschließend die entsprechende ISR auf, um den aktuellen Interrupt zu verarbeiten. Die meisten Hardwareplattformen verwenden Interruptmasken und Interruptprioritäten, um hardwarebasierte Interrupt-Synchronisierungsmethoden zu implementieren.
6. Die ISR führt alle erforderlichen Aufgaben aus, beispielsweise das Maskieren des aktuellen Interrupts, damit das Hardwaregerät keine weiteren Interrupts auslösen kann, die die aktuelle Verarbeitung beeinträchtigen, und gibt anschließend einen SYSINTR-Wert an den Ausnahmehandler zurück. Der SYSINTR-Wert ist eine logische Interrupt-ID.
7. Der Ausnahmehandler übergibt den SYSINTR-Wert an den Interrupthandler des Kernels, der das Event für den SYSINTR-Wert bestimmt, und den Event gegebenenfalls für die ISTs des Interrupts signalisiert.
8. Der Interrupthandler demaskiert alle Interrupts, außer den Interrupt, der gerade verarbeitet wird. Das Maskieren des aktuellen Interrupts verhindert, dass das aktuelle Gerät einen weiteren Interrupt verursacht, während der IST ausgeführt wird.
9. Der IST wird vom signalisierten Event aktiviert, um den Interrupt zu verarbeiten ohne andere Geräte zu blockieren.
10. Der IST ruft die Funktion *InterruptDone* auf, um den Interrupthandler zu benachrichtigen, dass der IST die Verarbeitung abgeschlossen hat und für ein weiteres Interruptevent verfügbar ist.

11. Der Interrupthandler ruft die Funktion *OEMInterruptDone* im OAL auf, um die Interruptverarbeitung abzuschließen und den Interrupt erneut zu aktivieren.

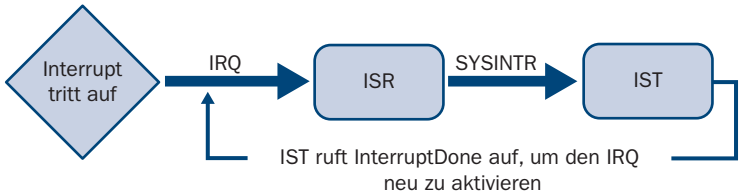


Abbildung 6.6 IRQs, ISRs, SYSINTRs und ISTs

Interrupt Service Routines

ISRs sind normalerweise kleine Codesegmente, die bei einem Hardwareinterrupt ausgeführt werden. Da der Kernel-Ausnahmehandler beim Ausführen der ISR alle Interrupts mit gleicher oder niedrigerer Priorität maskiert, muss die ISR so schnell wie möglich beendet und ein SYSINTR-Wert zurückgegeben werden, damit der Kernel alle IRQs mit minimaler Verzögerung erneut aktivieren (demaskieren) kann (außer den aktuellen Interrupt). Die Systemleistung kann wesentlich verringert werden, wenn zu viel Zeit mit ISRs verbracht wird, da auf einigen Geräten Interrupts verloren gehen oder Pufferüberläufe auftreten können. Ein weiterer wichtiger Aspekt ist, dass die ISR im Kernelmodus ausgeführt wird und nicht auf die Betriebssystem-APIs einer höheren Ebene zugreifen kann. Deshalb führen ISRs üblicherweise nur Basisaufgaben aus, beispielsweise das Kopieren von Daten aus den Hardwareregistern in die Speicherpuffer. In Windows Embedded CE wird die zeitaufwendige Interruptverarbeitung normalerweise in einem IST ausgeführt.

Die ISR bestimmt die Interruptquelle, maskiert oder demaskiert den Interrupt auf dem Gerät und gibt einen SYSINTR-Wert für den Interrupt zurück. Die ISR gibt *SYSINTR_NOP* zurück, um anzuzeigen, dass keine weitere Verarbeitung erforderlich ist. Der Kernel signalisiert folglich das Event für einen IST nicht, um den Interrupt zu verarbeiten. Wenn der Gerätetreiber jedoch einen IST verwendet, um den Interrupt zu verarbeiten, übergibt die ISR die logische Interrupt-ID an den Kernel. Der Kernel bestimmt und signalisiert das Interruptevent und der IST, der nach dem Aufruf der Funktion *WaitForSingleObject* fortgesetzt wird, führt die Verarbeitungsanweisungen in einer Schleife aus. Die Latenz zwischen der ISR und dem IST hängt von der Priorität des Threads und den anderen ausgeführten Threads ab (siehe Kapitel 3). ISTs werden normalerweise mit einer höheren Priorität ausgeführt.

Interrupt Service Threads

Ein IST ist ein regulärer Thread, der bei einem Interrupt zusätzliche Verarbeitungsvorgänge ausführt, nachdem die ISR beendet wurde. Die IST-Funktion umfasst eine Schleife und einen *WaitForSingleObject*-Aufruf, um den Thread unbegrenzt zu blockieren, bis der Kernel das angegebene IST-Event signalisiert. Bevor Sie das IST-Event verwenden können, müssen Sie die Funktion *InterruptInitialize* mit dem SYSINTR-Wert und einem Eventhandle als Parameter aufrufen, damit der CE-Kernel das Event ankündigen kann, wenn eine ISR den SYSINTR-Wert zurückgibt. In Kapitel 3 finden Sie detaillierte Informationen zur Multithread-Programmierung und Threadsynchronisierung basierend auf Events und anderen Kernelobjekten.

```
 CeSetThreadPriority(GetCurrentThread(), 200);

// Loop until told to stop
while(!pIst->stop)
{
    // Wait for the IST event.
    WaitForSingleObject(pIst->hevIrq, INFINITE)

    // Handle the interrupt.
    InterruptDone(pIst->sysIntr);
}
```

Nachdem der IST die IRQ-Verarbeitung abgeschlossen hat, muss er die Funktion *InterruptDone* aufrufen, um das System darüber zu informieren, dass der Interrupt verarbeitet wurde, der IST den nächsten IRQ verarbeiten kann und der Interrupt mit der Funktion *OEMInterruptDone* erneut aktiviert werden kann. In Tabelle 6.6 sind die OAL-Funktionen aufgeführt, die das System verwendet, um mit dem Interruptcontroller zu kommunizieren.

Tabelle 6.6 OAL-Funktionen für die Interruptverwaltung

Funktion	Beschreibung
OEMInterruptEnable	Wird vom Kernel als Antwort auf *InterruptInitialize* aufgerufen und aktiviert den angegebenen Interrupt im Interruptcontroller.
OEMInterruptDone	Wird vom Kernel als Antwort auf *InterruptDone* aufgerufen und demaskiert und bestätigt den Interrupt im Interruptcontroller.

Tabelle 6.6 OAL-Funktionen für die Interruptverwaltung (Fortsetzung)

Funktion	Beschreibung
OEMInterruptDisable	Deaktiviert den Interrupt im Interruptcontroller und wird als Antwort auf *InterruptDisable* aufgerufen.
OEMInterruptHandler	Identifiziert für ARM-Prozessoren den Interrupt-SYSINTR, der beim Abrufen des Status des Interruptcontrollers auftritt.
HookInterrupt	Registriert für andere Prozessoren als ARM eine Callback-Funktion für eine Interrupt-ID. Diese Funktion muss in der *OEMInit*-Funktion aufgerufen werden, um erforderliche Interrupts zu registrieren.
OEMInterruptHandlerFIQ	Wird für ARM-Prozessoren verwendet, um Interrupts für FIQ (Fast Interrupt) zu verarbeiten.

ACHTUNG **WaitForMultipleObjects-Einschränkung**

Verwenden Sie nicht die Funktion *WaitForMultipleObjects*, um auf ein Interruptevent zu warten. Wenn Sie auf mehrere Interruptevents warten müssen, erstellen Sie für jeden Interrupt einen IST.

Interrupt-IDs (IRQ und SYSINTR)

Jede Hardware-Interruptzeile entspricht einem IRQ-Wert in den Interruptcontroller-Registern. Jeder IRQ-Wert kann nur jeweils einer ISR zugewiesen werden, aber eine ISR kann mehrere IRQs zuordnen. Der Kernel muss die IRQs nicht verwalten, sondern lediglich die Events mit SYSINTR-Werten, die von der ISR für den IRQ zurückgegeben werden, bestimmen und signalisieren. Die Möglichkeit verschiedene SYSINTR-Werte von einer ISR zurückzugeben, ist die Basis für die Unterstützung mehrerer Geräte, die den gleichen Interrupt verwenden.

HINWEIS ***OEMInterruptHandler*** **und** ***HookInterrupt***

Zielgeräte, die nur einen einzigen IRQ unterstützen, beispielsweise ARM-Systeme, verwenden die Funktion *OEMInterruptHandler* als die ISR, um das eingebettete Peripheriegerät zu identifizieren, das den Interrupt ausgelöst hat. OEMs müssen diese Funktion als Bestandteil des OAL implementieren. Auf Plattformen, die mehrere IRQs unterstützen, beispielsweise Intel x86-Systeme, können Sie die IRQs individuellen ISRs zuweisen, indem Sie die Funktion *HookInterrupt* aufrufen.

Statische Interruptzuordnungen

Damit die ISR einen korrekten SYSINTR-Rückgabewert bestimmen kann, muss eine Zuordnung zwischen dem IRQ und der SYSINTR vorhanden sein, die in der OAL hartcodiert sein kann. Die Datei *Bsp_cfg.h* für das Geräteemulator-BSP veranschaulicht die Definition eines SYSINTR-Werts im OAL für ein Zielgerät in Bezug auf den *SYSINTR_FIRMWARE*-Wert. Beachten Sie beim Definieren weiterer IDs im OAL für ein benutzerdefiniertes Zielgerät, dass der Kernel alle Werte unter *SYSINTR_FIRMWARE* für die künftige Verwendung reserviert und der maximale Wert geringer als *SYSINTR_MAXIMUM* sein sollte.

Um den IRQs auf einem Zielgerät statische SYSINTR-Werte zuzuweisen, rufen Sie während der Systeminitialisierung die Funktion *OALIntrStaticTranslate* auf. Beispielsweise ruft das Geräteemulator-BSP die Funktion *OALIntrStaticTranslate* in der Funktion *BSPIntrInit* auf, um einen angepassten SYSINTR-Wert für die integrierte OHCI-Schnittstelle (Open Host Controller Interface) in den Interrupt-Zuordnungsarrays des Kernels (*g_oalSysIntr2Irq* und *g_oalIrq2SysIntr*) zu registrieren. Statische SYSINTR-Werte und Zuordnungen sind jedoch keine übliche Methode zum Zuweisen von IRQs mit SYSINTRs, da dies schwierig ist und OAL-Codeänderungen erfordert, um die benutzerdefinierte Interruptverarbeitung zu implementieren. Statische SYSINTR-Werte werden normalerweise nur für die wichtigsten Hardwarekomponenten eines Zielgeräts verwendet, wenn kein expliziter Gerätetreiber vorhanden ist und die ISR im OAL implementiert ist.

Dynamische Interruptzuordnungen

Sie müssen die SYSINTR-Werte im OAL nicht hartcodieren, wenn Sie die Funktion *KernelIoControl* in den Gerätetreibern mit dem IOCTL-Code *IOCTL_HAL_REQUEST_SYSINTR* aufrufen, um die IRQ/SYSINTR-Zuordnungen zu registrieren. Der Aufruf endet mit der Funktion *OALIntrRequestSysIntr*, die einen neuen SYSINTR für den angegebenen IRQ dynamisch zuweist und die IRQ- und SYSINTR-Zuordnungen anschließend in den Interrupt-Zuordnungsarrays des Kernels registriert. Das Suchen eines verfügbaren SYSINTR-Werts bis zu *SYSINTR_MAXIMUM* ist flexibler als statische SYSINTR-Zuweisungen, da diese Methode keine OAL-Änderungen erfordert, wenn Sie neue Treiber zum BSP hinzufügen.

Wenn *KernelIoControl* mit *IOCTL_HAL_REQUEST_SYSINTR* aufgerufen wird, wird eine 1:1-Beziehung zwischen dem IRQ und SYSINTR erstellt. Sollte die IRQ-SYSINTR-Zuordnungstabelle bereits einen Eintrag für den angegebenen IRQ enthalten, erstellt *OALIntrRequestSysIntr* keinen zweiten Eintrag. Um einen Eintrag

aus den Interrupt-Zuordnungstabellen zu entfernen (beispielsweise beim Entladen eines Treibers), rufen Sie *KernelIoControl* mit dem IOCTL-Code *IOCTL_HAL_REQUEST_SYSINTR* auf. *IOCTL_HAL_RELEASE_SYSINTR* hebt die Zuordnung des IRQ im SYSINTR-Wert auf.

Das folgende Codesegment veranschaulicht die Verwendung von *IOCTL_HAL_REQUEST_SYSINTR* und *IOCTL_HAL_RELEASE_SYSINTR*. Der benutzerdefinierte Wert (*dwLogintr*) wird an den OAL übergeben, der diesen Wert in einen SYSINTR-Wert umwandelt, der einem IST-Event zugewiesen wird.

```
DWORD dwLogintr = IRQ_VALUE;
DWORD dwSysintr = 0;
HANDLE hEvent   = NULL;
BOOL bResult    = TRUE;

// Create event to associate with the interrupt
m_hEvent = CreateEvent(NULL,FALSE,FALSE,NULL);
if (m_hDetectionEvent == NULL)
{
    return ERROR_VALUE;
}

// Ask the kernel (OAL) to associate an SYSINTR value to an IRQ
bResult = KernelIoControl(IOCTL_HAL_REQUEST_SYSINTR,
                          &dwLogintr, sizeof(dwLogintr),
                          &dwSysintr, sizeof(dwSysintr),
                          0);
if (bResult == FALSE)
{
    return ERROR_VALUE;
}

// Initialize interrupt and associate the SYSINTR value with the event.
bResult = InterruptInitialize(dwSysintr, hEvent,0,0);

if (bResult == FALSE)
{
    return ERROR_VALUE;
}

// Interrupt management loop
while(!m_bTerminateDetectionThread)
{
    // Wait for the event associated to the interrupt
    WaitForSingleObject(hEvent,INFINITE);

    // Add actual IST processing here

    // Acknowledge the interrupt
    InterruptDone(m_dwSysintr);
```

```
}


// Deinitialize interrupts will mask the interrupt
bResult = InterruptDisable(dwSysintr);

// Unregister SYSINTR
bResult = KernelIoControl(IOCTL_HAL_RELEASE_SYSINTR,
                          &dwSysintr, sizeof(dwSysintr),
                          NULL,0,
                          0);

// Close the event object
CloseHandle(hEvent);
```

Freigegebene Interruptzuordnungen

Die 1:1-Beziehung zwischen IRQ und SYSINTR bedeutet, dass Sie für einen IRQ nicht mehrere ISRs direkt registrieren können, um die Interruptfreigabe zu implementieren. Sie können mehrere ISRs jedoch indirekt zuordnen. Die Interrupt-Zuordnungstabellen ordnen einen ISR nur einer statischen ISR zu. In dieser ISR können Sie jedoch die Funktion *NKCallIntChain* aufrufen, um die ISRs zu durchlaufen, die über *LoadIntChainHandler* dynamisch registriert wurden. *NKCallIntChain* durchläuft die für den gemeinsam genutzten Interrupt registrierten ISRs und gibt den ersten SYSINTR-Wert zurück, der ungleich *SYSINTR_CHAIN* ist. Nachdem der geeignete SYSINTR-Wert für die aktuelle Interruptquelle bestimmt wurde, kann die statische ISR die logische Interrupt-ID an den Kernel übergeben, um das entsprechende IST-Event zu signalisieren. Die Funktion *LoadIntChainHandler* und installierbare ISRs werden später in dieser Lektion ausführlich beschrieben.

Kommunikation zwischen einer ISR und einem IST

Da die ISR und der IST zu verschiedenen Zeitpunkten und in verschiedenen Kontexten ausgeführt werden, müssen Sie die physischen und virtuellen Zuordnungen beachten, wenn eine ISR Daten an den IST übergibt. Eine ISR kopiert möglicherweise einzelne Bytes von einem Peripheriegerät in einen Eingabepuffer und gibt *SYSINTR_NOP*-Werte zurück, bis der Puffer voll ist. Die ISR gibt den tatsächlichen SYSINTR-Wert nur zurück, wenn der Eingabepuffer für den IST bereit ist. Der Kernel signalisiert das entsprechende IST-Event und der IST kopiert die Daten in einen Prozesspuffer.

Eine Methode zum Übertragen der Daten ist das Reservieren eines physischen Speicherbereichs in einer .bib-Datei. Die Datei *Config.bib* enthält mehrere Beispiele für

serielle Treiber und Debugtreiber. Die ISR kann die Funktion *OALPAtoVA* aufrufen, um die physische Adresse des reservierten Speicherbereichs in eine virtuelle Adresse umzuwandeln. Da die ISR im Kernelmodus ausgeführt wird, kann sie auf den reservierten Speicher zugreifen, um Daten vom Peripheriegerät zu puffern. Der IST ruft die Funktion *MmMapIoSpace* außerhalb des Kernels auf, um den physischen Speicher einer prozesspezifischen virtuellen Adresse zuzuordnen. *MmMapIoSpace* verwendet die Funktionen *VirtualAlloc* und *VirtualCopy*, um den physischen Speicher auf virtuellen Speicher abzubilden. Sie können *VirtualAlloc* und *VirtualCopy* jedoch direkt aufrufen, wenn Sie mehr Kontrolle über den Adresszuordnungsprozess benötigen.

Eine weitere Möglichkeit die Daten von einer ISR an einen IST zu übergeben, ist das dynamische Zuordnen von physischem Speicher im SDRAM unter Verwendung der Funktionen im Gerätetreiber. Diese Methode ist insbesondere für installierbare ISRs nützlich, die bei Bedarf im Kernel geladen werden. *AllocPhysMem* ordnet einen physischen zusammenhängenden Speicherbereich zu und gibt die physische Adresse zurück (oder schlägt fehl, wenn die zugeordnete Größe nicht verfügbar ist). Der Gerätetreiber kann über die Funktion *KernelIoControl* basierend auf einem benutzerdefinierten IOCTL-Code die physische Adresse an die ISR weitergeben. Die ISR wandelt unter Verwendung der Funktion *OALPAtoVA* die physische Adresse in eine virtuelle Adresse um. Der IST verwendet *MmMapIoSpace* oder die Funktionen *VirtualAlloc* und *VirtualCopy* wie für statisch reservierte Speicherbereiche.

Installierbare ISRs

Beachten Sie beim Anpassen von Windows Embedded CE für ein benutzerdefiniertes Zielgerät, dass der OAL so allgemein wie möglich gehalten werden sollte. Ansonsten müssen Sie den Code jedes Mal ändern, wenn Sie eine neue Komponente zum System hinzufügen. Um die Flexibilität und Anpassbarkeit sicherzustellen, unterstützt Windows Embedded CE installierbare ISRs (IISR), um Gerätetreiber bei Bedarf in den Kernelbereich zu laden, beispielsweise wenn Plug & Play-Peripheriegeräte angeschlossen werden. Installierbare ISRs sind außerdem eine Lösung für Prozessinterrupts, wenn mehrere Hardwaregeräte den gleichen Interrupt verwenden. Die ISR-Architektur hängt von DLLs ab, die den Code für die installierbare ISR enthalten und die Einsprungspunkte exportieren (siehe Tabelle 6.7).

Tabelle 6.7 Exportierte installierbare ISR DLL-Funktionen

Funktion	Beschreibung
ISRHandler	Diese Funktion enthält den installierbaren Interrupthandler. Der zurückgegebene Wert ist der SYSINTR-Wert für den IST, den Sie als Antwort auf den für die installierbare ISR registrierten IRQ mit der Funktion *LoadIntChainHandler* ausführen können. Der OAL muss das Verketten für mindestens diesen IRQ unterstützen. Das heißt, dass ein nicht verarbeiteter Interrupt mit einem weiteren Handler verkettet werden kann, wenn ein Interrupt auftritt.
CreateInstance	Diese Funktion wird aufgerufen, wenn eine installierbare ISR mit der Funktion *LoadIntChainHandler* geladen wird. Die Funktion gibt eine Instanzen-ID für die ISR zurück.
DestroyInstance	Diese Funktion wird aufgerufen, wenn eine installierbare ISR mit der Funktion *FreeIntChainHandler* entladen wird.
IOControl	Diese Funktion unterstützt die Kommunikation des IST mit der ISR.

HINWEIS Generische installierbare ISR (GIISR)

Um das Implementieren von installierbaren ISRs zu unterstützen, stellt Microsoft ein generisches IIS-Beispiel bereit, dass die meisten Anforderungen vieler Geräte erfüllt. Der Quellcode ist im folgenden Ordner gespeichert: *%_WINCEROOT%\Public\Common\Oak\Drivers\Giisr*.

Registrieren einer IISR

Die Funktion *LoadIntChainHandler* erwartet drei Parameter, die Sie angeben müssen, um eine installierbare ISR zu laden und zu registrieren. Der erste Parameter (*lpFilename*) gibt den Dateinamen der zu ladenden ISR DLL an. Der zweite Parameter (*lpszFunctionName*) gibt den Namen der Interrupthandlerfunktion an, und der dritte Parameter (*bIRQ*) definiert den IRQ, für den die installierbare ISR registriert werden soll. Als Antwort auf eine getrennte Hardwarekomponente kann ein Gerätetreiber eine installierbare ISR über die Funktion *FreeIntChainHandler* entladen.

Externe Abhängigkeiten und installierbare ISRs

Beachten Sie, dass *LoadIntChainHandler* die ISR DLLs in den Kernelbereich lädt. Das heißt, dass die installierbare ISR keine Betriebsystem-APIs einer höheren Ebene aufrufen und andere DLLs weder importieren noch implizit linken kann. Wenn die DLL explizit oder implizit mit anderen DLLs gelinkt ist oder die C-Laufzeitbibliothek verwendet, kann die DLL nicht geladen werden. Die installierbare ISR muss vollständig unabhängig sein.

Um sicherzustellen, dass eine installierbare ISR nicht mit den C-Laufzeitbibliotheken gelinkt ist, fügen Sie folgende Zeilen in der Sources-Datei im DLL-Teilprojekt ein:

```
NOMIPS16CODE=1
NOLIBC=1
```

Die Direktive **NOLIBC=1** stellt sicher, dass die C-Laufzeitbibliotheken nicht gelinkt sind. Die Option **NOMIPS16CODE=1** aktiviert die Compileroption **/QRimplicit-import**, die implizite Links zu anderen DLLs verhindert. Beachten Sie, dass diese Direktive absolut nichts mit der MIPS-CPU (Mikroprozessor ohne Interlocked Pipeline Stages) zu tun hat.

Zusammenfassung

Windows Embedded CE hängt von ISRs und ISTs ab, um die Interruptanforderungen von internen und externen Hardwarekomponenten zu erfüllen, die von der CPU neben der normalen Codeausführung verarbeitet werden müssen. ISRs werden normalerweise direkt im Kernel kompiliert oder in den beim Start geladenen Gerätetreibern implementiert und mit den entsprechenden IRQs über *HookInterrupt*-Aufrufe registriert. Sie können installierbare ISRs jedoch auch in ISR DLLs implementieren, die der Gerätetreiber bei Bedarf laden und mit der Funktion einem IRQ zuweisen kann. Installierbare ISRs ermöglichen außerdem die Unterstützung der Interruptfreigabe. Beispielsweise können Sie auf einem System mit nur einem IRQ, z.B. einem ARM-Gerät, die Funktion *OEMInterruptHandler* ändern, bei der es sich um eine statische ISR handelt, die weitere installierbare ISRs lädt (abhängig von der Hardwarekomponente, die den Interrupt auslöst).

Außer dem Unterschied, dass ISR DLLs nicht von externem Code abhängig sein dürfen, weisen ISRs und installierbare ISRs viele Ähnlichkeiten auf. Der Interrupthandler bestimmt die Interruptquelle, maskiert oder demaskiert den Interrupt auf dem Gerät und gibt einen SYSINTR-Wert für den IRQ zurück, um den Kernel über den ausgeführten IST zu benachrichtigen. Windows Embedded CE

verwaltet Interrupt-Zuordnungstabellen, die IRQs den SYSINTR-Werten zuweisen. Sie können im Quellcode statische SYSINTR-Werte definieren oder dynamische SYSINTR-Werte verwenden, die vom Kernel zur Laufzeit angefordert werden. Mit dynamischen SYSINTR-Werten können Sie die Portabilität Ihrer Anwendungen verbessern.

Entsprechend dem SYSINTR-Wert kann der Kernel ein IST-Event signalisieren, der ermöglicht, dass der Interrupt-Dienstthread nach dem Aufruf der Funktion *WaitForSingleObject* fortgesetzt wird. Wenn Sie die meisten IRQ-Verarbeitungsvorgänge im IST anstatt der ISR ausführen, wird eine optimale Systemleistung erreicht, da das System die Interruptquellen mit gleicher oder niedrigerer Priorität nur während der ISR-Ausführung blockiert. Nach Abschluss der ISR demaskiert der Kernel alle Interrupts, mit Ausnahme des Interrupts der gerade verarbeitet wird. Die aktuelle Interruptquelle bleibt blockiert, damit ein neuer Interrupt auf dem gleichen Gerät das aktuelle Interrupt-Verarbeitungsverfahren nicht stört. Wenn der IST abgeschlossen ist, muss der IST die Funktion *InterruptDone* aufrufen, um den Kernel zu benachrichtigen, dass er einen neuen Interrupt verarbeiten kann. Der Interrupthandler des Kernels kann anschließend den IRQ im Interruptcontroller erneut aktivieren.

Lektion 5: Implementieren der Energieverwaltung für einen Gerätetreiber

Wie in Kapitel 3 erklärt, ist die Energieverwaltung für Windows Embedded CE-Geräte unentbehrlich. Das Betriebssystem umfasst die Kernelkomponente *PM.dll* (Power Manager), die mit dem Geräte-Manager integriert ist, um Geräten das Verwalten ihrer Energiezustände und Anwendungen das Festlegen der Energiezustände für bestimmte Geräte zu ermöglichen. Die Hauptaufgabe des Power Managers ist das Optimieren des Energieverbrauchs und das Bereitstellen eines APIs für Systemkomponenten, Treiber und Anwendungen, um Energiebenachrichtigungen und die Energiesteuerung zu aktivieren. Obwohl der Power Manager keine strikten Anforderungen an den Energieverbrauch oder die Funktionen in einem bestimmten Energiestatus erzwingt, ist er hilfreich, um Energieverwaltungsfeatures zu einem Gerätetreiber hinzuzufügen, damit Sie den Status der Hardwarekomponenten mit einer Methode verwalten können, die dem Energiestatus des Zielgeräts entspricht. Weitere Informationen zum Power Manager, zu den Energiezuständen von Geräten und Systemen sowie den in Windows Embedded CE 6.0 unterstützten Energieverwaltungsfeatures finden Sie in Kapitel 3 „Systemprogrammierung."

Nach Abschluss dieser Lektion können Sie:

- Die Energieverwaltungsschnittstelle für Gerätetreiber identifizieren.
- Die Energieverwaltung in einem Gerätetreiber implementieren.

Veranschlagte Zeit für die Lektion: 30 Minuten.

Power Manager-Gerätetreiberschnittstelle

Der Power Manager kommuniziert mit den Treibern zum Zwecke der Energieverwaltung über die Funktionen *XXX_PowerUp*, *XXX_PowerDown* und *XXX_IOControl*. Der Gerätetreiber kann über die Funktion *DevicePowerNotify* eine Änderung des Geräteenergiestatus vom Power Manager anfordern. Der Power Manager ruft *XXX_IOControl* mit dem IOcontrol-Code *IOCTL_POWER_SET* auf und übergibt den angeforderten Geräteenergiestatus. Es kann den Anschein erwecken, dass es für einen Gerätetreiber zu kompliziert ist, den Energiestatus seines Gerät über den Power Manager zu ändern, aber dieses Verfahren stellt ein konsistentes Verhalten und Benutzerfreundlichkeit sicher. Wenn eine Anwendung einen bestimmten Energiestatus für ein Gerät anfordert, ruft der Power Manager den *IOCTL_POWER_SET*-Handler nach dem Aufruf der Funktion *DevicePowerNotify*

möglicherweise nicht auf. Gerätetreiber sollten deshalb nicht voraussetzen, dass erfolgreiche Aufrufe von *DevicePowerNotify* in einem Aufruf des *IOCTL_POWER_SET*-Handlers resultieren, oder dass *IOCTL_POWER_SET*-Aufrufe das Ergebnis eines *DevicePowerNotify*-Aufrufs sind. Der Power Manager kann Benachrichtigungen an einen Gerätetreiber senden, beispielsweise während eines Statusübergangs der Systemenergie. Um Energiebenachrichtigungen zu erhalten, muss der Gerätetreiber ankündigen, dass er für die Energieverwaltung aktiviert ist (entweder statisch über den Registrierungseintrag *IClass* im Unterschlüssel des Treibers oder dynamisch über die Funktion *AdvertiseInterface*.

XXX_PowerUp und XXX_PowerDown

Sie können die Streamschnittstellenfunktionen *XXX_PowerUp* und *XXX_PowerDown* verwenden, um Funktionen auszusetzen bzw. fortzusetzen. Der Kernel ruft *XXX_PowerDown* auf, bevor die CPU heruntergefahren wird. *XXX_PowerUp* wird aufgerufen, nachdem die CPU aktiviert wurde. Beachten Sie, dass das System in diesen Phasen im Singlethread-Modus ausgeführt wird und die meisten Systemaufrufe deaktiviert sind. Aus diesem Grund empfiehlt Microsoft die Verwendung der Funktion *XXX_IOControl* anstatt der Funktionen *XXX_PowerUp* und *XXX_PowerDown* zum Implementieren der Energieverwaltungsfeatures, einschließlich der Features zum Aussetzen und Fortsetzen des Geräts.

ACHTUNG Einschränkungen der Energieverwaltung

Wenn Sie Funktionen zum Aussetzen und Fortsetzen basierend auf den Funktionen *XXX_PowerUp* und *XXX_PowerDown* implementieren, sollten Sie keine System-APIs aufrufen. Vermeiden Sie insbesondere APIs, die Threads blockieren, beispielsweise *WaitForSingleObject*. Das Blockieren eines aktiven Threads im Singlethread-Modus verursacht einen nicht behebbaren Systemfehler.

IOControl

Die beste Methode zum Implementieren des Power Managers in einem Streamtreiber ist das Hinzufügen der Unterstützung für E/A-Energieverwaltungscodes zur Funktion *IOControl* des Treibers. Wenn der Power Manager über einen IClass-Registrierungseintrag oder die Funktion *AdvertiseInterface* über die Energieverwaltungsfunktionen des Treibers informiert wird, erhält der Treiber entsprechende Benachrichtigungen.

In Tabelle 6.8 sind die IOCTLs aufgeführt, die der Power Manager an einen Gerätetreiber senden kann, um Energieverwaltungsaufgaben auszuführen.

Tabelle 6.8 Energieverwaltungs-IOCTLs

Funktion	Beschreibung
IOCTL_POWER_CAPABILITIES	Fordert Informationen über die vom Treiber unterstützten Energiezustände an. Beachten Sie, dass der Treiber auf einen anderen Energiestatus festgelegt werden kann (D0 bis D4).
IOCTL_POWER_GET	Ruft den aktuellen Energiestatus des Treibers ab.
IOCTL_POWER_SET	Legt den Energiestatus des Treibers fest. Der Treiber ordnet den zurückgegebenen Energiestatuswert den Einstellungen zu und ändert den Gerätestatus. Der neue Energiestatus des Gerätetreibers sollte im Ausgabepuffer an den Power Manager zurückgegeben werden.
IOCTL_POWER_QUERY	Der Power Manger überprüft, ob der Treiber den Status des Geräts ändern kann. Diese Funktion ist veraltet.
IOCTL_REGISTER_POWER_RELATIONSHIP	Ermöglicht einem Gerätetreiber das Registrieren als Proxy für einen anderen Gerätetreiber, damit der Power Manager alle Energieanforderungen an diesen Gerätetreiber übergibt.

IClass Energieverwaltungsschnittstellen

Der Power Manager unterstützt einen *IClass*-Registrierungseintrag, den Sie im Unterschlüssel des Treibers konfigurieren können, um dem Treiber einen oder mehrere Geräteklassenwerte zuzuweisen. Der *IClass*-Wert entspricht einer GUID, die auf eine unter dem Registrierungsschlüssel *HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Power\Interfaces* definierte Schnittstelle verweist. Die wichtigste Schnittstelle für Treiberentwickler ist die Schnittstelle für generische Geräte mit der GUID {*A32942B7-920C-486b-B0E6-92A702A99B35*}, die für die Energiewaltung aktiviert sind. Wenn Sie diese GUID zum *IClass*-Registrierungseintrag des Gerätetreibers hinzufügen, sendet der Power Manager IOCTLs für Energiebenachrichtigungen an den Treiber (siehe Abbildung 6.7).

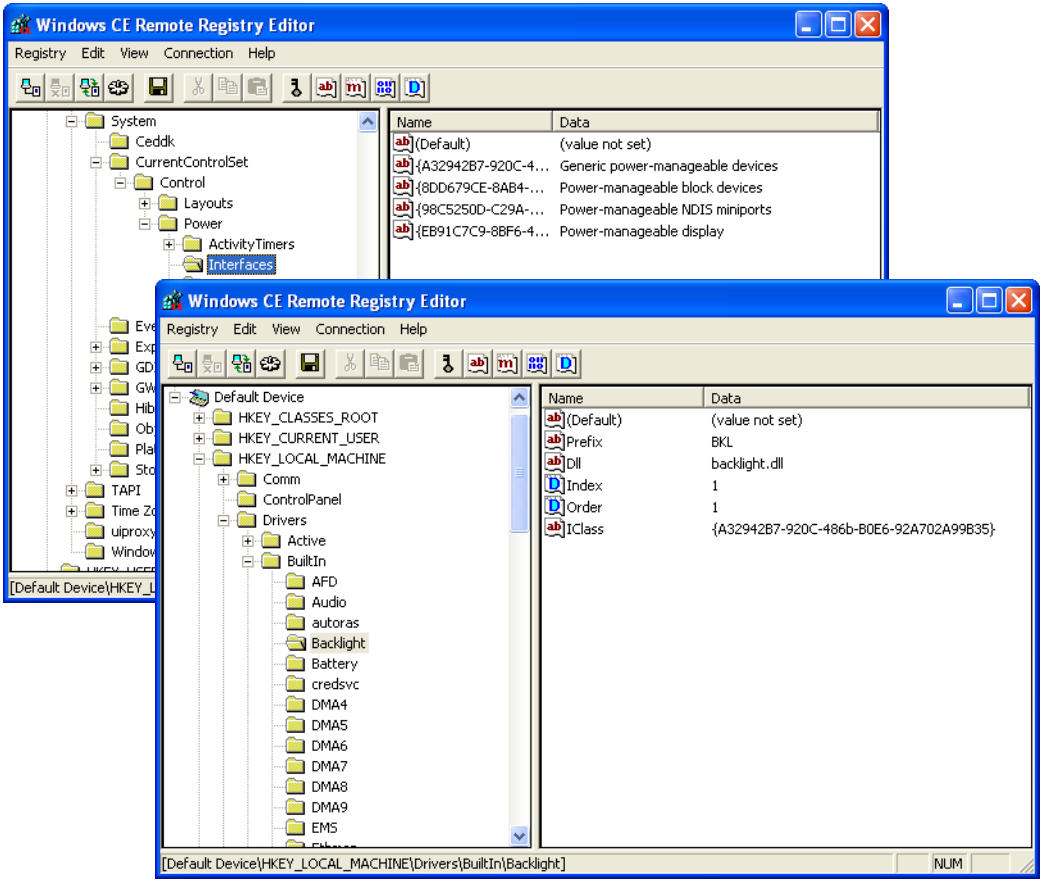


Abbildung 6.7 Konfigurieren des IClass-Registrierungseintrags zum Abrufen von Energieverwaltungsbenachrichtigungen.

WEITERE INFORMATIONEN Registrierungseinstellungen für die Energieverwaltung

Sie können den Standardenergiestatus für ein Gerät auch über die Registrierungseinstellungen und Geräteklassen konfigurieren (siehe Lektion 5 Implementieren der Energieverwaltung in Kapitel 3).

Zusammenfassung

Um in Windows Embedded CE die zuverlässige Energieverwaltung sicherzustellen, sollten die Gerätetreiber ihren internen Energiestatus nicht ohne Unterstützung durch den Power Manager wechseln. Betriebssystemkomponenten, Treiber und Anwendungen können die Funktion *DevicePowerNotify* aufrufen, um eine Energiestatusänderung anzufordern. Der Power Manager sendet daraufhin eine Anforderung zur Energiestatusänderung an den Treiber, wenn der Energiestatus dem aktuellen Status des Systems entspricht. Die empfohlene Methode zum Hinzufügen von Energieverwaltungsfunktionen zu einem Streamtreiber ist das Hinzufügen der Unterstützung für Energieverwaltungs-IOCTLs zur *XXX_IOControl*-Funktion. Die Funktionen *XXX_PowerUp* und *XXX_PowerDown* stellen nur begrenzte Möglichkeiten bereit, da der Power Manager diese Funktionen aufruft, wenn das System im Singlethread-Modus ausgeführt wird. Wenn der Treiber über einen *IClass*-Registrierungseintrag eine Energieverwaltungsschnittstelle ankündigt oder *AdvertiseInterface* aufruft, um unterstützte IOCTL-Schnittstellen dynamisch anzukündigen, sendet der Power Manager beim Auftreten von Energieevents IOCTLs an den Gerätetreiber.

Lektion 6: Grenzübergreifendes Marshalling von Daten

In Windows Embedded CE 6.0 verfügt jeder Prozess über einen separaten virtuellen Speicherbereich und Speicherkontext. Deshalb erfordert das Marshalling von Daten zwischen Prozessen einen Kopierprozess oder die Zuordnung von physischen Speicherbereichen. Windows Embedded CE 6.0 verarbeitet die meisten Informationen und stellt Systemfunktionen bereit, beispielsweise *OALPAtoVA* und *MmMapIoSpace*, um physische Speicheradressen virtuellen Speicheradressen zuzuordnen. Ein Treiberentwickler muss jedoch mit den Details des Marshallings von Daten vertraut sein, um sicherzustellen, dass das System zuverlässig und sicher arbeitet. Die Überprüfung von eingebetteten Zeigern und die richtige Handhabung des asynchronen Pufferzugriffs ist unbedingt erforderlich, damit Benutzeranwendungen Kernelmodustreiber nicht zum Ändern von Speicherregionen missbrauchen können, auf die die Anwendung keinen Zugriff haben sollte. Schlecht implementierte Kernelmodustreiber können eine Hintertür für bösartige Anwendungen öffnen, um die Kontrolle über das gesamte System zu übernehmen.

Nach Abschluss dieser Lektion können Sie:

- Puffer in Gerätetreibern reservieren und verwenden.
- Eingebettete Zeiger in einer Anwendung verwenden.
- Die eingebetteten Zeiger in einem Gerätetreiber überprüfen.

Veranschlagte Zeit für die Lektion: 30 Minuten.

Speicherzugriff

Windows Embedded CE wird in einem virtuellen Speicherkontext ausgeführt, der den physischen Speicher überlagert (siehe Abbildung 6.8). Das Betriebssystem verwendet VMM (Virtual Memory Manager) und die MMU (Memory Management Unit) des Prozessors für die Umwandlung von virtuellen Adressen in physische Adressen und zum Ausführen anderer Verwaltungsaufgaben für den Speicherzugriff.

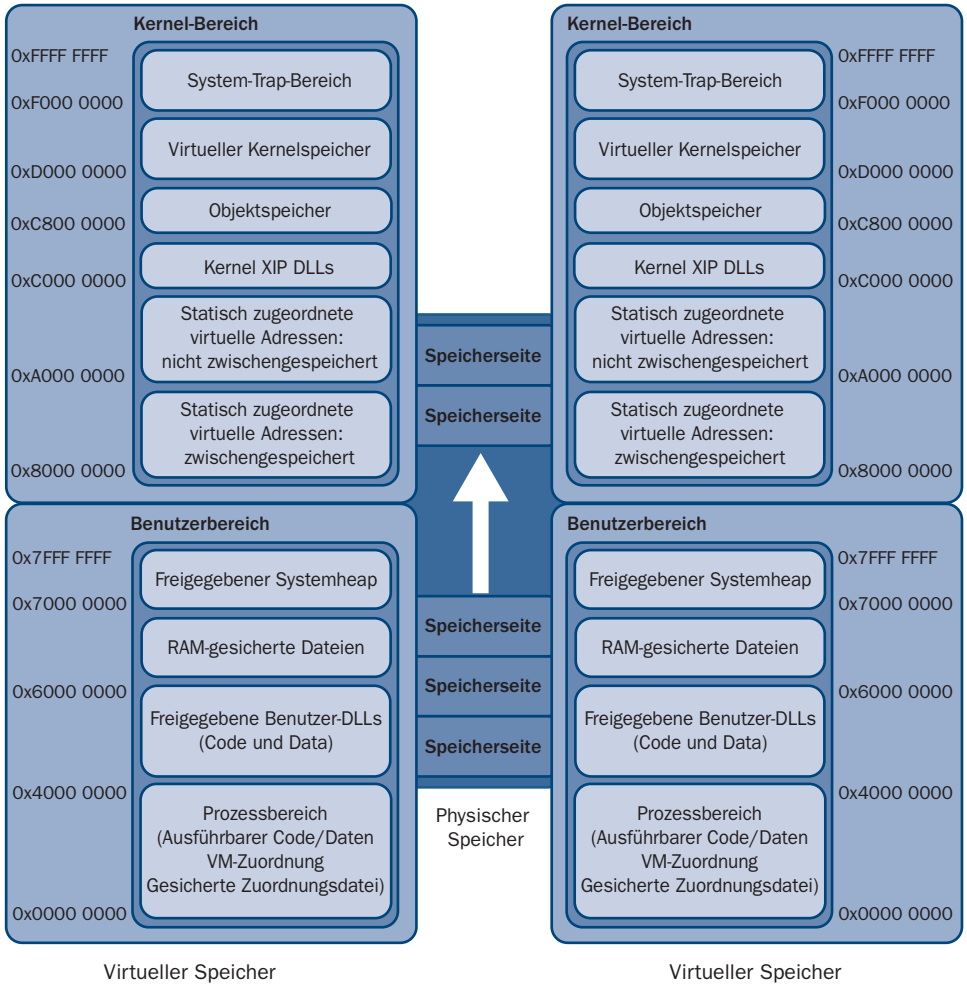


Abbildung 6.8 Virtuelle Speicherbereiche im Kernel- und Benutzerbereich

Physische Adressen können von der CPU ausschließlich während der Initialisierung verwendet werden, bevor der Kernel die MMU aktiviert. Das heißt jedoch nicht, dass auf den physischen Speicher nicht mehr zugegriffen werden kann. Jede virtuelle Speicherseite muss einer physischen Seite auf dem Zielgerät entsprechen. Die Prozesse in separaten virtuellen Adressbereichen erfordern eine Methode zum Zuordnen der gleichen physischen Speicherbereiche zu einem verfügbaren virtuellen Speicherbereich, um auf Daten gemeinsam zugreifen zu können. Die physische Adresse ist für alle auf dem System ausgeführten Prozesse identisch. Nur die virtuellen Adressen sind unterschiedlich. Durch Umwandeln der physischen Adresse

eines Prozesses in eine gültige virtuelle Adresse, können Prozesse auf den gleichen physischen Speicherbereich zugreifen und Daten prozessübergreifend nutzen.

Wie bereits erwähnt, können Kernelmodusroutinen, beispielsweise ISRs, *OALPAtoVA* aufrufen, um eine physische Adresse (PA) einer zwischengespeicherten oder nicht zwischengespeicherten virtuellen Adresse (VA) zuzuordnen. Da *OALPAtoVA* die physische Adresse einer virtuellen Adresse im Kernelbereich zuordnet, können Benutzermodusprozesse, beispielsweise ISTs, diese Funktion nicht verwenden. Auf den Kernelbereich kann im Benutzermodus nicht zugegriffen werden. Threads in Benutzermodusprozessen, beispielsweise ein IST, können die Funktion *MmMapIoSpace* aufrufen, um eine physische Adresse einer nicht ausgelagerten virtuellen Adresse im Benutzerbereich zuzuordnen. Die Funktion *MmMapIoSpace* erstellt einen neuen Eintrag in der MMU-Tabelle (TBL), wenn keine Übereinstimmung gefunden wird oder gibt eine vorhandene Zuordnung zurück. Wenn die Funktion *MmUnmapIoSpace* aufgerufen wird, kann der Benutzermodusprozess den Speicher wieder freigeben.

HINWEIS Einschränkungen des physischen Speicherzugriffs

Anwendungen und Benutzermodustreiber können nicht direkt auf den physischen Gerätespeicher zugreifen. Benutzermodusprozesse müssen die Funktion *HalTranslateBusAddress* aufrufen, um vor dem Aufruf der Funktion den physischen Gerätespeicher für den Bus einer physischen Systemspeicheradresse zuzuordnen. Um in nur einem Funktionsaufruf eine Busadresse in eine virtuelle Adresse zu konvertieren, verwenden Sie die Funktion *TransBusAddrToVirtual*, die die Funktionen *HalTranslateBusAddress* und *MmMapIoSpace* aufruft.

Reservieren von physischem Speicher

Um einen Speicherbereich für die Verwendung in einem Treiber oder im Kernel zu reservieren, stehen Ihnen zwei Methoden zur Verfügung:

- **Dynamisch über den Aufruf der Funktion** *AllocPhysMem* *AllocPhysMem* weist zusammenhängenden Speicher in einer Seite oder mehreren Seiten zu, die Sie wiederum zum virtuellen Speicher im Benutzerbereich zuordnen können, indem Sie die Funktion *MmMapIoSpace* oder *OALPAtoVA* aufrufen, abhängig davon, ob der Code im Benutzermodus oder im Kernelmodus ausgeführt wird. Da der physische Speicher in Einheiten von Speicherseiten reserviert wird, muss mindestens eine physische Speicherseite zugeordnet werden. Die Größe der Speicherseite hängt von der Hardwareplattform ab. Eine typische Seite ist 64 KB groß.

- **Statisch durch Erstellen eines *RESERVED*-Abschnitts in der Datei *Config.bib***
  Sie können physischen Speicher statisch reservieren, indem Sie den *MEMORY*-Abschnitt der .bib-Datei eines Run-Time Images verwenden, beispielsweise *Config.bib* im Ordner *BSP*. In Abbildung 6.9 ist diese Methode dargestellt. Die informationellen Namen der Speicherbereiche werden ausschließlich verwendet, um die im System definierten Speicherbereiche zu identifizieren. Die wichtigen Informationen sind die Adressdefinitionen und das *RESERVED*-Schlüsselwort. Entsprechend dieser Einstellungen schließt Windows Embedded CE die reservierten Bereiche aus dem Systemspeicher aus, damit diese von Peripheriegeräten für DMA und die Datenübertragung verwendet werden können. Es treten keine Zugriffskonflikte auf, da das System keine reservierten Speicherbereiche verwendet.

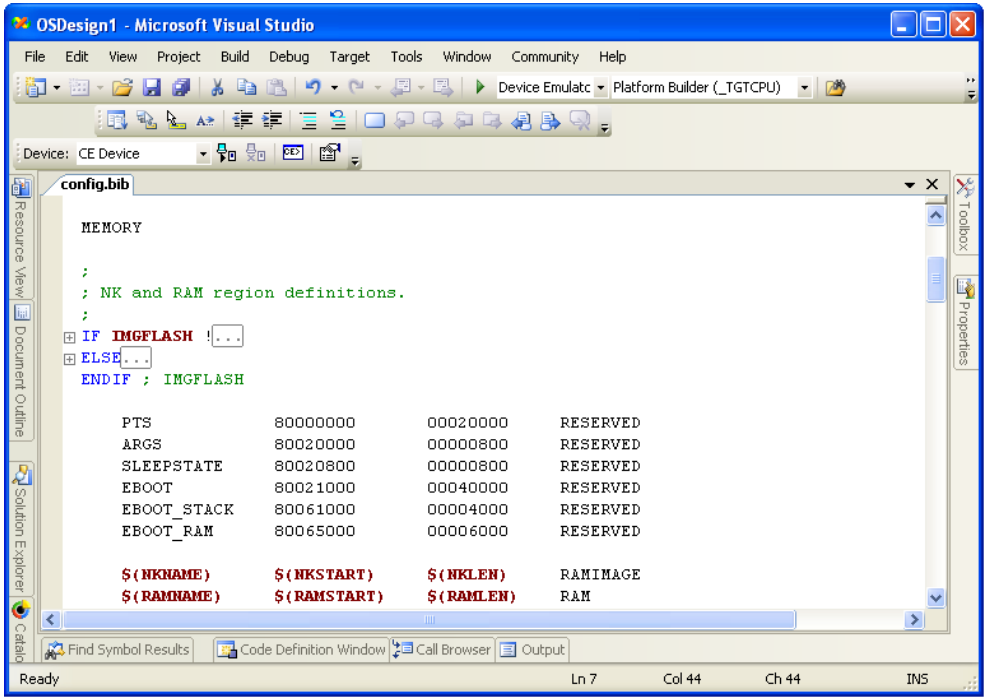


Abbildung 6.9 Definition der reservierten Speicherbereiche in einer Config.bib-Datei

Anwendungsaufruferpuffer

In Windows Embedded CE 6.0 werden Anwendungen und Gerätetreiber in unterschiedlichen Prozessbereichen ausgeführt. Beispielsweise lädt der Geräte-Manager die Streamtreiber in den Kernelprozess (*Nk.exe*) oder in den Hostprozess für Benutzermodustreiber (*Udevice.exe*), während alle Anwendungen in einem jeweils separaten Prozessbereich ausgeführt werden. Da Zeiger auf virtuelle Speicheradressen in einem Prozessbereich in anderen Prozessbereichen ungültig sind, müssen Sie Zeigerparameter zuordnen oder marshallen, wenn separate Prozesse für die Kommunikation und prozessübergreifende Datenübertragung auf den gleichen Pufferbereich im physischen Speicher zugreifen müssen.

Verwenden von Zeigerparametern

Ein Zeigerparameter ist ein Zeiger, den ein Aufrufer als Parameter an eine Funktion übergeben kann. Die *DeviceIoControl*-Parameter *lpInBuf* und *lpOutBuf* sind gute Beispiele. Anwendungen verwenden *DeviceIoControl* für direkte Eingabe- und Ausgabevorgänge. Ein Zeiger auf einen Eingabepuffer (*lpInBuf*) und ein Zeiger auf einen Ausgabepuffer (*lpOutBuf*) ermöglichen die Datenübertragung zwischen der Anwendung und dem Treiber. *DeviceIoControl* wird wie folgt in *Winbase.h* deklariert:

```
WINBASEAPI BOOL WINAPI DeviceIoControl (HANDLE hDevice,
                            DWORD dwIoControlCode,
                            __inout_bcount_opt(nInBufSize)LPVOID lpInBuf,
                            DWORD nInBufSize,
                            __inout_bcount_opt(nOutBufSize) LPVOID lpOutBuf,
                            DWORD nOutBufSize,
                            __out_opt LPDWORD lpBytesReturned,
                            __reserved LPOVERLAPPED lpOverlapped);
```

Zeigerparameter sind in Windows Embedded CE 6.0 einfach zu verwenden, da der Kernel die Zugriffsüberprüfungen und das Marshalling der Parameter automatisch ausführt. In der o.a. *DeviceIoControl*-Deklaration sind die Pufferparameter *lpInBuf* und *lpOutBuf* als E/A-Parameter einer bestimmten Größe definiert und *lpBytesReturned* ist nur ein Ausgabeparameter. Mit diesen Deklarationen stellt der Kernel sicher, dass eine Anwendung keine Adresse im schreibgeschützten Speicher (beispielsweise einen freigegebenen Heap, in den ausschließlich der Kernel schreiben kann) als E/A-Pufferzeiger oder Ausgabepufferzeiger übergibt. Wenn eine Anwendung eine solche Adresse übergibt, wird eine Ausnahme ausgelöst. Auf diese Art stellt Windows Embedded CE 6.0 sicher, dass eine Anwendung über einen Kernelmodustreiber keine höheren Zugriffsberechtigungen auf einen Speicherbereich erhält. Deshalb müssen Sie

treiberseitig den Zugriff für die Zeiger, die über die Streamschnittstellenfunktion *XXX_IOControl* (*pBufIn* and *pBufOut*) übergeben werden, nicht überprüfen.

Verwenden eingebetteter Zeiger

Eingebettete Zeiger werden vom Aufrufer über einen Speicherpuffer indirekt an eine Funktion übergeben. Beispielsweise kann eine Anwendung einen Zeiger im Eingabepuffer speichern, der in *DeviceIoControl* über den Parameterzeiger *lpInBuf* übergeben wird. Der Kernel überprüft und marshallt den Parameterzeiger *lpInBuf* automatisch. Das System kann den im Eingabepuffer eingebetteten Zeiger jedoch nicht identifizieren. Soweit der Kernel betroffen ist, enthält der Speicherpuffer Binärdaten. Windows Embedded CE 6.0 stellt keine Methode für die explite Angabe bereit, dass Zeiger in diesem Speicherblock enthalten sind.

Da eingebettete Zeiger die Zugriffsüberprüfungen des Kernels und Marhallhilfsfunktionen umgehen, müssen Sie die Zugriffsüberprüfungen und das Marshallen der in Gerätetreibern eingebetteten Zeiger vor der Verwendung manuell ausführen. Ansonsten werden möglicherweise Schwachstellen erzeugt, die von böswilligem Code genutzt werden können, um unzulässige Aktionen auszuführen und das gesamte System zu gefährden. Kernelmodustreiber verfügen über höhere Berechtigungen und können auf den Systemspeicher zugreifen, der für Benutzermoduscode nicht verfügbar ist.

Um sicherzustellen, dass der Aufruferprozess über die erforderlichen Zugriffsberechtigungen verfügt, den Zeiger marshallt und auf den Puffer zugreifen kann, rufen Sie die Funktion *CeOpenCallerBuffer* auf. *CeOpenCallerBuffer* überprüft die Zugriffsberechtigungen basierend darauf, ob der Aufrufer im Kernelmodus oder im Benutzermodus ausgeführt wird, weist eine neue virtuelle Adresse für den physischen Speicher des Aufruferpuffers zu und ordnet gegebenenfalls einen temporären Heappuffer zu, um eine Kopie des Aufruferpuffers zu erstellen. Da die Zuordnung des physischen Speichers das Zuweisen eines neuen virtuellen Adressbereichs im Treiber umfasst, müssen Sie die Funktion *CeCloseCallerBuffer* aufrufen, nachdem der Treiber die Verarbeitung abgeschlossen hat.

Pufferverarbeitung

Im Anschluss an die impliziten (Parameterzeiger) oder expliziten (eingebettete Zeiger) Zugriffsüberprüfungen und das Marshalling der Zeiger, kann der Gerätetreiber auf den Puffer zugreifen. Der Zugriff auf den Puffer ist jedoch nicht exklusiv. Der Gerätetreiber liest und schreibt die Daten im Puffer, aber der Aufrufer

führt diese Vorgänge möglicherweise gleichzeitig aus (siehe Abbildung 6.10). Beispielsweise können Sicherheitsprobleme auftreten, wenn ein Gerätespeicher gemarshallte Zeiger im Puffer des Aufrufers speichert. Ein zweiter Thread in der Anwendung kann den Zeiger ändern, um über den Treiber auf einen geschützten Speicherbereich zuzugreifen. Deshalb sollten Treiber immer sichere Kopien der Zeiger und Pufferwerte erstellen, die sie vom Aufrufer erhalten, und eingebettete Zeiger in lokalen Variablen kopieren, um asynchrone Änderungen zu vermeiden.

WICHTIG Asynchrone Pufferverarbeitung

Verwenden Sie Zeiger nicht im Aufruferpuffer, nachdem diese gemarshallt wurden. Speichern Sie außerdem keine gemarshallten Zeiger oder andere Variablen im Aufruferpuffer, die für die Treiberverarbeitung erforderlich sind. Kopieren Sie die Puffergrößenwerte in lokale Variablen, damit Aufrufer diese Werte nicht ändern können, um einen Pufferüberlauf zu verursachen. Eine Methode zum Verhindern von asynchronen Pufferänderungen durch den Aufrufer ist das Aufrufen der Funktion *CeOpenCallerBuffer* mit dem Parameter *ForceDuplicate* mit dem Wert *TRUE*, um die Daten aus dem Aufruferpuffer in einen temporären Heappuffer zu kopieren.

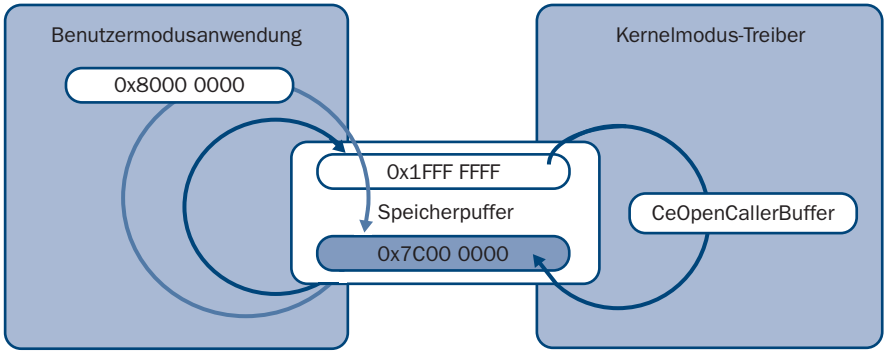


Abbildung 6.10 Ändern eines gemarshallten Zeigers in einem freigegebenen Puffer

Synchroner Zugriff

Der synchrone Speicherzugriff entspricht dem nicht gleichzeitigen Pufferzugriff. Der Thread des Aufrufers wartet bis der Funktionsaufruf zurückgegeben wird, beispielsweise *DeviceIoControl*. Im Aufruferprozess sind keine anderen Threads vorhanden, die auf den Puffer zugreifen, während der Treiber die Verarbeitung ausführt. In diesem Fall kann der Gerätetreiber eingebettete Zeiger und Parameterzeiger ohne weitere Maßnahmen verwenden (nachdem *CeOpenCallerBuffer* aufgerufen wurde).

Folgendes Beispiel stellt den synchronen Zugriff einer Anwendung auf einen Puffer dar: Der Beispielquellcode ist ein Auszug der *XXX_IOControl*-Funktion eines Streamtreibers:

```
BOOL SMP_IOControl(DWORD hOpenContext, DWORD dwCode,
                   PBYTE pBufIn, DWORD dwLenIn,
                   PBYTE pBufOut, DWORD dwLenOut,
                   PDWORD pdwActualOut)
{
    BYTE *lpBuff = NULL;

    ...

    if (dwCode == IOCTL_A_WRITE_FUNCTION)
    {
        // Check parameters
        if ( pBufIn == NULL || dwLenIn != sizeof(AN_INPUT_STRUCTURE))
        {
            DEBUGMSG(ZONE_IOCTL, (TEXT("Bad parameters\r\n")));
            return FALSE;
        }

        // Access input buffer
        hrMemAccessVal = CeOpenCallerBuffer((PVOID) &lpBuff,
                                            (PVOID) pBufIn,
                                             dwLenIn,
                                             ARG_I_PTR,
                                             FALSE);

        // Check hrMemAccessVal value
        // Access the pBufIn through lpBuff

        ...

        // Close the buffer when it is no longer needed
        CeCloseCallerBuffer((PVOID)lpBuff, (PVOID)pBufOut,
                            dwLenOut, ARG_I_PTR);

    }

...

}
```

Asynchroner Zugriff

Der asynchrone Pufferzugriff setzt voraus, dass mehrere Aufrufer und Treiberthreads sequentiell oder gleichzeitig auf den Puffer zugreifen. Für den sequenziellen Zugriff wird der Aufruferthread möglicherweise beendet, bevor der Treiberthread die Verarbeitung abgeschlossen hat. Beim Aufrufen der Marshalling-Hilfsfunktion

CeAllocAsynchronousBuffer müssen Sie den Puffer erneut marshallen, nachdem dieser von *CeOpenCallerBuffer* gemarshallt wurde, um sicherzustellen, dass der Puffer im Treiber auch dann verfügbar ist, wenn der Adressbereich des Aufrufers nicht verfügbar ist. Sie müssen die Funktion *CeFreeAsynchronousBuffer* aufrufen, nachdem der Treiber die Verarbeitung abgeschlossen hat.

Um sicherzustellen, dass der Gerätetreiber im Kernel- und im Benutzermodus funktioniert, gehen Sie wie folgt vor, um den asynchronen Pufferzugriff zu unterstützen:

- **Zeigerparameter** Übergeben Sie Zeigerparameter als skalare DWORD-Werte und rufen Sie anschließend *CeOpenCallerBuffer* und *CeAllocAsynchronousBuffer* auf, um Zugriffsüberprüfungen und das Marshalling auszuführen. Beachten Sie, dass Sie *CeAllocAsynchronousBuffer* für einen Zeigerparameter im Benutzermoduscode nicht aufrufen und keine asynchronen Schreibvorgänge des O_PTR- oder IO_PTR-Werts ausführen können.
- **Eingebettete Zeiger** Übergeben Sie eingebettete Zeiger an *CeOpenCallerBuffer* und *CeAllocAsynchronousBuffer*, um die Zugriffsüberprüfung und das Marshalling auszuführen.

Um den gleichzeitigen Zugriff zu unterstützen, müssen Sie im Anschluss an das Marshalling eine sichere Kopie des Puffers erstellen. Sie können die Funktion *CeOpenCallerBuffer* mit dem Parameter *ForceDuplicate* mit dem Wert *TRUE* und die Funktion *CeCloseCallerBuffer* aufrufen. Eine weitere Möglichkeit ist der Aufruf der Funktionen *CeAllocDuplicateBuffer* und *CeFreeDuplicateBuffer* für Puffer, die durch Parameterzeiger referenziert sind. Außerdem können Sie einen Zeiger oder Puffer in eine Stackvariablen kopieren oder mit der Funktion *VirtualAlloc* den Heapspeicher zuordnen und dann *memcpy* verwenden, um den Aufruferpuffer zu kopieren. Beachten Sie, dass das System von einer bösartigen Anwendung übernommen werden kann, wenn Sie keine sichere Kopie erstellen.

Ausnahmebehandlung

Ein weiterer wichtiger Aspekt, der für den asynchronen Pufferzugriff beachtet werden sollte, ist die Möglichkeit, dass eingebettete Zeiger nicht auf gültige Speicheradressen verweisen könnten. Beispielsweise kann eine Anwendung einen Zeiger an einen Treiber übergeben, der auf einen nicht zugeordneten oder reservierten Speicherbereich verweist, oder die Anwendung kann den Puffer asynchron freigeben. Um ein zuverlässiges System sicherzustellen und Speicherverlust zu verhindern, schließen Sie den Pufferzugriffscode in einen *__try*-Frame und den Bereinigungscode

zum Freigeben der Speicherzuordnungen in einen __*finally*-Block oder einen Ausnahmehandler ein. Weitere Informationen zur Ausnahmebehandlung finden Sie in Kapitel 3 „Systemprogrammierung."

Zusammenfassung

Windows Embedded CE 6.0 unterstützt die Interprozesskommunikation zwischen Anwendungen und Gerätetreibern über Kernelfeatures und Marshalling-Hilfsfunktionen, um die Arbeit des Treiberentwicklers zu vereinfachen. Der Kernel führt für Parameterzeiger alle Überprüfungen und das Zeigermarshalling automatisch aus. Nur die eingebetteten Zeiger erfordern zusätzliche Verarbeitungsschritte, da der Kernel den Inhalt der Anwendungspuffer, die an einen Treiber übergeben werden, nicht überprüfen kann. Das Überprüfen und Marshallen eines eingebetteten Zeigers beim asychronen Zugriff umfasst den Aufruf der Funktion *CeOpenCallerBuffer*. Der asynchrone Zugriff erfordert einen weiteren Aufruf der Funktion *CeAllocAsynchronousBuffer*, um den Zeiger noch einmal zu marshallen. Um sicherzustellen, dass der Treiber keine Systemanfälligkeiten verursacht, müssen Sie die Puffer richtig behandeln und eine sichere Kopie des Pufferinhalts erstellen, damit Aufrufer die Werte nicht ändern können. Verwenden Sie keine Zeiger oder Puffergrößenwerte im Puffer des Aufrufers, nachdem diese gemarshallt wurden. Speichern Sie gemarshallte Zeiger oder andere Variablen, die für die Treiberverarbeitung erforderlich sind, nicht im Aufruferpuffer.

Lektion 7: Verbessern der Treiberportabilität

Gerätetreiber erhöhen die Flexibilität und Portabilität des Betriebssystems. Idealerweise sind keine Codeänderungen erforderlich, um Gerätetreiber auf anderen Zielgeräten mit unterschiedlichen Kommunikationsanforderungen auszuführen. Um die Portabilität und Wiederverwendbarkeit von Treibern sicherzustellen, stehen Ihnen mehrere relativ unkomplizierte Methoden zur Verfügung. Eine Methode ist das Festlegen der Konfigurationseinstellungen in der Registrierung anstatt die Parameter im OAL oder dem Treiber hart zu codieren. Windows Embedded CE unterstützt eine mehrschichtige Architektur, die auf MDD und PDD basiert und die Sie im Design von Gerätetreibern einsetzen können. Es sind weitere Methoden zum Implementieren busagnostischer Treiber verfügbar, die Peripheriegeräte unabhängig vom Bustyp unterstützen.

Nach Abschluss dieser Lektion können Sie:

- Die Registrierungseinstellungen zum Verbessern der Portabilität und Wiederverwendbarkeit eines Gerätetreibers beschreiben.
- Einen Gerätetreiber mit einer busagnostischen Methode implementieren.

Veranschlagte Zeit für die Lektion: 15 Minuten.

Zugreifen auf Registrierungseinstellungen in einem Treiber

Um die Portabilität und Wiederverwendbarkeit eines Gerätetreibers zu verbessern, können Sie Registrierungseinträge im Unterschlüssel des Treibers konfigurieren. Beispielsweise können Sie E/A-zugeordnete Speicheradressen oder Einstellungen für installierbare ISRs festlegen, die der Gerätetreiber dynamisch lädt. Um auf die Einträge im Registrierungsschlüssel eines Gerätetreibers zuzugreifen, muss der Treiber den Pfad zu seinen Einstellungen ermitteln. Hierbei handelt es sich jedoch nicht unbedingt um den Schlüssel *HKEY_LOCAL_MACHINE\Drivers\BuiltIn*. Der korrekte Pfad ist im *Key*-Wert im Unterschlüssel des geladenen Treibers unter *HKEY_LOCAL_MACHINE\Drivers\Active* angegeben. Der Geräte-Manager übergibt den Pfad zum *Drivers\Active*-Unterschlüssel des Treibers im LPCTSTR-Parameter *pContext* an die Funktion *XXX_Init*. Der Gerätetreiber verwendet den LPCTSTR-Wert im Aufruf der Funktion *OpenDeviceKey*, um ein Handle für den Registrierungsschlüssel des Geräts abzurufen. Die *Key*-Werte des *Drivers\Active*-Unterschlüssels des Treibers müssen nicht direkt gelesen werden. Das von *OpenDeviceKey* zurückgegebene Handle verweist auf den Registrierungsschlüssel des Treibers. Sie

können dieses Handle wie jedes andere Registrierungshandle verwenden. Vergessen Sie nicht, das Handle zu schließen, wenn dieses nicht mehr benötigt wird.

TIPP XXX_Init-Funktion und Treibereinstellungen

Die Funktion *XXX_Init* ermöglicht das Bestimmen der Konfigurationseinstellungen für einen in der Registrierung definierten Treiber. Anstatt in nachfolgenden Streamfunktionsaufrufen wiederholt auf die Registrierung zuzugreifen, sollten Sie die Konfigurationsinformationen im Gerätekontext speichern, der vom Geräte-Manager als Antwort auf den *XXX_Init*-Aufruf erstellt und zurückgegeben wird.

Interruptspezifische Registrierungseinstellungen

Wenn der Gerätetreiber eine installierbare ISR für ein Gerät laden muss und Sie die Portabilität des Codes erhöhen möchten, registrieren Sie den ISR-Handler, den IRQ und die SYSINTR-Werte in Registrierungsschlüsseln, lesen Sie diese Werte beim Initialisieren des Treibers aus, überprüfen Sie, ob der IRQ und die SYSINTR-Werte gültig sind, und installieren Sie die ISR mit der Funktion *LoadIntChainHandler*.

In Tabelle 6.9 sind die entsprechenden Registrierungseinträge aufgeführt. Wenn Sie die Funktion *DDKReg_GetIsrInfo* aufrufen, können Sie die Werte lesen und dynamisch an die Funktion *LoadIntChainHandler* übergeben. Weitere Informationen zur Interruptverarbeitung in Gerätetreibern finden Sie in Lektion 4.

Tabelle 6.9 Interruptspezfische Registrierungseinträge für Gerätetreiber

Registrierungs-eintrag	Typ	Beschreibung
IRQ	REG_DWORD	Gibt den IRQ an, mit dem eine SYSINTR für einen IST im Treiber angefordert wird.
SYSINTR	REG_DWORD	Gibt einen SYSINTR-Wert zum Konfigurieren eines IST im Treiber an.
IsrDll	REG_SZ	Der Dateiname der DLL, die die installierbare ISR enthält.
IsrHandler	REG_SZ	Gibt den Einsprungsspunkt für die installierbare ISR an, die die DLL verwendet.

Speicherspezifische Registrierungseinstellungen

Speicherspezifische Registrierungswerte ermöglichen das Konfigurieren eines Geräts über die Registrierung. In Tabelle 6.10 sind die speicherspezifischen Registrierungsinformationen aufgeführt, die ein Treiber in einer DDKWINDOWINFO-Struktur durch den Aufruf der Funktion *DDKReg_GetWindowInfo* abrufen kann. Mit der Funktion *BusTransBusAddrToVirtual*-Funktion können Sie die Busadressen der im Speicher zugeordneten Fenster zu physischen Systemadressen zuordnen und anschließend mit *MnMapIoSpace* in virtuelle Adressen umwandeln.

Tabelle 6.10 Speicherspezifische Registrierungseinträge für Gerätetreiber

Registrierungseintrag	Typ	Beschreibung
IoBase	REG_DWORD	Eine Busbasis eines im Speicher zugeordneten Fensters, die vom Gerät verwendet wird.
IoLen	REG_DWORD	Gibt die Länge des im Speicher zugeordneten Fensters an, das in IoBase definiert ist.
MemBase	REG_MULTI_SZ	Eine Busbasis für mehrere im Speicher zugeordnete Fenster, die vom Gerät verwendet werden.
MemLen	REG_MULTI_SZ	Gibt die Länge der im Speicher zugeordneten Fenster an, die in MemBase definiert sind.

PCI-spezifische Registrierungseinstellungen

Eine weitere Registrierungshilfsfunktion, die Sie zum Auffüllen einer DDKPCIINFO-Struktur mit den Standardinformationen für die PCI-Treiberinstanz verwenden können, ist *DDKReg_GetPciInfo*. In Tabelle 6.11 sind die PCI-spezifischen Einstellungen aufgeführt, die Sie im Unterschlüssel eines Treibers konfigurieren können.

Tabelle 6.11 PCI-spezifische Registrierungseinträge für Gerätetreiber

Registrierungs-eintrag	Typ	Beschreibung
DeviceNumber	REG_DWORD	Die PCI-Gerätenummer.
FunctionNumber	REG_DWORD	Die PCI-Funktionsnummer des Geräts, die ein Singlefunktionsgerät auf einer PCI-Multifunktionskarte anzeigt.
InstanceIndex	REG_DWORD	Die Instanzennummer des Geräts.
DeviceID	REG_DWORD	Der Typ des Geräts.
ProgIF	REG_DWORD	Eine registrierungsspezifische Programmierungsschnittstelle, beispielsweise USB OHCI oder UHCI.
RevisionId	REG_DWORD	Die Versionsnummer des Geräts.
Subclass	REG_DWORD	Die Basisfunktion des Geräts, beispielsweise ein IDE-Controller.
SubSystemId	REG_DWORD	Der Typ der Karte oder des Teilsystems, das das Gerät verwendet.
SubVendorId	REG_DWORD	Der Anbieter der Karte oder des Teilsystems, das das Gerät verwendet.
VendorId	REG_MULTI_SZ	Der Hersteller des Geräts.

Entwickeln busagnostischer Treiber

Ähnlich wie die Einstellungen für installierbare ISRs, im Speicher zugeordnete Fenster und PCI-Geräteinstanzinformationen, können Sie die GPIO-Werte oder Zeitgebungseinstellungen für ein busagnostisches Treiberdesign in der Registrierung festlegen. Ein busagnostischer Treiber unterstützt mehrere Busimplementierungen für den gleichen Hardware-Chipset, beispielsweise PCI oder PCMCIA, ohne dass Codeänderungen erforderlich sind.

Um einen busagnostischen Treiber zu implementieren, gehen Sie wie folgt vor:

1. Verwalten Sie alle erforderlichen Konfigurationsparameter im Registrierungs-Unterschlüssel des Treibers und verwenden Sie die Windows Embedded CE-Registrierungshilfsfunktionen *DDKReg_GetIsrInfo*, *DDKReg_GetWindowInfo* und *DDKReg_GetPciInfo*, um diese Einstellungen während der Treiberinitialisierung abzurufen.
2. Rufen Sie *HalTranslateBusAddress* auf, um die busspezifischen Adressen in physische Systemadressen umzuwandeln, und rufen Sie anschließend *MmMapIoSpace* auf, um die physischen Adressen virtuellen Adressen zuzuordnen.
3. Setzen Sie die Hardware zurück, maskieren Sie den Interrupt und laden Sie eine installierbare ISR unter Verwendung der Funktion *LoadIntChainHandler* mit den aus *DDKReg_GetIsrInfo* abgerufenen Informationen.
4. Laden Sie unter Verwendung von *RegQueryValueEx* die Initialisierungseinstellungen für die installierbare ISR aus der Registrierung und übergeben Sie die Werte an die installierbare ISR, indem Sie *KernelLibIoControl* mit einer benutzerdefinierten IOCTL aufrufen. Beispielsweise verwendet die Generic Installable Interrupt Service Routine (GIISR) in Windows Embedded CE einen *IOCTL_GIISR_INFO*-Handler, um die Instanzeninformationen zu initialisieren, mit denen die GIISR erkennt, ob das Geräte-Interruptbit festgelegt ist und der entsprechende SYSINTR-Wert zurückgegeben werden muss. Der Quellcode ist im Ordner *C:\Wince600\Public\Common\Oak\Drivers\Giisr* gespeichert.
5. Starten Sie den IST, indem Sie die Funktion *CreateThread* aufrufen und den Interrupt demaskieren.

Zusammenfassung

Um die Portabilität eines Gerätetreibers zu verbessern, können Sie Registrierungseinträge im Unterschlüssel des Treibers konfigurieren. Windows Embedded CE umfasst mehrere Registrierungshilfsfunktionen, mit denen Sie die entsprechenden Einstellungen abrufen können, beispielsweise *DDKReg_GetIsrInfo*, *DDKReg_GetWindowInfo* und *DDKReg_GetPciInfo*. Diese Hilfsfunktionen rufen bestimmte Informationen für installierbare ISRs, die im Speicher zugeordneten Fenster und PCI-Geräteinstanzen ab. Mit *RegQueryValueEx* können Sie jedoch auch andere Registrierungseinträge abfragen. Um diese Registrierungsfunktionen zu verwenden, müssen Sie mit *OpenDeviceKey* ein Handle für den Unterschlüssel des Treibers abrufen. *OpenDeviceKey* erwartet einen Registrierungspfad, der vom Geräte-Manager in einem *XXX_Init*-Aufruf an den Treiber übergeben wird. Vergessen Sie nicht, das Registrierungshandle zu schließen, wenn dieses nicht mehr benötigt wird.

Lab 6: Entwickeln von Gerätetreibern

In diesem Lab implementieren Sie einen Streamtreiber, der eine aus 128 Unicode-Zeichen bestehende Zeichenfolge speichert und aus dem Speicher abruft. Im Begleitmaterial dieses Buchs ist eine Basisversion des Treibers verfügbar. Sie müssen lediglich den Code als Teilprojekt zu einem OS Design hinzufügen. Anschließend konfigurieren Sie die .bib-Datei und Registrierungseinstellungen, um den Treiber automatisch während des Starts zu laden, und erstellen eine WCE Console Application, um die Treiberfunktionalität zu testen. Im letzten Schritt fügen Sie die Unterstützung für die Energieverwaltung zum Zeichenfolgentreiber hinzu.

HINWEIS Detaillierte schrittweise Anleitungen

Um die Verfahren in diesem Lab erfolgreich auszuführen, lesen Sie das Dokument Schrittweise Anweisungen für Lab 6" im Begleitmaterial.

▶ Hinzufügen eines Streamschnittstellentreibers zu einem Run-Time Image

1. Klonen Sie das Geräteemulator-BSP und erstellen Sie ein OS Design, das auf diesem BSP basiert (siehe Lab 2 Erstellen und Bereitstellen eines Run-Time Images).
2. Kopieren Sie den Quellcode des Zeichenfolgentreibers, der auf der Begleit-CD im Ordner *\Labs\StringDriver\String* gespeichert ist, in den BSP-Ordner unter *%_WINCEROOT%\Platform\<BSPName>\Src\Drivers*. Der Ordner *String* sollte im Ordner *Drivers* erstellt werden. In diesem Ordner sollten sich die Dateien des Treibers auf der Begleit-CD befinden, beispielsweise *sources*, *string.c* und *string.def*. Sie können einen Treiber auch neu erstellen, aber die Verwendung des Beispieltreibers ist wesentlich zeitsparender.
3. Fügen Sie einen Eintrag zur Dirs-Datei im Ordner *Drivers* über dem neuen Ordner *String* hinzu, um den Zeichenfolgentreiber in den Buildprozess einzubeziehen.

ACHTUNG Include In Build-Option

Verwenden Sie im Solution Explorer nicht die Option Include In Build, um den Zeichenfolgentreiber in den Buildprozess einzubeziehen. Der Solution Explorer entfernt wichtige CESYSGEN-Direktiven aus der Dirs-Datei.

4. Fügen Sie einen Eintrag in die Datei *Platform.bib* ein, um den Zeichenfolgentreiber in *$(_FLATRELEASEDIR)* zum Run-Time Image hinzuzufügen. Markieren Sie das Treibermodul als ausgeblendete Systemdatei.

5. Fügen Sie folgende Codezeile zur Datei *Platform.reg* hinzu, um die .reg-Datei des Zeichenfolgentreibers in die Registrierung des Run-Time Images einzubeziehen.

```
#include "$(_TARGETPLATROOT)\SRC\DRIVERS\String\string.reg"
```

6. Erstellen Sie den Zeichenfolgentreiber, indem Sie mit der rechten Maustaste im Solution Explorer auf den Treiber klicken und **Build** auswählen.
7. Erstellen Sie ein neues Run-Time Image im Debug-Modus.

HINWEIS Erstellen eines Run-Time Images im Release-Modus

Um die Release-Version des Run-Time Images zu verwenden, müssen Sie die *DEBUGMSG*-Anweisungen im Treibercode in *RETAILMSG*-Anweisungen ändern, um Treibermeldungen auszugeben.

8. Öffnen Sie die generierte Datei *Nk.bin* im flachen *Release*-Verzeichnis, um zu überprüfen, ob die Datei *String.dll* und die Registrierungseinträge zum Laden des Treibers beim Start im Unterschlüssel *HKEY_LOCAL_MACHINE\Drivers\BuiltIn\String* vorhanden sind.
9. Laden Sie das generierte Image auf dem Geräteemulator.
10. Nachdem das Image gestartet wurde, öffnen Sie das Fenster **Modules**, indem Sie STRG+ALT+U drücken oder im Menü **Debug** von Visual Studio auf **Windows** zeigen und **Modules** auswählen. Überprüfen Sie, ob das System die Datei *string.dll* geladen hat (siehe Abbildung 6.11).

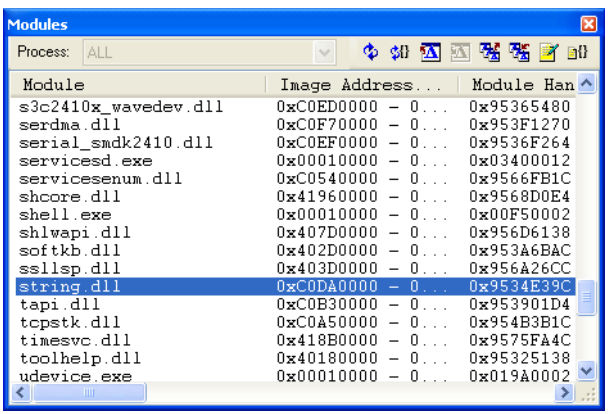


Abbildung 6.11 Das Fenster Modules mit geladenem Zeichenfolgentreiber

▶ Zugreifen auf den Treiber aus einer Anwendung

1. Erstellen Sie mit dem **Windows Embedded CE Subproject Wizard** ein neues **WCE Console Application**-Teilprojekt als Bestandteil des OS Designs. Wählen Sie **WCE Console Application** und die Vorlage **A Simple Windows Embedded CE Console Application** aus.
2. Klicken Sie mit der rechten Maustaste in der **Solution**-Ansicht auf den Namen des OS Designs und wählen Sie **Properties** aus, um die Imageeinstellungen im Teilprojekt zu ändern und das Teilprojekt aus dem Image auszuschließen.
3. Fügen Sie <*windows.h*> und <*winioctl.h*> ein.
4. Fügen Sie Code zur Anwendung hinzu, um mit *CreateFile* eine Instanz des Treibers zu öffnen. Übergeben Sie *GENERIC_READ|GENERIC_WRITE* für den zweiten *CreateFile*-Parameter (*dwDesiredAccess*). Übergeben Sie *OPEN_EXISTING* für den fünften Parameter (*dwCreationDisposition*). Wenn Sie den Treiber mit der $device-Namenskonvention öffnen, verwenden Sie einen doppelten Schrägstrich und geben Sie am Ende des Dateinamens keinen Doppelpunkt ein.

```
HANDLE hDrv = CreateFile(L"\\$device\\STR1",
                         GENERIC_READ|GENERIC_WRITE,
                         0, 0, OPEN_EXISTING, 0, 0);
```

5. Kopieren Sie die IOCTL-Headerdatei (*String_ioctl.h*) aus dem Ordner mit dem Zeichenfolgentreiber in den Ordner der neuen Anwendung und beziehen Sie die Headerdatei in die Quellcodedatei ein.
6. Deklarieren Sie eine Instanz der *PARMS_STRING*-Struktur, die in *String_iocontrol.h* definiert ist, damit Anwendungen mit folgendem Code eine Zeichenfolge im Treiber speichern können:

```
PARMS_STRING stringToStore;
wcscpy_s(stringToStore.szString,
         STR_MAX_STRING_LENGTH,
         L"Hello, driver!");
```

7. Rufen Sie *DeviceIoControl* mit den IOcontrol-Code *IOCTL_STRING_SET* auf, um die Zeichenfolge im Treiber zu speichern.
8. Erstellen Sie die Anwendung und wählen Sie im Menü **Target** die Option **Run Programs** aus, um die Anwendung auszuführen.
9. Im Debug-Fenster sollte die Meldung **Stored String "Hello, driver!"** angezeigt werden, wenn Sie die Anwendung ausführen (siehe Abbildung 6.12).

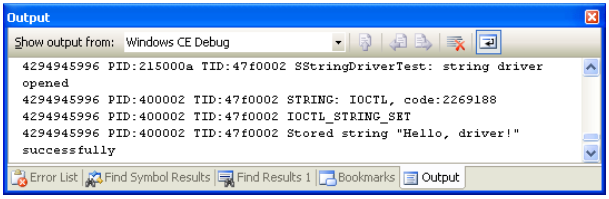


Abbildung 6.12 Eine Debugmeldung vom Zeichenfolgentreiber

▶ Hinzufügen der Unterstützung für die Energieverwaltung

1. Beenden Sie den Geräteemulator und trennen Sie die Verbindung.
2. Fügen Sie mit folgender Codezeile die *IClass* für generische Energieverwaltungsgeräte zum Registrierungsschlüssel des Zeichenfolgentreibers in *String.reg* hinzu:

   ```
   "IClass"=multi_sz:"{A32942B7-920C-486b-B0E6-92A702A99B35}"
   ```

3. Fügen Sie den Energieverwaltungscode, der in der Datei *StringDriverPowerCode.txt* unter *\Labs\StringDriver\Power* auf der Begleit-CD gespeichert ist, zur IOControl-Funktion des Zeichenfolgentreibers hinzu, um *IOCTL_POWER_GET*, *IOCTL_POWER_SET* und *IOCTL_POWER_CAPABILITIES* zu unterstützen.
4. Fügen Sie Code zum Gerätekontext des Zeichenfolgentreibers hinzu, um den aktuellen Energiestatus zu speichern:

   ```
   CEDEVICE_POWER_STATE CurrentDx;
   ```

5. Fügen Sie den Header *<pm.h>* zur Anwendung hinzu und rufen Sie *SetDevicePower* mit dem Namen des Zeichenfolgentreibers und verschiedenen Energiezuständen auf. Beispiel:

   ```
   SetDevicePower(L"STR1:", POWER_NAME, D2);
   ```

6. Führen Sie die Anwendung erneut aus. Beachten Sie die Debugmeldungen bezüglich der Energiezustände, wenn der Power Manager den Energiestatus des Zeichenfolgentreibers ändert (siehe Abbildung 6.13).

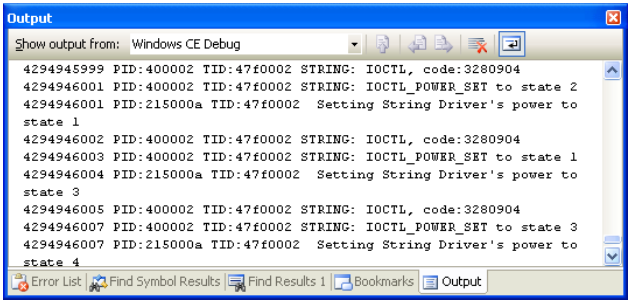


Abbildung 6.13 Energieverwaltungsmeldungen des Zeichenfolgentreibers

Lernzielkontrolle

Windows Embedded CE 6.0 basiert auf einem außergewöhnlich modularen Design und unterstützt ARM-, MIPS-, SH4- und x86-Boards in zahlreichen Hardwarekonfigurationen. Der CE-Kernel enthält den für das Betriebssystem wichtigsten Code. Der plattformspezifische Code befindet sich im OAL und in den Gerätetreibern. Gerätetreiber machen den größten Bestandteil eines BSP für ein OS Design aus. Anstatt direkt auf die Hardware zuzugreifen, lädt das Betriebssystem die entsprechenden Gerätetreiber und verwendet anschließend deren Funktionen und E/A-Dienste.

Windows Embedded CE-Gerätetreiber sind DLLs, die einem bekannten API entsprechen, damit sie vom Betriebssystem geladen werden können. Systemeigene CE-Treiber interagieren mit GWES und Streamtreiber interagieren mit dem Gerätemanager. Streamtreiber implementieren das Streamschnittstellen-API, damit ihre Ressourcen als spezielle Dateisystemressourcen dargestellt werden. Anwendungen können die Standarddateisystem-APIs verwenden, um diese Treiber zu verwenden. Das Streamschnittstellen-API unterstützt auch IOCTL-Handler, die praktisch sind, um einen Treiber mit dem Power Manager zu integrieren. Der Power Manager ruft *XXX_IOControl* mit dem IOControl-Code *IOCTL_POWER_SET* auf und übergibt den angeforderten Geräteenergiestatus.

Systemeigene Treiber und Streamtreiber können ein monolithisches oder schichtenbasiertes Treiberdesign verwenden. Das schichtenbasierte Design teilt den Gerätetreiber in einen MDD- und einen PDD-Bereich auf, um die Wiederverwendbarkeit des Codes zu erhöhen. Das schichtenbasierte Design unterstützt außerdem Treiberupdates. Windows Embedded CE umfasst eine flexible Architektur für die Interruptverarbeitung, die auf ISRs und ISTs basiert. Die Hauptaufgabe der ISR ist das Identfizieren der Interruptquelle und das Benachrichtigen des Kernels über den auszuführenden IST mit einem SYSINTR-Wert. Der IST führt die meisten Verarbeitungsvorgänge aus, beispielsweise die zeitaufwendigen Pufferkopierprozesse.

In Windows Embedded CE 6.0 stehen Ihnen zum Laden eines Treibers zwei Methoden zur Verfügung. Sie können die Registrierungseinstellungen des Treibers zum *BuiltIn*-Registrierungsschlüssel hinzufügen, um den Treiber automatisch während des Starts zu laden. Sie können den Treiber auch automatisch in einem *ActivateDeviceEx*-Aufruf laden. Abhängig von den Registrierungseinträgen des Treibers können Sie einen Treiber im Kernelmodus oder im Benutzermodus ausführen. Windows Embedded CE 6.0 umfasst einen Hostprozess für

Benutzermodustrieber und einen Reflector-Dienst, der das Ausführen von Kernelmodustreibern im Benutzermodus ohne Codeänderungen ermöglicht. Da Gerätetreiber in Windows Embedded CE 6.0 in anderen Prozessbereichen als Anwendungen ausgeführt werden, müssen Sie die Daten in physischen Speicherbereichen oder Kopierprozessen marshallen, um die Kommunikation zu unterstützen. Sie müssen eingebettete Zeiger überprüfen und marshallen, indem Sie *CeOpenCallerBuffer* und *CeAllocAsynchronousBuffer* aufrufen und den asynchronen Pufferzugriff so verarbeiten, dass eine Benutzeranwendung nicht auf einen Kernelmodustreiber zugreifen kann, um das System zu übernehmen.

Schlüsselbegriffe

Kennen Sie die Bedeutung der folgenden wichtigen Begriffe? Sie können Ihre Antworten überprüfen, wenn Sie die Begriffe im Glossar am Ende dieses Buches nachschlagen.

- IRQ
- SYSINTR
- IST
- ISR
- Benutzermodus
- Marshalling
- Streamschnittstelle
- Systemeigene Schnittstelle
- PDD
- MDD
- Monolithisch
- Busagnostisch

Empfohlene Vorgehensweise

Um die in diesem Kapitel behandelten Prüfungsschwerpunkte erfolgreich zu beherrschen, sollten Sie die folgenden Aufgaben durcharbeiten.

Erweitern Sie die Energieverwaltungsfeatures

Setzen Sie die Entwicklung des Energieverwaltungscodes für den Zeichenfolgentreiber fort.

- **Löschen Sie den Zeichenfolgenpuffer** Ändern Sie den Zeichenfolgentreiber, um den Inhalt des Zeichenfolgenpuffers zu löschen, wenn der Gerätetreiber in den Energiestatus D3 oder D4 übergeht.
- **Ändern Sie die Energiefunktionen** Überprüfen Sie was passiert, wenn Sie einen anderen POWER_CAPABILITIES-Wert an den Power Manager übergeben.

Weitere IOCTLs

Erweitern Sie die Features des Zeichenfolgentreibers, indem Sie weitere IOCTL-Handler hinzufügen.

- **Kehren Sie die gespeicherte Zeichenfolge um** Fügen Sie einen IOCTL hinzu, um den Inhalt der Zeichenfolge im Puffer umzukehren.
- **Verknüpfen Sie eine Zeichenfolge** Fügen Sie einen IOCTL hinzu, der eine zweite Zeichenfolge mit der gespeicherten Zeichenfolge verknüpft, ohne einen Pufferüberlauf zu verursachen.
- **Eingebettete Zeiger** Ersetzen Sie den Zeichenfolgenparameter durch einen Zeiger auf eine Zeichenfolge und greifen Sie mit *CeOpenCallerBuffer* auf den Zeiger zu.

Installierbare ISR

Lesen Sie die Produktdokumentation, um mehr über installierbare ISRs zu erfahren.

- **Vertiefen Sie Ihre Kenntnisse zu installierbaren ISRs** Weitere Informationen zu installierbaren ISRs finden Sie im Abschnitt „Installable ISRs and Device Drivers“ in der Windows Embedded CE 6.0-Dokumentation auf der Microsoft MSDN-Website unter *http://msdn2.microsoft.com/en-us/library/aa929596.aspx*.
- **Überprüfen Sie eine installierbare ISR** Suchen Sie nach einem Beispiel einer installierbaren ISR und überprüfen Sie deren Struktur. Ein guter Ausgangspunkt ist der GIISR-Code im Ordner *%_WINCEROOT%\Public\Common\Oak\Drivers\Giisr*.

Glossar

Application Programming Interface (API) Ein API ist eine Funktionsschnittstelle, die vom Betriebssystem oder einer Bibliothek bereitgestellt wird, um Anforderungen von Anwendungen zu unterstützen.

Application Verifier (AppVerifier) Mit AppVerifier können Entwickler verborgene Programmierfehler finden, beispielsweise Heapbeschädigungen und unzulässige Handles, die mit normalen Anwendungstests schwierig zu identifizieren sind.

Asynchroner Zugriff Mindestens zwei Threads greifen gleichzeitig auf den gleichen Puffer zu.

Ausnahme Eine Ausnahme ist eine nicht normale Situation, die während der Programmausführung auftritt.

Benutzermodus Die im Benutzermodus geladenen Treiber und Anwendungen werden im Benutzerspeicherbereich ausgeführt. In diesem Modus ausgeführten Treiber und Anwendungen können nicht direkt auf den Hardwarespeicher und nur bedingt auf bestimmte APIs und den Kernel zugreifen.

Binary Image Builder (.bib) Eine .bib-Datei legt die Module und Dateien in einem Run-Time Image fest.

Board Support Package (BSP) Ein BSP umfasst den hardwarespezifischen Code. Das BSP besteht normalerweise aus dem Boot Loader, dem OAL (OEM Adaptation Layer) und den boardspezifischen Gerätetreibern.

Boot Loader Ein Codesegment, das beim Prozessorstart ausgeführt wird, um den Prozessor zu initialisieren und das Betriebssystem zu starten.

Core Connectivity (CoreCon) Windows CE unterstützt eine einheitliche Kommunikationsinfrastruktur, die als Core Connectivity bezeichnet wird und umfassende Konnektivität für Downloads und das Debuggen bereitstellt.

Daten-Marshalling Ein Prozess für Daten, der die Zugriffsrechte und die Gültigkeit der Daten für einen anderen Prozess überprüft.

Debugger-Erweiterungsbefehle (CEDebugX) CeDebugX ist eine Erweiterung für den Debugger in Platform Builder. CeDebugX stellt detaillierte Informationen zum Systemstatus beim Erreichen eines Breakpoints bereit und analysiert Abstürze und Deadlocks.

Debugzone Ein Flag, das die Debugmeldungen bezüglich einer bestimmten Funktionen oder eines Treibermodus aktiviert oder deaktiviert.

Dirs-Datei Eine Dirs-Datei ist eine Textdatei, die die Unterverzeichnisse angibt, die den Quellcode für den Buildprozess enthalten.

Eingebetteter Zeiger Ein Zeiger, der in eine Speicherstruktur eingebettet ist.

Event Ein Synchronisierungsobjekt, das von Threads und dem Kernel verwendet wird, um andere Threads im System zu benachrichtigen.

Gerätetreiber Ein Gerätetreiber ist Software zum Verwalten des Gerätebetriebs durch das Zusammenfassen der Funktionen eines physischen oder virtuellen Geräts.

Iltiming Iltiming legt die ISR- und IST-Latenzen auf einem Windows Embedded CE-System fest.

Interrupt Ein Interrupt setzt das System vorübergehend aus, um anzuzeigen, dass ein Verarbeitungsvorgang erforderlich ist. Jedem Interrupt auf einem System ist ein IRQ-Wert zugeordnet, der wiederhum mindestens einer ISR zugewiesen ist.

Interrupt Service Routine (ISR) Eine ISR ist eine Softwareroutine, die von der Hardware beim Auftreten eines Interrupts aktiviert wird. ISRs überprüfen einen Interrupt und legen die Interruptverarbeitung mittels einem SYSINTR-Wert fest, der einem IST zugeordnet wird.

Interrupt Service Thread (IST) Der IST ist ein Thread, der die meisten Interruptverarbeitungsvorgänge ausführt. Das Betriebssystem aktiviert den IST, wenn ein Interrupt verarbeitet werden muss. Nachdem dem ISTs ein SYSINTR-Wert zugewiesen wurde, kann der SYSINTR-Wert von einer ISR zurückgegeben werden. Anschließend wird der entsprechende IST ausgeführt.

IRQ (Interrupt Request) IRQ-Werte werden in der Hardware den Interrupts zugeordnet. Jeder IRQ-Wert kann mindestens einer ISR zugeordnet werden, die das System ausführt, um den entsprechenden ausgelösten Interrupt zu verarbeiten.

Katalog Ein Container mit den Komponenten, die ein auswählbares Feature eines OS Designs für den Benutzer repräsentieren.

Kerneldebugger Der Kerneldebugger integriert die Funktionen, die zum Konfigurieren der Verbindung mit einem Zielgerät und zum Herunterladen eines Run-Time Images auf das Zielgerät erforderlich sind. Der Kerneldebugger ermöglicht das Debuggen des Betriebssystems, der Treiber und der Anwendungen.

Kernel Independent Transport Layer (KITL)
Der KITL bietet eine einfache Methode, um die Debugdienste zu unterstützen.

Kernelmodustreiber Ein Treiber, der im Speicherbereich des Kernels ausgeführt wird.

Kernel Tracker Dieses Tool zeigt OS-Events und Anwendungsevents, die auf einem Windows Embedded CE-Gerät auftreten, auf einem Entwicklungscomputer an.

Klonen Beim Klonen wird eine exakte Kopie von Dateien generiert, um die Originaldateien zu sichern, bevor Änderungen vorgenommen werden. Der Code in einem PUBLIC-Ordner sollte vor jeder Änderung geklont werden.

Komponente Ein CE-Feature, das über den Katalog zu einem OS Design hinzugefügt oder aus diesem entfernt werden kann.

Kritischer Abschnitt (Critical Section) Ein Objekt mit einem Synchronisierungsprozess, das einem Mutexobjekt ähnlich ist. Der Unterschied ist, dass ausschließlich die Threads in einem einzigen Prozess auf einen kritischen Abschnitt zugreifen können.

Mehrschichtiger Treiber Ein Treiber, der in mehrere Layer aufgeteilt ist, um das Verwalten und Wiederverwenden des Codes zu vereinfachen.

Model Device Driver (MDD) Der MDD-Layer eines mehrschichtigen Treibers hat eine Standardschnittstelle zum Betriebssystemlayer und PDD-Layer und führt alle hardwareunabhängigen, treiberspezifischen Prozesse aus.

Monolithischer Treiber Ein Treiber, der nicht in unterschiedliche Layer aufgeteilt ist. Als monolithischer Treiber kann jeder Treiber bezeichnet werden, der nicht der MDD- oder PDD-Standardlayerarchitektur von CE entspricht, auch wenn der Treiber kein eigenes Layerschema besitzt.

Mutex Ein Mutexobjekt ist ein Synchronisierungsobjekt, dessen Status auf signalisiert (wenn es nicht zu einem Thread gehört)

oder auf nicht signalisiert (wenn es zu einem Thread gehört) festgelegt ist. Ein Mutex kann nur einem Thread zugewiesen sein. Ein Mutex repräsentiert eine Ressource, auf die nur jeweils ein Thread zugreifen kann, beispielsweise eine globale Variable oder ein Hardwaregerät.

OEM Adaptation Layer (OAL) Ein OAL ist ein logischer Codelayer zwischen dem Windows Embedded CE-Kernel und der Hardware des Zielgeräts. Der OAL wird physisch mit den Kernelbibliotheken gelinkt, um die ausführbare Kerneldatei zu erstellen.

Operating System Benchmark (OSBench) Dieses Tool wird zum Messen der Leistung des Schedulers verwendet.

OS Design Ein Projekt im Platform Builder für Windows Embedded CE6 R2, das ein benutzerdefiniertes, binäres Run-Time Image des Betriebssystems Windows Embedded CE6 R2 generiert.

Platform Dependent Driver (PDD) Der PDD-Layer eines mehrschichtigen Treibers ist direkt mit der Hardware gelinkt und führt die hardwarespezifische Verarbeitung aus.

Power Manager Der Power Manager steuert den Energieverbrauch des Systems, indem er dem System oder einzelnen Treibern einen Energiestatus zwischen D0 (aktiviert) oder D4 (deaktiviert) zuweist. Außerdem koordiniert der Power Manager die Statusübergänge basierend auf der Benutzer- und Systemaktivität sowie bestimmten Anforderungen.

Production Quality OAL (PQOAL) Der PQOAL ist eine standardisierte OAL-Struktur, die die Entwicklung eines OAL vereinfacht und beschleunigt. PQOAL verbessert über Codebibliotheken, Verzeichnisstruktur, die die Wiederverwendung des Codes unterstützen, zentralisierte Konfigurationsdateien und eine konsistente prozessor- und hardwareübergreifende Architektur die Erstellung von OAL-Komponenten.

Prozess Ein Prozess ist ein Programm in Windows Embedded CE. Jeder Prozess kann mehrere Threads umfassen. Ein Prozess kann im Benutzer- oder Kernelbereich ausgeführt werden.

Quick Fix Engineering (QFE) Diese Windows Embedded CE-Patches sind auf der Microsoft-Website verfügbar. QFEs beheben Probleme und stellen neue Features bereit.

Reflector-Dienst Dieser Dienst führt Anforderungen für Benutzermodustreibers aus, um diesen den Zugriff auf den Kernel und die Hardware zu ermöglichen.

Registrierung Der Informationspeicher in Windows Embedded CE, der die Konfigurationsinformationen für Hardware- und Softwarekomponenten enthält.

Remote Performance Monitor Diese Anwendung überwacht die Echtzeit-Leistung des Betriebssystems. Außerdem werden die Speicherbelegung, Netzwerklatenzen und andere Elemente überwacht.

Run-Time Image Die Binärdatei, die auf einem Hardwaregerät bereitgestellt wird. Diese Datei enthält alle Dateien, die das Betriebssystem für Anwendungen und Treiber benötigt.

Semaphore Ein Semaphore-Objekt ist ein Synchronisierungsobjekt, das den Zugriff auf Hardware- und Softwareressourcen schützt, um nur einer bestimmten Anzahl gleichzeitiger Threads den Zugriff zu gestatten. Das Semaphore hat einen Wert zwischen 0 und einem angegebenen Maximalwert. Der Wert wird dekrementiert, wenn ein Thread das Warten auf das Semaphore-Objekt beendet und inkrementiert, wenn ein Thread das Semaphore freigibt. Wenn der Wert 0 erreicht wird, kann kein

Thread auf die Ressource zugreifen, die vom Semaphore geschützt wird. Der Status eines Semaphore ist auf signalisiert (der Wert ist größer als 0) oder auf nicht signalisiert (der Wert ist 0) festgelegt.

Shell Die Shell ist die Software, die Benutzerinteraktionen mit dem Gerät interpretiert. Die Shell wird beim Gerätestart aktiviert. Die Standardshell, AYGShell, umfasst einen Desktop, ein Startmenü und eine Symbolleiste, die den Desktopversionen von Windows ähnlich sind.

Software Development Kit (SDK) Das SDK ermöglicht den Entwicklern von Drittanbietern das Erstellen von Anwendungen für ein benutzerdefinierten Windows Embedded CE6 R2 Run-Time Image.

Sources-Datei Eine Sources-Datei ist eine Textdatei, die in einem Unterverzeichnis die Makrodefinitionen für den Quellcode festlegt. Build.exe verwendet diese Makrodefinitionen, um zu bestimmen, wie der Quellcode kompiliert und gelinkt wird.

Streamschnittstellentreiber Ein Streamschnittstellentreiber ist ein Treiber, der die Streamschnittstellenfunktionen unabhängig vom Gerät bereitstellt. Alle Treiber, außer die von GWES verwalteten systemeigenen Treiber, exportieren eine Streamschnittstelle.

Systemeigener Treiber Touchscreen-, Tastatur- und Anzeigetreiber sind die einzigen systemeigenen Treiber in Windows Embedded CE. Diese Treiber werden von GWES anstatt dem Geräte-Manager verwaltet.

Teilprojekt Eine Gruppe von Dateien, die in einem OS Design integriert und wiederverwendet oder aus dem OS Design entfernt werden können.

Umgebungsvariable Eine Windows-Umgebungsvariable, die Features aktiviert oder deaktiviert. Diese Variable wird normalerweise verwendet, um das Buildsystem und das OS Design im Katalog zu konfigurieren.

Synchronisierungsobjekte Das Handle eines Synchronisierungsobjekts kann in einer Wait-Funktion angegeben werden, um die Ausführung mehrerer Threads zu koordinieren.

Synchroner Zugriff Mindestens zwei separate Threads verwenden den gleichen Puffer. Auf den Puffer kann nur jeweils ein Thread zugreifen. Ein anderer Thread kann erst auf den Puffer zugreifen, nachdem der aktuelle Thread den Puffer freigegeben hat.

Sysgen Die Sysgen-Phase ist der erste Schritt im Buildprozess, in dem die Public- und BSP-Ordner gefiltert werden. In der Sysgen-Phase werden die zu den OS Design-Komponenten gehörenden Dateien identifiziert. Während dieser Phase werden die im OS Design ausgewählten Komponenten mit ausführbaren Dateien gelinkt und in den OS Design-Ordner kopiert.

Sysgen-Variable Eine Direktive des CE-Buildprozesses in der Sysgen-Phase, in der ausgewählte CE-Features gelinkt werden.

Sysintr Der Wert, der einem IRQ entspricht. Sysintr wird verwendet, um ein Event zu signalisieren. Dieser Wert wird von einer ISR in Beantwortung eines Interrupts zurückgegeben.

Target Control-Shell Eine Shell in Platform Builder für Visual Studio, die den Zugriff auf Debuggerbefehle ermöglicht. Die Target Control-Shell ist verfügbar, wenn die Verbindung mit dem Zielsystem über KITL hergestellt wird.

Thread Die kleinste Softwareeinheit, die der Scheduler auf dem Betriebssystem verwalten kann. Ein Treiber oder eine Anwendung kann mehrere Threads umfassen.

Virtueller Speicher Der virtuelle Speicher ist eine Methode zum Abstrahieren des physischen Speichers eines Systems, damit dieser für Prozesse zusammenhängend ist. Für jeden Prozess in Windows Embedded CE 6.0 R2 sind zwei GB virtueller Speicher verfügbar. Dieser Speicher muss im virtuellen Speicherbereich des Prozesses mit MmMapIoSpace oder OALPAtoVA zugeordnet werden, damit ein Prozess auf den physischen Speicher zugreifen kann.

Windows Embedded CE Test Kit (CETK) Das CETK wird verwendet, um die für Windows Embedded CE entwickelten Gerätetreiber zu testen.

Stichwortverzeichnis

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

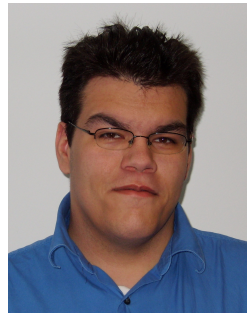
Z

Über die Autoren

Nicolas Besson

Nicolas Besson hat mehr als sieben Jahre technische Erfahrung mit Windows Embedded CE-Technologien. Er spezialisiert sich derzeit auf die Softwareentwicklung und das Projektmanagement bei Adeneo, einem Microsoft Gold Embedded Partner, der weltweit vertreten ist und sich auf Windows Embedded CE-Technologien konzentriert. Nicolas ist seit zwei Jahren ein Microsoft eMVP. Um sein Wissen weiterzugeben, hält er Schulungen für Unternehmen in aller Welt ab. In seinem Blog unter *http://nicolasbesson.blogspot.com* finden Sie Insider-Informationen zu Windows Embedded CE-Technologien.

Ray Marcilla



Ray Marcilla ist ein Entwickler von eingebetteter Software in der amerikanischen Niederlassung von Adeneo in Bellevue, Washington. Ray verfügt über langjährige Erfahrung in der Anwendungsentwicklung mit nativem und .NET-Code sowie in der CE-Treiberentwicklung. Außerdem wirkt er an technischen CE-Präsentationen und Workshops mit. Ray hat an zahlreichen interessanten Projekten für ARM- und x86-Entwicklungsplattformen gearbeitet. In seiner Freizeit studiert er Fremdsprachen. Ray spricht fließend Japanisch und hat gute Kenntnisse in Koreanisch und Französisch.

Rajesh Kakde

Rajesh arbeitet seit 2001 mit Windows Embedded CE. Er hat in mehreren Ländern in verschiedenen Branchen gearbeitet, einschließlich der Unterhaltungselektronik und mit Echtzeit-Geräten im Industrieeinsatz. Er hat langjährige Erfahrung in der Entwicklung von BSPs und Treibern sowie in der Anwendungsentwicklung.

Ray ist derzeit als Senior Windows Embedded Consultant bei der Adeneo Corp. beschäftigt, wo er seine technischen Fachkenntnisse bezüglich BSPs und Treibern einsetzt und verschiedene OEM-Projekte verwaltet. Außerdem hält er Workshops und Schulungen für Windows Embedded CE-Workshops ab, in denen er Ideen austauscht und Benutzern seine Leidenschaft für die Technologie vermittelt.